Understanding The Linux Virtual Memory Manager

Mel Gorman

July 9, 2007
Preface

Linux is developed with a stronger practical emphasis than a theoretical one. When new algorithms or changes to existing implementations are suggested, it is common to request code to match the argument. Many of the algorithms used in the Virtual Memory (VM) system were designed by theorists but the implementations have now diverged from the theory considerably. In part, Linux does follow the traditional development cycle of design to implementation but it is more common for changes to be made in reaction to how the system behaved in the “real-world” and intuitive decisions by developers.

This means that the VM performs well in practice but there is very little VM specific documentation available except for a few incomplete overviews in a small number of websites, except the web site containing an earlier draft of this book of course! This has lead to the situation where the VM is fully understood only by a small number of core developers. New developers looking for information on how it functions are generally told to read the source and little or no information is available on the theoretical basis for the implementation. This requires that even a casual observer invest a large amount of time to read the code and study the field of Memory Management.

This book, gives a detailed tour of the Linux VM as implemented in 2.4.22 and gives a solid introduction of what to expect in 2.6. As well as discussing the implementation, the theory it is is based on will also be introduced. This is not intended to be a memory management theory book but it is often much simpler to understand why the VM is implemented in a particular fashion if the underlying basis is known in advance.

To complement the description, the appendix includes a detailed code commentary on a significant percentage of the VM. This should drastically reduce the amount of time a developer or researcher needs to invest in understanding what is happening inside the Linux VM. As VM implementations tend to follow similar code patterns even between major versions. This means that with a solid understanding of the 2.4 VM, the later 2.5 development VMs and the final 2.6 release will be decipherable in a number of weeks.

The Intended Audience

Anyone interested in how the VM, a core kernel subsystem, works will find answers to many of their questions in this book. The VM, more than any other subsystem,
affects the overall performance of the operating system. It is also one of the most poorly understood and badly documented subsystem in Linux, partially because there is, quite literally, so much of it. It is very difficult to isolate and understand individual parts of the code without first having a strong conceptual model of the whole VM, so this book intends to give a detailed description of what to expect without going to the source.

This material should be of prime interest to new developers interested in adapting the VM to their needs and to readers who simply would like to know how the VM works. It also will benefit other subsystem developers who want to get the most from the VM when they interact with it and operating systems researchers looking for details on how memory management is implemented in a modern operating system. For others, who are just curious to learn more about a subsystem that is the focus of so much discussion, they will find an easy to read description of the VM functionality that covers all the details without the need to plough through source code.

However, it is assumed that the reader has read at least one general operating system book or one general Linux kernel orientated book and has a general knowledge of C before tackling this book. While every effort is made to make the material approachable, some prior knowledge of general operating systems is assumed.

Book Overview

In chapter 1, we go into detail on how the source code may be managed and deciphered. Three tools will be introduced that are used for the analysis, easy browsing and management of code. The main tools are the Linux Cross Referencing (LXR) tool which allows source code to be browsed as a web page and CodeViz for generating call graphs which was developed while researching this book. The last tool, PatchSet is for managing kernels and the application of patches. Applying patches manually can be time consuming and the use of version control software such as CVS (http://www.cshome.org/) or BitKeeper (http://www.bitmover.com) are not always an option. With this tool, a simple specification file determines what source to use, what patches to apply and what kernel configuration to use.

In the subsequent chapters, each part of the Linux VM implementation will be discussed in detail, such as how memory is described in an architecture independent manner, how processes manage their memory, how the specific allocators work and so on. Each will refer to the papers that describe closest the behaviour of Linux as well as covering in depth the implementation, the functions used and their call graphs so the reader will have a clear view of how the code is structured. At the end of each chapter, there will be a “What’s New” section which introduces what to expect in the 2.6 VM.

The appendices are a code commentary of a significant percentage of the VM. It gives a line by line description of some of the more complex aspects of the VM. The style of the VM tends to be reasonably consistent, even between major releases of the kernel so an in-depth understanding of the 2.4 VM will be an invaluable aid to understanding the 2.6 kernel when it is released.
What’s New in 2.6

At the time of writing, 2.6.0-test4 has just been released so 2.6.0-final is due “any month now” which means December 2003 or early 2004. Fortunately the 2.6 VM, in most ways, is still quite recognisable in comparison to 2.4. However, there is some new material and concepts in 2.6 and it would be pity to ignore them so to address this, hence the “What’s New in 2.6” sections. To some extent, these sections presume you have read the rest of the book so only glance at them during the first reading. If you decide to start reading 2.5 and 2.6 VM code, the basic description of what to expect from the “Whats New” sections should greatly aid your understanding. It is important to note that the sections are based on the 2.6.0-test4 kernel which should not change significantly before 2.6. As they are still subject to change though, you should still treat the “What’s New” sections as guidelines rather than definite facts.

Companion CD

A companion CD is included with this book which is intended to be used on systems with GNU/Linux installed. Mount the CD on /cdrom as followed;

    root@joshua:/$ mount /dev/cdrom /cdrom -o exec

A copy of Apache 1.3.27 (http://www.apache.org/) has been built and configured to run but it requires the CD be mounted on /cdrom/. To start it, run the script /cdrom/start_server. If there are no errors, the output should look like:

    mel@joshua:~$ /cdrom/start_server
Starting CodeViz Server: done
Starting Apache Server: done

The URL to access is http://localhost:10080/

If the server starts successfully, point your browser to http://localhost:10080 to avail of the CDs web services. Some features included with the CD are:

- A web server started is available which is started by /cdrom/start_server. After starting it, the URL to access is http://localhost:10080. It has been tested with Red Hat 7.3 and Debian Woody;

- The whole book is included in HTML, PDF and plain text formats from /cdrom/docs. It includes a searchable index for functions that have a commentary available. If a function is searched for that does not have a commentary, the browser will be automatically redirected to LXR;

- A web browsable copy of the Linux 2.4.22 source is available courtesy of LXR
- Generate call graphs with an online version of the CodeViz tool.

- The VM Regress, CodeViz and patchset packages which are discussed in Chapter 1 are available in /cdrom/software. gcc-3.0.4 is also provided as it is required for building CodeViz.

To shutdown the server, run the script /cdrom/stop_server and the CD may then be unmounted.

Typographic Conventions

The conventions used in this document are simple. New concepts that are introduced as well as URLs are in italicised font. Binaries and package names are are in bold. Structures, field names, compile time defines and variables are in a constant-width font. At times when talking about a field in a structure, both the structure and field name will be included like page→list for example. Filenames are in a constant-width font but include files have angle brackets around them like <linux/mm.h> and may be found in the include/ directory of the kernel source.

Acknowledgments

The compilation of this book was not a trivial task. This book was researched and developed in the open and it would be remiss of me not to mention some of the people who helped me at various intervals. If there is anyone I missed, I apologise now.

First, I would like to thank John O’Gorman who tragically passed away while the material for this book was being researched. It was his experience and guidance that largely inspired the format and quality of this book.

Secondly, I would like to thank Mark L. Taub from Prentice Hall PTR for giving me the opportunity to publish this book. It has being a rewarding experience and it made trawling through all the code worthwhile. Massive thanks go to my reviewers who provided clear and detailed feedback long after I thought I had finished writing. Finally, on the publishers front, I would like to thank Bruce Perens for allowing me to publish under the Bruce Peren’s Open Book Series (http://www.perens.com/Books).

With the technical research, a number of people provided invaluable insight. Abhishek Nayani, was a source of encouragement and enthusiasm early in the research. Ingo Oeser kindly provided invaluable assistance early on with a detailed explanation on how data is copied from userspace to kernel space including some valuable historical context. He also kindly offered to help me if I felt I ever got lost in the twisty maze of kernel code. Scott Kaplan made numerous corrections to a number of systems from non-contiguous memory allocation, to page replacement policy. Jonathon Corbet provided the most detailed account of the history of the kernel development with the kernel page he writes for Linux Weekly News. Zack Brown, the chief behind Kernel Traffic, is the sole reason I did not drown in kernel
related mail. IBM, as part of the Equinox Project, provided an xSeries 350 which was invaluable for running my own test kernels on machines larger than what I previously had access to. Finally, Patrick Healy was crucial to ensuring that this book was consistent and approachable to people who are familiar, but not experts, on Linux or memory management.

A number of people helped with smaller technical issues and general inconsistencies where material was not covered in sufficient depth. They are Mult Ben-Yehuda, Parag Sharma, Matthew Dobson, Roger Luethi, Brian Lowe and Scott Crosby. All of them sent corrections and queries on different parts of the document which ensured too much prior knowledge was assumed.

Carl Spalletta sent a number of queries and corrections to every aspect of the book in its earlier online form. Steve Greenland sent a large number of grammar corrections. Philipp Marek went above and beyond being helpful sending over 90 separate corrections and queries on various aspects. Long after I thought I was finished, Aris Sotiropoulos sent a large number of small corrections and suggestions. The last person, whose name I cannot remember but is an editor for a magazine sent me over 140 corrections against an early version to the document. You know who you are, thanks.

Eleven people sent a few corrections, though small, were still missed by several of my own checks. They are Marek Januszewski, Amit Shah, Adrian Stanciu, Andy Isaacson, Jean Francois Martinez, Glen Kaukola, Wolfgang Oertl, Michael Babcock, Kirk True, Chuck Luciano and David Wilson.

On the development of VM Regress, there were nine people who helped me keep it together. Danny Faught and Paul Larson both sent me a number of bug reports and helped ensure it worked with a variety of different kernels. Cliff White, from the OSDL labs ensured that VM Regress would have a wider application than my own test box. Dave Olien, also associated with the OSDL labs was responsible for updating VM Regress to work with 2.5.64 and later kernels. Albert Cahalan sent all the information I needed to make it function against later proc utilities. Finally, Andrew Morton, Rik van Riel and Scott Kaplan all provided insight on what direction the tool should be developed to be both valid and useful.

The last long list are people who sent me encouragement and thanks at various intervals. They are Martin Bligh, Paul Rolland, Mohamed Ghouse, Samuel Chessman, Ersin Er, Mark Hoy, Michael Martin, Martin Gallwey, Ravi Parimi, Daniel Colt, Ahdna Shafi, Xiong Quanren, Dave Airlie, Der Herr Hofrat, Ida Hallgren, Manu Anand, Eugene Teo, Diego Calleja and Ed Cashin. Thanks, the encouragement was heartening.

In conclusion, I would like to thank a few people without whom, I would not have completed this. I would like to thank my parents who kept me going long after I should have been earning enough money to support myself. I would like to thank my girlfriend Karen, who patiently listened to rants, tech babble, angsting over the book and made sure I was the person with the best toys. Kudos to friends who dragged me away from the computer periodically and kept me relatively sane, including Daren who is cooking me dinner as I write this. Finally, I would like to thank the thousands of hackers that have contributed to GNU, the Linux kernel
and other Free Software projects over the years who without I would not have an excellent system to write about. It was an inspiration to me to see such dedication when I first started programming on my own PC 6 years ago after finally figuring out that Linux was not an application for Windows used for reading email.
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4 Memory Regions</td>
<td>60</td>
</tr>
<tr>
<td>4.5 Exception Handling</td>
<td>75</td>
</tr>
<tr>
<td>4.6 Page Faulting</td>
<td>76</td>
</tr>
<tr>
<td>4.7 Copying To/From Userspace</td>
<td>82</td>
</tr>
<tr>
<td>4.8 What’s New in 2.6</td>
<td>84</td>
</tr>
<tr>
<td>5 Boot Memory Allocator</td>
<td>89</td>
</tr>
<tr>
<td>5.1 Representing the Boot Map</td>
<td>90</td>
</tr>
<tr>
<td>5.2 Initialising the Boot Memory Allocator</td>
<td>92</td>
</tr>
<tr>
<td>5.3 Allocating Memory</td>
<td>93</td>
</tr>
<tr>
<td>5.4 Freeing Memory</td>
<td>94</td>
</tr>
<tr>
<td>5.5 Retiring the Boot Memory Allocator</td>
<td>95</td>
</tr>
<tr>
<td>5.6 What’s New in 2.6</td>
<td>97</td>
</tr>
<tr>
<td>6 Physical Page Allocation</td>
<td>98</td>
</tr>
<tr>
<td>6.1 Managing Free Blocks</td>
<td>98</td>
</tr>
<tr>
<td>6.2 Allocating Pages</td>
<td>99</td>
</tr>
<tr>
<td>6.3 Free Pages</td>
<td>102</td>
</tr>
<tr>
<td>6.4 Get Free Page (GFP) Flags</td>
<td>103</td>
</tr>
<tr>
<td>6.5 Avoiding Fragmentation</td>
<td>106</td>
</tr>
<tr>
<td>6.6 What’s New In 2.6</td>
<td>106</td>
</tr>
<tr>
<td>7 Non-Contiguous Memory Allocation</td>
<td>110</td>
</tr>
<tr>
<td>7.1 Describing Virtual Memory Areas</td>
<td>110</td>
</tr>
<tr>
<td>7.2 Allocating A Non-Contiguous Area</td>
<td>111</td>
</tr>
<tr>
<td>7.3 Freeing A Non-Contiguous Area</td>
<td>113</td>
</tr>
<tr>
<td>7.4 What’s New in 2.6</td>
<td>114</td>
</tr>
<tr>
<td>8 Slab Allocator</td>
<td>115</td>
</tr>
<tr>
<td>8.1 Caches</td>
<td>118</td>
</tr>
<tr>
<td>8.2 Slabs</td>
<td>129</td>
</tr>
<tr>
<td>8.3 Objects</td>
<td>135</td>
</tr>
<tr>
<td>8.4 Sizes Cache</td>
<td>137</td>
</tr>
<tr>
<td>8.5 Per-CPU Object Cache</td>
<td>138</td>
</tr>
<tr>
<td>8.6 Slab Allocator Initialisation</td>
<td>141</td>
</tr>
<tr>
<td>8.7 Interfacing with the Buddy Allocator</td>
<td>142</td>
</tr>
<tr>
<td>8.8 What’s New In 2.6</td>
<td>142</td>
</tr>
<tr>
<td>9 High Memory Management</td>
<td>144</td>
</tr>
<tr>
<td>9.1 Managing the PKMap Address Space</td>
<td>144</td>
</tr>
<tr>
<td>9.2 Mapping High Memory Pages</td>
<td>145</td>
</tr>
<tr>
<td>9.3 Mapping High Memory Pages Atomically</td>
<td>147</td>
</tr>
<tr>
<td>9.4 Bounce Buffers</td>
<td>148</td>
</tr>
<tr>
<td>9.5 Emergency Pools</td>
<td>150</td>
</tr>
<tr>
<td>9.6 What’s New in 2.6</td>
<td>151</td>
</tr>
</tbody>
</table>
CONTENTS

B Describing Physical Memory 201
  B.1 Initialising Zones ........................................ 202
  B.2 Page Operations ........................................... 216

C Page Table Management 221
  C.1 Page Table Initialisation ................................. 222
  C.2 Page Table Walking ....................................... 230

D Process Address Space 232
  D.1 Process Memory Descriptors ............................... 236
  D.2 Creating Memory Regions .................................. 243
  D.3 Searching Memory Regions ................................. 293
  D.4 Locking and Unlocking Memory Regions ...................... 299
  D.5 Page Faulting ............................................ 313
  D.6 Page-Related Disk IO ...................................... 341

E Boot Memory Allocator 381
  E.1 Initialising the Boot Memory Allocator .................... 382
  E.2 Allocating Memory ......................................... 385
  E.3 Freeing Memory ............................................ 395
  E.4 Retiring the Boot Memory Allocator ....................... 397

F Physical Page Allocation 404
  F.1 Allocating Pages ........................................... 405
  F.2 Allocation Helper Functions ............................... 418
  F.3 Free Pages .................................................. 420
  F.4 Free Helper Functions ..................................... 425

G Non-Contiguous Memory Allocation 426
  G.1 Allocating A Non-Contiguous Area ......................... 427
  G.2 Freeing A Non-Contiguous Area ............................ 437

H Slab Allocator 442
  H.1 Cache Manipulation ........................................ 444
  H.2 Slabs ........................................................ 464
  H.3 Objects ..................................................... 472
  H.4 Sizes Cache ................................................ 487
  H.5 Per-CPU Object Cache ...................................... 490
  H.6 Slab Allocator Initialisation ............................... 498
  H.7 Interfacing with the Buddy Allocator ...................... 499

I High Memory Management 500
  I.1 Mapping High Memory Pages ................................. 502
  I.2 Mapping High Memory Pages Atomically ...................... 508
  I.3 Unmapping Pages ............................................ 510
  I.4 Unmapping High Memory Pages Atomically ................... 512
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I.5</td>
<td>Bounce Buffers</td>
<td>513</td>
</tr>
<tr>
<td>I.6</td>
<td>Emergency Pools</td>
<td>521</td>
</tr>
<tr>
<td><strong>J</strong></td>
<td>Page Frame Reclamation</td>
<td>523</td>
</tr>
<tr>
<td>J.1</td>
<td>Page Cache Operations</td>
<td>525</td>
</tr>
<tr>
<td>J.2</td>
<td>LRU List Operations</td>
<td>535</td>
</tr>
<tr>
<td>J.3</td>
<td>Refilling <code>inactive_list</code></td>
<td>540</td>
</tr>
<tr>
<td>J.4</td>
<td>Reclaiming Pages from the LRU Lists</td>
<td>542</td>
</tr>
<tr>
<td>J.5</td>
<td>Shrinking all caches</td>
<td>550</td>
</tr>
<tr>
<td>J.6</td>
<td>Swapping Out Process Pages</td>
<td>554</td>
</tr>
<tr>
<td>J.7</td>
<td>Page Swap Daemon</td>
<td>565</td>
</tr>
<tr>
<td><strong>K</strong></td>
<td>Swap Management</td>
<td>570</td>
</tr>
<tr>
<td>K.1</td>
<td>Scanning for Free Entries</td>
<td>572</td>
</tr>
<tr>
<td>K.2</td>
<td>Swap Cache</td>
<td>577</td>
</tr>
<tr>
<td>K.3</td>
<td>Swap Area IO</td>
<td>584</td>
</tr>
<tr>
<td>K.4</td>
<td>Activating a Swap Area</td>
<td>594</td>
</tr>
<tr>
<td>K.5</td>
<td>Deactivating a Swap Area</td>
<td>606</td>
</tr>
<tr>
<td><strong>L</strong></td>
<td>Shared Memory Virtual Filesystem</td>
<td>620</td>
</tr>
<tr>
<td>L.1</td>
<td>Initialising <code>shmfs</code></td>
<td>622</td>
</tr>
<tr>
<td>L.2</td>
<td>Creating Files in <code>tmpfs</code></td>
<td>628</td>
</tr>
<tr>
<td>L.3</td>
<td>File Operations in <code>tmpfs</code></td>
<td>632</td>
</tr>
<tr>
<td>L.4</td>
<td>Inode Operations in <code>tmpfs</code></td>
<td>646</td>
</tr>
<tr>
<td>L.5</td>
<td>Page Faulting within a Virtual File</td>
<td>655</td>
</tr>
<tr>
<td>L.6</td>
<td>Swap Space Interaction</td>
<td>667</td>
</tr>
<tr>
<td>L.7</td>
<td>Setting up Shared Regions</td>
<td>674</td>
</tr>
<tr>
<td>L.8</td>
<td>System V IPC</td>
<td>678</td>
</tr>
<tr>
<td><strong>M</strong></td>
<td>Out of Memory Management</td>
<td>685</td>
</tr>
<tr>
<td>M.1</td>
<td>Determining Available Memory</td>
<td>686</td>
</tr>
<tr>
<td>M.2</td>
<td>Detecting and Recovering from OOM</td>
<td>688</td>
</tr>
<tr>
<td><strong>Reference</strong></td>
<td>Bibliography</td>
<td>694</td>
</tr>
<tr>
<td></td>
<td>Code Commentary Index</td>
<td>698</td>
</tr>
<tr>
<td></td>
<td>Index</td>
<td>703</td>
</tr>
</tbody>
</table>
Code Commentary Contents
List of Figures

1.1 Example Patch ...................................................... 7
2.1 Relationship Between Nodes, Zones and Pages ..................... 15
2.2 Zone Watermarks .................................................... 19
2.3 Call Graph: setup_memory() .......................................... 20
2.4 Sleeping On a Locked Page .......................................... 21
2.5 Call Graph: free_area_init() ......................................... 23
3.1 Page Table Layout .................................................... 33
3.2 Linear Address Bit Size Macros ...................................... 34
3.3 Linear Address Size and Mask Macros ............................... 34
3.4 Call Graph: paging_init() ........................................... 40
4.1 Kernel Address Space ................................................ 53
4.2 Data Structures related to the Address Space ....................... 55
4.3 Memory Region Flags ................................................ 63
4.4 Call Graph: sys_mmap2() ............................................. 67
4.5 Call Graph: get_unmapped_area() ................................... 68
4.6 Call Graph: insert_vm_struct() ...................................... 70
4.7 Call Graph: sys_mremap() ........................................... 72
4.8 Call Graph: move_vma() ............................................. 72
4.9 Call Graph: move_page_tables() ..................................... 73
4.10 Call Graph: sys_mlock() ............................................. 74
4.11 Call Graph: do_munmap() ............................................ 74
4.12 Call Graph: do_page_fault() ......................................... 78
4.13 do_page_fault() Flow Diagram ...................................... 79
4.14 Call Graph: handle_mm_fault() ..................................... 80
4.15 Call Graph: do_no_page() ........................................... 80
4.16 Call Graph: do_swap_page() ......................................... 81
4.17 Call Graph: do_wp_page() ........................................... 82
5.1 Call Graph: alloc_bootmem() ......................................... 93
5.2 Call Graph: mem_init() .............................................. 95
5.3 Initialising mem_map and the Main Physical Page Allocator ....... 96
6.1 Free page block management .......................................... 99
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2</td>
<td>Allocating physical pages</td>
<td>101</td>
</tr>
<tr>
<td>6.3</td>
<td>Call Graph: alloc_pages()</td>
<td>101</td>
</tr>
<tr>
<td>6.4</td>
<td>Call Graph: __free_pages()</td>
<td>102</td>
</tr>
<tr>
<td>7.1</td>
<td>vmalloc Address Space</td>
<td>111</td>
</tr>
<tr>
<td>7.2</td>
<td>Call Graph: vmalloc()</td>
<td>112</td>
</tr>
<tr>
<td>7.3</td>
<td>Relationship between vmalloc(), alloc_page() and Page Faulting</td>
<td>113</td>
</tr>
<tr>
<td>7.4</td>
<td>Call Graph: vfree()</td>
<td>114</td>
</tr>
<tr>
<td>8.1</td>
<td>Layout of the Slab Allocator</td>
<td>116</td>
</tr>
<tr>
<td>8.2</td>
<td>Slab page containing Objects Aligned to L1 CPU Cache</td>
<td>117</td>
</tr>
<tr>
<td>8.3</td>
<td>Call Graph: kmem_cache_create()</td>
<td>126</td>
</tr>
<tr>
<td>8.4</td>
<td>Call Graph: kmem_cache_reap()</td>
<td>127</td>
</tr>
<tr>
<td>8.5</td>
<td>Call Graph: kmem_cache_shrink()</td>
<td>128</td>
</tr>
<tr>
<td>8.6</td>
<td>Call Graph: __kmem_cache_shrink()</td>
<td>128</td>
</tr>
<tr>
<td>8.7</td>
<td>Call Graph: kmem_cache_destroy()</td>
<td>129</td>
</tr>
<tr>
<td>8.8</td>
<td>Page to Cache and Slab Relationship</td>
<td>130</td>
</tr>
<tr>
<td>8.9</td>
<td>Slab With Descriptor On-Slab</td>
<td>131</td>
</tr>
<tr>
<td>8.10</td>
<td>Slab With Descriptor Off-Slab</td>
<td>132</td>
</tr>
<tr>
<td>8.11</td>
<td>Call Graph: kmem_cache_grow()</td>
<td>132</td>
</tr>
<tr>
<td>8.12</td>
<td>Initialised kmem_bufctl_t Array</td>
<td>133</td>
</tr>
<tr>
<td>8.13</td>
<td>Call Graph: kmem_slab_destroy()</td>
<td>135</td>
</tr>
<tr>
<td>8.14</td>
<td>Call Graph: kmem_cache_alloc()</td>
<td>136</td>
</tr>
<tr>
<td>8.15</td>
<td>Call Graph: kmem_cache_free()</td>
<td>136</td>
</tr>
<tr>
<td>8.16</td>
<td>Call Graph: kmalloc()</td>
<td>138</td>
</tr>
<tr>
<td>8.17</td>
<td>Call Graph: kfree()</td>
<td>138</td>
</tr>
<tr>
<td>9.1</td>
<td>Call Graph: kmap()</td>
<td>146</td>
</tr>
<tr>
<td>9.2</td>
<td>Call Graph: kunmap()</td>
<td>148</td>
</tr>
<tr>
<td>9.3</td>
<td>Call Graph: create_bounce()</td>
<td>149</td>
</tr>
<tr>
<td>9.4</td>
<td>Call Graph: bounce_end_io_read/write()</td>
<td>150</td>
</tr>
<tr>
<td>9.5</td>
<td>Acquiring Pages from Emergency Pools</td>
<td>151</td>
</tr>
<tr>
<td>10.1</td>
<td>Page Cache LRU Lists</td>
<td>155</td>
</tr>
<tr>
<td>10.2</td>
<td>Call Graph: generic_file_read()</td>
<td>158</td>
</tr>
<tr>
<td>10.3</td>
<td>Call Graph: add_to_page_cache()</td>
<td>159</td>
</tr>
<tr>
<td>10.4</td>
<td>Call Graph: shrink_caches()</td>
<td>163</td>
</tr>
<tr>
<td>10.5</td>
<td>Call Graph: swap_out()</td>
<td>163</td>
</tr>
<tr>
<td>10.6</td>
<td>Call Graph: kswapd()</td>
<td>165</td>
</tr>
<tr>
<td>11.1</td>
<td>Storing Swap Entry Information in swp_entry_t</td>
<td>172</td>
</tr>
<tr>
<td>11.2</td>
<td>Call Graph: get_swap_page()</td>
<td>173</td>
</tr>
<tr>
<td>11.3</td>
<td>Call Graph: add_to_swap_cache()</td>
<td>174</td>
</tr>
<tr>
<td>11.4</td>
<td>Adding a Page to the Swap Cache</td>
<td>175</td>
</tr>
<tr>
<td>11.5</td>
<td>Call Graph: read_swap_cache_async()</td>
<td>177</td>
</tr>
<tr>
<td>11.6</td>
<td>Call Graph: sys_writepage()</td>
<td>178</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

12.1 Call Graph: init_tmpfs() ............................................. 183
12.2 Call Graph: shmem_create() ........................................ 187
12.3 Call Graph: shmem_nopage() ....................................... 188
12.4 Traversing Indirect Blocks in a Virtual File .................. 189
12.5 Call Graph: shmem_zero_setup() ................................. 191
12.6 Call Graph: sys_shmget() ........................................ 192

13.1 Call Graph: out_of_memory() ...................................... 195

14.1 Broad Overview on how VM Sub-Systems Interact ............... 199

D.1 Call Graph: mmput() ............................................... 241

E.1 Call Graph: free_bootmem() ....................................... 395

H.1 Call Graph: enable_all_cpucaches() ............................... 490
List of Tables

1.1 Kernel size as an indicator of complexity ........................................ 1
2.1 Flags Describing Page Status ..................................................... 30
2.2 Macros For Testing, Setting and Clearing page—flags Status Bits .... 31
3.1 Page Table Entry Protection and Status Bits ............................. 36
3.2 Translation Lookaside Buffer Flush API ..................................... 43
3.3 Translation Lookaside Buffer Flush API (cont) ............................ 44
3.4 Cache and TLB Flush Ordering .................................................. 45
3.5 CPU Cache Flush API ............................................................. 46
3.6 CPU D-Cache and I-Cache Flush API ........................................ 47
4.1 System Calls Related to Memory Regions .................................. 56
4.2 Functions related to memory region descriptors ......................... 59
4.3 Memory Region VMA API ....................................................... 69
4.4 Reasons For Page Faulting ...................................................... 77
4.5 Accessing Process Address Space API ....................................... 83
5.1 Boot Memory Allocator API for UMA Architectures .................... 90
5.2 Boot Memory Allocator API for NUMA Architectures ................... 91
6.1 Physical Pages Allocation API .................................................. 100
6.2 Physical Pages Free API ........................................................ 102
6.3 Low Level GFP Flags Affecting Zone Allocation ......................... 103
6.4 Low Level GFP Flags Affecting Allocator behaviour ................. 104
6.5 Low Level GFP Flag Combinations For High Level Use ............ 104
6.6 High Level GFP Flags Affecting Allocator Behaviour ................ 105
6.7 Process Flags Affecting Allocator behaviour ........................... 106
7.1 Non-Contiguous Memory Allocation API ................................ 112
7.2 Non-Contiguous Memory Free API ........................................... 113
8.1 Slab Allocator API for caches .................................................. 118
8.2 Internal cache static flags ....................................................... 123
8.3 Cache static flags set by caller ............................................... 123
8.4 Cache static debug flags ......................................................... 124
8.5 Cache Allocation Flags ........................................................ 124
8.6 Cache Constructor Flags ........................................... 125
9.1 High Memory Mapping API ......................................... 147
9.2 High Memory Unmapping API ..................................... 147
10.1 Page Cache API .................................................. 157
10.2 LRU List API ...................................................... 160
11.1 Swap Cache API .................................................. 176
Chapter 1

Introduction

Linux is a relatively new operating system that has begun to enjoy a lot of attention from the business, academic and free software worlds. As the operating system matures, its feature set, capabilities and performance grow but so, out of necessity does its size and complexity. The table in Figure 1.1 shows the size of the kernel source code in bytes and lines of code of the mm/ part of the kernel tree. This does not include the machine dependent code or any of the buffer management code and does not even pretend to be an accurate metric for complexity but still serves as a small indicator.

<table>
<thead>
<tr>
<th>Version</th>
<th>Release Date</th>
<th>Total Size</th>
<th>Size of mm/</th>
<th>Line count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>March 13th, 1992</td>
<td>5.9MiB</td>
<td>96KiB</td>
<td>3109</td>
</tr>
<tr>
<td>1.2.13</td>
<td>February 8th, 1995</td>
<td>11MiB</td>
<td>136KiB</td>
<td>4531</td>
</tr>
<tr>
<td>2.0.39</td>
<td>January 9th 2001</td>
<td>35MiB</td>
<td>204KiB</td>
<td>6792</td>
</tr>
<tr>
<td>2.2.22</td>
<td>September 16th, 2002</td>
<td>93MiB</td>
<td>292KiB</td>
<td>9554</td>
</tr>
<tr>
<td>2.4.22</td>
<td>August 25th, 2003</td>
<td>181MiB</td>
<td>436KiB</td>
<td>15724</td>
</tr>
<tr>
<td>2.6.0-test4</td>
<td>August 22nd, 2003</td>
<td>261MiB</td>
<td>604KiB</td>
<td>21714</td>
</tr>
</tbody>
</table>

Table 1.1: Kernel size as an indicator of complexity

As is the habit of open source developers in general, new developers asking questions are sometimes told to refer directly to the source with the “polite” acronym RTFS¹ or else are referred to the kernel newbies mailing list (http://www.kernelnewbies.org). With the Linux Virtual Memory (VM) manager, this used to be a suitable response as the time required to understand the VM could be measured in weeks and the books available devoted enough time to the memory management chapters to make the relatively small amount of code easy to navigate.

The books that describe the operating system such as Understanding the Linux Kernel [BC00] [BC03], tend to cover the entire kernel rather than one topic with the notable exception of device drivers [RC01]. These books, particularly Understanding

¹Read The Flaming Source. It doesn’t really stand for Flaming but there could be children watching.
1.1 Getting Started

The Linux Kernel, provide invaluable insight into kernel internals but they miss the details which are specific to the VM and not of general interest. For example, it is detailed in this book why \texttt{ZONE\_NORMAL} is exactly 896MiB and exactly how per-cpu caches are implemented. Other aspects of the VM, such as the boot memory allocator and the virtual memory filesystem which are not of general kernel interest are also covered by this book.

Increasingly, to get a comprehensive view on how the kernel functions, one is required to read through the source code line by line. This book tackles the VM specifically so that this investment of time to understand it will be measured in weeks and not months. The details which are missed by the main part of the book will be caught by the code commentary.

In this chapter, there will be an informal introduction to the basics of acquiring information on an open source project and some methods for managing, browsing and comprehending the code. If you do not intend to be reading the actual source, you may skip to Chapter 2.

1.1 Getting Started

One of the largest initial obstacles to understanding code is deciding where to start and how to easily manage, browse and get an overview of the overall code structure. If requested on mailing lists, people will provide some suggestions on how to proceed but a comprehensive methodology is rarely offered aside from suggestions to keep reading the source until it makes sense. In the following sections, some useful rules of thumb for open source code comprehension will be introduced and specifically on how they may be applied to the kernel.

1.1.1 Configuration and Building

With any open source project, the first step is to download the source and read the installation documentation. By convention, the source will have a \texttt{INSTALL} or \texttt{README} file at the top-level of the source tree [FF02]. In fact, some automated build tools such as \texttt{automake} require the install file to exist. These files will contain instructions for configuring and installing the package or will give a reference to where more information may be found. Linux is no exception as it includes a \texttt{README} which describes how the kernel may be configured and built.

The second step is to build the software. In earlier days, the requirement for many projects was to edit the \texttt{Makefile} by hand but this is rarely the case now. Free software usually uses at least \texttt{autoconf} to automate testing of the build environment and \texttt{automake} to simplify the creation of \texttt{Makefiles} so building is often as simple as:

\begin{verbatim}
me@joshua: project $ ./configure && make
\end{verbatim}

\footnote{http://www.gnu.org/software/autoconf/}
\footnote{http://www.gnu.org/software/automake/}
Some older projects, such as the Linux kernel, use their own configuration tools and some large projects such as the Apache webserver have numerous configuration options but usually the configure script is the starting point. In the case of the kernel, the configuration is handled by the Makefiles and supporting tools. The simplest means of configuration is to:

mel@joshua: linux-2.4.22 $ make config

This asks a long series of questions on what type of kernel should be built. Once all the questions have been answered, compiling the kernel is simply:

mel@joshua: linux-2.4.22 $ make bzImage && make modules

A comprehensive guide on configuring and compiling a kernel is available with the Kernel HOWTO\(^4\) and will not be covered in detail with this book. For now, we will presume you have one fully built kernel and it is time to begin figuring out how the new kernel actually works.

### 1.1.2 Sources of Information

Open Source projects will usually have a home page, especially since free project hosting sites such as [http://www.sourceforge.net](http://www.sourceforge.net) are available. The home site will contain links to available documentation and instructions on how to join the mailing list, if one is available. Some sort of documentation will always exist, even if it is as minimal as a simple README file, so read whatever is available. If the project is old and reasonably large, the web site will probably feature a Frequently Asked Questions (FAQ).

Next, join the development mailing list and lurk, which means to subscribe to a mailing list and read it without posting. Mailing lists are the preferred form of developer communication followed by, to a lesser extent, Internet Relay Chat (IRC) and online newgroups, commonly referred to as UseNet. As mailing lists often contain discussions on implementation details, it is important to read at least the previous months archives to get a feel for the developer community and current activity. The mailing list archives should be the first place to search if you have a question or query on the implementation that is not covered by available documentation. If you have a question to ask the developers, take time to research the questions and ask it the “Right Way” [RM01]. While there are people who will answer “obvious” questions, it will not do your credibility any favours to be constantly asking questions that were answered a week previously or are clearly documented.

Now, how does all this apply to Linux? First, the documentation. There is a README at the top of the source tree and a wealth of information is available in the Documentation/ directory. There also is a number of books on UNIX design [Vah96], Linux specifically [BC00] and of course this book to explain what to expect in the code.

---

one of the best online sources of information available on kernel development is the “Kernel Page” in the weekly edition of Linux Weekly News (http://www.lwn.net). It also reports on a wide range of Linux related topics and is worth a regular read. The kernel does not have a home web site as such but the closest equivalent is http://www.kernelnuebies.org which is a vast source of information on the kernel that is invaluable to new and experienced people alike.

here is a FAQ available for the Linux Kernel Mailing List (LKML) at http://www.tux.org/lkml/ that covers questions, ranging from the kernel development process to how to join the list itself. The list is archived at many sites but a common choice to reference is http://marc.theaimsgroup.com/?l=linux-kernel. Be aware that the mailing list is very high volume list which can be a very daunting read but a weekly summary is provided by the Kernel Traffic site at http://kt.zork.net/kernel-traffic/.

The sites and sources mentioned so far contain general kernel information but there are memory management specific sources. There is a Linux-MM web site at http://www.linux-mm.org which contains links to memory management specific documentation and a linux-mm mailing list. The list is relatively light in comparison to the main list and is archived at http://mail.nl.linux.org/linux-mm/.

The last site that to consult is the Kernel Trap site at http://www.kerneltrap.org. The site contains many useful articles on kernels in general. It is not specific to Linux but it does contain many Linux related articles and interviews with kernel developers.

As is clear, there is a vast amount of information that is available that may be consulted before resorting to the code. With enough experience, it will eventually be faster to consult the source directly but when getting started, check other sources of information first.

1.2 Managing the Source

The mainline or stock kernel is principally distributed as a compressed tape archive (.tar.bz) file which is available from your nearest kernel source repository, in Ireland’s case ftp://ftp.ie.kernel.org/. The stock kernel is always considered to be the one released by the tree maintainer. For example, at time of writing, the stock kernels for 2.2.x are those released by Alan Cox\(^5\), for 2.4.x by Marcelo Tosatti and for 2.5.x by Linus Torvalds. At each release, the full tar file is available as well as a smaller patch which contains the differences between the two releases. Patching is the preferred method of upgrading because of bandwidth considerations. Contributions made to the kernel are almost always in the form of patches which are unified diffs generated by the GNU tool diff.

**Why patches** Sending patches to the mailing list initially sounds clumsy but it is remarkable efficient in the kernel development environment. The principal advantage of patches is that it is much easier to read what changes have been made than to

\(^5\)Last minute update, Alan is just after announcing he was going on sabbatical and will no longer maintain the 2.2.x tree. There is no maintainer at the moment.
compare two full versions of a file side by side. A developer familiar with the code can easily see what impact the changes will have and if it should be merged. In addition, it is very easy to quote the email that includes the patch and request more information about it.

**Subtrees** At various intervals, individual influential developers may have their own version of the kernel distributed as a large patch to the main tree. These subtrees generally contain features or cleanups which have not been merged to the mainstream yet or are still being tested. Two notable subtrees is the -rmap tree maintained by Rik Van Riel, a long time influential VM developer and the -mm tree maintained by Andrew Morton, the current maintainer of the stock development VM. The -rmap tree contains a large set of features that for various reasons are not available in the mainline. It is heavily influenced by the FreeBSD VM and has a number of significant differences to the stock VM. The -mm tree is quite different to -rmap in that it is a testing tree with patches that are being tested before merging into the stock kernel.

**BitKeeper** In more recent times, some developers have started using a source code control system called BitKeeper (http://www.bitmover.com), a proprietary version control system that was designed with the Linux as the principal consideration. BitKeeper allows developers to have their own distributed version of the tree and other users may “pull” sets of patches called *changesets* from each others trees. This distributed nature is a very important distinction from traditional version control software which depends on a central server.

BitKeeper allows comments to be associated with each patch which is displayed as part of the release information for each kernel. For Linux, this means that the email that originally submitted the patch is preserved making the progress of kernel development and the meaning of different patches a lot more transparent. On release, a list of the patch titles from each developer is announced as well as a detailed list of all patches included.

As BitKeeper is a proprietary product, email and patches are still considered the only method for generating discussion on code changes. In fact, some patches will not be considered for acceptance unless there is first some discussion on the main mailing list as code quality is considered to be directly related to the amount of peer review [Ray02]. As the BitKeeper maintained source tree is exported in formats accessible to open source tools like CVS, patches are still the preferred means of discussion. It means that no developer is required to use BitKeeper for making contributions to the kernel but the tool is still something that developers should be aware of.
1.2.1 Diff and Patch

The two tools for creating and applying patches are **diff** and **patch**, both of which are GNU utilities available from the GNU website\(^6\). **diff** is used to generate patches and **patch** is used to apply them. While the tools have numerous options, there is a "preferred usage".

Patches generated with **diff** should always be **unified diff**, include the C function that the change affects and be generated from one directory above the kernel source root. A unified diff include more information that just the differences between two lines. It begins with a two line header with the names and creation date of the two files that **diff** is comparing. After that, the "diff" will consist of one or more "hunks". The beginning of each hunk is marked with a line beginning with @@ which includes the starting line in the source code and how many lines there is before and after the hunk is applied. The hunk includes "context" lines which show lines above and below the changes to aid a human reader. Each line begins with a +, - or blank. If the mark is +, the line is added. If a -, the line is removed and a blank is to leave the line alone as it is there just to provide context. The reasoning behind generating from one directory above the kernel root is that it is easy to see quickly what version the patch has been applied against and it makes the scripting of applying patches easier if each patch is generated the same way.

Let us take for example, a very simple change has been made to **mm/page_alloc.c** which adds a small piece of commentary. The patch is generated as follows. Note that this command should be all one line minus the backslashes.

```
mel@joshua: kernels/ $ diff -up \
   linux-2.4.22-clean/mm/page_alloc.c \
   linux-2.4.22-mel/mm/page_alloc.c > example.patch
```

This generates a unified context diff (-u switch) between two files and places the patch in example.patch as shown in Figure 1.2.1. It also displays the name of the affected C function.

From this patch, it is clear even at a casual glance what files are affected (**page_alloc.c**), what line it starts at (76) and the new lines added are clearly marked with a +. In a patch, there may be several "hunks" which are marked with a line starting with @@. Each hunk will be treated separately during patch application.

Broadly speaking, patches come in two varieties; plain text such as the one above which are sent to the mailing list and compressed patches that are compressed with either **gzip** (.gz extension) or **bzip2** (.bz2 extension). It is usually safe to assume that patches were generated one directory above the root of the kernel source tree. This means that while the patch is generated one directory above, it may be applied with the option -p1 while the current directory is the kernel source tree root.

\(^6\) [http://www.gnu.org](http://www.gnu.org)
1.2.1 Diff and Patch

--- linux-2.4.22-clean/mm/page_alloc.c Thu Sep 4 03:53:15 2003
+++ linux-2.4.22-mel/mm/page_alloc.c Thu Sep 3 03:54:07 2003
@@ -76,8 +76,23 @@
    * triggers coalescing into a block of larger size.
    *
    * @page: The first page of the block to be freed
+ * @order: 2^order number of pages are freed
+ *
+ * This function returns the pages allocated by __alloc_pages and tries to
+ * merge buddies if possible. Do not call directly, use free_pages()
+ */
+/**
+ *
+ * __free_pages_ok - Returns pages to the buddy allocator
+ * @page: The first page of the block to be freed
+ * @order: 2^order number of pages are freed
+ *
+ Figure 1.1: Example Patch

Broadly speaking, this means a plain text patch to a clean tree can be easily
applied as follows:

mel@joshua: kernels/ $ cd linux-2.4.22-clean/
mel@joshua: linux-2.4.22-clean/ $ patch -p1 < ../example.patch
patching file mm/page_alloc.c
mel@joshua: linux-2.4.22-clean/ $

To apply a compressed patch, it is a simple extension to just decompress the
patch to standard out (stdout) first.

mel@joshua: linux-2.4.22-mel/ $ gzip -dc ../example.patch.gz | patch -p1

If a hunk can be applied but the line numbers are different, the hunk number
and the number of lines needed to offset will be output. These are generally safe
warnings and may be ignored. If there are slight differences in the context, it will be
applied and the level of “fuzziness” will be printed which should be double checked. If a hunk fails to apply, it will be saved to filename.c.rej and the original file will be saved to filename.c.orig and have to be applied manually.

1.2.2 Basic Source Management with PatchSet

The untarring of sources, management of patches and building of kernels is initially interesting but quickly palls. To cut down on the tedium of patch management, a simple tool was developed while writing this book called PatchSet which is designed the easily manage the kernel source and patches eliminating a large amount of the tedium. It is fully documented and freely available from http://www.csn.ul.ie/~mel/projects/patchset/ and on the companion CD.

**Downloading** Downloading kernels and patches in itself is quite tedious and scripts are provided to make the task simpler. First, the configuration file etc/patchset.conf should be edited and the KERNEL_MIRROR parameter updated for your local http://www.kernel.org/ mirror. Once that is done, use the script download to download patches and kernel sources. A simple use of the script is as follows

```
mel@joshua: patchset/ $ download 2.4.18
  # Will download the 2.4.18 kernel source

mel@joshua: patchset/ $ download -p 2.4.19
  # Will download a patch for 2.4.19

mel@joshua: patchset/ $ download -p -b 2.4.20
  # Will download a bzip2 patch for 2.4.20
```

Once the relevant sources or patches have been downloaded, it is time to configure a kernel build.

**Configuring Builds** Files called set configuration files are used to specify what kernel source tar to use, what patches to apply, what kernel configuration (generated by make config) to use and what the resulting kernel is to be called. A sample specification file to build kernel 2.4.20-rmap15f is:

```
linux-2.4.18.tar.gz
2.4.20-rmap15f
config_generic

1 patch-2.4.19.gz
1 patch-2.4.20.bz2
1 2.4.20-rmap15f
```
1.2.2 Basic Source Management with PatchSet

This first line says to unpack a source tree starting with linux-2.4.18.tar.gz. The second line specifies that the kernel will be called 2.4.20-rmap15f. 2.4.20 was selected for this example as rmap patches against a later stable release were not available at the time of writing. To check for updated rmap patches, see http://surriel.com/patches/. The third line specifies which kernel .config file to use for compiling the kernel. Each line after that has two parts. The first part says what patch depth to use i.e. what number to use with the -p switch to patch. As discussed earlier in Section 1.2.1, this is usually 1 for applying patches while in the source directory. The second is the name of the patch stored in the patches directory. The above example will apply two patches to update the kernel from 2.4.18 to 2.4.20 before building the 2.4.20-rmap15f kernel tree.

If the kernel configuration file required is very simple, then use the createset script to generate a set file for you. It simply takes a kernel version as a parameter and guesses how to build it based on available sources and patches.

mel@joshua: patchset/ $ createset 2.4.20

Building a Kernel The package comes with three scripts. The first script, called make-kernel.sh, will unpack the kernel to the kernels/ directory and build it if requested. If the target distribution is Debian, it can also create Debian packages for easy installation by specifying the -d switch. The second, called make-gengraph.sh, will unpack the kernel but instead of building an installable kernel, it will generate the files required to use CodeViz, discussed in the next section, for creating call graphs. The last, called make-lxr.sh, will install a kernel for use with LXR.

Generating Diffs Ultimately, you will need to see the difference between files in two trees or generate a “diff” of changes you have made yourself. Three small scripts are provided to make this task easier. The first is setclean which sets the source tree to compare from. The second is setworking to set the path of the kernel tree you are comparing against or working on. The third is difftree which will generate diffs against files or directories in the two trees. To generate the diff shown in Figure 1.2.1, the following would have worked;

mel@joshua: patchset/ $ setclean linux-2.4.22-clean
mel@joshua: patchset/ $ setworking linux-2.4.22-mel
mel@joshua: patchset/ $ difftree mm/page_alloc.c

The generated diff is a unified diff with the C function context included and complies with the recommended usage of diff. Two additional scripts are available which are very useful when tracking changes between two trees. They are diffstruct and difffunc. These are for printing out the differences between individual structures and functions. When used first, the -f switch must be used to record what source file the structure or function is declared in but it is only needed the first time.
1.3 Browsing the Code

When code is small and manageable, it is not particularly difficult to browse through
the code as operations are clustered together in the same file and there is not much
coupling between modules. The kernel unfortunately does not always exhibit this
behaviour. Functions of interest may be spread across multiple files or contained as
inline functions in headers. To complicate matters, files of interest may be buried
beneath architecture specific directories making tracking them down time consum-
ing.

One solution for easy code browsing is ctags(\url{http://ctags.sourceforge.net/})
which generates tag files from a set of source files. These tags can be used to
jump to the C file and line where the identifier is declared with editors such as
Vi and Emacs. In the event there is multiple instances of the same tag, such as
with multiple functions with the same name, the correct one may be selected from
a list. This method works best when one is editing the code as it allows very fast
navigation through the code to be confined to one terminal window.

A more friendly browsing method is available with the Linux Cross-Referencing
(LXR) tool hosted at \url{http://lxr.linux.no/}. This tool provides the ability to represent
source code as browsable web pages. Identifiers such as global variables, macros
and functions become hyperlinks. When clicked, the location where it is defined is
displayed along with every file and line referencing the definition. This makes code
navigation very convenient and is almost essential when reading the code for the
first time.

The tool is very simple to install and and browsable version of the kernel 2.4.22
source is available on the CD included with this book. All code extracts throughout
the book are based on the output of LXR so that the line numbers would be clearly
visible in excerpts.

1.3.1 Analysing Code Flow

As separate modules share code across multiple C files, it can be difficult to see
what functions are affected by a given code path without tracing through all the
code manually. For a large or deep code path, this can be extremely time consuming
to answer what should be a simple question.

One simple, but effective tool to use is CodeViz which is a call graph gen-
erator and is included with the CD. It uses a modified compiler for either C or
C++ to collect information necessary to generate the graph. The tool is hosted at
\url{http://www.csn.ul.ie/~mel/projects/codeviz/}.

During compilation with the modified compiler, files with a .cdep extension are
generated for each C file. This .cdep file contains all function declarations and
calls made in the C file. These files are distilled with a program called genfull
to generate a full call graph of the entire source code which can be rendered with \texttt{dot},
part of the GraphViz project hosted at \url{http://www.graphviz.org/}.

In the kernel compiled for the computer this book was written on, there were a
total of 40,165 entries in the full.graph file generated by \texttt{genfull}. This call graph
1.3.2 Simple Graph Generation

is essentially useless on its own because of its size so a second tool is provided called gengraph. This program, at basic usage, takes the name of one or more functions as an argument and generates postscript file with the call graph of the requested function as the root node. The postscript file may be viewed with ghostview or gv.

The generated graphs can be to unnecessary depth or show functions that the user is not interested in, therefore there are three limiting options to graph generation. The first is limit by depth where functions that are greater than \( N \) levels deep in a call chain are ignored. The second is to totally ignore a function so it will not appear on the call graph or any of the functions they call. The last is to display a function, but not traverse it which is convenient when the function is covered on a separate call graph or is a known API whose implementation is not currently of interest.

All call graphs shown in these documents are generated with the CodeViz tool as it is often much easier to understand a subsystem at first glance when a call graph is available. It has been tested with a number of other open source projects based on C and has wider application than just the kernel.

1.3.2 Simple Graph Generation

If both PatchSet and CodeViz are installed, the first call graph in this book shown in Figure 3.4 can be generated and viewed with the following set of commands. For brevity, the output of the commands is omitted:

```
mel@joshua: patchset $ download 2.4.22
mel@joshua: patchset $ createset 2.4.22
mel@joshua: patchset $ make-gengraph.sh 2.4.22
mel@joshua: patchset $ cd kernels/linux-2.4.22
mel@joshua: linux-2.4.22 $ gengraph -t -s "alloc_bootmem_low_pages \zone_sizes_init" -f paging_init
mel@joshua: linux-2.4.22 $ gv paging_init.ps
```

1.4 Reading the Code

When a new developer or researcher asks how to start reading the code, they are often recommended to start with the initialisation code and work from there. This may not be the best approach for everyone as initialisation is quite architecture dependent and requires detailed hardware knowledge to decipher it. It also gives very little information on how a subsystem like the VM works as it is during the late stages of initialisation that memory is set up in the way the running system sees it.

The best starting point to understanding the VM is this book and the code commentary. It describes a VM that is reasonably comprehensive without being overly complicated. Later VMs are more complex but are essentially extensions of the one described here.
For when the code has to be approached afresh with a later VM, it is always best to start in an isolated region that has the minimum number of dependencies. In the case of the VM, the best starting point is the Out Of Memory (OOM) manager in mm/oom_kill.c. It is a very gentle introduction to one corner of the VM where a process is selected to be killed in the event that memory in the system is low. It is because it touches so many different aspects of the VM that is covered last in this book! The second subsystem to then examine is the non-contiguous memory allocator located in mm/vmalloc.c and discussed in Chapter 7 as it is reasonably contained within one file. The third system should be physical page allocator located in mm/page_alloc.c and discussed in Chapter 6 for similar reasons. The fourth system of interest is the creation of VMAs and memory areas for processes discussed in Chapter 4. Between these systems, they have the bulk of the code patterns that are prevalent throughout the rest of the kernel code making the deciphering of more complex systems such as the page replacement policy or the buffer IO much easier to comprehend.

The second recommendation that is given by experienced developers is to benchmark and test the VM. There are many benchmark programs available but commonly used ones are ConTest (http://members.optusnet.com.au/ekolivas/contest/), SPEC (http://www.specbench.org/), lmbench (http://www.bitmover.com/lmbench/) and dbench (http://freshmeat.net/projects/dbench/). For many purposes, these benchmarks will fit the requirements.

Unfortunately it is difficult to test just the VM accurately and benchmarking it is frequently based on timing a task such as a kernel compile. A tool called VM Regress is available at http://www.csn.ul.ie/~mel/vmregress/ that lays the foundation required to build a fully fledged testing, regression and benchmarking tool for the VM. It uses a combination of kernel modules and userspace tools to test small parts of the VM in a reproducible manner and has one benchmark for testing the page replacement policy using a large reference string. It is intended as a framework for the development of a testing utility and has a number of Perl libraries and helper kernel modules to do much of the work but is still in the early stages of development so use with care.

1.5 Submitting Patches

There are two files, SubmittingPatches and CodingStyle, in the Documentation/ directory which cover the important basics. However, there is very little documentation describing how to get patches merged. This section will give a brief introduction on how, broadly speaking, patches are managed.

First and foremost, the coding style of the kernel needs to be adhered to as having a style inconsistent with the main kernel will be a barrier to getting merged regardless of the technical merit. Once a patch has been developed, the first problem is to decide where to send it. Kernel development has a definite, if non-apparent, hierarchy of who handles patches and how to get them submitted. As an example, we'll take the case of 2.5.x development.
1.5 Submitting Patches

The first check to make is if the patch is very small or trivial. If it is, post it to the main kernel mailing list. If there is no bad reaction, it can be fed to what is called the *Trivial Patch Monkey*\(^7\). The trivial patch monkey is exactly what it sounds like; it takes small patches and feeds them en-masse to the correct people. This is best suited for documentation, commentary or one-liner patches.

Patches are managed through what could be loosely called a set of rings with Linus in the very middle having the final say on what gets accepted into the main tree. Linus, with rare exceptions, accepts patches only from who he refers to as his "lieutenants", a group of around 10 people who he trusts to "feed" him correct code. An example lieutenant is Andrew Morton, the VM maintainer at time of writing.

An example lieutenant is Andrew Morton, the VM maintainer at time of writing. Any change to the VM has to be accepted by Andrew before it will get to Linus. These people are generally maintainers of a particular system but sometimes will "feed" him patches from another subsystem if they feel it is important enough.

Each of the lieutenants are active developers on different subsystems. Just like Linus, they have a small set of developers they trust to be knowledgeable about the patch they are sending but will also pick up patches which affect their subsystem more readily. Depending on the subsystem, the list of people they trust will be heavily influenced by the list of maintainers in the MAINTAINERS file. The second major area of influence will be from the subsystem specific mailing list if there is one. The VM does not have a list of maintainers but it does have a mailing list\(^8\).

The maintainers and lieutenants are crucial to the acceptance of patches. Linus, broadly speaking, does not appear to wish to be convinced with argument alone on the merit for a significant patch but prefers to hear it from one of his lieutenants, which is understandable considering the volume of patches that exists.

In summary, a new patch should be emailed to the subsystem mailing list cc'd to the main list to generate discussion. If there is no reaction, it should be sent to the maintainer for that area of code if there is one and to the lieutenant if there is not. Once it has been picked up by a maintainer or lieutenant, chances are it will be merged. The important key is that patches and ideas must be released early and often so developers have a chance to look at it while it is still manageable. There are notable cases where massive patches merging with the main tree because there were long periods of silence with little or no discussion. A recent example of this is the Linux Kernel Crash Dump project which still has not been merged into the main stream because there has not enough favorable feedback from lieutenants or strong support from vendors.


\(^8\) [http://www.linux-mm.org/mailinglists.shtml](http://www.linux-mm.org/mailinglists.shtml)
Chapter 2

Describing Physical Memory

Linux is available for a wide range of architectures so there needs to be an architecture-independent way of describing memory. This chapter describes the structures used to keep account of memory banks, pages and the flags that affect VM behaviour.

The first principal concept prevalent in the VM is *Non-Uniform Memory Access (NUMA)*. With large scale machines, memory may be arranged into banks that incur a different cost to access depending on the "distance" from the processor. For example, there might be a bank of memory assigned to each CPU or a bank of memory very suitable for DMA near device cards.

Each bank is called a *node* and the concept is represented under Linux by a *struct pglist_data* even if the architecture is UMA. This struct is always referenced to by its typedef *pg_data_t*. Every node in the system is kept on a NULL terminated list called *pgdat_list* and each node is linked to the next with the field *pg_data_t*→*node_next*. For UMA architectures like PC desktops, only one static *pg_data_t* structure called *contig_page_data* is used. Nodes will be discussed further in Section 2.1.

Each node is divided up into a number of blocks called *zones* which represent ranges within memory. Zones should not be confused with zone based allocators as they are unrelated. A zone is described by a *struct zone_struct*, typedefed to *zone_t* and each one is of type *ZONE_DMA*, *ZONE_NORMAL* or *ZONE_HIGHMEM*. Each zone type suitable a different type of usage. *ZONE_DMA* is memory in the lower physical memory ranges which certain ISA devices require. Memory within *ZONE_NORMAL* is directly mapped by the kernel into the upper region of the linear address space which is discussed further in Section 4.1. *ZONE_HIGHMEM* is the remaining available memory in the system and is not directly mapped by the kernel.

With the x86 the zones are:

- **ZONE_DMA**: First 16 MiB of memory
- **ZONE_NORMAL**: 16 MiB - 896 MiB
- **ZONE_HIGHMEM**: 896 MiB - End

It is important to note that many kernel operations can only take place using *ZONE_NORMAL* so it is the most performance critical zone. Zones are discussed further in Section 2.2. Each physical page frame is represented by a *struct page* and all the
2.1 No des

As we have mentioned, each node in memory is described by a `pg_data_t` which is a typedef for a `struct pglist_data`. When allocating a page, Linux uses a node-local allocation policy to allocate memory from the node closest to the running CPU. As processes tend to run on the same CPU, it is likely the memory from the current node will be used. The struct is declared as follows in `<linux/mmzone.h>`:

Figure 2.1: Relationship Between Nodes, Zones and Pages

As the amount of memory directly accessible by the kernel (ZONE_NORMAL) is limited in size, Linux supports the concept of High Memory which is discussed further in Section 2.5. This chapter will discuss how nodes, zones and pages are represented before introducing high memory management.

2.1 Nodes

structs are kept in a global `mem_map` array which is usually stored at the beginning of ZONE_NORMAL or just after the area reserved for the loaded kernel image in low memory machines. `struct pages` are discussed in detail in Section 2.4 and the global `mem_map` array is discussed in detail in Section 3.7. The basic relationship between all these structs is illustrated in Figure 2.1.
We now briefly describe each of these fields:

**node_zones** The zones for this node, ZONE_HIGHMEM, ZONE_NORMAL, ZONE_DMA;

**node_zonelists** This is the order of zones that allocations are preferred from. build_zonelists() in mm/page_alloc.c sets up the order when called by free_area_init_core(). A failed allocation in ZONE_HIGHMEM may fall back to ZONE_NORMAL or back to ZONE_DMA;

**nr_zones** Number of zones in this node, between 1 and 3. Not all nodes will have three. A CPU bank may not have ZONE_DMA for example;

**node_mem_map** This is the first page of the struct page array representing each physical frame in the node. It will be placed somewhere within the global mem_map array;

**valid_addr_bitmap** A bitmap which describes “holes” in the memory node that no memory exists for. In reality, this is only used by the Sparc and Sparc64 architectures and ignored by all others;

**bdata** This is only of interest to the boot memory allocator discussed in Chapter 5;

**node_start_paddr** The starting physical address of the node. An unsigned long does not work optimally as it breaks for ia32 with Physical Address Extension (PAE) for example. PAE is discussed further in Section 2.5. A more suitable solution would be to record this as a Page Frame Number (PFN). A PFN is simply in index within physical memory that is counted in page-sized units. PFN for a physical address could be trivially defined as (page Phys_addr >> PAGE_SHIFT);

**node_start_mapnr** This gives the page offset within the global mem_map. It is calculated in free_area_init_core() by calculating the number of pages between mem_map and the local mem_map for this node called lmem_map;
2.2 Zones

The total number of pages in this zone;

The Node ID (NID) of the node, starts at 0;

Pointer to next node in a NULL terminated list.

All nodes in the system are maintained on a list called pgdat_list. The nodes are placed on this list as they are initialised by the init_bootmem_core() function, described later in Section 5.2.1. Up until late 2.4 kernels (> 2.4.18), blocks of code that traversed the list looked something like:

```c
pg_data_t * pgdat;
pgdat = pgdat_list;
do {
    /* do something with pgdata_t */
    ...
} while ((pgdat = pgdat->node_next));
```

In more recent kernels, a macro for_each_pgdat(), which is trivially defined as a for loop, is provided to improve code readability.

2.2 Zones

Zones are described by a struct zone_struct and is usually referred to by its typedef zone_t. It keeps track of information like page usage statistics, free area information and locks. It is declared as follows in <linux/mmzone.h>:

```c
typedef struct zone_struct {
    spinlock_t lock;
    unsigned long free_pages;
    unsigned long pages_min, pages_low, pages_high;
    int need_balance;
    free_area_t free_area[MAX_ORDER];
    wait_queue_head_t * wait_table;
    unsigned long wait_table_size;
    unsigned long wait_table_shift;
    struct pglist_data *zone_pgdat;
    struct page *zone_mem_map;
    unsigned long zone_start_paddr;
    unsigned long zone_start_mapnr;
    char *name;
    unsigned long size;
} zone_t;
```
This is a brief explanation of each field in the struct.

**lock**  Spinlock to protect the zone from concurrent accesses;

**free_pages**  Total number of free pages in the zone;

**pages_min, pages_low, pages_high**  These are zone watermarks which are described in the next section;

**need_balance**  This flag that tells the pageout kswapd to balance the zone. A zone is said to need balance when the number of available pages reaches one of the zone watermarks. Watermarks is discussed in the next section;

**free_area**  Free area bitmaps used by the buddy allocator;

**wait_table**  A hash table of wait queues of processes waiting on a page to be freed. This is of importance to wait_on_page() and unlock_page(). While processes could all wait on one queue, this would cause all waiting processes to race for pages still locked when woken up. A large group of processes contending for a shared resource like this is sometimes called a thundering herd. Wait tables are discussed further in Section 2.2.3;

**wait_table_size**  Number of queues in the hash table which is a power of 2;

**wait_table_shift**  Defined as the number of bits in a long minus the binary logarithm of the table size above;

**zone_pgdat**  Points to the parent pg_data_t;

**zone_mem_map**  The first page in the global mem_map this zone refers to;

**zone_start_paddr**  Same principle as node_start_paddr;

**zone_start_mapnr**  Same principle as node_start_mapnr;

**name**  The string name of the zone, “DMA”, “Normal” or “HighMem”

**size**  The size of the zone in pages.

### 2.2.1 Zone Watermarks

When available memory in the system is low, the pageout daemon kswapd is woken up to start freeing pages (see Chapter 10). If the pressure is high, the process will free up memory synchronously, sometimes referred to as the direct-reclaim path. The parameters affecting pageout behaviour are similar to those by FreeBSD [McK96] and Solaris [MM01].

Each zone has three watermarks called pages_low, pages_min and pages_high which help track how much pressure a zone is under. The relationship between them is illustrated in Figure 2.2. The number of pages for pages_min is calculated in the
2.2.1 Zone Watermarks

function `free_area_init_core()` during memory init and is based on a ratio to the size of the zone in pages. It is calculated initially as `ZoneSizeInPages/128`. The lowest value it will be is 20 pages (80K on an x86) and the highest possible value is 255 pages (1MiB on an x86).

**Figure 2.2: Zone Watermarks**

- **pages_low** When `pages_low` number of free pages is reached, `kswapd` is woken up by the buddy allocator to start freeing pages. This is equivalent to when `lotsfree` is reached in Solaris and `freemin` in FreeBSD. The value is twice the value of `pages_min` by default;

- **pages_min** When `pages_min` is reached, the allocator will do the `kswapd` work in a synchronous fashion, sometimes referred to as the `direct-reclaim` path. There is no real equivalent in Solaris but the closest is the `desfree` or `minfree` which determine how often the pageout scanner is woken up;

- **pages_high** Once `kswapd` has been woken to start freeing pages it will not consider the zone to be “balanced” when `pages_high` pages are free. Once
2.2.2 Calculating The Size of Zones

the watermark has been reached, \texttt{kswapd} will go back to sleep. In Solaris, this is called \texttt{lotsfree} and in BSD, it is called \texttt{free_target}. The default for \texttt{pages\_high} is three times the value of \texttt{pages\_min}.

Whatever the pageout parameters are called in each operating system, the meaning is the same, it helps determine how hard the pageout daemon or processes work to free up pages.

\textbf{2.2.2 Calculating The Size of Zones}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{setup_memory.png}
\caption{Call Graph: \texttt{setup\_memory()}}
\end{figure}

The PFN is an offset, counted in pages, within the physical memory map. The first PFN usable by the system, \texttt{min\_low\_pfn} is located at the beginning of the first page after _end which is the end of the loaded kernel image. The value is stored as a file scope variable in \texttt{mm/bootmem.c} for use with the boot memory allocator.

How the last page frame in the system, \texttt{max\_pfn}, is calculated is quite architecture specific. In the x86 case, the function \texttt{find\_max\_pfn()} reads through the whole e820 map for the highest page frame. The value is also stored as a file scope variable in \texttt{mm/bootmem.c}. The e820 is a table provided by the BIOS describing what physical memory is available, reserved or non-existent.

The value of \texttt{max\_low\_pfn} is calculated on the x86 with \texttt{find\_max\_low\_pfn()} and it marks the end of ZONE_NORMAL. This is the physical memory directly accessible by the kernel and is related to the kernel/userspace split in the linear address space marked by \texttt{PAGE\_OFFSET}. The value, with the others, is stored in \texttt{mm/bootmem.c}. Note that in low memory machines, the \texttt{max\_pfn} will be the same as the \texttt{max\_low\_pfn}. 
With the three variables `min_low_pfn`, `max_low_pfn` and `max_pfn`, it is straightforward to calculate the start and end of high memory and place them as file scope variables in `arch/i386/mm/init.c` as `highstart_pfn` and `highend_pfn`. The values are used later to initialise the high memory pages for the physical page allocator as we will much later in Section 5.5.

### 2.2.3 Zone Wait Queue Table

When IO is being performed on a page, such as during page-in or page-out, it is locked to prevent accessing it with inconsistent data. Processes wishing to use it have to join a wait queue before it can be accessed by calling `wait_on_page()`. When the IO is completed, the page will be unlocked with `UnlockPage()` and any process waiting on the queue will be woken up. Each page could have a wait queue but it would be very expensive in terms of memory to have so many separate queues so instead, the wait queue is stored in the `zone_t`.

It is possible to have just one wait queue in the zone but that would mean that all processes waiting on any page in a zone would be woken up when one was unlocked. This would cause a serious *thundering herd* problem. Instead, a hash table of wait queues is stored in `zone_t->wait_table`. In the event of a hash collision, processes may still be woken unnecessarily but collisions are not expected to occur frequently.

---

**Figure 2.4: Sleeping On a Locked Page**

The table is allocated during `free_area_init_core()`. The size of the table is calculated by `wait_table_size()` and stored in the `zone_t->wait_table_size`. The maximum size it will be is 4096 wait queues. For smaller tables, the size of the table is the minimum power of 2 required to store `NoPages / PAGES_PER_WAITQUEUE` number of queues, where `NoPages` is the number of pages in the zone and `PAGE_PER_WAITQUEUE` is defined to be 256. In other words, the size of the table is calculated as the integer component of the following equation:
2.3 Zone Initialisation

\[
\text{wait}_\text{table}\_\text{size} = \log_2\left(\frac{\text{NoPages} \times 2}{\text{PAGE\_PER\_WAITQUEUE}} - 1\right)
\]

The field \text{zone\_t}→\text{wait\_table\_shift} is calculated as the number of bits a page address must be shifted right to return an index within the table. The function \text{page\_waitqueue()} is responsible for returning which wait queue to use for a page in a zone. It uses a simple multiplicative hashing algorithm based on the virtual address of the \text{struct page} being hashed.

It works by simply multiplying the address by \text{GOLDEN\_RATIO\_PRIME} and shifting the result \text{zone\_t}→\text{wait\_table\_shift} bits right to index the result within the hash table. \text{GOLDEN\_RATIO\_PRIME}[\text{Lev00}] is the largest prime that is closest to the golden ratio[\text{Knuth68}] of the largest integer that may be represented by the architecture.

2.3 Zone Initialisation

The zones are initialised after the kernel page tables have been fully setup by \text{paging\_init()}. Page table initialisation is covered in Section 3.6. Predictably, each architecture performs this task differently but the objective is always the same, to determine what parameters to send to either \text{free\_area\_init()} for UMA architectures or \text{free\_area\_init\_node()} for NUMA. The only parameter required for UMA is \text{zones\_size}. The full list of parameters:

- \text{nid} is the Node ID which is the logical identifier of the node whose zones are being initialised;
- \text{pgdat} is the node’s \text{pg\_data\_t} that is being initialised. In UMA, this will simply be \text{contig\_page\_data};
- \text{pmap} is set later by \text{free\_area\_init\_core()} to point to the beginning of the local \text{lmem\_map} array allocated for the node. In NUMA, this is ignored as NUMA treats \text{mem\_map} as a virtual array starting at \text{PAGE\_OFFSET}. In UMA, this pointer is the global \text{mem\_map} variable which is now \text{mem\_map} gets initialised in UMA.
- \text{zones\_sizes} is an array containing the size of each zone in pages;
- \text{zone\_start\_paddr} is the starting physical address for the first zone;
- \text{zone\_holes} is an array containing the total size of memory holes in the zones;

It is the core function \text{free\_area\_init\_core()} which is responsible for filling in each \text{zone\_t} with the relevant information and the allocation of the \text{mem\_map} array for the node. Note that information on what pages are free for the zones is not determined at this point. That information is not known until the boot memory allocator is being retired which will be discussed much later in Chapter 5.
2.3.1 Initialising mem_map

The mem_map area is created during system startup in one of two fashions. On NUMA systems, the global mem_map is treated as a virtual array starting at PAGE_OFFSET. free_area_init_node() is called for each active node in the system which allocates the portion of this array for the node being initialised. On UMA systems, free_area_init() is uses contig_page_data as the node and the global mem_map as the "local" mem_map for this node. The callgraph for both functions is shown in Figure 2.5.

![Figure 2.5: Call Graph: free_area_init()](image)

The core function free_area_init_core() allocates a local lmem_map for the node being initialised. The memory for the array is allocated from the boot memory allocator with alloc_bootmem_node() (see Chapter 5). With UMA architectures, this newly allocated memory becomes the global mem_map but it is slightly different for NUMA.

NUMA architectures allocate the memory for lmem_map within their own memory node. The global mem_map never gets explicitly allocated but instead is set to PAGE_OFFSET where it is treated as a virtual array. The address of the local map is stored in pg_data_t→node_mem_map which exists somewhere within the virtual mem_map. For each zone that exists in the node, the address within the virtual mem_map for the zone is stored in zone_t→zone_mem_map. All the rest of the code then treats mem_map as a real array as only valid regions within it will be used by nodes.

2.4 Pages

Every physical page frame in the system has an associated struct page which is used to keep track of its status. In the 2.2 kernel [BC00], this structure resembled
it's equivalent in System V [GC94] but like the other UNIX variants, the structure changed considerably. It is declared as follows in `<linux/mm.h>`:

```c
typedef struct page {
    struct list_head list;
    struct address_space *mapping;
    unsigned long index;
    struct page *next_hash;
    atomic_t count;
    unsigned long flags;
    struct list_head lru;
    struct page **pprev_hash;
    struct buffer_head *buffers;

    #if defined(CONFIG_HIGHMEM) || defined(WANT_PAGE_VIRTUAL)
    void *virtual;
    #endif /* CONFIG_HIGMEM || WANT_PAGE_VIRTUAL */
} mem_map_t;
```

Here is a brief description of each of the fields:

- **list**  Pages may belong to many lists and this field is used as the list head. For example, pages in a mapping will be in one of three circular linked links kept by the `address_space`. These are `clean_pages`, `dirty_pages` and `locked_pages`. In the slab allocator, this field is used to store pointers to the slab and cache the page belongs to. It is also used to link blocks of free pages together;

- **mapping**  When files or devices are memory mapped, their inode has an associated `address_space`. This field will point to this address space if the page belongs to the file. If the page is anonymous and `mapping` is set, the `address_space` is `swapper_space` which manages the swap address space;

- **index**  This field has two uses and it depends on the state of the page what it means. If the page is part of a file mapping, it is the offset within the file. If the page is part of the swap cache this will be the offset within the `address_space` for the swap address space (`swapper_space`). Secondly, if a block of pages is being freed for a particular process, the order (power of two number of pages being freed) of the block being freed is stored in `index`. This is set in the function `_free_pages_ok();`

- **next_hash**  Pages that are part of a file mapping are hashed on the inode and offset. This field links pages together that share the same hash bucket;

- **count**  The reference count to the page. If it drops to 0, it may be freed. Any greater and it is in use by one or more processes or is in use by the kernel like when waiting for IO;
flags These are flags which describe the status of the page. All of them are declared in `<linux/mm.h>` and are listed in Table 2.1. There are a number of macros defined for testing, clearing and setting the bits which are all listed in Table 2.2. The only really interesting one is `SetPageUptodate()` which calls an architecture specific function `arch_set_page_uptodate()` if it is defined before setting the bit;

lru For the page replacement policy, pages that may be swapped out will exist on either the `active_list` or the `inactive_list` declared in `page_alloc.c`. This is the list head for these LRU lists. These two lists are discussed in detail in Chapter 10;

pprev_hash This complement to `next_hash` so that the hash can work as a doubly linked list;

buffers If a page has buffers for a block device associated with it, this field is used to keep track of the `buffer_head`. An anonymous page mapped by a process may also have an associated `buffer_head` if it is backed by a swap file. This is necessary as the page has to be synced with backing storage in block sized chunks defined by the underlying filesystem;

virtual Normally only pages from `ZONE_NORMAL` are directly mapped by the kernel. To address pages in `ZONE_HIGHMEM`, `kmap()` is used to map the page for the kernel which is described further in Chapter 9. There are only a fixed number of pages that may be mapped. When it is mapped, this is its virtual address;

The type `mem_map_t` is a typedef for `struct page` so it can be easily referred to within the `mem_map` array.

2.4.1 Mapping Pages to Zones

Up until as recently as kernel 2.4.18, a `struct page` stored a reference to its zone with `page→zone` which was later considered wasteful, as even such a small pointer consumes a lot of memory when thousands of `struct page`s exist. In more recent kernels, the `zone` field has been removed and instead the top `ZONE_SHIFT` (8 in the x86) bits of the `page→flags` are used to determine the zone a page belongs to. First a `zone_table` of zones is set up. It is declared in `mm/page_alloc.c` as:

```
33 zone_t *zone_table[MAX_NR_ZONES*MAX_NR_NODES];
34 EXPORT_SYMBOL(zone_table);
```

`MAX_NR_ZONES` is the maximum number of zones that can be in a node, i.e. 3. `MAX_NR_NODES` is the maximum number of nodes that may exist. The function `EXPORT_SYMBOL()` makes `zone_table` accessible to loadable modules. This table is treated like a multi-dimensional array. During `free_area_init_core()`, all the pages in a node are initialised. First it sets the value for the table
2.5 High Memory

As the address space usable by the kernel (ZONE_NORMAL) is limited in size, the kernel has support for the concept of High Memory. Two thresholds of high memory exist on 32-bit x86 systems, one at 4GiB and a second at 64GiB. The 4GiB limit is related to the amount of memory that may be addressed by a 32-bit physical address. To access memory between the range of 1GiB and 4GiB, the kernel temporarily maps pages from high memory into ZONE_NORMAL with kmap(). This is discussed further in Chapter 9.

The second limit at 64GiB is related to Physical Address Extension (PAE) which is an Intel invention to allow more RAM to be used with 32 bit systems. It makes 4 extra bits available for the addressing of memory, allowing up to $2^{36}$ bytes (64GiB) of memory to be addressed.

PAE allows a processor to address up to 64GiB in theory but, in practice, processes in Linux still cannot access that much RAM as the virtual address space is still only 4GiB. This has led to some disappointment from users who have tried to malloc() all their RAM with one process.

Secondly, PAE does not allow the kernel itself to have this much RAM available. The struct page used to describe each page frame still requires 44 bytes and this uses kernel virtual address space in ZONE_NORMAL. That means that to describe 1GiB of memory, approximately 11MiB of kernel memory is required. Thus, with 16GiB, 176MiB of memory is consumed, putting significant pressure on ZONE_NORMAL. This does not sound too bad until other structures are taken into account which use ZONE_NORMAL. Even very small structures such as Page Table Entries (PTEs) require about 16MiB in the worst case. This makes 16GiB about the practical limit for available physical memory Linux on an x86. If more memory needs to be accessed, the advice given is simple and straightforward, buy a 64 bit machine.

2.6 What’s New In 2.6

Nodes At first glance, there has not been many changes made to how memory is described but the seemingly minor changes are wide reaching. The node descriptor pg_data_t has a few new fields which are as follows:
node_start_pfn replaces the node_start_paddr field. The only difference is that the new field is a PFN instead of a physical address. This was changed as PAE architectures can address more memory than 32 bits can address so nodes starting over 4GiB would be unreachable with the old field;

kswapd_wait is a new wait queue for kswapd. In 2.4, there was a global wait queue for the page swapper daemon. In 2.6, there is one kswapdN for each node where N is the node identifier and each kswapd has its own wait queue with this field.

The node_size field has been removed and replaced instead with two fields. The change was introduced to recognise the fact that nodes may have “holes” in them where there is no physical memory backing the address.

node_present_pages is the total number of physical pages that are present in the node.

node_spanned_pages is the total area that is addressed by the node, including any holes that may exist.

Zones Even at first glance, zones look very different. They are no longer called zone_t but instead referred to as simply struct zone. The second major difference is the LRU lists. As we’ll see in Chapter 10, kernel 2.4 has a global list of pages that determine the order pages are freed or paged out. These lists are now stored in the struct zone. The relevant fields are:

lru_lock is the spinlock for the LRU lists in this zone. In 2.4, this is a global lock called pagemap_lru_lock;

active_list is the active list for this zone. This list is the same as described in Chapter 10 except it is now per-zone instead of global;

inactive_list is the inactive list for this zone. In 2.4, it is global;

refill_counter is the number of pages to remove from the active_list in one pass. Only of interest during page replacement;

nr_active is the number of pages on the active_list;

nr_inactive is the number of pages on the inactive_list;

all_unreclaimable is set to 1 if the pageout daemon scans through all the pages in the zone twice and still fails to free enough pages;

pages_scanned is the number of pages scanned since the last bulk amount of pages has been reclaimed. In 2.6, lists of pages are freed at once rather than freeing pages individually which is what 2.4 does;
pressure measures the scanning intensity for this zone. It is a decaying average which affects how hard a page scanner will work to reclaim pages.

Three other fields are new but they are related to the dimensions of the zone. They are:

- **zone_start_pfn** is the starting PFN of the zone. It replaces the zone_start_paddr and zone_start_mapnr fields in 2.4;
- **spanned_pages** is the number of pages this zone spans, including holes in memory which exist with some architectures;
- **present_pages** is the number of real pages that exist in the zone. For many architectures, this will be the same value as spanned_pages.

The next addition is **struct per_cpu_pageset** which is used to maintain lists of pages for each CPU to reduce spinlock contention. The zone→pageset field is a NR_CPU sized array of struct per_cpu_pageset where NR_CPU is the compiled upper limit of number of CPUs in the system. The per-cpu struct is discussed further at the end of the section.

The last addition to **struct zone** is the inclusion of padding of zeros in the struct. Development of the 2.6 VM recognised that some spinlocks are very heavily contended and are frequently acquired. As it is known that some locks are almost always acquired in pairs, an effort should be made to ensure they use different cache lines which is a common cache programming trick [Sea00]. These padding in the struct zone are marked with the ZONE_PADDING() macro and are used to ensure the zone→lock, zone→lru_lock and zone→pageset fields use different cache lines.

**Pages** The first noticeable change is that the ordering of fields has been changed so that related items are likely to be in the same cache line. The fields are essentially the same except for two additions. The first is a new union used to create a PTE chain. PTE chains are are related to page table management so will be discussed at the end of Chapter 3. The second addition is of page→private field which contains private information specific to the mapping. For example, the field is used to store a pointer to a buffer_head if the page is a buffer page. This means that the page→buffers field has also been removed. The last important change is that page→virtual is no longer necessary for high memory support and will only exist if the architecture specifically requests it. How high memory pages are supported is discussed further in Chapter 9.

**Per-CPU Page Lists** In 2.4, only one subsystem actively tries to maintain per-cpu lists for any object and that is the Slab Allocator, discussed in Chapter 8. In 2.6, the concept is much more wide-spread and there is a formalised concept of hot and cold pages.
The `struct per_cpu_pageset`, declared in `<linux/mmzone.h>`, has one field which is an array with two elements of type `per_cpu_pages`. The zeroth element of this array is for hot pages and the first element is for cold pages where hot and cold determines how “active” the page is currently in the cache. When it is known for a fact that the pages are not to be referenced soon, such as with IO readahead, they will be allocated as cold pages.

The `struct per_cpu_pages` maintains a count of the number of pages currently in the list, a high and low watermark which determine when the set should be refilled or pages freed in bulk, a variable which determines how many pages should be allocated in one block and finally, the actual list head of pages.

To build upon the per-cpu page lists, there is also a per-cpu page accounting mechanism. There is a `struct page_state` that holds a number of accounting variables such as the `pgalloc` field which tracks the number of pages allocated to this CPU and `pswpin` which tracks the number of swap reads. The struct is heavily commented in `<linux/page-flags.h>`. A single function `mod_page_state()` is provided for updating fields in the `page_state` for the running CPU and three helper macros are provided called `inc_page_state()`, `dec_page_state()` and `sub_page_state()`.
2.6 What’s New In 2.6

<table>
<thead>
<tr>
<th>Bit name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PG_active</td>
<td>This bit is set if a page is on the active_list LRU and cleared when it is removed. It marks a page as being hot.</td>
</tr>
<tr>
<td>PG_arch_1</td>
<td>Quoting directly from the code: PG_arch_1 is an architecture specific page state bit. The generic code guarantees that this bit is cleared for a page when it first is entered into the page cache. This allows an architecture to defer the flushing of the D-Cache (See Section 3.9) until the page is mapped by a process.</td>
</tr>
<tr>
<td>PG_checked</td>
<td>Only used by the Ext2 filesystem</td>
</tr>
<tr>
<td>PG_dirty</td>
<td>This indicates if a page needs to be flushed to disk. When a page is written to that is backed by disk, it is not flushed immediately, this bit is needed to ensure a dirty page is not freed before it is written out.</td>
</tr>
<tr>
<td>PG_error</td>
<td>If an error occurs during disk I/O, this bit is set</td>
</tr>
<tr>
<td>PG_fs_1</td>
<td>Bit reserved for a filesystem to use for it’s own purposes. Currently, only NFS uses it to indicate if a page is in sync with the remote server or not</td>
</tr>
<tr>
<td>PG_highmem</td>
<td>Pages in high memory cannot be mapped permanently by the kernel. Pages that are in high memory are flagged with this bit during mem_initO.</td>
</tr>
<tr>
<td>PG_launder</td>
<td>This bit is important only to the page replacement policy. When the VM wants to swap out a page, it will set this bit and call the writepageO function. When scanning, if it encounters a page with this bit and PG_locked set, it will wait for the I/O to complete.</td>
</tr>
<tr>
<td>PG_locked</td>
<td>This bit is set when the page must be locked in memory for disk I/O. When I/O starts, this bit is set and released when it completes.</td>
</tr>
<tr>
<td>PG_lru</td>
<td>If a page is on either the active_list or the inactive_list, this bit will be set.</td>
</tr>
<tr>
<td>PG_referenced</td>
<td>If a page is mapped and it is referenced through the mapping, index hash table, this bit is set. It is used during page replacement for moving the page around the LRU lists.</td>
</tr>
<tr>
<td>PG_reserved</td>
<td>This is set for pages that can never be swapped out. It is set by the boot memory allocator (See Chapter 5) for pages allocated during system startup. Later it is used to flag empty pages or ones that do not even exist.</td>
</tr>
<tr>
<td>PG_slab</td>
<td>This will flag a page as being used by the slab allocator.</td>
</tr>
<tr>
<td>PG_skip</td>
<td>Used by some architectures to skip over parts of the address space with no backing physical memory.</td>
</tr>
<tr>
<td>PG_unused</td>
<td>This bit is literally unused</td>
</tr>
<tr>
<td>PG_uptodate</td>
<td>When a page is read from disk without error, this bit will be set.</td>
</tr>
</tbody>
</table>

Table 2.1: Flags Describing Page Status
Table 2.2: Macros For Testing, Setting and Clearing page—flags Status Bits

<table>
<thead>
<tr>
<th>Bit name</th>
<th>Set</th>
<th>Test</th>
<th>Clear</th>
</tr>
</thead>
<tbody>
<tr>
<td>PG_active</td>
<td>SetPageActive()</td>
<td>PageActive()</td>
<td>ClearPageActive()</td>
</tr>
<tr>
<td>PG_arch_1</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>PG_checked</td>
<td>SetPageChecked()</td>
<td>PageChecked()</td>
<td>ClearPageDirty()</td>
</tr>
<tr>
<td>PG_dirty</td>
<td>SetPageDirty()</td>
<td>PageDirty()</td>
<td>ClearPageError()</td>
</tr>
<tr>
<td>PG_error</td>
<td>SetPageError()</td>
<td>PageError()</td>
<td>n/a</td>
</tr>
<tr>
<td>PG_highmem</td>
<td>n/a</td>
<td>PageHighMem()</td>
<td>ClearPageLaunder()</td>
</tr>
<tr>
<td>PG_launder</td>
<td>SetPageLaunder()</td>
<td>PageLaunder()</td>
<td>UnlockPage()</td>
</tr>
<tr>
<td>PG_locked</td>
<td>LockPage()</td>
<td>PageLocked()</td>
<td>TestClearPageLRU()</td>
</tr>
<tr>
<td>PG_lru</td>
<td>TestSetPageLRU()</td>
<td>PageLRU()</td>
<td>ClearPageReferenced()</td>
</tr>
<tr>
<td>PG_referenced</td>
<td>SetPageReferenced()</td>
<td>PageReferenced()</td>
<td>ClearPageReserved()</td>
</tr>
<tr>
<td>PG_reserved</td>
<td>SetPageReserved()</td>
<td>PageReserved()</td>
<td>n/a</td>
</tr>
<tr>
<td>PG_skip</td>
<td>n/a</td>
<td>n/a</td>
<td>PageClearSlab()</td>
</tr>
<tr>
<td>PG_slab</td>
<td>PageSetSlab()</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>PG_unused</td>
<td>n/a</td>
<td>PageUptodate()</td>
<td>ClearPageUptodate()</td>
</tr>
<tr>
<td>PG_uptodate</td>
<td>SetPageUptodate()</td>
<td>PageUptodate()</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 3

Page Table Management

Linux layers the machine independent/dependent layer in an unusual manner in comparison to other operating systems [CP99]. Other operating systems have objects which manage the underlying physical pages such as the pmap object in BSD. Linux instead maintains the concept of a three-level page table in the architecture independent code even if the underlying architecture does not support it. While this is conceptually easy to understand, it also means that the distinction between different types of pages is very blurry and page types are identified by their flags or what lists they exist on rather than the objects they belong to.

Architectures that manage their Memory Management Unit (MMU) differently are expected to emulate the three-level page tables. For example, on the x86 without PAE enabled, only two page table levels are available. The Page Middle Directory (PMD) is defined to be of size 1 and “folds back” directly onto the Page Global Directory (PGD) which is optimised out at compile time. Unfortunately, for architectures that do not manage their cache or Translation Lookaside Buffer (TLB) automatically, hooks for machine dependent have to be explicitly left in the code for when the TLB and CPU caches need to be altered and flushed even if they are null operations on some architectures like the x86. These hooks are discussed further in Section 3.8.

This chapter will begin by describing how the page table is arranged and what types are used to describe the three separate levels of the page table followed by how a virtual address is broken up into its component parts for navigating the table. Once covered, it will be discussed how the lowest level entry, the Page Table Entry (PTE) and what bits are used by the hardware. After that, the macros used for navigating a page table, setting and checking attributes will be discussed before talking about how the page table is populated and how pages are allocated and freed for the use with page tables. The initialisation stage is then discussed which shows how the page tables are initialised during boot strapping. Finally, we will cover how the TLB and CPU caches are utilised.
3.1 Describing the Page Directory

Each process a pointer (mm_struct→pgd) to its own Page Global Directory (PGD) which is a physical page frame. This frame contains an array of type pgd_t which is an architecture specific type defined in <asm/page.h>. The page tables are loaded differently depending on the architecture. On the x86, the process page table is loaded by copying mm_struct→pgd into the cr3 register which has the side effect of flushing the TLB. In fact this is how the function __flush_tlb() is implemented in the architecture dependent code.

Each active entry in the PGD table points to a page frame containing an array of Page Middle Directory (PMD) entries of type pmd_t which in turn points to page frames containing Page Table Entries (PTE) of type pte_t, which finally points to page frames containing the actual user data. In the event the page has been swapped out to backing storage, the swap entry is stored in the PTE and used by do_swap_page() during page fault to find the swap entry containing the page data. The page table layout is illustrated in Figure 3.1.

![Figure 3.1: Page Table Layout](image)

Any given linear address may be broken up into parts to yield offsets within these three page table levels and an offset within the actual page. To help break up the linear address into its component parts, a number of macros are provided in triplets for each page table level, namely a SHIFT, a SIZE and a MASK macro. The SHIFT macros specifies the length in bits that are mapped by each level of the page.
3.1 Describing the Page Directory

tables as illustrated in Figure 3.2.

Figure 3.2: Linear Address Bit Size Macros

The \texttt{MASK} values can be ANDd with a linear address to mask out all the upper bits and is frequently used to determine if a linear address is aligned to a given level within the page table. The \texttt{SIZE} macros reveal how many bytes are addressed by each entry at each level. The relationship between the \texttt{SIZE} and \texttt{MASK} macros is illustrated in Figure 3.3.

Figure 3.3: Linear Address Size and Mask Macros

For the calculation of each of the triplets, only \texttt{SHIFT} is important as the other two are calculated based on it. For example, the three macros for page level on the x86 are:

\begin{verbatim}
5 #define PAGE_SHIFT 12
6 #define PAGE_SIZE (1UL << PAGE_SHIFT)
7 #define PAGE_MASK (~(PAGE_SIZE-1))
\end{verbatim}

\texttt{PAGE_SHIFT} is the length in bits of the offset part of the linear address space which is 12 bits on the x86. The size of a page is easily calculated as $2^{\text{PAGE\_SHIFT}}$ which is the equivalent of the code above. Finally the mask is calculated as the negation of the bits which make up the \texttt{PAGE\_SIZE - 1}. If a page needs to be aligned on a page boundary, \texttt{PAGE\_ALIGN()} is used. This macro adds \texttt{PAGE\_SIZE - 1} to
3.2 Describing a Page Table Entry

the address before simply ANDing it with the\texttt{PAGE\_MASK} to zero out the page offset bits.

\texttt{PMD\_SHIFT} is the number of bits in the linear address which are mapped by the second level part of the table. The \texttt{PMD\_SIZE} and \texttt{PMD\_MASK} are calculated in a similar way to the page level macros.

\texttt{PGDIR\_SHIFT} is the number of bits which are mapped by the top, or first level, of the page table. The \texttt{PGDIR\_SIZE} and \texttt{PGDIR\_MASK} are calculated in the same manner as above.

The last three macros of importance are the \texttt{PTRS\_PER\_x} which determine the number of entries in each level of the page table. \texttt{PTRS\_PER\_PGD} is the number of pointers in the PGD, 1024 on an x86 without PAE. \texttt{PTRS\_PER\_PMD} is for the PMD, 1 on the x86 without PAE and \texttt{PTRS\_PER\_PTE} is for the lowest level, 1024 on the x86.

3.2 Describing a Page Table Entry

As mentioned, each entry is described by the structs \texttt{pte\_t}, \texttt{pmd\_t} and \texttt{pgd\_t} for PTEs, PMDs and PGDs respectively. Even though these are often just unsigned integers, they are defined as structs for two reasons. The first is for type protection so that they will not be used inappropriately. The second is for features like PAE on the x86 where an additional 4 bits is used for addressing more than 4GiB of memory. To store the protection bits, \texttt{pgprot\_t} is defined which holds the relevant flags and is usually stored in the lower bits of a page table entry.

For type casting, 4 macros are provided in \texttt{asm/page.h}, which takes the above types and returns the relevant part of the structs. They are \texttt{pte\_val()}, \texttt{pmd\_val()}, \texttt{pgd\_val()} and \texttt{pgprot\_val()}. To reverse the type casting, 4 more macros are provided \texttt{__pte()}, \texttt{__pmd()}, \texttt{__pgd()} and \texttt{__pgprot()}

Where exactly the protection bits are stored is architecture dependent. For illustration purposes, we will examine the case of an x86 architecture without PAE enabled but the same principles apply across architectures. On an x86 with no PAE, the \texttt{pte\_t} is simply a 32 bit integer within a struct. Each \texttt{pte\_t} points to an address of a page frame and all the addresses pointed to are guaranteed to be page aligned. Therefore, there are \texttt{PAGE\_SHIFT} (12) bits in that 32 bit value that are free for status bits of the page table entry. A number of the protection and status bits are listed in Table 3.1 but what bits exist and what they mean varies between architectures.

These bits are self-explanatory except for the \texttt{_PAGE\_PROTNONE} which we will discuss further. On the x86 with Pentium III and higher, this bit is called the \textit{Page Attribute Table (PAT)} while earlier architectures such as the Pentium II had this bit reserved. The PAT bit is used to indicate the size of the page the PTE is referencing. In a PGD entry, this same bit is instead called the \textit{Page Size Exception (PSE)} bit so obviously these bits are meant to be used in conjunction.

As Linux does not use the PSE bit for user pages, the PAT bit is free in the PTE for other purposes. There is a requirement for having a page resident in memory but inaccessible to the userspace process such as when a region is protected.
3.3 Using Page Table Entries

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>_PAGE_PRESENT</td>
<td>Page is resident in memory and not swapped out</td>
</tr>
<tr>
<td>_PAGE_PROTNONE</td>
<td>Page is resident but not accessible</td>
</tr>
<tr>
<td>_PAGE_RW</td>
<td>Set if the page may be written to</td>
</tr>
<tr>
<td>_PAGE_USER</td>
<td>Set if the page is accessible from user space</td>
</tr>
<tr>
<td>_PAGE_DIRTY</td>
<td>Set if the page is written to</td>
</tr>
<tr>
<td>_PAGE_ACCESSED</td>
<td>Set if the page is accessed</td>
</tr>
</tbody>
</table>

Table 3.1: Page Table Entry Protection and Status Bits

with mprotect() with the PROT_NONE flag. When the region is to be protected, the _PAGE_PRESENT bit is cleared and the _PAGE_PROTNONE bit is set. The macro pte_present() checks if either of these bits are set and so the kernel itself knows the PTE is present, just inaccessible to userspace which is a subtle, but important point. As the hardware bit _PAGE_PRESENT is clear, a page fault will occur if the page is accessed so Linux can enforce the protection while still knowing the page is resident if it needs to swap it out or the process exits.

3.3 Using Page Table Entries

Macros are defined in <asm/pgtable.h> which are important for the navigation and examination of page table entries. To navigate the page directories, three macros are provided which break up a linear address space into its component parts. pgd_offset() takes an address and the mm_struct for the process and returns the PGD entry that covers the requested address. pmd_offset() takes a PGD entry and an address and returns the relevant PMD. pte_offset() takes a PMD and returns the relevant PTE. The remainder of the linear address provided is the offset within the page. The relationship between these fields is illustrated in Figure 3.1.

The second round of macros determine if the page table entries are present or may be used.

- pte_none(), pmd_none() and pgd_none() return 1 if the corresponding entry does not exist;
- pte_present(), pmd_present() and pgd_present() return 1 if the corresponding page table entries have the PRESENT bit set;
- pte_clear(), pmd_clear() and pgd_clear() will clear the corresponding page table entry;
- pmd_bad() and pgd_bad() are used to check entries when passed as input parameters to functions that may change the value of the entries. Whether it returns 1 varies between the few architectures that define these macros but for those that actually define it, making sure the page entry is marked as present and accessed are the two most important checks.
There are many parts of the VM which are littered with page table walk code and it is important to recognise it. A very simple example of a page table walk is the function `follow_page()` in `mm/memory.c`. The following is an excerpt from that function, the parts unrelated to the page table walk are omitted:

```c
407  pgd_t *pgd;
408  pmd_t *pmd;
409  pte_t *ptep, pte;
410
411  pgd = pgd_offset(mm, address);
412  if (pgd_none(*pgd) || pgd_bad(*pgd))
413      goto out;
414
415  pmd = pmd_offset(pgd, address);
416  if (pmd_none(*pmd) || pmd_bad(*pmd))
417      goto out;
418
419  ptep = pte_offset(pmd, address);
420  if (!ptep)
421      goto out;
422
423  pte = *ptep;
```

It simply uses the three offset macros to navigate the page tables and the `_none()` and `_bad()` macros to make sure it is looking at a valid page table.

The third set of macros examine and set the permissions of an entry. The permissions determine what a userspace process can and cannot do with a particular page. For example, the kernel page table entries are never readable by a userspace process.

- The read permissions for an entry are tested with `pte_read()`, set with `pte_mkread()` and cleared with `pte_rprotect();`
- The write permissions are tested with `pte_write()`, set with `pte_mkwrite()` and cleared with `pte_wrprotect();`
- The execute permissions are tested with `pte_exec()`, set with `pte_mkexec()` and cleared with `pte_exprotect()`. It is worth nothing that with the x86 architecture, there is no means of setting execute permissions on pages so these three macros act the same way as the read macros;
- The permissions can be modified to a new value with `pte_modify()` but its use is almost non-existent. It is only used in the function `change_pte_range()` in `mm/mprotect.c`. 

3.4 Translating and Setting Page Table Entries

The fourth set of macros examine and set the state of an entry. There are only two bits that are important in Linux, the dirty bit and the accessed bit. To check these bits, the macros `pte_dirty()` and `pte_young()` macros are used. To set the bits, the macros `pte_mkdirty()` and `pte_mkyoung()` are used. To clear them, the macros `pte_mkclean()` and `pte_old()` are available.

3.5 Allocating and Freeing Page Tables

The last set of functions deal with the allocation and freeing of page tables. Page tables, as stated, are physical pages containing an array of entries and the allocation and freeing of physical pages is a relatively expensive operation, both in terms of time and the fact that interrupts are disabled during page allocation. The allocation and deletion of page tables, at any of the three levels, is a very frequent operation so it is important the operation is as quick as possible.

Hence the pages used for the page tables are cached in a number of different lists called quicklists. Each architecture implements these caches differently but the principles used are the same. For example, not all architectures cache PGDs because the allocation and freeing of them only happens during process creation and exit. As both of these are very expensive operations, the allocation of another page is negligible.

PGDs, PMDs and PTEs have two sets of functions each for the allocation and freeing of page tables. The allocation functions are `pgd_alloc()`, `pmd_alloc()` and `pte_alloc()` respectively and the free functions are, predictably enough, called `pgd_free()`, `pmd_free()` and `pte_free()`.

Broadly speaking, the three implement caching with the use of three caches called `pgd_quicklist`, `pmd_quicklist` and `pte_quicklist`. Architectures implement these three lists in different ways but one method is through the use of a
LIFO type structure. Ordinarily, a page table entry contains points to other pages containing page tables or data. While cached, the first element of the list is used to point to the next free page table. During allocation, one page is popped off the list and during free, one is placed as the new head of the list. A count is kept of how many pages are used in the cache.

The quick allocation function from the pgd_quicklist is not externally defined outside of the architecture although get_pgd_fast() is a common choice for the function name. The cached allocation function for PMDs and PTEs are publicly defined as pmd_alloc_one_fast() and pte_alloc_one_fast().

If a page is not available from the cache, a page will be allocated using the physical page allocator (see Chapter 6). The functions for the three levels of page tables are get_pgd_slow(), pmd_alloc_one() and pte_alloc_one().

Obviously a large number of pages may exist on these caches and so there is a mechanism in place for pruning them. Each time the caches grow or shrink, a counter is incremented or decremented and it has a high and low watermark. check_pgt_cache() is called in two places to check these watermarks. When the high watermark is reached, entries from the cache will be freed until the cache size returns to the low watermark. The function is called after clear_page_tables() when a large number of page tables are potentially reached and is also called by the system idle task.

3.6 Kernel Page Tables

When the system first starts, paging is not enabled as page tables do not magically initialise themselves. Each architecture implements this differently so only the x86 case will be discussed. The page table initialisation is divided into two phases. The bootstrap phase sets up page tables for just 8MiB so the paging unit can be enabled. The second phase initialises the rest of the page tables. We discuss both of these phases below.

3.6.1 Bootstrapping

The assembler function startup_32() is responsible for enabling the paging unit in arch/i386/kernel/head.S. While all normal kernel code in vmlinux is compiled with the base address at PAGE_OFFSET + 1MiB, the kernel is actually loaded beginning at the first megabyte (0x00100000) of memory. The first megabyte is used by some devices for communication with the BIOS and is skipped. The bootstrap code in this file treats 1MiB as its base address by subtracting __PAGE_OFFSET from any address until the paging unit is enabled so before the paging unit is enabled, a page table mapping has to be established which translates the 8MiB of physical memory to the virtual address PAGE_OFFSET.

Initialisation begins with statically defining at compile time an array called swapper_pg_dir which is placed using linker directives at 0x00101000. It then establishes page table entries for 2 pages, pg0 and pg1. If the processor supports
3.6.2 Finalising

the Page Size Extension (PSE) bit, it will be set so that pages will be translated are 4MiB pages, not 4KiB as is the normal case. The first pointers to pg0 and pg1 are placed to cover the region 1-9MiB the second pointers to pg0 and pg1 are placed at PAGE_OFFSET+1MiB. This means that when paging is enabled, they will map to the correct pages using either physical or virtual addressing for just the kernel image. The rest of the kernel page tables will be initialised by paging_init().

Once this mapping has been established, the paging unit is turned on by setting a bit in the cr0 register and a jump takes places immediately to ensure the Instruction Pointer (EIP register) is correct.

3.6.2 Finalising

The function responsible for finalising the page tables is called paging_init(). The call graph for this function on the x86 can be seen on Figure 3.4.

![Figure 3.4: Call Graph: paging_init()](image)

The function first calls pagetable_init() to initialise the page tables necessary to reference all physical memory in ZONE_DMA and ZONE_NORMAL. Remember that high memory in ZONE_HIGMEM cannot be directly referenced and mappings are set up for it temporarily. For each pgd_t used by the kernel, the boot memory allocator (see Chapter 5) is called to allocate a page for the PMDs and the PSE bit will be set if available to use 4MiB TLB entries instead of 4KiB. If the PSE bit is not supported, a page for PTEs will be allocated for each pmd_t. If the CPU supports the PGE flag, it also will be set so that the page table entry will be global and visible to all processes.

Next, pagetable_init() calls fixrange_init() to setup the fixed address space mappings at the end of the virtual address space starting at FIXADDR_START. These mappings are used for purposes such as the local APIC and the atomic kmap-pings between FIX_KMAP_BEGIN and FIX_KMAP_END required by kmap_atomic(). Finally, the function calls fixrange_init() to initialise the page table entries required for normal high memory mappings with kmap().
3.7 Mapping addresses to a struct page

Once pagetable_init() returns, the page tables for kernel space are now full initialised so the static PGD (swapper_pg_dir) is loaded into the CR3 register so that the static table is now being used by the paging unit.

The next task of the paging_init() is responsible for calling kmap_init() to initialise each of the PTEs with the PAGE_KERNEL protection flags. The final task is to call zone_sizes_init() which initialises all the zone structures used.

3.7 Mapping addresses to a struct page

There is a requirement for Linux to have a fast method of mapping virtual addresses to physical addresses and for mapping struct pages to their physical address. Linux achieves this by knowing where, in both virtual and physical memory, the global mem_map array is as the global array has pointers to all struct pages representing physical memory in the system. All architectures achieve this with very similar mechanisms but for illustration purposes, we will only examine the x86 carefully. This section will first discuss how physical addresses are mapped to kernel virtual addresses and then what this means to the mem_map array.

3.7.1 Mapping Physical to Virtual Kernel Addresses

As we saw in Section 3.6, Linux sets up a direct mapping from the physical address 0 to the virtual address PAGE_OFFSET at 3GiB on the x86. This means that any virtual address can be translated to the physical address by simply subtracting PAGE_OFFSET which is essentially what the function virt_to_phys() with the macro __pa() does:

```c
/* from <asm-i386/page.h> */
#define __pa(x) ((unsigned long)(x)-PAGE_OFFSET)
```

```
/* from <asm-i386/io.h> */
static inline unsigned long virt_to_phys(volatile void * address)
{
    return __pa(address);
}
```

Obviously the reverse operation involves simply adding PAGE_OFFSET which is carried out by the function phys_to_virt() with the macro __va(). Next we see how this helps the mapping of struct pages to physical addresses.

3.7.2 Mapping struct pages to Physical Addresses

As we saw in Section 3.6.1, the kernel image is located at the physical address 1MiB, which of course translates to the virtual address PAGE_OFFSET + 0x00100000 and a virtual region totaling about 8MiB is reserved for the image which is the region that can be addressed by two PGDs. This would imply that the first available memory to use is located at 0xC0800000 but that is not the case. Linux tries to reserve the first
16MiB of memory for ZONE_DMA so first virtual area used for kernel allocations is actually 0xC1000000. This is where the global mem_map is usually located. ZONE_DMA will be still get used, but only when absolutely necessary.

Physical addresses are translated to struct pages by treating them as an index into the mem_map array. Shifting a physical address PAGE_SHIFT bits to the right will treat it as aPFN from physical address 0 which is also an index within the mem_map array. This is exactly what the macro virt_to_page() does which is declared as follows in <asm-i386/page.h>:

```c
#define virt_to_page(kaddr) (mem_map + (__pa(kaddr) >> PAGE_SHIFT))
```

The macro virt_to_page() takes the virtual address kaddr, converts it to the physical address with __pa(), converts it into an array index by bit shifting it right PAGE_SHIFT bits and indexing into the mem_map by simply adding them together. No macro is available for converting struct pages to physical addresses but at this stage, it should be obvious to see how it could be calculated.

### 3.8 Translation Lookaside Buffer (TLB)

Initially, when the processor needs to map a virtual address to a physical address, it must traverse the full page directory searching for the PTE of interest. This would normally imply that each assembly instruction that references memory actually requires several separate memory references for the page table traversal [Tan01]. To avoid this considerable overhead, architectures take advantage of the fact that most processes exhibit a locality of reference or, in other words, large numbers of memory references tend to be for a small number of pages. They take advantage of this reference locality by providing a Translation Lookaside Buffer (TLB) which is a small associative memory that caches virtual to physical page table resolutions.

Linux assumes that the most architectures support some type of TLB although the architecture independent code does not cares how it works. Instead, architecture dependant hooks are dispersed throughout the VM code at points where it is known that some hardware with a TLB would need to perform a TLB related operation. For example, when the page tables have been updated, such as after a page fault has completed, the processor may need to be update the TLB for that virtual address mapping.

Not all architectures require these type of operations but because some do, the hooks have to exist. If the architecture does not require the operation to be performed, the function for that TLB operation will a null operation that is optimised out at compile time.

A quite large list of TLB API hooks, most of which are declared in <asm/pgtable.h>, are listed in Tables 3.2 and 3.3 and the APIs are quite well documented in the kernel source by Documentation/cachetlb.txt [Mil00]. It is possible to have just one TLB flush function but as both TLB flushes and TLB refills are very expensive operations, unnecessary TLB flushes should be avoided if at all possible. For example,
when context switching, Linux will avoid loading new page tables using *Lazy TLB Flushing*, discussed further in Section 4.3.

```c
void flush_tlb_all(void)

This flushes the entire TLB on all processors running in the system making it the most expensive TLB flush operation. After it completes, all modifications to the page tables will be visible globally. This is required after the kernel page tables, which are global in nature, have been modified such as after `vfree()` (See Chapter 7) completes or after the PKMap is flushed (See Chapter 9).

```c
void flush_tlb_mm(struct mm_struct *mm)

This flushes all TLB entries related to the userspace portion (i.e. below `PAGE_OFFSET`) for the requested mm context. In some architectures, such as MIPS, this will need to be performed for all processors but usually it is confined to the local processor. This is only called when an operation has been performed that affects the entire address space, such as after all the address mapping have been duplicated with `dup_mmap()` for fork or after all memory mappings have been deleted with `exit_mmap()`.

```c
void flush_tlb_range(struct mm_struct *mm, unsigned long start,
unsigned long end)

As the name indicates, this flushes all entries within the requested userspace range for the mm context. This is used after a new region has been moved or changeh as during `mremap()` which moves regions or `mprotect()` which changes the permissions. The function is also indirectly used during unmapping a region with `munmap()` which calls `tlb_finish_mmu()` which tries to use `flush_tlb_range()` intelligently. This API is provided for architectures that can remove ranges of TLB entries quickly rather than iterating with `flush_tlb_page()`.

Table 3.2: Translation Lookaside Buffer Flush API

3.9 Level 1 CPU Cache Management

As Linux manages the CPU Cache in a very similar fashion to the TLB, this section covers how Linux utilises and manages the CPU cache. CPU caches, like TLB caches, take advantage of the fact that programs tend to exhibit a locality of reference [Sea00] [CS98]. To avoid having to fetch data from main memory for each reference, the CPU will instead cache very small amounts of data in the CPU cache. Frequently, there is two levels called the Level 1 and Level 2 CPU caches. The Level 2 CPU caches are larger but slower than the L1 cache but Linux only concerns itself with the Level 1 or L1 cache.
void flush_tlb_page(struct vm_area_struct *vma, unsigned long addr)

Predictably, this API is responsible for flushing a single page from the TLB. The two most common usage of it is for flushing the TLB after a page has been faulted in or has been paged out.

void flush_tlb_pgtables(struct mm_struct *mm, unsigned long start, unsigned long end)

This API is called with the page tables are being torn down and freed. Some platforms cache the lowest level of the page table, i.e., the actual page frame storing entries, which needs to be flushed when the pages are being deleted. This is called when a region is being unmapped and the page directory entries are being reclaimed.

void update_mmu_cache(struct vm_area_struct *vma, unsigned long addr, pte_t pte)

This API is only called after a page fault completes. It tells the architecture dependent code that a new translation now exists at pte for the virtual address addr. It is up to each architecture how this information should be used. For example, Sparc64 uses the information to decide if the local CPU needs to flush it's data cache or does it need to send an IPI to a remote processor.

CPU caches are organised into lines. Each line is typically quite small, usually 32 bytes and each line is aligned to it's boundary size. In other words, a cache line of 32 bytes will be aligned on a 32 byte address. With Linux, the size of the line is L1_CACHE_BYTES which is defined by each architecture.

How addresses are mapped to cache lines vary between architectures but the mappings come under three headings, direct mapping, associative mapping and set associative mapping. Direct mapping is the simplest approach where each block of memory maps to only one possible cache line. With associative mapping, any block of memory can map to any cache line. Set associative mapping is a hybrid approach where any block of memory can map to any line but only within a subset of the available lines. Regardless of the mapping scheme, they each have one thing in common, addresses that are close together and aligned to the cache size are likely to use different lines. Hence Linux employs simple tricks to try and maximise cache usage

- Frequently accessed structure fields are at the start of the structure to increase the chance that only one line is needed to address the common fields;
- Unrelated items in a structure should try to be at least cache size bytes apart to avoid false sharing between CPUs;
- Objects in the general caches, such as the mm_struct cache, are aligned to the
L1 CPU cache to avoid false sharing.

If the CPU references an address that is not in the cache, a cache miss occurs and the data is fetched from main memory. The cost of cache misses is quite high as a reference to cache can typically be performed in less than 10ns where a reference to main memory typically will cost between 100ns and 200ns. The basic objective is then to have as many cache hits and as few cache misses as possible.

Just as some architectures do not automatically manage their TLBs, some do not automatically manage their CPU caches. The hooks are placed in locations where the virtual to physical mapping changes, such as during a page table update. The CPU cache flushes should always take place first as some CPUs require a virtual to physical mapping to exist when the virtual address is being flushed from the cache. The three operations that require proper ordering are important is listed in Table 3.4.

<table>
<thead>
<tr>
<th>Flushing Full MM</th>
<th>Flushing Range</th>
<th>Flushing Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>flush_cache_mm()</code></td>
<td><code>flush_cache_range()</code></td>
<td><code>flush_cache_page()</code></td>
</tr>
<tr>
<td>Change all page tables</td>
<td>Change page table range</td>
<td>Change single PTE</td>
</tr>
<tr>
<td><code>flush_tlb_mm()</code></td>
<td><code>flush_tlb_range()</code></td>
<td><code>flush_tlb_page()</code></td>
</tr>
</tbody>
</table>

Table 3.4: Cache and TLB Flush Ordering

The API used for flushing the caches are declared in `<asm/pgtable.h>` and are listed in Tables 3.5. In many respects, it is very similar to the TLB flushing API.

It does not end there though. A second set of interfaces is required to avoid virtual aliasing problems. The problem is that some CPUs select lines based on the virtual address meaning that one physical address can exist on multiple lines leading to cache coherency problems. Architectures with this problem may try and ensure that shared mappings will only use addresses as a stop-gap measure. However, a proper API to address is problem is also supplied which is listed in Table 3.6.

## 3.10 What’s New In 2.6

Most of the mechanics for page table management are essentially the same for 2.6 but the changes that have been introduced are quite wide reaching and the implementations in-depth.

### MMU-less Architecture Support

A new file has been introduced called `mm/nommu.c`. This source file contains replacement code for functions that assume the existence of a MMU like `mmmap()` for example. This is to support architectures, usually microcontrollers, that have no MMU. Much of the work in this area was developed by the uCLinux Project ([http://www.uclinux.org](http://www.uclinux.org)).
void flush_cache_all(void)

This flushes the entire CPU cache system making it the most severe flush operation to use. It is used when changes to the kernel page tables, which are global in nature, are to be performed.

void flush_cache_mm(struct mm_struct mm)

This flushes all entries related to the address space. On completion, no cache lines will be associated with mm.

void flush_cache_range(struct mm_struct *mm, unsigned long start, unsigned long end)

This flushes lines related to a range of addresses in the address space. Like it's TLB equivalent, it is provided in case the architecture has an efficient way of flushing ranges instead of flushing each individual page.

void flush_cache_page(struct vm_area_struct *vma, unsigned long vmaddr)

This is for flushing a single page sized region. The VMA is supplied as the mm_struct is easily accessible via vma->vm_mm. Additionally, by testing for the VM_EXEC flag, the architecture will know if the region is executable for caches that separate the instructions and data caches. VMAs are described further in Chapter 4.

Table 3.5: CPU Cache Flush API

Reverse Mapping  The most significant and important change to page table management is the introduction of Reverse Mapping (rmap). Referring to it as “rmap” is deliberate as it is the common usage of the “acronym” and should not be confused with the -rmap tree developed by Rik van Riel which has many more alterations to the stock VM than just the reverse mapping.

In a single sentence, rmap grants the ability to locate all PTEs which map a particular page given just the struct page. In 2.4, the only way to find all PTEs which map a shared page, such as a memory mapped shared library, is to linearly search all page tables belonging to all processes. This is far too expensive and Linux tries to avoid the problem by using the swap cache (see Section 11.4). This means that with many shared pages, Linux may have to swap out entire processes regardless of the page age and usage patterns. 2.6 instead has a PTE chain associated with every struct page which may be traversed to remove a page from all page tables that reference it. This way, pages in the LRU can be swapped out in an intelligent manner without resorting to swapping entire processes.

As might be imagined by the reader, the implementation of this simple concept is a little involved. The first step in understanding the implementation is the union pte that is a field in struct page. This has union has two fields, a pointer to a struct pte_chain called chain and a pte_addr_t called direct. The union
3.10 What’s New In 2.6

void flush_page_to_ram(unsigned long address)

This is a deprecated API which should no longer be used and in fact will be removed totally for 2.6. It is covered here for completeness and because it is still used. The function is called when a new physical page is about to be placed in the address space of a process. It is required to avoid writes from kernel space being invisible to userspace after the mapping occurs.

void flush_dcache_page(struct page *page)

This function is called when the kernel writes to or copies from a page cache page as these are likely to be mapped by multiple processes.

void flush_icache_range(unsigned long address, unsigned long endaddr)

This is called when the kernel stores information in addresses that is likely to be executed, such as when a kernel module has been loaded.

void flush_icache_user_range(struct vm_area_struct *vma, struct page *page, unsigned long addr, int len)

This is similar to flush_icache_range() except it is called when a userspace range is affected. Currently, this is only used for ptrace() (used when debugging) when the address space is being accessed by access_process_vm().

void flush_icache_page(struct vm_area_struct *vma, struct page *page)

This is called when a page-cache page is about to be mapped. It is up to the architecture to use the VMA flags to determine whether the I-Cache or D-Cache should be flushed.

Table 3.6: CPU D-Cache and I-Cache Flush API

is an optimisation whereby direct is used to save memory if there is only one PTE mapping the entry, otherwise a chain is used. The type pte_addr_t varies between architectures but whatever its type, it can be used to locate a PTE, so we will treat it as a pte_t for simplicity.

The struct pte_chain is a little more complex. The struct itself is very simple but it is compact with overloaded fields and a lot of development effort has been spent on making it small and efficient. Fortunately, this does not make it indecipherable.

First, it is the responsibility of the slab allocator to allocate and manage struct pte_chains as it is this type of task the slab allocator is best at. Each struct pte_chain can hold up to NRPTET pointers to PTE structures. Once that many PTEs have been filled, a struct pte_chain is allocated and added to the chain.

The struct pte_chain has two fields. The first is unsigned long next_and_idx which has two purposes. When next_and_idx is ANDed with NRPTET, it returns the
number of PTEs currently in this struct pte_chain indicating where the next free
slot is. When next_and_idx is ANDed with the negation of NRPTE (i.e. ∼NRPTE), a
pointer to the next struct pte_chain in the chain is returned. This is basically
how a PTE chain is implemented.

To give a taste of the rmap intricacies, we’ll give an example of what happens
when a new PTE needs to map a page. The basic process is to have the caller
allocate a new pte_chain with pte_chain_alloc(). This allocated chain is passed
with the struct page and the PTE to page_add_rmap(). If the existing PTE
chain associated with the page has slots available, it will be used and the pte_chain
allocated by the caller returned. If no slots were available, the allocated pte_chain
will be added to the chain and NULL returned.

There is a quite substantial API associated with rmap, for tasks such as creat-
ing chains and adding and removing PTEs to a chain, but a full listing is beyond
the scope of this section. Fortunately, the API is confined to mm/rmap.c and the
functions are heavily commented so their purpose is clear.

There are two main benefits, both related to pageout, with the introduction
of reverse mapping. The first is with the setup and tear-down of pagetables. As
will be seen in Section 11.4, pages being paged out are placed in a swap cache and
information is written into the PTE necessary to find the page again. This can lead
to multiple minor faults as pages are put into the swap cache and then faulted again
by a process. With rmap, the setup and removal of PTEs is atomic. The second
major benefit is when pages need to paged out, finding all PTEs referencing the
pages is a simple operation but impractical with 2.4, hence the swap cache.

Reverse mapping is not without its cost though. The first, and obvious one, is the
additional space requirements for the PTE chains. Arguably, the second is a CPU
cost associated with reverse mapping but it has not been proved to be significant.
What is important to note though is that reverse mapping is only a benefit when
pageouts are frequent. If the machines workload does not result in much pageout
or memory is ample, reverse mapping is all cost with little or no benefit. At the
time of writing, the merits and downsides to rmap is still the subject of a number
of discussions.

Object-Based Reverse Mapping The reverse mapping required for each page
can have very expensive space requirements. To compound the problem, many
of the reverse mapped pages in a VMA will be essentially identical. One way of
addressing this is to reverse map based on the VMAs rather than individual pages.
That is, instead of having a reverse mapping for each page, all the VMAs which
map a particular page would be traversed and unmapped the page from each. Note
that objects in this case refers to the VMAs, not an object in the object-orientated
sense of the word. At the time of writing, this feature has not been merged yet
and was last seen in kernel 2.5.68-mm1 but there is a strong incentive to have it

\footnote{Told you it was compact.}

\footnote{Don’t blame me, I didn’t name it. In fact the original patch for this feature came with the
comment “From Dave. Crappy name”}
available if the problems with it can be resolved. For the very curious, the patch for just file/device backed objrmap at this release is available\(^3\) but it is only for the very very curious reader.

There are two tasks that require all PTEs that map a page to be traversed. The first task is `page_referenced()` which checks all PTEs that map a page to see if the page has been referenced recently. The second task is when a page needs to be unmapped from all processes with `try_to_unmap()`. To complicate matters further, there are two types of mappings that must be reverse mapped, those that are backed by a file or device and those that are anonymous. In both cases, the basic objective is to traverse all VMAs which map a particular page and then walk the page table for that VMA to get the PTE. The only difference is how it is implemented. The case where it is backed by some sort of file is the easiest case and was implemented first so we’ll deal with it first. For the purposes of illustrating the implementation, we’ll discuss how `page_referenced()` is implemented.

`page_referenced()` calls `page_referenced_obj()` which is the top level function for finding all PTEs within VMAs that map the page. As the page is mapped for a file or device, `page->mapping` contains a pointer to a valid `address_space`. The `address_space` has two linked lists which contain all VMAs which use the mapping with the `address_space->i_mmap` and `address_space->i_mmap_shared` fields. For every VMA that is on these linked lists, `page_referenced_obj_one()` is called with the VMA and the page as parameters. The function `page_referenced_obj_one()` first checks if the page is in an address managed by this VMA and if so, traverses the page tables of the `mm_struct` using the VMA (`vma->vm_mm`) until it finds the PTE mapping the page for that `mm_struct`.

Anonymous page tracking is a lot trickier and was implemented in a number of stages. It only made a very brief appearance and was removed again in 2.5.65-mm4 as it conflicted with a number of other changes. The first stage in the implementation was to use `page->mapping` and `page->index` fields to track `mm_struct` and `address` pairs. These fields previously had been used to store a pointer to `swapper_space` and a pointer to the `swp_entry_t` (See Chapter 11). Exactly how it is addressed is beyond the scope of this section but the summary is that `swp_entry_t` is stored in `page->private`.

`try_to_unmap_obj()` works in a similar fashion but obviously, all the PTEs that reference a page with this method can do so without needing to reverse map the individual pages. There is a serious search complexity problem that is preventing it being merged. The scenario that describes the problem is as follows;

Take a case where 100 processes have 100 VMAs mapping a single file. To unmapped a single page in this case with object-based reverse mapping would require 10,000 VMAs to be searched, most of which are totally unnecessary. With page based reverse mapping, only 100 `pte_chain` slots need to be examined, one for each process. An optimisation was introduced to order VMAs in the `address_space` by virtual address but the search for a single page is still far too expensive for

---

\(^3\)ftp://ftp.kernel.org/pub/linux/kernel/people/akpm/patches/2.5/2.5.68/2.5.68-mm2/experimental
3.10 What's New In 2.6

object-based reverse mapping to be merged.

**PTEs in High Memory** In 2.4, page table entries exist in ZONE_NORMAL as the kernel needs to be able to address them directly during a page table walk. This was acceptable until it was found that, with high memory machines, ZONE_NORMAL was being consumed by the third level page table PTEs. The obvious answer is to move PTEs to high memory which is exactly what 2.6 does.

As we will see in Chapter 9, addressing information in high memory is far from free, so moving PTEs to high memory is a compile time configuration option. In short, the problem is that the kernel must map pages from high memory into the lower address space before it can be used but there is a very limited number of slots available for these mappings introducing a troublesome bottleneck. However, for applications with a large number of PTEs, there is little other option. At time of writing, a proposal has been made for having a User Kernel Virtual Area (UKVA) which would be a region in kernel space private to each process but it is unclear if it will be merged for 2.6 or not.

To take the possibility of high memory mapping into account, the macro pte_offset() from 2.4 has been replaced with pte_offset_map() in 2.6. If PTEs are in low memory, this will behave the same as pte_offset() and return the address of the PTE. If the PTE is in high memory, it will first be mapped into low memory with kmap_atomic() so it can be used by the kernel. This PTE must be unmapped as quickly as possible with pte_unmap().

In programming terms, this means that page table walk code looks slightly different. In particular, to find the PTE for a given address, the code now reads as (taken from mm/memory.c);

```c
640         ptep = pte_offset_map(pmd, address);
641         if (!ptep)
642             goto out;
643
644         pte = *ptep;
645         pte_unmap(ptep);
```

Additionally, the PTE allocation API has changed. Instead of pte_alloc(), there is now a pte_alloc_kernel() for use with kernel PTE mappings and pte_alloc_map() for userspace mapping. The principal difference between them is that pte_alloc_kernel() will never use high memory for the PTE.

In memory management terms, the overhead of having to map the PTE from high memory should not be ignored. Only one PTE may be mapped per CPU at a time, although a second may be mapped with pte_offset_map_nested(). This introduces a penalty when all PTEs need to be examined, such as during zap_page_range() when all PTEs in a given range need to be unmapped.

At time of writing, a patch has been submitted which places PMDs in high memory using essentially the same mechanism and API changes. It is likely that it will be merged.
3.10 What’s New In 2.6

**Huge TLB Filesystem**  Most modern architectures support more than one page size. For example, on many x86 architectures, there is an option to use 4KiB pages or 4MiB pages. Traditionally, Linux only used large pages for mapping the actual kernel image and nowhere else. As TLB slots are a scarce resource, it is desirable to be able to take advantages of the large pages especially on machines with large amounts of physical memory.

In 2.6, Linux allows processes to use “huge pages”, the size of which is determined by `HPAGE_SIZE`. The number of available huge pages is determined by the system administrator by using the `/proc/sys/vm/nr_hugepages` proc interface which ultimately uses the function `set_hugetlb_mem_size()`. As the success of the allocation depends on the availability of physically contiguous memory, the allocation should be made during system startup.

The root of the implementation is a **Huge TLB Filesystem (hugetlbfs)** which is a pseudo-filesystem implemented in `fs/hugetlbfs/inode.c`. Basically, each file in this filesystem is backed by a huge page. During initialisation, `init_hugetlbfs_fs()` registers the file system and mounts it as an internal filesystem with `kern_mount()`.

There are two ways that huge pages may be accessed by a process. The first is by using `shmget()` to setup a shared region backed by huge pages and the second is the call `mmap()` on a file opened in the huge page filesystem.

When a shared memory region should be backed by huge pages, the process should call `shmget()` and pass `SHM_HUGETLB` as one of the flags. This results in `hugetlb_zero_setup()` being called which creates a new file in the root of the internal hugetlb filesystem. A file is created in the root of the internal filesystem. The name of the file is determined by an atomic counter called `hugetlbfs_counter` which is incremented every time a shared region is setup.

To create a file backed by huge pages, a filesystem of type hugetlbfs must first be mounted by the system administrator. Instructions on how to perform this task are detailed in `Documentation/vm/hugetlbpage.txt`. Once the filesystem is mounted, files can be created as normal with the system call `open()`. When `mmap()` is called on the open file, the `file_operations` struct `hugetlbfs_file_operations` ensures that `hugetlbfs_file_mmap()` is called to setup the region properly.

Huge TLB pages have their own function for the management of page tables, address space operations and filesystem operations. The names of the functions for page table management can all be seen in `<linux/hugetlb.h>` and they are named very similar to their “normal” page equivalents. The implementation of the hugetlb functions are located near their normal page equivalents so are easy to find.

**Cache Flush Management**  The changes here are minimal. The API function `flush_page_to_ram()` has being totally removed and a new API `flush_dcache_range()` has been introduced.
Chapter 4

Process Address Space

One of the principal advantages of virtual memory is that each process has its own virtual address space, which is mapped to physical memory by the operating system. In this chapter we will discuss the process address space and how Linux manages it.

Zero page The kernel treats the userspace portion of the address space very differently to the kernel portion. For example, allocations for the kernel are satisfied immediately and are visible globally no matter what process is on the CPU. `vmalloc()` is partially an exception as a minor page fault will occur to sync the process page tables with the reference page tables, but the page will still be allocated immediately upon request. With a process, space is simply reserved in the linear address space by pointing a page table entry to a read-only globally visible page filled with zeros. On writing, a page fault is triggered which results in a new page being allocated, filled with zeros, placed in the page table entry and marked writable. It is filled with zeros so that the new page will appear exactly the same as the global zero-filled page.

The userspace portion is not trusted or presumed to be constant. After each context switch, the userspace portion of the linear address space can potentially change except when a Lazy TLB switch is used as discussed later in Section 4.3. As a result of this, the kernel must be prepared to catch all exception and addressing errors raised from userspace. This is discussed in Section 4.5.

This chapter begins with how the linear address space is broken up and what the purpose of each section is. We then cover the structures maintained to describe each process, how they are allocated, initialised and then destroyed. Next, we will cover how individual regions within the process space are created and all the various functions associated with them. That will bring us to exception handling related to the process address space, page faulting and the various cases that occur to satisfy a page fault. Finally, we will cover how the kernel safely copies information to and from userspace.
4.1 Linear Address Space

From a user perspective, the address space is a flat linear address space but predictably, the kernel's perspective is very different. The address space is split into two parts, the userspace part which potentially changes with each full context switch and the kernel address space which remains constant. The location of the split is determined by the value of \texttt{PAGE_OFFSET} which is at 0xC0000000 on the x86. This means that 3GiB is available for the process to use while the remaining 1GiB is always mapped by the kernel. The linear virtual address space as the kernel sees it is illustrated in Figure 4.1.

Figure 4.1: Kernel Address Space

8MiB (the amount of memory addressed by two PGDs) is reserved at \texttt{PAGE_OFFSET} for loading the kernel image to run. 8MiB is simply a reasonable amount of space to reserve for the purposes of loading the kernel image. The kernel image is placed in this reserved space during kernel page tables initialisation as discussed in Section 3.6.1. Somewhere shortly after the image, the \texttt{mem_map} for UMA architectures, as discussed in Chapter 2, is stored. The location of the array is usually at the 16MiB mark to avoid using ZONE_DMA but not always. With NUMA architectures, portions of the virtual \texttt{mem_map} will be scattered throughout this region and where they are actually located is architecture dependent.

The region between \texttt{PAGE_OFFSET} and \texttt{VMALLOC_START} - \texttt{VMALLOC_OFFSET} is the physical memory map and the size of the region depends on the amount of available RAM. As we saw in Section 3.6, page table entries exist to map physical memory to the virtual address range beginning at \texttt{PAGE_OFFSET}. Between the physical memory map and the vmalloc address space, there is a gap of space \texttt{VMALLOC_OFFSET} in size, which on the x86 is 8MiB, to guard against out of bounds errors. For illustration, on a x86 with 32MiB of RAM, \texttt{VMALLOC_START} will be located at \texttt{PAGE_OFFSET} + 0x02000000 + 0x00800000.

In low memory systems, the remaining amount of the virtual address space, minus a 2 page gap, is used by \texttt{vmalloc()} for representing non-contiguous memory allocations in a contiguous virtual address space. In high-memory systems, the \texttt{vmalloc} area extends as far as \texttt{PKMAP_BASE} minus the two page gap and two extra regions are introduced. The first, which begins at \texttt{PKMAP_BASE}, is an area
reserv ed for the mapping of high memory pages into low memory with kmap() as discussed in Chapter 9. The second is for fixed virtual address mappings which extends from \texttt{FIXADDR\_START} to \texttt{FIXADDR\_TOP}. Fixed virtual addresses are needed for subsystems that need to know the virtual address at compile time such as the \textit{Advanced Programmable Interrupt Controller (APIC)}. \texttt{FIXADDR\_TOP} is statically defined to be \texttt{0xFFFFE000} on the x86 which is one page before the end of the virtual address space. The size of the fixed mapping region is calculated at compile time in \texttt{__FIXADDR\_SIZE} and used to index back from \texttt{FIXADDR\_TOP} to give the start of the region \texttt{FIXADDR\_START}.

The region required for \texttt{vmalloc()}, \texttt{kmap()} and the fixed virtual address mapping is what limits the size of \texttt{ZONE\_NORMAL}. As the running kernel needs these functions, a region of at least \texttt{Vmalloc\_RESERVE} will be reserved at the top of the address space. \texttt{Vmalloc\_RESERVE} is architecture specific but on the x86, it is defined as 128MiB. This is why \texttt{ZONE\_NORMAL} is generally referred to being only 896MiB in size; it is the 1GiB of the upper portion of the linear address space minus the minimum 128MiB that is reserved for the \texttt{vmalloc} region.

\section{Managing the Address Space}

The address space usable by the process is managed by a high level \texttt{mm\_struct} which is roughly analogous to the \texttt{vmspace} struct in BSD [McK96]. Each address space consists of a number of page-aligned regions of memory that are in use. They never overlap and represent a set of addresses which contain pages that are related to each other in terms of protection and purpose. These regions are represented by a \texttt{struct vm\_area\_struct} and are roughly analogous to the \texttt{vm\_map\_entry} struct in BSD. For clarity, a region may represent the process heap for use with \texttt{malloc()}, a memory mapped file such as a shared library or a block of anonymous memory allocated with \texttt{mmap()}. The pages for this region may still have to be allocated, be active and resident or have been paged out.

If a region is backed by a file, its \texttt{vm\_file} field will be set. By traversing \texttt{vm\_file}→\texttt{f\_dentry}→\texttt{d\_inode}→\texttt{i\_mapping}, the associated address_space for the region may be obtained. The address_space has all the filesystem specific information required to perform page-based operations on disk.

The relationship between the different address space related structures is illustrated in 4.2. A number of system calls are provided which affect the address space and regions. These are listed in Table 4.1.

\section{Process Address Space Descriptor}

The process address space is described by the \texttt{mm\_struct} struct meaning that only one exists for each process and is shared between userspace threads. In fact, threads are identified in the task list by finding all \texttt{task\_structs} which have pointers to the same \texttt{mm\_struct}. 
A unique \texttt{mm_struct} is not needed for kernel threads as they will never page fault or access the userspace portion. The only exception is page faulting within the vmalloc space. The page fault handling code treats this as a special case and updates the current page table with information in the master page table. As a \texttt{mm_struct} is not needed for kernel threads, the \texttt{task_struct-mm} field for kernel threads is always NULL. For some tasks such as the boot idle task, the \texttt{mm_struct} is never setup but for kernel threads, a call to \texttt{daemonize()} will call \texttt{exit_mm()} to decrement the usage counter.

As TLBFlushes are extremely expensive, especially with architectures such as the PPC, a technique called lazy TLB is employed which avoids unnecessary TLB flushes by processes which do not access the userspace page tables as the kernel portion of the address space is always visible. The call to \texttt{switch_mm()}, which results in a TLB flush, is avoided by “borrowing” the \texttt{mm_struct} used by the previous task and placing it in \texttt{task_struct-active_mm}. This technique has made large improvements to context switches times.

When entering lazy TLB, the function \texttt{enter_lazy_tlb()} is called to ensure that a \texttt{mm_struct} is not shared between processors in SMP machines, making it a NULL operation on UP machines. The second time use of lazy TLB is during
4.3 Process Address Space Descriptor

<table>
<thead>
<tr>
<th>System Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fork()</code></td>
<td>Creates a new process with a new address space. All the pages are marked COW and are shared between the two processes until a page fault occurs to make private copies</td>
</tr>
<tr>
<td><code>clone()</code></td>
<td><code>clone()</code> allows a new process to be created that shares parts of its context with its parent and is how threading is implemented in Linux. <code>clone()</code> without the <code>CLONE_VM</code> set will create a new address space which is essentially the same as <code>fork()</code></td>
</tr>
<tr>
<td><code>mmap()</code></td>
<td><code>mmap()</code> creates a new region within the process linear address space</td>
</tr>
<tr>
<td><code>mremap()</code></td>
<td>Remaps or resizes a region of memory. If the virtual address space is not available for the mapping, the region may be moved unless the move is forbidden by the caller.</td>
</tr>
<tr>
<td><code>munmap()</code></td>
<td>This destroys part or all of a region. If the region been unmapped is in the middle of an existing region, the existing region is split into two separate regions</td>
</tr>
<tr>
<td><code>shmat()</code></td>
<td>This attaches a shared memory segment to a process address space</td>
</tr>
<tr>
<td><code>shmdt()</code></td>
<td>Removes a shared memory segment from an address space</td>
</tr>
<tr>
<td><code>execve()</code></td>
<td>This loads a new executable file replacing the current address space</td>
</tr>
<tr>
<td><code>exit()</code></td>
<td>Destroys an address space and all regions</td>
</tr>
</tbody>
</table>

Table 4.1: System Calls Related to Memory Regions

process exit when `start_lazy_tlb()` is used briefly while the process is waiting to be reaped by the parent.

The struct has two reference counts called `mm_users` and `mm_count` for two types of "users". `mm_users` is a reference count of processes accessing the userspace portion of for this `mm_struct`, such as the page tables and file mappings. Threads and the `swap_out()` code for instance will increment this count making sure a `mm_struct` is not destroyed early. When it drops to 0, `exit_mmap()` will delete all mappings and tear down the page tables before decrementing the `mm_count`.

`mm_count` is a reference count of the "anonymous users" for the `mm_struct` initialised at 1 for the "real" user. An anonymous user is one that does not necessarily care about the userspace portion and is just borrowing the `mm_struct`. Example users are kernel threads which use lazy TLB switching. When this count drops to 0, the `mm_struct` can be safely destroyed. Both reference counts exist because anonymous users need the `mm_struct` to exist even if the userspace mappings get destroyed and there is no point delaying the teardown of the page tables.

The `mm_struct` is defined in `<linux/sched.h>` as follows:
struct mm_struct {
    struct vm_area_struct * mmap;
    rb_root_t mm_rb;
    struct vm_area_struct * mmap_cache;
    pgd_t * pgd;
    atomic_t mm_users;
    atomic_t mm_count;
    int map_count;
    struct rw_semaphore mmap_sem;
    spinlock_t page_table_lock;
    struct list_head mmlist;
    unsigned long start_code, end_code, start_data, end_data;
    unsigned long start_brk, brk, start_stack;
    unsigned long arg_start, arg_end, env_start, env_end;
    unsigned long rss, total_vm, locked_vm;
    unsigned long def_flags;
    unsigned long cpu_vm_mask;
    unsigned long swap_address;
    unsigned dumpable:1;
    /* Architecture-specific MM context */
    mm_context_t context;
};

The meaning of each of the field in this sizeable struct is as follows:

**mmap** The head of a linked list of all VMA regions in the address space;

**mm_rb** The VMAs are arranged in a linked list and in a red-black tree for fast lookups. This is the root of the tree;

**mmap_cache** The VMA found during the last call to `find_vma()` is stored in this field on the assumption that the area will be used again soon;

**pgd** The Page Global Directory for this process;

**mm_users** A reference count of users accessing the userspace portion of the address space as explained at the beginning of the section;

**mm_count** A reference count of the anonymous users for the mm_struct starting at 1 for the ‘real’ user as explained at the beginning of this section;

**map_count** Number of VMAs in use;
4.3 Process Address Space Descriptor

**mmap_sem** This is a long lived lock which protects the VMA list for readers and writers. As users of this lock require it for a long time and may need to sleep, a spinlock is inappropriate. A reader of the list takes this semaphore with `down_read()`. If they need to write, it is taken with `down_write()` and the `page_table_lock` spinlock is later acquired while the VMA linked lists are being updated;

**page_table_lock** This protects most fields on the `mm_struct`. As well as the page tables, it protects the RSS (see below) count and the VMA from modification;

**mmalist** All `mm_structs` are linked together via this field;

**start_code, end_code** The start and end address of the code section;

**start_data, end_data** The start and end address of the data section;

**start_brk, brk** The start and end address of the heap;

**start_stack** Predictably enough, the start of the stack region;

**arg_start, arg_end** The start and end address of command line arguments;

**env_start, env_end** The start and end address of environment variables;

**rss** Resident Set Size (RSS) is the number of resident pages for this process. It should be noted that the global zero page is not accounted for by RSS;

**total_vm** The total memory space occupied by all VMA regions in the process;

**locked_vm** The number of resident pages locked in memory;

**def_flags** Only one possible value, `VM_LOCKED`. It is used to determine if all future mappings are locked by default or not;

**cpu_vm_mask** A bitmask representing all possible CPUs in an SMP system. The mask is used by an *InterProcessor Interrupt (IPI)* to determine if a processor should execute a particular function or not. This is important during TLB flush for each CPU;

**swap_address** Used by the pageout daemon to record the last address that was swapped from when swapping out entire processes;

**dumpable** Set by `prctl()`, this flag is important only when tracing a process;

**context** Architecture specific MMU context.

There are a small number of functions for dealing with `mm_structs`. They are described in Table 4.2.
4.3.1 Allocating a Descriptor

Two functions are provided to allocate a mm_struct. To be slightly confusing, they are essentially the same but with small important differences. allocate_mm() is just a preprocessor macro which allocates a mm_struct from the slab allocator (see Chapter 8). mm_alloc() allocates from slab and then calls mm_init() to initialise it.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mm_init()</td>
<td>Initialises a mm_struct by setting starting values for each field, allocating a PGD, initialising spinlocks etc.</td>
</tr>
<tr>
<td>allocate_mm()</td>
<td>Allocates a mm_struct() from the slab allocator</td>
</tr>
<tr>
<td>mm_alloc()</td>
<td>Allocates a mm_struct using allocate_mm() and calls mm_init() to initialise it</td>
</tr>
<tr>
<td>exit_mmap()</td>
<td>Walks through a mm_struct and unmaps all VMAs associated with it</td>
</tr>
<tr>
<td>copy_mm()</td>
<td>Makes an exact copy of the current tasks mm_struct for a new task. This is only used during fork</td>
</tr>
<tr>
<td>free_mm()</td>
<td>Returns the mm_struct to the slab allocator</td>
</tr>
</tbody>
</table>

Table 4.2: Functions related to memory region descriptors

4.3.2 Initialising a Descriptor

The initial mm_struct in the system is called init_mm() and is statically initialised at compile time using the macro INIT_MM().

```c
#define INIT_MM(name) \
{ \
  mm_rb: RB_ROOT, \n  pgd: swapper_pg_dir, \n  mm_users: ATOMIC_INIT(2), \n  mm_count: ATOMIC_INIT(1), \n  mmap_sem: __RWSEM_INITIALIZER(name.mmap_sem), \n  page_table_lock: SPIN_LOCK_UNLOCKED, \n  mmlist: LIST_HEAD_INIT(name.mmlist), \n}
```

Once it is established, new mm_structs are created using their parent mm_struct as a template. The function responsible for the copy operation is copy_mm() and it uses init_mm() to initialise process specific fields.

4.3.3 Destroying a Descriptor

While a new user increments the usage count with atomic_inc(&mm->mm_users), it is decremented with a call to mmput(). If the mm_users count reaches zero, all
4.4 Memory Regions

The full address space of a process is rarely used, only sparse regions are. Each region is represented by a `vm_area_struct` which never overlap and represent a set of addresses with the same protection and purpose. Examples of a region include a read-only shared library loaded into the address space or the process heap. A full list of mapped regions a process has may be viewed via the proc interface at `/proc/PID/maps` where PID is the process ID of the process that is to be examined.

The region may have a number of different structures associated with it as illustrated in Figure 4.2. At the top, there is the `vm_area_struct` which on its own is enough to represent anonymous memory.

If the region is backed by a file, the `struct file` is available through the `vm_file` field which has a pointer to the `struct inode`. The inode is used to get the `struct address_space` which has all the private information about the file including a set of pointers to filesystem functions which perform the filesystem specific operations such as reading and writing pages to disk.

The struct `vm_area_struct` is declared as follows in `<linux/mm.h>`:

the mapped regions are destroyed with `exit_mmap()` and the page tables destroyed as there is no longer any users of the userspace portions. The `mm_count` count is decremented with `mmdrop()` as all the users of the page tables and VMAs are counted as one `mm_struct` user. When `mm_count` reaches zero, the `mm_struct` will be destroyed.
struct vm_area_struct {
    struct mm_struct * vm_mm;
    unsigned long vm_start;
    unsigned long vm_end;

    struct vm_area_struct *vm_next; /* linked list of VM areas per task, sorted by address */

    pgprot_t vm_page_prot;
    unsigned long vm_flags;

    rb_node_t vm_rb; /* Function pointers to deal with this struct. */

    struct vm_area_struct *vm_next_share;
    struct vm_area_struct **vm_pprev_share; /* Information about our backing store: */

    unsigned long vm_pgoff;
    struct file * vm_file;
    unsigned long vm_raend;
    void * vm_private_data;
};

4.4 Memory Regions

- **vm_mm**: The mm_struct this VMA belongs to;
- **vm_start**: The starting address of the region;
- **vm_end**: The end address of the region;
- **vm_next**: All the VMAs in an address space are linked together in an address-ordered singly linked list via this field. It is interesting to note that the VMA list is one of the very rare cases where a singly linked list is used in the kernel;
- **vm_page_prot**: The protection flags that are set for each PTE in this VMA. The different bits are described in Table 3.1;
- **vm_flags**: A set of flags describing the protections and properties of the VMA. They are all defined in `<linux/mm.h>` and are described in Table 4.3;
- **vm_rb**: As well as being in a linked list, all the VMAs are stored on a red-black tree for fast lookups. This is important for page fault handling when finding the correct region quickly is important, especially for a large number of mapped regions;
4.4.1 Memory Region Operations

*vm_next_share* Shared VMA regions based on file mappings (such as shared libraries) linked together with this field;

*vm_pprev_share* The complement of *vm_next_share*;

*vm_ops* The *vm_ops* field contains function pointers for *open()* and *close()* and *nopage()* functionalities. These are needed for syncing with information from the disk;

*vm_pgoff* This is the page aligned offset within a file that is memory mapped;

*vm_file* The struct file pointer to the file being mapped;

*vm_raend* This is the end address of a read-ahead window. When a fault occurs, a number of additional pages after the desired page will be paged in. This field determines how many additional pages are faulted in;

*vm_private_data* Used by some device drivers to store private information. Not of concern to the memory manager.

All the regions are linked together on a linked list ordered by address via the *vm_next* field. When searching for a free area, it is a simple matter of traversing the list but a frequent operation is to search for the VMA for a particular address such as during page faulting for example. In this case, the red-black tree is traversed as it has $O(\log N)$ search time on average. The tree is ordered so that lower addresses than the current node are on the left leaf and higher addresses are on the right.

### 4.4.1 Memory Region Operations

There are three operations which a VMA may support called *open()* and *close()* and *nopage()* functionalities. It supports these with a *vm_operations_struct* in the VMA called *vma->vm_ops*. The struct contains three function pointers and is declared as follows in `<linux/mm.h>`:

```c
133 struct vm_operations_struct {
134     void (*open)(struct vm_area_struct * area);
135     void (*close)(struct vm_area_struct * area);
136     struct page *(*nopage)(struct vm_area_struct * area,
                               unsigned long address,
                               int unused);
137 };
```

The *open()* and *close()* functions are will be called every time a region is created or deleted. These functions are only used by a small number of devices, one filesystem and System V shared regions which need to perform additional operations when regions are opened or closed. For example, the System V *open()* callback will increment the number of VMAs using a shared segment (*shp->shm_nattch*).

The main operation of interest is the *nopage()* callback. This callback is used during a page-fault by *do_no_page()* and is responsible for locating the
## 4.4.1 Memory Region Operations

<table>
<thead>
<tr>
<th>Protection Flags</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM_READ</td>
<td>Pages may be read</td>
</tr>
<tr>
<td>VM_WRITE</td>
<td>Pages may be written</td>
</tr>
<tr>
<td>VM_EXEC</td>
<td>Pages may be executed</td>
</tr>
<tr>
<td>VM_SHARED</td>
<td>Pages may be shared</td>
</tr>
<tr>
<td>VM_DONTCOPY</td>
<td>VMA will not be copied on fork</td>
</tr>
<tr>
<td>VM_DONTEXPAND</td>
<td>Prevents a region being resized. Flag is unused</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mmap Related Flags</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM_MAYREAD</td>
<td>Allow the VM_READ flag to be set</td>
</tr>
<tr>
<td>VM_MAYWRITE</td>
<td>Allow the VM_WRITE flag to be set</td>
</tr>
<tr>
<td>VM_MAYEXEC</td>
<td>Allow the VM_EXEC flag to be set</td>
</tr>
<tr>
<td>VM_MAYSHARE</td>
<td>Allow the VM_SHARE flag to be set</td>
</tr>
<tr>
<td>VM_GROWSDOWN</td>
<td>Shared segment (probably stack) may grow down</td>
</tr>
<tr>
<td>VM_GROWSUP</td>
<td>Shared segment (probably heap) may grow up</td>
</tr>
<tr>
<td>VM_SHM</td>
<td>Pages are used by shared SHM memory segment</td>
</tr>
<tr>
<td>VM_DENYWRITE</td>
<td>What MAP_DENYWRITE for mmap() translates to. Now unused</td>
</tr>
<tr>
<td>VM_EXECUTABLE</td>
<td>What MAP_EXECUTABLE for mmap() translates to. Now unused</td>
</tr>
<tr>
<td>VM_STACK_FLAGS</td>
<td>Flags used by setup_arg_flags() to setup the stack</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Locking Flags</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM_LOCKED</td>
<td>If set, the pages will not be swapped out. Set by mlock()</td>
</tr>
<tr>
<td>VM_IO</td>
<td>Signals that the area is a mmaped region for IO to a device. It will also prevent the region being core dumped</td>
</tr>
<tr>
<td>VM_RESERVED</td>
<td>Do not swap out this region, used by device drivers</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>madvise() Flags</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM_SEQ_READ</td>
<td>A hint that pages will be accessed sequentially</td>
</tr>
<tr>
<td>VM_RAND_READ</td>
<td>A hint stating that readahead in the region is useless</td>
</tr>
</tbody>
</table>

Figure 4.3: Memory Region Flags
4.4.2 File/Device backed memory regions

Most files that are mapped will use a generic vm_operations_struct() called generic_file_vm_ops. It registers only a nopage() function called filemap_nopage(). This nopage() function will either locate the page in the page cache or read the information from disk. The struct is declared as follows in mm/filemap.c:

```c
static struct vm_operations_struct generic_file_vm_ops = {
    nopage:    filemap_nopage,
};
```

4.4.2 File/Device backed memory regions

In the event the region is backed by a file, the vm_file leads to an associated address_space as shown in Figure 4.2. The struct contains information of relevance to the filesystem such as the number of dirty pages which must be flushed to disk. It is declared as follows in `<linux/fs.h>`:

```c
struct address_space {
    struct list_head clean_pages;
    struct list_head dirty_pages;
    struct list_head locked_pages;
    unsigned long nrpages;
    struct address_space_operations *a_ops;
    struct inode *host;
    struct vm_area_struct *i_mmap;
    struct vm_area_struct *i_mmap_shared;
    spinlock_t i_shared_lock;
    int gfp_mask;
};
```

A brief description of each field is as follows:

**clean_pages** List of clean pages that need no synchronisation with backing storage;

**dirty_pages** List of dirty pages that need synchronisation with backing storage;

**locked_pages** List of pages that are locked in memory;

**nrpages** Number of resident pages in use by the address space;

**a_ops** A struct of function for manipulating the filesystem. Each filesystem provides it’s own address_space_operations although they sometimes use generic functions;

**host** The host inode the file belongs to;
4.4.2 File/Device backed memory regions

i_mmap A list of private mappings using this address_space;

i_mmap_shared A list of VMAs which share mappings in this address_space;

i_shared_lock A spinlock to protect this structure;

gfp_mask The mask to use when calling alloc_pages() for new pages.

Periodically the memory manager will need to flush information to disk. The memory manager does not know and does not care how information is written to disk, so the a_ops struct is used to call the relevant functions. It is declared as follows in <linux/fs.h>:

```c
struct address_space_operations {
    int (*writepage)(struct page *);
    int (*readpage)(struct file *, struct page *);
    int (*sync_page)(struct page *);
    int (*prepare_write)(struct file *, struct page *, unsigned, unsigned);
    int (*commit_write)(struct file *, struct page *, unsigned, unsigned);
    int (*prepare_write)(struct address_space *, long);
    int (*commit_write)(struct address_space *, struct page *, unsigned, unsigned);
    int (*bmap)(struct address_space *, long);
    int (*flushpage) (struct page *, unsigned long);
    int (*releasepage) (struct page *, int);
    int (*direct_IO)(int, struct inode *, struct kiobuf *, unsigned long, int);
    int (*direct_fileIO)(int, struct file *, struct kiobuf *, unsigned long, int);
    int (*removepage)(struct page *);
};
```

These fields are all function pointers which are described as follows;

**writepage** Write a page to disk. The offset within the file to write to is stored within the page struct. It is up to the filesystem specific code to find the block. See buffer.c:block_write_full_page();

**readpage** Read a page from disk. See buffer.c:block_read_full_page();
4.4.3 Creating A Memory Region

sync_page Sync a dirty page with disk. See buffer.c:block_sync_page();

prepare_write This is called before data is copied from userspace into a page that will be written to disk. With a journaled filesystem, this ensures the filesystem log is up to date. With normal filesystems, it makes sure the needed buffer pages are allocated. See buffer.c:block_prepare_write();

commit_write After the data has been copied from userspace, this function is called to commit the information to disk. See buffer.c:block_commit_write();

bmap Maps a block so that raw IO can be performed. Mainly of concern to filesystem specific code although it is also when swapping out pages that are backed by a swap file instead of a swap partition;

flushpage This makes sure there is no IO pending on a page before releasing it. See buffer.c:discard_bh_page();

releasepage This tries to flush all the buffers associated with a page before freeing the page itself. See try_to_freeBuffers();

direct_IO This function is used when performing direct IO to an inode. The #define exists so that external modules can determine at compile-time if the function is available as it was only introduced in 2.4.21

direct_fileIO Used to perform direct IO with a struct file. Again, the #define exists for external modules as this API was only introduced in 2.4.22

removepage An optional callback that is used when a page is removed from the page cache in remove_page_from_inode_queue()

4.4.3 Creating A Memory Region

The system call mmap() is provided for creating new memory regions within a process. For the x86, the function calls sys_mmap2() which calls do_mmap2() directly with the same parameters. do_mmap2() is responsible for acquiring the parameters needed by do_mmap_pgoff(), which is the principle function for creating new areas for all architectures.

do_mmap2() first clears the MAP_DENYWRITE and MAP_EXECUTABLE bits from the flags parameter as they are ignored by Linux, which is confirmed by the mmap() manual page. If a file is being mapped, do_mmap2() will look up the struct file based on the file descriptor passed as a parameter and acquire the mm_struct→mmap_sem semaphore before calling do_mmap_pgoff().

do_mmap_pgoff() begins by performing some basic sanity checks. It first checks the appropriate filesystem or device functions are available if a file or device is being mapped. It then ensures the size of the mapping is page aligned and that it does not attempt to create a mapping in the kernel portion of the address space. It then makes sure the size of the mapping does not overflow the range of pgoff and finally that the process does not have too many mapped regions already.
Finding a Mapped Memory Region

4.4.4 Finding a Mapped Memory Region

A common operation is to find the VMA a particular address belongs to, such as during operations like page faulting, and the function responsible for this is `find_vma()`. The function `find_vma()` and other API functions affecting memory regions are listed in Table 4.3.

It first checks the `mmap_cache` field which caches the result of the last call to `find_vma()` as it is quite likely the same region will be needed a few times in succession. If it is not the desired region, the red-black tree stored in the `mm_rb` field is traversed. If the desired address is not contained within any VMA, the function will return the VMA closest to the requested address so it is important callers double check to ensure the returned VMA contains the desired address.

A second function called `find_vma_prev()` is provided which is functionally the same as `find_vma()` except that it also returns a pointer to the VMA preceding the
4.4.5 Finding a Free Memory Region

When a new area is to be memory mapped, a free region has to be found that is large enough to contain the new mapping. The function responsible for finding a free area is get_unmapped_area().

As the call graph in Figure 4.5 indicates, there is little work involved with finding an unmapped area. The function is passed a number of parameters. A struct file is passed representing the file or device to be mapped as well as pgoff which is the offset within the file that is been mapped. The requested address for the mapping is passed as well as its length. The last parameter is the protection flags for the area.

If a device is being mapped, such as a video card, the associated f_op->get_unmapped_area() is used. This is because devices or files may have additional requirements for mapping that generic code can not be aware of, such as the address having to be aligned to a particular virtual address.

If there are no special requirements, the architecture specific function arch_get_unmapped_area() is called. Not all architectures provide their own function. For those that don't, there is a generic version provided in mm/mmap.c.
4.4.6 Inserting a memory region

The principal function for inserting a new memory region is `insert_vm_struct()` whose call graph can be seen in Figure 4.6. It is a very simple function which
4.4.6 Inserting a memory region

The function `insert_vm_struct()` is rarely used as it does not increase the `map_count` field. Instead, the function commonly used is `__insert_vm_struct()` which performs the same tasks except that it increments `map_count`.

Two varieties of linking functions are provided, `vma_link()` and `__vma_link()`. `vma_link()` is intended for use when no locks are held. It will acquire all the necessary locks, including locking the file if the VMA is a file mapping before calling `__vma_link()` which places the VMA in the relevant lists.

It is important to note that many functions do not use the `insert_vm_struct()` functions but instead prefer to call `find_vma_prepare()` themselves followed by a later `vma_link()` to avoid having to traverse the tree multiple times.

The linking in `__vma_link()` consists of three stages which are contained in three separate functions. `__vma_link_list()` inserts the VMA into the linear, singly linked list. If it is the first mapping in the address space (i.e. `prev` is NULL), it will become the red-black tree root node. The second stage is linking the node into the red-black tree with `__vma_link_rb()`. The final stage is fixing up the file share mapping with `__vma_link_file()` which basically inserts the VMA into the linked list of VMAs via the `vm_pprev_share` and `vm_next_share` fields.
4.4.7 Merging contiguous regions

Merging VMAs

Linux used to have a function called `merge_segments()` [Hac02] which was responsible for merging adjacent regions of memory together if the file and permissions matched. The objective was to remove the number of VMAs required, especially as many operations resulted in a number of mappings been created such as calls to `sys_mprotect()`. This was an expensive operation as it could result in large portions of the mappings been traversed and was later removed as applications, especially those with many mappings, spent a long time in `merge_segments()`.

The equivalent function which exists now is called `vma_merge()` and it is only used in two places. The first is user `sys_mmap()` which calls it if an anonymous region is being mapped, as anonymous regions are frequently mergable. The second time is during `do_brk()` which is expanding one region into a newly allocated one where the two regions should be merged. Rather than merging two regions, the function `vma_merge()` checks if an existing region may be expanded to satisfy the new allocation negating the need to create a new region. A region may be expanded if there are no file or device mappings and the permissions of the two areas are the same.

Regions are merged elsewhere, although no function is explicitly called to perform the merging. The first is during a call to `sys_mprotect()` during the fixup of areas where the two regions will be merged if the two sets of permissions are the same after the permissions in the affected region change. The second is during a call to `move_vma()` when it is likely that similar regions will be located beside each other.

4.4.8 Remapping and moving a memory region

Moving VMAs

`mremap()` is a system call provided to grow or shrink an existing memory mapping. This is implemented by the function `sys_mremap()` which may move a memory region if it is growing or it would overlap another region and `MREMAP_FIXED` is not specified in the flags. The call graph is illustrated in Figure 4.7.

If a region is to be moved, `do_mremap()` first calls `get_unmapped_area()` to find a region large enough to contain the new resized mapping and then calls `move_vma()` to move the old VMA to the new location. See Figure 4.8 for the call graph to `move_vma()`.

First `move_vma()` checks if the new location may be merged with the VMAs adjacent to the new location. If they can not be merged, a new VMA is allocated literally one PTE at a time. Next `move_page_tables()` is called (see Figure 4.9 for its call graph) which copies all the page table entries from the old mapping to the new one. While there may be better ways to move the page tables, this method makes error recovery trivial as backtracking is relatively straightforward.

The contents of the pages are not copied. Instead, `zap_page_range()` is called to swap out or remove all the pages from the old mapping and the normal page fault handling code will swap the pages back in from backing storage or from files or will
4.4.9 Locking a Memory Region

Linux can lock pages from an address range into memory via the system call \texttt{mlock()} which is implemented by \texttt{sys_mlock()} whose call graph is shown in Figure 4.10. At a high level, the function is simple; it creates a VMA for the address range to be locked, sets the \texttt{VM_LOCKED} flag on it and forces all the pages to be present with \texttt{make_pages_present()}. A second system call \texttt{mlockall()} which maps to \texttt{sys_mlockall()} is also provided which is a simple extension to do the same work as \texttt{sys_mlock()} except for every VMA on the calling process. Both functions rely on the core function \texttt{do_mlock()} to perform the real work of finding the affected VMAs and deciding what function is needed to fix up the regions as described later.

There are some limitations to what memory may be locked. The address range must be page aligned as VMAs are page aligned. This is addressed by simply rounding the range up to the nearest page aligned range. The second proviso is that the process limit \texttt{RLIMIT_MLOCK} imposed by the system administrator may not be exceeded. The last proviso is that each process may only lock half of physical memory at a time. This is a bit non-functional as there is nothing to stop a process forking a number of times and each child locking a portion but as only root processes are allowed to lock pages, it does not make much difference. It is safe to presume that a root process is trusted and knows what it is doing. If it does not, the system administrator with the resulting broken system probably deserves it and gets to keep
4.4.10 Unlocking the region

Unlocking VMAs

The system calls `munlock()` and `munlockall()` provide the corollary for the locking functions and map to `sys_munlock()` and `sys_munlockall()` respectively. The functions are much simpler than the locking functions as they do not have to make numerous checks. They both rely on the same `do_mmap()` function to fix up the regions.

4.4.11 Fixing up regions after locking

When locking or unlocking, VMAs will be affected in one of four ways, each of which must be fixed up by `mlock_fixup()`. The locking may affect the whole VMA in which case `mlock_fixup_all()` is called. The second condition, handled by `mlock_fixup_start()`, is where the start of the region is locked, requiring that a new VMA be allocated to map the new area. The third condition, handled by `mlock_fixup_end()`, is predictably enough where the end of the region is locked. Finally, `mlock_fixup_middle()` handles the case where the middle of a region is mapped requiring two new VMAs to be allocated.

It is interesting to note that VMAs created as a result of locking are never merged, even when unlocked. It is presumed that processes which lock regions will need to lock the same regions over and over again and it is not worth the processor power to constantly merge and split regions.
4.4.12 Deleting a memory region

The function responsible for deleting memory regions, or parts thereof, is `do_munmap()`. It is a relatively simple operation in comparison to the other memory region related operations and is basically divided up into three parts. The first is to fix up the red-black tree for the region that is about to be unmapped. The second is to release the pages and PTEs related to the region to be unmapped and the third is to fix up the regions if a hole has been generated.

To ensure the red-black tree is ordered correctly, all VMAs to be affected by the unmap are placed on a linked list called `free` and then deleted from the red-black tree with `rb_erase()`. The regions if they still exist will be added with their new addresses later during the fixup.

Next the linked list VMAs on `free` is walked through and checked to ensure it is not a partial unmapping. Even if a region is just to be partially un-
4.4.13 Deleting all memory regions

During process exit, it is necessary to unmap all VMAs associated with a mm_struct. The function responsible is exit_mmap(). It is a very simply function which flushes the CPU cache before walking through the linked list of VMAs, unmapping each of them in turn and freeing up the associated pages before flushing the TLB and deleting the page table entries. It is covered in detail in the Code Commentary.

4.5 Exception Handling

A very important part of VM is how kernel address space exceptions that are not bugs are caught\(^1\). This section does not cover the exceptions that are raised with errors such as divide by zero, we are only concerned with the exception raised as the result of a page fault. There are two situations where a bad reference may occur. The first is where a process sends an invalid pointer to the kernel via a system call which the kernel must be able to safely trap as the only check made initially is that the address is below \texttt{PAGE_OFFSET}. The second is where the kernel uses \texttt{copy_from_user()} or \texttt{copy_to_user()} to read or write data from userspace.

At compile time, the linker creates an exception table in the \texttt{__ex_table} section of the kernel code segment which starts at \texttt{__start___ex_table} and ends at \texttt{__stop___ex_table}. Each entry is of type \texttt{exception_table_entry} which is a pair consisting of an execution point and a fixup routine. When an exception occurs that the page fault handler cannot manage, it calls \texttt{search_exception_table()} to see if a fixup routine has been provided for an error at the faulting instruction. If module support is compiled, each modules exception table will also be searched.

If the address of the current exception is found in the table, the corresponding location of the fixup code is returned and executed. We will see in Section 4.7 how this is used to trap bad reads and writes to userspace.

\(^1\)Many thanks go to Ingo Oeser for clearing up the details of how this is implemented.
4.6 Page Faulting

Pages in the process linear address space are not necessarily resident in memory. For example, allocations made on behalf of a process are not satisfied immediately as the space is just reserved within the `vm_area_struct`. Other examples of non-resident pages include the page having been swapped out to backing storage or writing a read-only page.

Linux, like most operating systems, has a Demand Fetch policy as its fetch policy for dealing with pages that are not resident. This states that the page is only fetched from backing storage when the hardware raises a page fault exception which the operating system traps and allocates a page. The characteristics of backing storage imply that some sort of page prefetching policy would result in less page faults [MM87] but Linux is fairly primitive in this respect. When a page is paged in from swap space, a number of pages after it, up to $2^{\text{page-cluster}}$ are read in by `swapin_readahead()` and placed in the swap cache. Unfortunately there is only a chance that pages likely to be used soon will be adjacent in the swap area making it a poor prepageing policy. Linux would likely benefit from a prepageing policy that adapts to program behaviour [KMC02].

There are two types of page fault, major and minor faults. Major page faults occur when data has to be read from disk which is an expensive operation, else the fault is referred to as a minor, or soft page fault. Linux maintains statistics on the number of these types of page faults with the `task_struct→maj_flt` and `task_struct→min_flt` fields respectively.

The page fault handler in Linux is expected to recognise and act on a number of different types of page faults listed in Table 4.4 which will be discussed in detail later in this chapter.

Each architecture registers an architecture-specific function for the handling of page faults. While the name of this function is arbitrary, a common choice is `do_page_fault()` whose call graph for the x86 is shown in Figure 4.12.

This function is provided with a wealth of information such as the address of the fault, whether the page was simply not found or was a protection error, whether it was a read or write fault and whether it is a fault from user or kernel space. It is responsible for determining which type of fault has occurred and how it should be handled by the architecture-independent code. The flow chart, in Figure 4.13, shows broadly speaking what this function does. In the figure, identifiers with a colon after them corresponds to the label as shown in the code.

`handle_mm_fault()` is the architecture independent top level function for faulting in a page from backing storage, performing COW and so on. If it returns 1, it was a minor fault, 2 was a major fault, 0 sends a `SIGBUS` error and any other value invokes the out of memory handler.

4.6.1 Handling a Page Fault

Once the exception handler has decided the fault is a valid page fault in a valid memory region, the architecture-independent function `handle_mm_fault()`, whose
### 4.6.1 Handling a Page Fault

<table>
<thead>
<tr>
<th>Exception</th>
<th>Type</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Region valid but page not allocated</td>
<td>Minor</td>
<td>Allocate a page frame from the physical page allocator</td>
</tr>
<tr>
<td>Region not valid but is beside an expandable region like the stack</td>
<td>Minor</td>
<td>Expand the region and allocate a page</td>
</tr>
<tr>
<td>Page swapped out but present in swap cache</td>
<td>Minor</td>
<td>Re-establish the page in the process page tables and drop a reference to the swap cache</td>
</tr>
<tr>
<td>Page swapped out to backing storage</td>
<td>Major</td>
<td>Find where the page with information stored in the PTE and read it from disk</td>
</tr>
<tr>
<td>Page write when marked read-only</td>
<td>Minor</td>
<td>If the page is a COW page, make a copy of it, mark it writable and assign it to the process. If it is in fact a bad write, send a SIGSEGV signal</td>
</tr>
<tr>
<td>Region is invalid or process has no permissions to access</td>
<td>Error</td>
<td>Send a SIGSEGV signal to the process</td>
</tr>
<tr>
<td>Fault occurred in the kernel portion address space</td>
<td>Minor</td>
<td>If the fault occurred in the vmalloc area of the address space, the current process page tables are updated against the master page table held by init_mm. This is the only valid kernel page fault that may occur</td>
</tr>
<tr>
<td>Fault occurred in the userspace region while in kernel mode</td>
<td>Error</td>
<td>If a fault occurs, it means a kernel system did not copy from userspace properly and caused a page fault. This is a kernel bug which is treated quite severely.</td>
</tr>
</tbody>
</table>

Table 4.4: Reasons For Page Faulting

The call graph is shown in Figure 4.14, takes over. It allocates the required page table entries if they do not already exist and calls `handle_pte_fault()`.

Based on the properties of the PTE, one of the handler functions shown in Figure 4.14 will be used. The first stage of the decision is to check if the PTE is marked not present or if it has been allocated with which is checked by `pte_present()` and `pte_none()`. If no PTE has been allocated (`pte_none()` returned true), `do_no_page()` is called which handles Demand Allocation. Otherwise it is a page...
4.6.2 Demand Allocation

When a process accesses a page for the very first time, the page has to be allocated and possibly filled with data by the do_no_page() function. If the vm_operations_struct associated with the parent VMA (vma→vm_ops) provides a nopage() function, it is called. This is of importance to a memory mapped device such as a video card which needs to allocate the page and supply data on access or to a mapped file which must retrieve its data from backing storage. We will first discuss the case where the faulting page is anonymous as this is the simplest case.

Handling anonymous pages If vm_area_struct→vm_ops field is not filled or a nopage() function is not supplied, the function do_anonymous_page() is called to handle an anonymous access. There are only two cases to handle, first time read
4.6.2 Demand Allocation

Figure 4.13: do_page_fault() Flow Diagram

and first time write. As it is an anonymous page, the first read is an easy case as no data exists. In this case, the system-wide empty_zero_page, which is just a page of zeros, is mapped for the PTE and the PTE is write protected. The write protection is set so that another page fault will occur if the process writes to the page. On the x86, the global zero-filled page is zeroed out in the function mem_init().

If this is the first write to the page alloc_page() is called to allocate a free page (see Chapter 6) and is zero filled by clear_user_highpage(). Assuming the page was successfully allocated, the Resident Set Size (RSS) field in the mm_struct will be incremented; flush_page_to_ram() is called as required when a page has been inserted into a userspace process by some architectures to ensure cache coherency. The page is then inserted on the LRU lists so it may be reclaimed later by the page reclaiming code. Finally the page table entries for the process are updated for the new mapping.
4.6.2 Demand Allocation

Handling file/device backed pages If backed by a file or device, a `nopage()` function will be provided within the VMAs `vm_operations_struct`. In the file-backed case, the function `filemap_nopage()` is frequently the `nopage()` function for allocating a page and reading a page-sized amount of data from disk. Pages backed by a virtual file, such as those provided by `shmfs`, will use the function `shmem_nopage()` (See Chapter 12). Each device driver provides a different `nopage()` whose internals are unimportant to us here as long as it returns a valid `struct page` to use.

On return of the page, a check is made to ensure a page was successfully allocated and appropriate errors returned if not. A check is then made to see if an early COW break should take place. An early COW break will take place if the fault is a write to the page and the `VM_SHARED` flag is not included in the managing VMA. An early break is a case of allocating a new page and copying the data across before reducing the reference count to the page returned by the `nopage()` function.
4.6.3 Demand Paging

In either case, a check is then made with `pte_none()` to ensure there is not a PTE already in the page table that is about to be used. It is possible with SMP that two faults would occur for the same page at close to the same time and as the spinlocks are not held for the full duration of the fault, this check has to be made at the last instant. If there has been no race, the PTE is assigned, statistics updated and the architecture hooks for cache coherency called.

4.6.3 Demand Paging

When a page is swapped out to backing storage, the function `do_swap_page()` is responsible for reading the page back in, with the exception of virtual files which are covered in Section 12. The information needed to find it is stored within the PTE itself. The information within the PTE is enough to find the page in swap. As pages may be shared between multiple processes, they cannot always be swapped out immediately. Instead, when a page is swapped out, it is placed within the swap cache.

Figure 4.16: Call Graph: `do_swap_page()`

A shared page cannot be swapped out immediately because there is no way of mapping a `struct page` to the PTEs of each process it is shared between. Searching the page tables of all processes is simply far too expensive. It is worth noting that the late 2.5.x kernels and 2.4.x with a custom patch have what is called *Reverse Mapping (RMAP)* which is discussed at the end of the chapter.

With the swap cache existing, it is possible that when a fault occurs it still exists in the swap cache. If it is, the reference count to the page is simply increased and it is placed within the process page tables again and registers as a minor page fault.

If the page exists only on disk `swapin_readahead()` is called which reads in the requested page and a number of pages after it. The number of pages read in is determined by the variable `page_cluster` defined in `mm/swap.c`. On low memory machines with less than 16MiB of RAM, it is initialised as 2 or 3 otherwise. The number of pages read in is \(2^{\text{page-cluster}}\) unless a bad or empty swap entry is encountered. This works on the premise that a seek is the most expensive operation in time so once the seek has completed, the succeeding pages should also be read in.
4.6.4 Copy On Write (COW) Pages

Once upon time, the full parent address space was duplicated for a child when a process forked. This was an extremely expensive operation as it is possible a significant percentage of the process would have to be swapped in from backing storage. To avoid this considerable overhead, a technique called Copy-On-Write (COW) is employed.

During fork, the PTEs of the two processes are made read-only so that when a write occurs there will be a page fault. Linux recognises a COW page because even though the PTE is write protected, the controlling VMA shows the region is writable. It uses the function do_wp_page() to handle it by making a copy of the page and assigning it to the writing process. If necessary, a new swap slot will be reserved for the page. With this method, only the page table entries have to be copied during a fork.

4.7 Copying To/From Userspace

It is not safe to access memory in the process address space directly as there is no way to quickly check if the page addressed is resident or not. Linux relies on the MMU to raise exceptions when the address is invalid and have the Page Fault Exception handler catch the exception and fix it up. In the x86 case, assembler is provided by the __copy_user() to trap exceptions where the address is totally useless. The location of the fixup code is found when the function search_exception_table() is called. Linux provides an ample API (mainly macros) for copying data to and from the user address space safely as shown in Table 4.5.

All the macros map on to assembler functions which all follow similar patterns of implementation so for illustration purposes, we'll just trace how copy_from_user() is implemented on the x86.

If the size of the copy is known at compile time, copy_from_user() calls _constant_copy_from_user() else _generic_copy_from_user() is used. If the size is known, there are different assembler optimisations to copy data in 1, 2 or 4
unsigned long copy_from_user(void *to, const void *from, unsigned long n)
    Copies n bytes from the user address(from) to the kernel address space(to)

unsigned long copy_to_user(void *to, const void *from, unsigned long n)
    Copies n bytes from the kernel address(from) to the user address space(to)

void copy_user_page(void *to, void *from, unsigned long address)
    This copies data to an anonymous or COW page in userspace. Ports are
    responsible for avoiding D-cache aliases. It can do this by using a kernel virtual
    address that would use the same cache lines as the virtual address.

void clear_user_page(void *page, unsigned long address)
    Similar to copy_user_page() except it is for zeroing a page

global void get_user(void *to, void *from)
    Copies an integer value from userspace (from) to kernel space (to)

void put_user(void *from, void *to)
    Copies an integer value from kernel space (from) to userspace (to)

long strncpy_from_user(char *dst, const char *src, long count)
    Copies a null terminated string of at most count bytes long from userspace
    (src) to kernel space (dst)

long strlen_user(const char *s, long n)
    Returns the length, upper bound by n, of the userspace string including the
    terminating NULL

int access_ok(int type, unsigned long addr, unsigned long size)
    Returns non-zero if the userspace block of memory is valid and zero otherwise

<table>
<thead>
<tr>
<th>Table 4.5: Accessing Process Address Space API</th>
</tr>
</thead>
</table>

byte strides otherwise the distinction between the two copy functions is not important.

The generic copy function eventually calls the function __copy_user_zeroing() in
<asm-i386/uaccess.h> which has three important parts. The first part is the
assembler for the actual copying of size number of bytes from userspace. If any
page is not resident, a page fault will occur and if the address is valid, it will get
swapped in as normal. The second part is “fixup” code and the third part is the
__ex_table mapping the instructions from the first part to the fixup code in the
second part.

These pairings, as described in Section 4.5, copy the location of the copy instruc-
tions and the location of the fixup code the kernel exception handle table by the
linker. If an invalid address is read, the function do_page_fault() will fall through,
call search_exception_table() and find the EIP where the faulty read took place
and jump to the fixup code which copies zeros into the remaining kernel space, fixes
up registers and returns. In this manner, the kernel can safely access userspace with
no expensive checks and letting the MMU hardware handle the exceptions.

All the other functions that access userspace follow a similar pattern.

4.8 What’s New in 2.6

Linear Address Space  The linear address space remains essentially the same as
2.4 with no modifications that cannot be easily recognised. The main change is the
addition of a new page usable from userspace that has been entered into the fixed
address virtual mappings. On the x86, this page is located at 0xFFFFF000 and called
the syscall page. Code is located at this page which provides the optimal method
for entering kernel-space from userspace. A userspace program now should use call
0xFFFFF000 instead of the traditional int 0x80 when entering kernel space.

struct mm_struct  This struct has not changed significantly. The first change is
the addition of a free_area_cache field which is initialised as TASK_UNMAPPED_BASE.
This field is used to remember where the first hole is in the linear address space to
improve search times. A small number of fields have been added at the end of the
struct which are related to core dumping and beyond the scope of this book.

struct vm_area_struct  This struct also has not changed significantly. The main
differences is that the vm_next_share and vm_pprev_share has been replaced
with a proper linked list with a new field called simply shared. The vm_raend
has been removed altogether as file readahead is implemented very differently in
2.6. Readahead is mainly managed by a struct file_ra_state struct stored in
struct file→f_ra. How readahead is implemented is described in a lot of detail
in mm/readahead.c.

struct address_space  The first change is relatively minor. The gfp_mask field
has been replaced with a flags field where the first __GFP_BITS_SHIFT bits are
used as the gfp_mask and accessed with mapping_gfp_mask(). The remaining bits
are used to store the status of asynchronous IO. The two flags that may be set are
AS_EIO to indicate an IO error and AS_ENOSPC to indicate the filesystem ran out of
space during an asynchronous write.

This struct has a number of significant additions, mainly related to the page
cache and file readahead. As the fields are quite unique, we'll introduce them in
detail:

page_tree  This is a radix tree of all pages in the page cache for this mapping
indexed by the block the data is located on the physical disk. In 2.4, searching
the page cache involved traversing a linked list, in 2.6, it is a radix tree lookup
which considerably reduces search times. The radix tree is implemented in
lib/radix-tree.c;

`page_lock` Spinlock protecting `page_tree`;

`io_pages` When dirty pages are to be written out, they are added to this
list before `do_writepages()` is called. As explained in the comment above
`mpage_writepages()` in `fs/mpage.c`, pages to be written out are placed on
this list to avoid deadlocking by locking already locked by IO;

`dirtied_when` This field records, in jiffies, the first time an inode was dirtied.
This field determines where the inode is located on the `super_block→s_dirty`
list. This prevents a frequently dirtied inode remaining at the top of the list
and starving writeout on other inodes;

`backing_dev_info` This field records readahead related information. The struct
is declared in `include/linux/backing-dev.h` with comments explaining the
fields;

`private_list` This is a private list available to the `address_space`. If the helper
functions `mark_buffer_dirty_inode()` and `sync_mapping_buffers()` are
used, this list links `buffer_heads` via the `buffer_head→b_assoc_buffers`
field;

`private_lock` This spinlock is available for the `address_space`. The use of
this lock is very convoluted but some of the uses are explained in the long
ChangeLog for 2.5.17 (`http://lwn.net/2002/0523/a/2.5.17.php3`). but it is
mainly related to protecting lists in other mappings which share buffers in
this mapping. The lock would not protect this `private_list`, but it would
protect the `private_list` of another `address_space` sharing buffers with this
mapping;

`assoc_mapping` This is the `address_space` which backs buffers contained in
this mappings `private_list`;

`truncate_count` is incremented when a region is being truncated by the function
`invalidate_mmap_range()`. The counter is examined during page fault by
`do_no_page()` to ensure that a page is not faulted in that was just invalidated.

struct `address_space_operations` Most of the changes to this struct initially
look quite simple but are actually quite involved. The changed fields are:

`writepage` The `writepage()` callback has been changed to take an additional pa-
parameter `struct writeback_control`. This struct is responsible for recording
information about the writeback such as if it is congested or not, if the writer
is the page allocator for direct reclaim or `kupdated` and contains a handle to
the backing `backing_dev_info` to control readahead;
4.8 What's New in 2.6

writepages Moves all pages from dirty_pages to io_pages before writing them all out;

set_page_dirty is an address_space specific method of dirtying a page. This is mainly used by the backing storage address_space_operations and for anonymous shared pages where there are no buffers associated with the page to be dirtied;

readpages Used when reading in pages so that readahead can be accurately controlled;

bmap This has been changed to deal with disk sectors rather than unsigned longs for devices larger than $2^{32}$ bytes.

invalidatepage This is a renaming change. block_flushpage() and the callback flushpage() has been renamed to block_invalidatepage() and invalidatepage();

direct_IO This has been changed to use the new IO mechanisms in 2.6. The new mechanisms are beyond the scope of this book;

Memory Regions The operation of mmap() has two important changes. The first is that it is possible for security modules to register a callback. This callback is called security_file_mmap() which looks up a security_ops struct for the relevant function. By default, this will be a NULL operation.

The second is that there is much stricter address space accounting code in place. vm_area structs which are to be accounted will have the VM_ACCOUNT flag set, which will be all userspace mappings. When userspace regions are created or destroyed, the functions vm_acct_memory() and vm_unacct_memory() update the variable vm_committed_space. This gives the kernel a much better view of how much memory has been committed to userspace.

4GiB/4GiB User/Kernel Split One limitation that exists for the 2.4.x kernels is that the kernel has only 1GiB of virtual address space available which is visible to all processes. At time of writing, a patch has been developed by Ingo Mohan which allows the kernel to optionally have it's own full 4GiB address space. The patches are available from http://redhat.com/mingo/4g-patches/ and are included in the -mm test trees but it is unclear if it will be merged into the mainstream or not.

This feature is intended for 32 bit systems that have very large amounts (>16GiB) of RAM. The traditional 3/1 split adequately supports up to 1GiB of RAM. After that, high-memory support allows larger amounts to be supported by temporarily mapping high-memory pages but with more RAM, this forms a significant bottleneck. For example, as the amount of physical RAM approached the 60GiB

See http://lwn.net/Articles/39283/ for the first announcement of the patch.
range, almost the entire of low memory is consumed by `mem_map`. By giving the
kernel it's own 4GiB virtual address space, it is much easier to support the memory
but the serious penalty is that there is a per-syscall TLB flush which heavily impacts
performance.

With the patch, there is only a small 16MiB region of memory shared between
userspace and kernelspace which is used to store the GDT, IDT, TSS, LDT, vsyscall
page and the kernel stack. The code for doing the actual switch between the pageta-
bles is then contained in the trampoline code for entering/existing kernelspace.
There are a few changes made to the core core such as the removal of direct pointers
for accessing userspace buffers but, by and large, the core kernel is unaffected by
this patch.

### Non-Linear VMA Population

In 2.4, a VMA backed by a file is populated in a linear fashion. This can be optionally changed in 2.6 with the introduction of
the `MAP_POPULATE` flag to `mmap()` and the new system call `remap_file_pages()`,
implemented by `sys_remap_file_pages()`. This system call allows arbitrary pages
in an existing VMA to be remapped to an arbitrary location on the backing file by
manipulating the page tables.

On page-out, the non-linear address for the file is encoded within the PTE so that
it can be installed again correctly on page fault. How it is encoded is architecture
specific so two macros are defined called `pgoff_to_pte()` and `pte_to_pgoff()` for
the task.

This feature is largely of benefit to applications with a large number of mappings
such as database servers and virtualising applications such as emulators. It was
introduced for a number of reasons. First, VMAs are per-process and can have
considerable space requirements, especially for applications with a large number of
mappings. Second, the search `get_unmapped_area()` uses for finding a free area
in the virtual address space is a linear search which is very expensive for large
numbers of mappings. Third, non-linear mappings will pre-fault most of the pages
into memory where as normal mappings may cause a major fault for each page
although can be avoided by using the new flag `MAP_POPULATE` flag with `mmap()` or
my using `mlock()`. The last reason is to avoid sparse mappings which, at worst
case, would require one VMA for every file page mapped.

However, this feature is not without some serious drawbacks. The first is that
the system calls `truncate()` and `mincore()` are broken with respect to non-linear
mappings. Both system calls depend depend on `vm_area_struct->vm_pgoff` which
is meaningless for non-linear mappings. If a file mapped by a non-linear mapping
is truncated, the pages that exists within the VMA will still remain. It has been
proposed that the proper solution is to leave the pages in memory but make them
anonymous but at the time of writing, no solution has been implemented.

The second major drawback is TLB invalidations. Each remapped page will re-
quire that the MMU be told the remapping took place with `flush_icache_page()`
but the more important penalty is with the call to `flush_tlb_page()`. Some pro-
cessors are able to invalidate just the TLB entries related to the page but other
processors implement this by flushing the entire TLB. If re-mappings are frequent, the performance will degrade due to increased TLB misses and the overhead of constantly entering kernel space. In some ways, these penalties are the worst as the impact is heavily processor dependant.

It is currently unclear what the future of this feature, if it remains, will be. At the time of writing, there is still on-going arguments on how the issues with the feature will be fixed but it is likely that non-linear mappings are going to be treated very differently to normal mappings with respect to pageout, truncation and the reverse mapping of pages. As the main user of this feature is likely to be databases, this special treatment is not likely to be a problem.

**Page Faulting** The changes to the page faulting routines are more cosmetic than anything else other than the necessary changes to support reverse mapping and PTEs in high memory. The main cosmetic change is that the page faulting routines return self explanatory compile time definitions rather than magic numbers. The possible return values for handle_mm_fault() are VM_FAULT_MINOR, VM_FAULT_MAJOR, VM_FAULT_SIGBUS and VM_FAULT_OOM.
Chapter 5

Boot Memory Allocator

It is impractical to statically initialise all the core kernel memory structures at compile time as there are simply far too many permutations of hardware configurations. Yet to set up even the basic structures requires memory as even the physical page allocator, discussed in the next chapter, needs to allocate memory to initialise itself. But how can the physical page allocator allocate memory to initialise itself?

To address this, a specialised allocator called the Boot Memory Allocator is used. It is based on the most basic of allocators, a First Fit allocator which uses a bitmap to represent memory \cite{Tan01} instead of linked lists of free blocks. If a bit is 1, the page is allocated and 0 if unallocated. To satisfy allocations of sizes smaller than a page, the allocator records the Page Frame Number (PFN) of the last allocation and the offset the allocation ended at. Subsequent small allocations are “merged” together and stored on the same page.

The reader may ask why this allocator is not used for the running system. One compelling reason is that although the first fit allocator does not suffer badly from fragmentation \cite{JW98}, memory frequently has to linearly searched to satisfy an allocation. As this is examining bitmaps, it gets very expensive, especially as the first fit algorithm tends to leave many small free blocks at the beginning of physical memory which still get scanned for large allocations, thus making the process very wasteful \cite{WJNB95}.

There are two very similar but distinct APIs for the allocator. One is for UMA architectures, listed in Table 5.1 and the other is for NUMA, listed in Table 5.2. The principle difference is that the NUMA API must be supplied with the node affected by the operation but as the callers of these APIs exist in the architecture dependant layer, it is not a significant problem.

This chapter will begin with a description of the structure the allocator uses to describe the physical memory available for each node. We will then illustrate how the limits of physical memory and the sizes of each zone are discovered before talking about how the information is used to initialise the boot memory allocator structures. The allocation and free routines will then be discussed before finally talking about how the boot memory allocator is retired.
5.1 Representing the Boot Map

A `bootmem_data` struct exists for each node of memory in the system. It contains the information needed for the boot memory allocator to allocate memory for a node such as the bitmap representing allocated pages and where the memory is located. It is declared as follows in `<linux/bootmem.h>`:

```c
unsigned long init_bootmem(unsigned long start, unsigned long page)
    This initialises the memory between 0 and the PFN page. The beginning of usable memory is at the PFN start
void reserve_bootmem(unsigned long addr, unsigned long size)
    Mark the pages between the address `addr` and `addr+size` reserved. Requests to partially reserve a page will result in the full page being reserved
void free_bootmem(unsigned long addr, unsigned long size)
    Mark the pages between the address `addr` and `addr+size` free
void * alloc_bootmem(unsigned long size)
    Allocate `size` number of bytes from `ZONE_NORMAL`. The allocation will be aligned to the L1 hardware cache to get the maximum benefit from the hardware cache
void * alloc_bootmem_low(unsigned long size)
    Allocate `size` number of bytes from `ZONE_DMA`. The allocation will be aligned to the L1 hardware cache
void * alloc_bootmem_pages(unsigned long size)
    Allocate `size` number of bytes from `ZONE_NORMAL` aligned on a page size so that full pages will be returned to the caller
void * alloc_bootmem_low_pages(unsigned long size)
    Allocate `size` number of bytes from `ZONE_NORMAL` aligned on a page size so that full pages will be returned to the caller
unsigned long bootmem_bootmap_pages(unsigned long pages)
    Calculate the number of pages required to store a bitmap representing the allocation state of `pages` number of pages
unsigned long free_all_bootmem()
    Used at the boot allocator end of life. It cycles through all pages in the bitmap. For each one that is free, the flags are cleared and the page is freed to the physical page allocator (See next chapter) so the runtime allocator can set up its free lists
```

Table 5.1: Boot Memory Allocator API for UMA Architectures
5.1 Representing the Boot Map

unsigned long init_bootmem_node(pg_data_t *pgdat, unsigned long freepfn, unsigned long startpfn, unsigned long endpfn)

For use with NUMA architectures. It initialise the memory between PFNs startpfn and endpfn with the first usable PFN at freepfn. Once initialised, the pgdat node is inserted into the pgdat_list.

void reserve_bootmem_node(pg_data_t *pgdat, unsigned long physaddr, unsigned long size)

Mark the pages between the address addr and addr+size on the specified node pgdat reserved. Requests to partially reserve a page will result in the full page being reserved.

void free_bootmem_node(pg_data_t *pgdat, unsigned long physaddr, unsigned long size)

Mark the pages between the address addr and addr+size on the specified node pgdat free.

void * alloc_bootmem_node(pg_data_t *pgdat, unsigned long size)

Allocate size number of bytes from ZONE_NORMAL on the specified node pgdat. The allocation will be aligned to the L1 hardware cache to get the maximum benefit from the hardware cache.

void * alloc_bootmem_pages_node(pg_data_t *pgdat, unsigned long size)

Allocate size number of bytes from ZONE_NORMAL on the specified node pgdat aligned on a page size so that full pages will be returned to the caller.

void * alloc_bootmem_low_pages_node(pg_data_t *pgdat, unsigned long size)

Allocate size number of bytes from ZONE_NORMAL on the specified node pgdat aligned on a page size so that full pages will be returned to the caller.

unsigned long free_all_bootmem_node(pg_data_t *pgdat)

Used at the boot allocator end of life. It cycles through all pages in the bitmap for the specified node. For each one that is free, the page flags are cleared and the page is freed to the physical page allocator (See next chapter) so the runtime allocator can set up its free lists.

Table 5.2: Boot Memory Allocator API for NUMA Architectures

25 typedef struct bootmem_data {
26   unsigned long node_boot_start;
27   unsigned long node_low_pfn;
28   void *node_bootmem_map;
29   unsigned long last_offset;
30   unsigned long last_pos;
31 } bootmem_data_t;
5.2 Initialising the Boot Memory Allocator

The fields of this struct are as follows:

- **node_boot_start**: This is the starting physical address of the represented block;
- **node_low_pfn**: This is the end physical address, in other words, the end of the ZONE_NORMAL this node represents;
- **node_bootmem_map**: This is the location of the bitmap representing allocated or free pages with each bit;
- **last_offset**: This is the offset within the page of the end of the last allocation. If 0, the page used is full;
- **last_pos**: This is the the PFN of the page used with the last allocation. Using this with the last_offset field, a test can be made to see if allocations can be merged with the page used for the last allocation rather than using up a full new page.

5.2 Initialising the Boot Memory Allocator

Each architecture is required to supply a setup_arch() function which, among other tasks, is responsible for acquiring the necessary parameters to initialise the boot memory allocator.

Each architecture has its own function to get the necessary parameters. On the x86, it is called setup_memory(), as discussed in Section 2.2.2, but on other architectures such as MIPS or Sparc, it is called bootmem_init() or the case of the PPC, do_init_bootmem(). Regardless of the architecture, the tasks are essentially the same. The parameters it calculates are:

- **min_low_pfn**: This is the lowest PFN that is available in the system;
- **max_low_pfn**: This is the highest PFN that may be addressed by low memory (ZONE_NORMAL);
- **highstart_pfn**: This is the PFN of the beginning of high memory (ZONE_HIGHMEM);
- **highend_pfn**: This is the last PFN in high memory;
- **max_pfn**: Finally, this is the last PFN available to the system.

5.2.1 Initialising bootmem_data

Once the limits of usable physical memory are discovered by setup_memory(), one of two boot memory initialisation functions is selected and provided with the start and end PFN for the node to be initialised. init_bootmem(), which initialises contig_page_data, is used by UMA architectures, while init_bootmem_node()
is for NUMA to initialise a specified node. Both function are trivial and rely on `init_bootmem_core()` to do the real work.

The first task of the core function is to insert this `pgdat_data_t` into the `pgdat_list` as at the end of this function, the node is ready for use. It then records the starting and end address for this node in its associated `bootmem_data_t` and allocates the bitmap representing page allocations. The size in bytes, hence the division by 8, of the bitmap required is calculated as:

\[
\text{mapsize} = \frac{(\text{end}_{\_\text{pfn}} - \text{start}_{\_\text{pfn}}) + 7}{8}
\]

The bitmap is stored at the physical address pointed to by `bootmem_data_t->node_boot_start` and the virtual address to the map is placed in `bootmem_data_t->node_bootmem_map`. As there is no architecture independent way to detect ‘holes’ in memory, the entire bitmap is initialised to 1, effectively marking all pages allocated. It is up to the architecture dependent code to set the bits of usable pages to 0 although, in reality, the Sparc architecture is the only one which uses this bitmap. In the case of the x86, the function `register_bootmem_low_pages()` reads through the e820 map and calls `free_bootmem()` for each usable page to set the bit to 0 before using `reserve_bootmem()` to reserve the pages needed by the actual bitmap.

5.3 Allocating Memory

The `reserve_bootmem()` function may be used to reserve pages for use by the caller but is very cumbersome to use for general allocations. There are four functions provided for easy allocations on UMA architectures called `alloc_bootmem()`, `alloc_bootmem_low()`, `alloc_bootmem_pages()` and `alloc_bootmem_low_pages()` which are fully described in Table 5.1. All of these macros call `__alloc_bootmem()` with different parameters. The call graph for these functions is shown in in Figure 5.1.

Figure 5.1: Call Graph: `alloc_bootmem()`
Similar functions exist for NUMA which take the node as an additional parameter, as listed in Table 5.2. They are called `alloc_bootmem_node()`, `alloc_bootmem_pages_node()` and `alloc_bootmem_low_pages_node()`. All of these macros call `__alloc_bootmem_node()` with different parameters.

The parameters to either `__alloc_bootmem()` and `__alloc_bootmem_node()` are essentially the same. They are

- **pgdat** This is the node to allocate from. It is omitted in the UMA case as it is assumed to be `contig_page_data`;
- **size** This is the size in bytes of the requested allocation;
- **align** This is the number of bytes that the request should be aligned to. For small allocations, they are aligned to `SMP_CACHE_BYTES`, which on the x86 will align to the L1 hardware cache;
- **goal** This is the preferred starting address to begin allocating from. The “low” functions will start from physical address 0 whereas the others will begin from `MAX_DMA_ADDRESS` which is the maximum address DMA transfers may be made from on this architecture.

The core function for all the allocation APIs is `__alloc_bootmem_core()`. It is a large function but with simple steps that can be broken down. The function linearly scans memory starting from the `goal` address for a block of memory large enough to satisfy the allocation. With the API, this address will either be 0 for DMA-friendly allocations or `MAX_DMA_ADDRESS` otherwise.

The clever part, and the main bulk of the function, deals with deciding if this new allocation can be merged with the previous one. It may be merged if the following conditions hold:

- The page used for the previous allocation (`bootmem_data→pos`) is adjacent to the page found for this allocation;
- The previous page has some free space in it (`bootmem_data→offset != 0`);
- The alignment is less than `PAGE_SIZE`.

Regardless of whether the allocations may be merged or not, the `pos` and `offset` fields will be updated to show the last page used for allocating and how much of the last page was used. If the last page was fully used, the `offset` is 0.

## 5.4 Freeing Memory

In contrast to the allocation functions, only two free functions are provided which are `free_bootmem()` for UMA and `free_bootmem_node()` for NUMA. They both call `free_bootmem_core()` with the only difference being that a `pgdat` is supplied with NUMA.
The core function is relatively simple in comparison to the rest of the allocator. For each full page affected by the free, the corresponding bit in the bitmap is set to 0. If it already was 0, \texttt{BUG()} is called to show a double-free occurred. \texttt{BUG()} is used when an unrecoverable error due to a kernel bug occurs. It terminates the running process and causes a kernel oops which shows a stack trace and debugging information that a developer can use to fix the bug.

An important restriction with the free functions is that only full pages may be freed. It is never recorded when a page is partially allocated so if only partially freed, the full page remains reserved. This is not as major a problem as it appears as the allocations always persist for the lifetime of the system; However, it is still an important restriction for developers during boot time.

### 5.5 Retiring the Boot Memory Allocator

Late in the bootstrapping process, the function \texttt{start_kernel()} is called which knows it is safe to remove the boot allocator and all its associated data structures. Each architecture is required to provide a function \texttt{mem_init()} that is responsible for destroying the boot memory allocator and its associated structures.

---

**Figure 5.2: Call Graph: \texttt{mem_init()}**

The purpose of the function is quite simple. It is responsible for calculating the dimensions of low and high memory and printing out an informational message to the user as well as performing final initialisations of the hardware if necessary. On the x86, the principal function of concern for the VM is the \texttt{free_pages_init()}. 
5.5 Retiring the Boot Memory Allocator

This function first tells the boot memory allocator to retire itself by calling \texttt{free\_all\_bootmem()} for UMA architectures or \texttt{free\_all\_bootmem\_node()} for NUMA. Both call the core function \texttt{free\_all\_bootmem\_core()} with different parameters. The core function is simple in principle and performs the following tasks:

- For all unallocated pages known to the allocator for this node:
  - Clear the \texttt{PG\_reserved} flag in its struct page;
  - Set the count to 1;
  - Call \texttt{__free\_pages()} so that the buddy allocator (discussed next chapter) can build its free lists.

- Free all pages used for the bitmap and give them to the buddy allocator.

At this stage, the buddy allocator now has control of all the pages in low memory which leaves only the high memory pages. After \texttt{free\_all\_bootmem()} returns, it first counts the number of reserved pages for accounting purposes. The remainder of the \texttt{free\_pages\_init()} function is responsible for the high memory pages. However, at this point, it should be clear how the global \texttt{mem\_map} array is allocated, initialised and the pages given to the main allocator. The basic flow used to initialise pages in low memory in a single node system is shown in Figure 5.3.

![Figure 5.3: Initialising mem\_map and the Main Physical Page Allocator](image-url)

Once \texttt{free\_all\_bootmem()} returns, all the pages in \texttt{ZONE\_NORMAL} have been given to the buddy allocator. To initialise the high memory pages, \texttt{free\_pages\_init()} calls \texttt{one\_highpage\_init()} for every page between \texttt{highstart\_pf} and \texttt{highend\_pf}.
one_highpage_init() simple clears the PG_reserved flag, sets the PG_highmem flag, sets the count to 1 and calls __free_pages() to release it to the buddy allocator in the same manner free_all_bootmem_core() did.

At this point, the boot memory allocator is no longer required and the buddy allocator is the main physical page allocator for the system. An interesting feature to note is that not only is the data for the boot allocator removed but also all code that was used to bootstrap the system. All initialisation function that are required only during system start-up are marked __init such as the following:

321 unsigned long __init free_all_bootmem (void)

All of these functions are placed together in the .init section by the linker. On the x86, the function free_initmem() walks through all pages from __init_begin to __init_end and frees up the pages to the buddy allocator. With this method, Linux can free up a considerable amount of memory that is used by bootstrapping code that is no longer required. For example, 27 pages were freed while booting the kernel running on the machine this document is composed on.

5.6 What’s New in 2.6

The boot memory allocator has not changed significantly since 2.4 and is mainly concerned with optimisations and some minor NUMA related modifications. The first optimisation is the addition of a last_success field to the bootmem_data_t struct. As the name suggests, it keeps track of the location of the last successful allocation to reduce search times. If an address is freed before last_success, it will be changed to the freed location.

The second optimisation is also related to the linear search. When searching for a free page, 2.4 test every bit which is expensive. 2.6 instead tests if a block of BITS_PER_LONG is all ones. If it’s not, it will test each of the bits individually in that block. To help the linear search, nodes are ordered in order of their physical addresses by init_bootmem().

The last change is related to NUMA and contiguous architectures. Contiguous architectures now define their own init_bootmem() function and any architecture can optionally define their own reserve_bootmem() function.
Chapter 6

Physical Page Allocation

This chapter describes how physical pages are managed and allocated in Linux. The principal algorithm used is the Binary Buddy Allocator, devised by Knutlon [Kno65] and further described by Knuth [Knu68]. It has been shown to be extremely fast in comparison to other allocators [KB85].

This is an allocation scheme which combines a normal power-of-two allocator with free buffer coalescing [Vah96] and the basic concept behind it is quite simple. Memory is broken up into large blocks of pages where each block is a power of two number of pages. If a block of the desired size is not available, a large block is broken up in half and the two blocks are buddies to each other. One half is used for the allocation and the other is free. The blocks are continuously halved as necessary until a block of the desired size is available. When a block is later freed, the buddy is examined and the two coalesced if it is free.

This chapter will begin with describing how Linux remembers what blocks of memory are free. After that the methods for allocating and freeing pages will be discussed in details. The subsequent section will cover the flags which affect the allocator behaviour and finally the problem of fragmentation and how the allocator handles it will be covered.

6.1 Managing Free Blocks

As stated, the allocator maintains blocks of free pages where each block is a power of two number of pages. The exponent for the power of two sized block is referred to as the order. An array of free_area_t structs are maintained for each order that points to a linked list of blocks of pages that are free as indicated by Figure 6.1.

Hence, the 0th element of the array will point to a list of free page blocks of size $2^0$ or 1 page, the 1st element will be a list of $2^1$ (2) pages up to $2^{\text{MAX\_ORDER}-1}$ number of pages, where the \text{MAX\_ORDER} is currently defined as 10. This eliminates the chance that a larger block will be split to satisfy a request where a smaller block would have sufficed. The page blocks are maintained on a linear linked list via page→list.

Each zone has a free_area_t struct array called free_area[\text{MAX\_ORDER}]. It is
Figure 6.1: Free page block management

declared in `<linux/mm.h>` as follows:

```c
typedef struct free_area_struct {
    struct list_head free_list;
    unsigned long *map;
} free_area_t;
```

The fields in this struct are simply:

- **free_list**: A linked list of free page blocks;
- **map**: A bitmap representing the state of a pair of buddies.

Linux saves memory by only using one bit instead of two to represent each pair of buddies. Each time a buddy is allocated or freed, the bit representing the pair of buddies is toggled so that the bit is zero if the pair of pages are both free or both full and 1 if only one buddy is in use. To toggle the correct bit, the macro `MARK_USED()` in `page_alloc.c` is used which is declared as follows:

```c
#define MARK_USED(index, order, area) 
    __change_bit((index) >> (1+(order)), (area)->map)
```

`index` is the index of the page within the global `mem_map` array. By shifting it right by `1+order` bits, the bit within `map` representing the pair of buddies is revealed.

## 6.2 Allocating Pages

Linux provides a quite sizable API for the allocation of page frames. All of them take a `gfp_mask` as a parameter which is a set of flags that determine how the allocator will behave. The flags are discussed in Section 6.4.

The allocation API functions all use the core function `__alloc_pages()` but the APIs exist so that the correct node and zone will be chosen. Different users will
### Table 6.1: Physical Pages Allocation API

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct page * alloc_page(unsigned int gfp_mask)</td>
<td>Allocate a single page and return a struct address</td>
</tr>
<tr>
<td>struct page * alloc_pages(unsigned int gfp_mask, unsigned int order)</td>
<td>Allocate $2^\text{order}$ number of pages and returns a struct page</td>
</tr>
<tr>
<td>unsigned long get_free_page(unsigned int gfp_mask)</td>
<td>Allocate a single page, zero it and return a virtual address</td>
</tr>
<tr>
<td>unsigned long __get_free_page(unsigned int gfp_mask)</td>
<td>Allocate a single page and return a virtual address</td>
</tr>
<tr>
<td>unsigned long __get_free_pages(unsigned int gfp_mask, unsigned int order)</td>
<td>Allocate $2^\text{order}$ number of pages and return a virtual address</td>
</tr>
<tr>
<td>struct page * __get_dma_pages(unsigned int gfp_mask, unsigned int order)</td>
<td>Allocate $2^\text{order}$ number of pages from the DMA zone and return a struct page</td>
</tr>
</tbody>
</table>

require different zones such as ZONE_DMA for certain device drivers or ZONE_NORMAL for disk buffers and callers should not have to be aware of what node is being used. A full list of page allocation APIs are listed in Table 6.1.

Allocations are always for a specified order, 0 in the case where a single page is required. If a free block cannot be found of the requested order, a higher order block is split into two buddies. One is allocated and the other is placed on the free list for the lower order. Figure 6.2 shows where a $2^4$ block is split and how the buddies are added to the free lists until a block for the process is available.

When the block is later freed, the buddy will be checked. If both are free, they are merged to form a higher order block and placed on the higher free list where its buddy is checked and so on. If the buddy is not free, the freed block is added to the free list at the current order. During these list manipulations, interrupts have to be disabled to prevent an interrupt handler manipulating the lists while a process has them in an inconsistent state. This is achieved by using an interrupt safe spinlock.

The second decision to make is which memory node or pg_data_t to use. Linux uses a node-local allocation policy which aims to use the memory bank associated with the CPU running the page allocating process. Here, the function _alloc_pages() is what is important as this function is different depending on whether the kernel is built for a UMA (function in mm/page_alloc.c) or NUMA (function in mm/numa.c) machine.

Regardless of which API is used,_alloc_pages() in mm/page_alloc.c is the heart of the allocator. This function, which is never called directly, examines the
6.2 Allocating Pages

Figure 6.2: Allocating physical pages

selected zone and checks if it is suitable to allocate from based on the number of available pages. If the zone is not suitable, the allocator may fall back to other zones. The order of zones to fall back on are decided at boot time by the function build_zonelists() but generally ZONE_HIGHMEM will fall back to ZONE_NORMAL and that in turn will fall back to ZONE_DMA. If number of free pages reaches the pages_low watermark, it will wake kswapd to begin freeing up pages from zones and if memory is extremely tight, the caller will do the work of kswapd itself.

Figure 6.3: Call Graph: alloc_pages()

Once the zone has finally been decided on, the function rmqueue() is called to allocate the block of pages or split higher level blocks if one of the appropriate size is not available.
6.3 Free Pages

The API for the freeing of pages is a lot simpler and exists to help remember the order of the block to free as one disadvantage of a buddy allocator is that the caller has to remember the size of the original allocation. The API for freeing is listed in Table 6.2.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void __free_pages(struct page *page, unsigned int order)</td>
<td>Free an order number of pages from the given page</td>
</tr>
<tr>
<td>void __free_page(struct page *page)</td>
<td>Free a single page</td>
</tr>
<tr>
<td>void free_page(void *addr)</td>
<td>Free a page from the given virtual address</td>
</tr>
</tbody>
</table>

Table 6.2: Physical Pages Free API

The principal function for freeing pages is __free_pages_ok() and it should not be called directly. Instead the function __free_pages() is provided which performs simple checks first as indicated in Figure 6.4.

When a buddy is freed, Linux tries to coalesce the buddies together immediately if possible. This is not optimal as the worst case scenario will have many coalitions followed by the immediate splitting of the same blocks [Vah96].

To detect if the buddies can be merged or not, Linux checks the bit corresponding to the affected pair of buddies in free area—map. As one buddy has just been freed by this function, it is obviously known that at least one buddy is free. If the bit in the map is 0 after toggling, we know that the other buddy must also be free because
if the bit is 0, it means both buddies are either both free or both allocated. If both are free, they may be merged.

Calculating the address of the buddy is a well known concept [Knuth]. As the allocations are always in blocks of size \(2^k\), the address of the block, or at least its offset within `zone_mem_map` will also be a power of \(2^k\). The end result is that there will always be at least \(k\) number of zeros to the right of the address. To get the address of the buddy, the \(k\)th bit from the right is examined. If it is 0, then the buddy will have this bit flipped. To get this bit, Linux creates a mask which is calculated as

\[
\text{mask} = (\sim 0 << k)
\]

The mask we are interested in is

\[
\text{imask} = 1 + \sim \text{mask}
\]

Linux takes a shortcut in calculating this by noting that

\[
\text{imask} = -\text{mask} = 1 + \sim \text{mask}
\]

Once the buddy is merged, it is removed for the free list and the newly coalesced pair moves to the next higher order to see if it may also be merged.

### 6.4 Get Free Page (GFP) Flags

A persistent concept through the whole VM is the Get Free Page (GFP) flags. These flags determine how the allocator and `kswapd` will behave for the allocation and freeing of pages. For example, an interrupt handler may not sleep so it will not have the `__GFP_WAIT` flag set as this flag indicates the caller may sleep. There are three sets of GFP flags, all defined in `<linux/mm.h>`.

The first of the three is the set of zone modifiers listed in Table 6.3. These flags indicate that the caller must try to allocate from a particular zone. The reader will note there is not a zone modifier for `ZONE_NORMAL`. This is because the zone modifier flag is used as an offset within an array and 0 implicitly means allocate from `ZONE_NORMAL`.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__GFP_DMA</code></td>
<td>Allocate from <code>ZONE_DMA</code> if possible</td>
</tr>
<tr>
<td><code>__GFP_HIGHMEM</code></td>
<td>Allocate from <code>ZONE_HIGHMEM</code> if possible</td>
</tr>
<tr>
<td><code>GFP_DMA</code></td>
<td>Alias for <code>__GFP_DMA</code></td>
</tr>
</tbody>
</table>

Table 6.3: Low Level GFP Flags Affecting Zone Allocation

The next flags are action modifiers listed in Table 6.4. They change the behaviour of the VM and what the calling process may do. The low level flags on their own are too primitive to be easily used.
### 6.4 Get Free Page (GFP) Flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__GFP_WAIT</td>
<td>Indicates that the caller is not high priority and can sleep or reschedule</td>
</tr>
<tr>
<td>__GFP_HIGH</td>
<td>Used by a high priority or kernel process. Kernel 2.2.x used it to determine if a process could access emergency pools of memory. In 2.4.x kernels, it does not appear to be used.</td>
</tr>
<tr>
<td>__GFP_IO</td>
<td>Indicates that the caller can perform low level IO. In 2.4.x, the main effect this has is determining if <code>try_to_free_buffers()</code> can flush buffers or not. It is used by at least one journaled filesystem.</td>
</tr>
<tr>
<td>__GFP_HIGHIO</td>
<td>Determines that IO can be performed on pages mapped in high memory. Only used in <code>try_to_free_buffers()</code></td>
</tr>
<tr>
<td>__GFP_FS</td>
<td>Indicates if the caller can make calls to the filesystem layer. This is used when the caller is filesystem related, the buffer cache for instance, and wants to avoid recursively calling itself</td>
</tr>
</tbody>
</table>

Table 6.4: Low Level GFP Flags Affecting Allocator behaviour

It is difficult to know what the correct combinations are for each instance so a few high level combinations are defined and listed in Table 6.5. For clarity the __GFP__ is removed from the table combinations so, the __GFP_HIGH__ flag will read as HIGH below. The combinations to form the high level flags are listed in Table 6.6 To help understand this, take GFP_ATOMIC as an example. It has only the __GFP_HIGH__ flag set. This means it is high priority, will use emergency pools (if they exist) but will not sleep, perform IO or access the filesystem. This flag would be used by an interrupt handler for example.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Low Level Flag Combination</th>
</tr>
</thead>
<tbody>
<tr>
<td>GFP_ATOMIC</td>
<td>HIGH</td>
</tr>
<tr>
<td>GFP_NOIO</td>
<td>HIGH</td>
</tr>
<tr>
<td>GFP_NOHIGHIO</td>
<td>HIGH</td>
</tr>
<tr>
<td>GFP_NOFS</td>
<td>HIGH</td>
</tr>
<tr>
<td>GFP_KERNEL</td>
<td>HIGH</td>
</tr>
<tr>
<td>GFP_NFS</td>
<td>HIGH</td>
</tr>
<tr>
<td>GFP_USER</td>
<td>WAIT</td>
</tr>
<tr>
<td>GFP_HIGHUSER</td>
<td>WAIT</td>
</tr>
<tr>
<td>GFP_KSWAPD</td>
<td>WAIT</td>
</tr>
</tbody>
</table>

Table 6.5: Low Level GFP Flag Combinations For High Level Use
### 6.4.1 Process Flags

A process may also set flags in the `task_struct` which affects allocator behaviour. The full list of process flags are defined in `<linux/sched.h>` but only the ones affecting VM behaviour are listed in Table 6.7.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GFP_ATOMIC</td>
<td>This flag is used whenever the caller cannot sleep and must be serviced if at all possible. Any interrupt handler that requires memory must use this flag to avoid sleeping or performing IO. Many subsystems during init will use this system such as buffer_init() and inode_init()</td>
</tr>
<tr>
<td>GFP_NOIO</td>
<td>This is used by callers who are already performing an IO related function. For example, when the loop back device is trying to get a page for a buffer head, it uses this flag to make sure it will not perform some action that would result in more IO. If fact, it appears the flag was introduced specifically to avoid a deadlock in the loopback device.</td>
</tr>
<tr>
<td>GFP_NOHIGHIO</td>
<td>This is only used in one place in alloc_bounce_page() during the creating of a bounce buffer for IO in high memory</td>
</tr>
<tr>
<td>GFP_NOFS</td>
<td>This is only used by the buffer cache and filesystems to make sure they do not recursively call themselves by accident</td>
</tr>
<tr>
<td>GFP_KERNEL</td>
<td>The most liberal of the combined flags. It indicates that the caller is free to do whatever it pleases. Strictly speaking the difference between this flag and GFP_USER is that this could use emergency pools of pages but that is a no-op on 2.4.x kernels</td>
</tr>
<tr>
<td>GFP_USER</td>
<td>Another flag of historical significance. In the 2.2.x series, an allocation was given a LOW, MEDIUM or HIGH priority. If memory was tight, a request with GFP_USER (low) would fail whereas the others would keep trying. Now it has no significance and is not treated any different to GFP_KERNEL.</td>
</tr>
<tr>
<td>GFP_HIGHUSER</td>
<td>This flag indicates that the allocator should allocate from ZONE_HIGHMEM if possible. It is used when the page is allocated on behalf of a user process</td>
</tr>
<tr>
<td>GFP_NFS</td>
<td>This flag is defunct. In the 2.0.x series, this flag determined what the reserved page size was. Normally 20 free pages were reserved. If this flag was set, only 5 would be reserved. Now it is not treated differently anywhere</td>
</tr>
<tr>
<td>GFP_KSWAPD</td>
<td>More historical significance. In reality this is not treated any different to GFP_KERNEL</td>
</tr>
</tbody>
</table>

Table 6.6: High Level GFP Flags Affecting Allocator Behaviour
6.5 Avoiding Fragmentation

One important problem that must be addressed with any allocator is the problem of internal and external fragmentation. External fragmentation is the inability to service a request because the available memory exists only in small blocks. Internal fragmentation is defined as the wasted space where a large block had to be assigned to service a small request. In Linux, external fragmentation is not a serious problem as large requests for contiguous pages are rare and usually `vmalloc()` (see Chapter 7) is sufficient to service the request. The lists of free blocks ensure that large blocks do not have to be split unnecessarily.

Internal fragmentation is the single most serious failing of the binary buddy system. While fragmentation is expected to be in the region of 28% [WJNB95], it has been shown that it can be in the region of 60%, in comparison to just 1% with the first fit allocator [JW98]. It has also been shown that using variations of the buddy system will not help the situation significantly [PN77]. To address this problem, Linux uses a slab allocator [Bon94] to carve up pages into small blocks of memory for allocation [Tan01] which is discussed further in Chapter 8. With this combination of allocators, the kernel can ensure that the amount of memory wasted due to internal fragmentation is kept to a minimum.

6.6 What’s New In 2.6

Allocating Pages The first noticeable difference seems cosmetic at first. The function `alloc_pages()` is now a macro and defined in `<linux/gfp.h>` instead of

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PF_MEMALLOC</td>
<td>This flags the process as a memory allocator. <code>kswapd</code> sets this flag and it is set for any process that is about to be killed by the <code>Out Of Memory (OOM)</code> killer which is discussed in Chapter 13. It tells the buddy allocator to ignore zone watermarks and assign the pages if at all possible</td>
</tr>
<tr>
<td>PF_MEMDIE</td>
<td>This is set by the OOM killer and functions the same as the <code>PF_MEMALLOC</code> flag by telling the page allocator to give pages if at all possible as the process is about to die</td>
</tr>
<tr>
<td>PF_FREE_PAGES</td>
<td>Set when the buddy allocator calls <code>try_to_free_pages()</code> itself to indicate that free pages should be reserved for the calling process in <code>__free_pages_ok()</code> instead of returning to the free lists</td>
</tr>
</tbody>
</table>

Table 6.7: Process Flags Affecting Allocator behaviour
a function defined in `<linux/mm.h>`. The new layout is still very recognisable and the main difference is a subtle but important one. In 2.4, there was specific code dedicated to selecting the correct node to allocate from based on the running CPU but 2.6 removes this distinction between NUMA and UMA architectures.

In 2.6, the function `alloc_pages()` calls `numa_node_id()` to return the logical ID of the node associated with the current running CPU. This NID is passed to `__alloc_pages()` which calls `NODE_DATA()` with the NID as a parameter. On UMA architectures, this will unconditionally result in `contig_page_data` being returned but NUMA architectures instead set up an array which `NODE_DATA()` uses NID as an offset into. In other words, architectures are responsible for setting up a CPU ID to NUMA memory node mapping. This is effectively still a node-local allocation policy as is used in 2.4 but it is a lot more clearly defined.

**Per-CPU Page Lists** The most important addition to the page allocation is the addition of the per-cpu lists, first discussed in Section 2.6.

In 2.4, a page allocation requires an interrupt safe spinlock to be held while the allocation takes place. In 2.6, pages are allocated from a `struct per_cpu_pageset` by `buffered_rmqueue()`. If the low watermark (`per_cpu_pageset->low`) has not been reached, the pages will be allocated from the pageset with no requirement for a spinlock to be held. Once the low watermark is reached, a large number of pages will be allocated in bulk with the interrupt safe spinlock held, added to the per-cpu list and then one returned to the caller.

Higher order allocations, which are relatively rare, still require the interrupt safe spinlock to be held and there will be no delay in the splits or coalescing. With 0 order allocations, splits will be delayed until the low watermark is reached in the per-cpu set and coalescing will be delayed until the high watermark is reached.

However, strictly speaking, this is not a lazy buddy algorithm [BL89]. While pagesets introduce a merging delay for order-0 allocations, it is a side-effect rather than an intended feature and there is no mechanism available to drain the pagesets and merge the buddies. In other words, despite the per-cpu and new accounting code which bulks up the amount of code in `mm/page_alloc.c`, the core of the buddy algorithm remains the same as it was in 2.4.

The implication of this change is straightforward; the number of times the spinlock protecting the buddy lists must be acquired is reduced. Higher order allocations are relatively rare in Linux so the optimisation is for the common case. This change will be noticeable on large number of CPU machines but will make little difference to single CPUs. There are a few issues with pagesets but they are not recognised as a serious problem. The first issue is that high order allocations may fail if the pagesets hold order-0 pages that would normally be merged into higher order contiguous blocks. The second is that an order-0 allocation may fail if memory is low, the current CPU pageset is empty and other CPU’s pagesets are full, as no mechanism exists for reclaiming pages from “remote” pagesets. The last potential problem is that buddies of newly freed pages could exist in other pagesets leading to possible fragmentation problems.
Freeing Pages Two new API functions have been introduced for the freeing of pages called `free_hot_page()` and `free_cold_page()`. Predictably, the determination if the freed pages are placed on the hot or cold lists in the per-cpu pagesets. However, while the `free_cold_page()` is exported and available for use, it is actually never called.

Order-0 page frees from `__free_pages()` and frees resulting from page cache releases by `__page_cache_release()` are placed on the hot list where as higher order allocations are freed immediately with `__free_pages_ok()`. Order-0 are usually related to userspace and are the most common type of allocation and free. By keeping them local to the CPU lock contention will be reduced as most allocations will also be of order-0.

Eventually, lists of pages must be passed to `free_pages_bulk()` or the pageset lists would hold all free pages. This `free_pages_bulk()` function takes a list of page block allocations, the order of each block and the count number of blocks to free from the list. There are two principal cases where this is used. The first is higher order frees passed to `__free_pages_ok()`. In this case, the page block is placed on a linked list, of the specified order and a count of 1. The second case is where the high watermark is reached in the pageset for the running CPU. In this case, the pageset is passed, with an order of 0 and a count of `pageset->batch`.

Once the core function `__free_pages_bulk()` is reached, the mechanisms for freeing pages is to the buddy lists is very similar to 2.4.

GFP Flags There are still only three zones, so the zone modifiers remain the same but three new GFP flags have been added that affect how hard the VM will work, or not work, to satisfy a request. The flags are:

`__GFP_NOFAIL` This flag is used by a caller to indicate that the allocation should never fail and the allocator should keep trying to allocate indefinitely.

`__GFP_REPEAT` This flag is used by a caller to indicate that the request should try to repeat the allocation if it fails. In the current implementation, it behaves the same as `__GFP_NOFAIL` but later the decision might be made to fail after a while.

`__GFP_NORETRY` This flag is almost the opposite of `__GFP_NOFAIL`. It indicates that if the allocation fails it should just return immediately.

At time of writing, they are not heavily used but they have just been introduced and are likely to be used more over time. The `__GFP_REPEAT` flag in particular is likely to be heavily used as blocks of code which implement this flags behaviour exist throughout the kernel.

The next GFP flag that has been introduced is an allocation modifier called `__GFP_COLD` which is used to ensure that cold pages are allocated from the per-cpu lists. From the perspective of the VM, the only user of this flag is the function `page_cache_alloc_cold()` which is mainly used during IO readahead. Usually page allocations will be taken from the hot pages list.
The last new flag is `__GFP_NO_GROW`. This is an internal flag used only by the slab allocator (discussed in Chapter 8) which aliases the flag to `SLAB_NO_GROW`. It is used to indicate when new slabs should never be allocated for a particular cache. In reality, the GFP flag has just been introduced to complement the old `SLAB_NO_GROW` flag which is currently unused in the main kernel.
Chapter 7

Non-Contiguous Memory Allocation

It is preferable when dealing with large amounts of memory to use physically contiguous pages in memory both for cache related and memory access latency reasons. Unfortunately, due to external fragmentation problems with the buddy allocator, this is not always possible. Linux provides a mechanism via `vmalloc()` where non-contiguous physically memory can be used that is contiguous in virtual memory.

An area is reserved in the virtual address space between `VMALLOC_START` and `VMALLOC_END`. The location of `VMALLOC_START` depends on the amount of available physical memory but the region will always be at least `VMALLOC_RESERVE` in size, which on the x86 is 128MiB. The exact size of the region is discussed in Section 4.1.

The page tables in this region are adjusted as necessary to point to physical pages which are allocated with the normal physical page allocator. This means that allocation must be a multiple of the hardware page size. As allocations require altering the kernel page tables, there is a limitation on how much memory can be mapped with `vmalloc()` as only the virtual addresses space between `VMALLOC_START` and `VMALLOC_END` is available. As a result, it is used sparingly in the core kernel. In 2.4.22, it is only used for storing the swap map information (see Chapter 11) and for loading kernel modules into memory.

This small chapter begins with a description of how the kernel tracks which areas in the vmalloc address space are used and how regions are allocated and freed.

7.1 Describing Virtual Memory Areas

The vmalloc address space is managed with a resource map allocator [Vah96]. The struct `vm_struct` is responsible for storing the base, size pairs. It is defined in `<linux/vmalloc.h>` as:

```c
struct vm_struct {
    unsigned long flags;
    void * addr;
    unsigned long size;
    struct vm_struct * next;
};
```
A fully-fledged VMA could have been used but it contains extra information that does not apply to vmalloc areas and would be wasteful. Here is a brief description of the fields in this small struct.

**flags** These set either to **VM_ALLOC**, in the case of use with `vmalloc()` or **VM_IOREMAP** when ioremap is used to map high memory into the kernel virtual address space;

**addr** This is the starting address of the memory block;

**size** This is, predictably enough, the size in bytes;

**next** is a pointer to the next `vm_struct`. They are ordered by address and the list is protected by the `vmlist_lock` lock.

As is clear, the areas are linked together via the `next` field and are ordered by address for simple searches. Each area is separated by at least one page to protect against overruns. This is illustrated by the gaps in Figure 7.1.

![Figure 7.1: vmalloc Address Space](image)

When the kernel wishes to allocate a new area, the `vm_struct` list is searched linearly by the function `get_vm_area()`. Space for the struct is allocated with `kmalloc()`. When the virtual area is used for remapping an area for IO (commonly referred to as `ioremap`), this function will be called directly to map the requested area.

### 7.2 Allocating A Non-Contiguous Area

The functions `vmalloc()`, `vmalloc_dma()` and `vmalloc_32()` are provided to allocate a memory area that is contiguous in virtual address space. They all take a single parameter `size` which is rounded up to the next page alignment. They all return a linear address for the new allocated area.

As is clear from the call graph shown in Figure 7.2, there are two steps to allocating the area. The first step taken by `get_vm_area()` is to find a region large enough to store the request. It searches through a linear linked list of `vm_structs` and returns a new struct describing the allocated region.

The second step is to allocate the necessary PGD entries with `vmalloc_area_pages()`, PMD entries with `alloc_area_pmd()` and PTE entries with `alloc_area_pte()` before finally allocating the page with `alloc_page()`.
The page table updated by `vmalloc()` is not the current process but the reference page table stored at `init_mm->pgd`. This means that a process accessing the `vmalloc` area will cause a page fault exception as its page tables are not pointing to the correct area. There is a special case in the page fault handling code which knows that the fault occurred in the `vmalloc` area and updates the current process page tables using information from the master page table. How the use of `vmalloc()` relates to the
7.3 Freeing A Non-Contiguous Area

The function vfree() is responsible for freeing a virtual area. It linearly searches the list of vm_structs looking for the desired region and then calls vmfree_area_pages() on the region of memory to be freed.

`vmfree_area_pages()` is the exact opposite of vmalloc_area_pages(). It walks the page tables freeing up the page table entries and associated pages for the region.

```
void vfree(void *addr)
    Free a region of memory allocated with vmalloc(), vmalloc_dma() or vmalloc_32()
```

Table 7.2: Non-Contiguous Memory Free API
Non-contiguous memory allocation remains essentially the same in 2.6. The main difference is a slightly different internal API which affects when the pages are allocated. In 2.4, `vmalloc_area_pages()` is responsible for beginning a page table walk and then allocating pages when the PTE is reached in the function `alloc_area_pte()`. In 2.6, all the pages are allocated in advance by `__vmalloc()` and placed in an array which is passed to `map_vm_area()` for insertion into the kernel page tables.

The `get_vm_area()` API has changed very slightly. When called, it behaves the same as previously as it searches the entire `vmalloc` virtual address space for a free area. However, a caller can search just a subset of the `vmalloc` address space by calling `__get_vm_area()` directly and specifying the range. This is only used by the ARM architecture when loading modules.

The last significant change is the introduction of a new interface `vmap()` for the insertion of an array of pages in the `vmalloc` address space and is only used by the sound subsystem core. This interface was backported to 2.4.22 but it is totally unused. It is either the result of an accidental backport or was merged to ease the application of vendor-specific patches that require `vmap()`.
Chapter 8

Slab Allocator

In this chapter, the general-purpose allocator is described. It is a slab allocator which is very similar in many respects to the general kernel allocator used in Solaris [MM01]. Linux’s implementation is heavily based on the first slab allocator paper by Bonwick [Bon94] with many improvements that bear a close resemblance to those described in his later paper [BA01]. We will begin with a quick overview of the allocator followed by a description of the different structures used before giving an in-depth tour of each task the allocator is responsible for.

The basic idea behind the slab allocator is to have caches of commonly used objects kept in an initialised state available for use by the kernel. Without an object-based allocator, the kernel will spend much of its time allocating, initialising and freeing the same object. The slab allocator aims to cache the freed object so that the basic structure is preserved between uses [Bon94].

The slab allocator consists of a variable number of caches that are linked together on a doubly linked circular list called a cache chain. A cache, in the context of the slab allocator, is a manager for a number of objects of a particular type like the mm_struct or fs_cache cache and is managed by a struct kmem_cache_s discussed in detail later. The caches are linked via the next field in the cache struct.

Each cache maintains blocks of contiguous pages in memory called slabs which are carved up into small chunks for the data structures and objects the cache manages. The relationship between these different structures is illustrated in Figure 8.1.

The slab allocator has three principle aims:

- The allocation of small blocks of memory to help eliminate internal fragmentation that would be otherwise caused by the buddy system;
- The caching of commonly used objects so that the system does not waste time allocating, initialising and destroying objects. Benchmarks on Solaris showed excellent speed improvements for allocations with the slab allocator in use [Bon94];
- The better utilisation of hardware cache by aligning objects to the L1 or L2 caches.
To help eliminate internal fragmentation normally caused by a binary buddy allocator, two sets of caches of small memory buffers ranging from $2^5$ (32) bytes to $2^{17}$ (131072) bytes are maintained. One cache set is suitable for use with DMA devices. These caches are called size-N and size-N(DMA) where N is the size of the allocation, and a function `kmalloc()` (see Section 8.4.1) is provided for allocating them. With this, the single greatest problem with the low level page allocator is addressed. The sizes caches are discussed in further detail in Section 8.4.

The second task of the slab allocator is to maintain caches of commonly used objects. For many structures used in the kernel, the time needed to initialise an object is comparable to, or exceeds, the cost of allocating space for it. When a new slab is created, a number of objects are packed into it and initialised using a constructor if available. When an object is freed, it is left in its initialised state so that object allocation will be quick.

The final task of the slab allocator is hardware cache utilization. If there is space left over after objects are packed into a slab, the remaining space is used to color the slab. Slab coloring is a scheme which attempts to have objects in different slabs use different lines in the cache. By placing objects at a different starting offset within the slab, it is likely that objects will use different lines in the CPU cache helping ensure that objects from the same slab cache will be unlikely to flush each other.
With this scheme, space that would otherwise be wasted fulfills a new function. Figure 8.2 shows how a page allocated from the buddy allocator is used to store objects that using coloring to align the objects to the L1 CPU cache.

Figure 8.2: Slab page containing Objects Aligned to L1 CPU Cache

Linux does not attempt to color page allocations based on their physical address [Kes91], or order where objects are placed such as those described for data [GAV95] or code segments [HK97] but the scheme used does help improve cache line usage. Cache colouring is further discussed in Section 8.1.5. On an SMP system, a further step is taken to help cache utilization where each cache has a small array of objects reserved for each CPU. This is discussed further in Section 8.5.

The slab allocator provides the additional option of slab debugging if the option is set at compile time with `CONFIG_SLAB_DEBUG`. Two debugging features are providing called red zoning and object poisoning. With red zoning, a marker is placed at either end of the object. If this mark is disturbed, the allocator knows the object where a buffer overflow occurred and reports it. Poisoning an object will fill it with a predefined bit pattern (defined `0x5A` in `mm/slab.c`) at slab creation and after a free. At allocation, this pattern is examined and if it is changed, the allocator knows that the object was used before it was allocated and flags it.

The small, but powerful, API which the allocator exports is listed in Table 8.1.
### Table 8.1: Slab Allocator API for caches

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kmem_cache_t * kmem_cache_create(const char *name, size_t size, size_t offset, unsigned long flags, void (<em>ctor)(void</em>, kmem_cache_t *, unsigned long), void (<em>dtor)(void</em>, kmem_cache_t *, unsigned long))</td>
<td>Creates a new cache and adds it to the cache chain</td>
</tr>
<tr>
<td>int kmem_cache_reap(int gfp_mask)</td>
<td>Scans at most REAP_SCANLEN caches and selects one for reaping all per-cpu objects and free slabs from. Called when memory is tight</td>
</tr>
<tr>
<td>int kmem_cache_shrink(kmem_cache_t *cachep)</td>
<td>This function will delete all per-cpu objects associated with a cache and delete all slabs in the slabs_free list. It returns the number of pages freed.</td>
</tr>
<tr>
<td>void * kmem_cache_alloc(kmem_cache_t *cachep, int flags)</td>
<td>Allocate a single object from the cache and return it to the caller</td>
</tr>
<tr>
<td>void kmem_cache_free(kmem_cache_t *cachep, void *objp)</td>
<td>Free an object and return it to the cache</td>
</tr>
<tr>
<td>void * kmalloc(size_t size, int flags)</td>
<td>Allocate a block of memory from one of the sizes cache</td>
</tr>
<tr>
<td>void kfree(const void *objp)</td>
<td>Free a block of memory allocated with kmalloc</td>
</tr>
<tr>
<td>int kmem_cache_destroy(kmem_cache_t *cachep)</td>
<td>Destroys all objects in all slabs and frees up all associated memory before removing the cache from the chain</td>
</tr>
</tbody>
</table>

### 8.1 Caches

One cache exists for each type of object that is to be cached. For a full list of caches available on a running system, run `cat /proc/slabinfo`. This file gives some basic information on the caches. An excerpt from the output of this file looks like;
8.1 Caches

slabinfo - version: 1.1 (SMP)

<table>
<thead>
<tr>
<th>Cache</th>
<th>Num-active-objs</th>
<th>Total-objs</th>
<th>Obj-size</th>
<th>Num-active-slabs</th>
<th>Total-slabs</th>
<th>Num-pages-per-slab</th>
</tr>
</thead>
<tbody>
<tr>
<td>kmem_cache</td>
<td>80</td>
<td>80</td>
<td>248</td>
<td>5</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>urp_priv</td>
<td>0</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>tcp_bind_bucket</td>
<td>15</td>
<td>226</td>
<td>32</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>ino_cache</td>
<td>5714</td>
<td>5992</td>
<td>512</td>
<td>856</td>
<td>856</td>
<td>1</td>
</tr>
<tr>
<td>dentry_cache</td>
<td>5160</td>
<td>5160</td>
<td>128</td>
<td>172</td>
<td>172</td>
<td>1</td>
</tr>
<tr>
<td>mm_struct</td>
<td>240</td>
<td>240</td>
<td>160</td>
<td>10</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>vm_area_struct</td>
<td>3911</td>
<td>4480</td>
<td>96</td>
<td>112</td>
<td>112</td>
<td>1</td>
</tr>
<tr>
<td>size-64(DMA)</td>
<td>0</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>size-64</td>
<td>432</td>
<td>1357</td>
<td>64</td>
<td>23</td>
<td>23</td>
<td>1</td>
</tr>
<tr>
<td>size-32(DMA)</td>
<td>17</td>
<td>113</td>
<td>32</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>size-32</td>
<td>850</td>
<td>2712</td>
<td>32</td>
<td>24</td>
<td>24</td>
<td>1</td>
</tr>
</tbody>
</table>

Each of the column fields correspond to a field in the `struct kmem_cache_s` structure. The columns listed in the excerpt above are:

- **Cache-name**: A human readable name such as “tcp_bind_bucket);
- **Num-active-objs**: Number of objects that are in use;
- **Total-objs**: How many objects are available in total including unused;
- **Obj-size**: The size of each object, typically quite small;
- **Num-active-slabs**: Number of slabs containing objects that are active;
- **Total-slabs**: How many slabs in total exist;
- **Num-pages-per-slab**: The pages required to create one slab, typically 1.

If SMP is enabled like in the example excerpt, two more columns will be displayed after a colon. They refer to the per CPU cache described in Section 8.5. The columns are:

- **Limit**: This is the number of free objects the pool can have before half of it is given to the global free pool;
- **Batchcount**: The number of objects allocated for the processor in a block when no objects are free.

To speed allocation and freeing of objects and slabs they are arranged into three lists: `slabs_full`, `slabs_partial` and `slabs_free`. `slabs_full` has all its objects in use. `slabs_partial` has free objects in it and so is a prime candidate for allocation of objects. `slabs_free` has no allocated objects and so is a prime candidate for slab destruction.
8.1.1 Cache Descriptor

All information describing a cache is stored in a struct `kmem_cache_s` declared in `mm/slab.c`. This is an extremely large struct and so will be described in parts.

```
struct kmem_cache_s {
    struct list_head slabs_full;
    struct list_head slabs_partial;
    struct list_head slabs_free;
    unsigned int objsize;
    unsigned int flags;
    unsigned int num;
    spinlock_t spinlock;
    #ifdef CONFIG_SMP
    unsigned int batchcount;
    #endif
    unsigned int gfporder;
    unsigned int gfpflags;
    size_t colour;
    unsigned int colour_off;
    unsigned int colour_next;
    kmem_cache_t *slabp_cache;
    unsigned int growing;
    unsigned int dflags;
    void (*ctor)(void *, kmem_cache_t *, unsigned long);
    void (*dtor)(void *, kmem_cache_t *, unsigned long);
}
```

Most of these fields are of interest when allocating or freeing objects.

- **slabs_***: These are the three lists where the slabs are stored as described in the previous section;
- **objsize**: This is the size of each object packed into the slab;
- **flags**: These flags determine how parts of the allocator will behave when dealing with the cache. See Section 8.1.2;
- **num**: This is the number of objects contained in each slab;
- **spinlock**: A spinlock protecting the structure from concurrent accesses;
- **batchcount**: This is the number of objects that will be allocated in batch for the per-cpu caches as described in the previous section.
This block deals with fields of interest when allocating or freeing slabs from the cache.

**gfporder** This indicates the size of the slab in pages. Each slab consumes $2^{gfporder}$ pages as these are the allocation sizes the buddy allocator provides;

**gfpflags** The GFP flags used when calling the buddy allocator to allocate pages are stored here. See Section 6.4 for a full list;

**colour** Each slab stores objects in different cache lines if possible. Cache colouring will be further discussed in Section 8.1.5;

**colour_off** This is the byte alignment to keep slabs at. For example, slabs for the size-X cachxes are aligned on the L1 cache;

**colour_next** This is the next colour line to use. This value wraps back to 0 when it reaches colour;

**growing** This flag is set to indicate if the cache is growing or not. If it is, it is much less likely this cache will be selected to reap free slabs under memory pressure;

**dflags** These are the dynamic flags which change during the cache lifetime. See Section 8.1.3;

**ctor** A complex object has the option of providing a constructor function to be called to initialise each new object. This is a pointer to that function and may be NULL;

**dtor** This is the complementing object destructor and may be NULL;

**failures** This field is not used anywhere in the code other than being initialised to 0.

These are set during cache creation

**name** This is the human readable name of the cache;

**next** This is the next cache on the cache chain.
8.1.1 Cache Descriptor

**cpudata** This is the per-cpu data and is discussed further in Section 8.5.

```c
233 #if STATS
234    unsigned long num_active;
235    unsigned long num_allocations;
236    unsigned long high_mark;
237    unsigned long grown;
238    unsigned long reaped;
239    unsigned long errors;
240 #ifdef CONFIG_SMP
241    atomic_t allochit;
242    atomic_t allocmiss;
243    atomic_t freehit;
244    atomic_t freemiss;
245 #endif
246 #endif
247 
```

These figures are only available if the `CONFIG_SLAB_DEBUG` option is set during compile time. They are all beancounters and not of general interest. The statistics for `/proc/slabinfo` are calculated when the proc entry is read by another process by examining every slab used by each cache rather than relying on these fields to be available.

- **num_active** The current number of active objects in the cache is stored here;
- **num_allocations** A running total of the number of objects that have been allocated on this cache is stored in this field;
- **high_mark** This is the highest value `num_active` has had to date;
- **grown** This is the number of times `kmem_cache_grow()` has been called;
- **reaped** The number of times this cache has been reaped is kept here;
- **errors** This field is never used;
- **allochit** This is the total number of times an allocation has used the per-cpu cache;
- **allocmiss** To complement `allochit`, this is the number of times an allocation has missed the per-cpu cache;
- **freehit** This is the number of times a free was placed on a per-cpu cache;
- **freemiss** This is the number of times an object was freed and placed on the global pool.
8.1.2 Cache Static Flags

A number of flags are set at cache creation time that remain the same for the lifetime of the cache. They affect how the slab is structured and how objects are stored within it. All the flags are stored in a bitmask in the flags field of the cache descriptor. The full list of possible flags that may be used are declared in `<linux/slab.h>`.

There are three principle sets. The first set is internal flags which are set only by the slab allocator and are listed in Table 8.2. The only relevant flag in the set is the CFGS_OFF_SLAB flag which determines where the slab descriptor is stored.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFGS_OFF_SLAB</td>
<td>Indicates that the slab managers for this cache are kept off-slab. This is discussed further in Section 8.2.1</td>
</tr>
<tr>
<td>CFLGS_OPTIMIZE</td>
<td>This flag is only ever set and never used</td>
</tr>
</tbody>
</table>

Table 8.2: Internal cache static flags

The second set are set by the cache creator and they determine how the allocator treats the slab and how objects are stored. They are listed in Table 8.3.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLAB_HWCACHE_ALIGN</td>
<td>Align the objects to the L1 CPU cache</td>
</tr>
<tr>
<td>SLAB_MUST_HWCACHE_ALIGN</td>
<td>Force alignment to the L1 CPU cache even if it is very wasteful or slab debugging is enabled</td>
</tr>
<tr>
<td>SLAB_NO_REAP</td>
<td>Never reap slabs in this cache</td>
</tr>
<tr>
<td>SLAB_CACHE_DMA</td>
<td>Allocate slabs with memory from ZONE_DMA</td>
</tr>
</tbody>
</table>

Table 8.3: Cache static flags set by caller

The last flags are only available if the compile option CONFIG_SLAB_DEBUG is set. They determine what additional checks will be made to slabs and objects and are primarily of interest only when new caches are being developed.

To prevent callers using the wrong flags a CREATE_MASK is defined in `mm/slab.c` consisting of all the allowable flags. When a cache is being created, the requested flags are compared against the CREATE_MASK and reported as a bug if invalid flags are used.

8.1.3 Cache Dynamic Flags

The dflags field has only one flag, DFLGS_GROWN, but it is important. The flag is set during `kmem_cache_grow()` so that `kmem_cache_reap()` will be unlikely to choose
### 8.1.4 Cache Allocation Flags

These flags correspond to the GFP page flag options for allocating pages for slabs. Callers sometimes call with either SLAB_* or GFP_* flags, but they really should use only SLAB_* flags. They correspond directly to the flags described in Section 6.4 so will not be discussed in detail here. It is presumed the existence of these flags are for clarity and in case the slab allocator needed to behave differently in response to a particular flag but in reality, there is no difference.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLAB_ATOMIC</td>
<td>Equivalent to GFP_ATOMIC</td>
</tr>
<tr>
<td>SLAB_DMA</td>
<td>Equivalent to GFP_DMA</td>
</tr>
<tr>
<td>SLAB_KERNEL</td>
<td>Equivalent to GFP_KERNEL</td>
</tr>
<tr>
<td>SLAB_NFS</td>
<td>Equivalent to GFP_NFS</td>
</tr>
<tr>
<td>SLAB_NOFS</td>
<td>Equivalent to GFP_NOFS</td>
</tr>
<tr>
<td>SLAB_NOHIGHIO</td>
<td>Equivalent to GFP_NOHIGHIO</td>
</tr>
<tr>
<td>SLAB_NOIO</td>
<td>Equivalent to GFP_NOIO</td>
</tr>
<tr>
<td>SLAB_USER</td>
<td>Equivalent to GFP_USER</td>
</tr>
</tbody>
</table>

Table 8.5: Cache Allocation Flags

A very small number of flags may be passed to constructor and destructor functions which are listed in Table 8.6.

### 8.1.5 Cache Colouring

To utilise hardware cache better, the slab allocator will offset objects in different slabs by different amounts depending on the amount of space left over in the slab. The offset is in units of \texttt{BYTES\_PER\_WORD} unless \texttt{SLAB\_HWCACHE\_ALIGN} is set in which
8.1.6 Cache Creation

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLABCTOR_CONSTRUCTOR</td>
<td>Set if the function is being called as a constructor for caches which use the same function as a constructor and a destructor</td>
</tr>
<tr>
<td>SLABCTOR_ATOMIC</td>
<td>Indicates that the constructor may not sleep</td>
</tr>
<tr>
<td>SLABCTOR_VERIFY</td>
<td>Indicates that the constructor should just verify the object is initialised correctly</td>
</tr>
</tbody>
</table>

Table 8.6: Cache Constructor Flags

case it is aligned to blocks of L1_CACHE_BYTES for alignment to the L1 hardware cache.

During cache creation, it is calculated how many objects can fit on a slab (see Section 8.2.7) and how many bytes would be wasted. Based on wastage, two figures are calculated for the cache descriptor

\text{colour} \quad \text{This is the number of different offsets that can be used;}

\text{colour\_off} \quad \text{This is the multiple to offset each objects by in the slab.}

With the objects offset, they will use different lines on the associative hardware cache. Therefore, objects from slabs are less likely to overwrite each other in memory.

The result of this is best explained by an example. Let us say that s\_mem (the address of the first object) on the slab is 0 for convenience, that 100 bytes are wasted on the slab and alignment is to be at 32 bytes to the L1 Hardware Cache on a Pentium II.

In this scenario, the first slab created will have its objects start at 0. The second will start at 32, the third at 64, the fourth at 96 and the fifth will start back at 0. With this, objects from each of the slabs will not hit the same hardware cache line on the CPU. The value of colour is 3 and colour\_off is 32.

8.1.6 Cache Creation

The function kmem_cache_create() is responsible for creating new caches and adding them to the cache chain. The tasks that are taken to create a cache are

- Perform basic sanity checks for bad usage;
- Perform debugging checks if \text{CONFIG\_SLAB\_DEBUG} is set;
- Allocate a kmem_cache_t from the cache_cache slab cache;
- Align the object size to the word size;
- Calculate how many objects will fit on a slab;
- Align the object size to the hardware cache;
8.1.7 Cache Reaping

- Calculate colour offsets;
- Initialise remaining fields in cache descriptor;
- Add the new cache to the cache chain.

Figure 8.3 shows the call graph relevant to the creation of a cache; each function is fully described in the Code Commentary.

8.1.7 Cache Reaping

When a slab is freed, it is placed on the slabs_free list for future use. Caches do not automatically shrink themselves so when kswapd notices that memory is tight, it calls kmem_cache_reap() to free some memory. This function is responsible for selecting a cache that will be required to shrink its memory usage. It is worth noting that cache reaping does not take into account what memory node or zone is under pressure. This means that with a NUMA or high memory machine, it is possible the kernel will spend a lot of time freeing memory from regions that are under no memory pressure but this is not a problem for architectures like the x86 which has only one bank of memory.

The call graph in Figure 8.4 is deceptively simple as the task of selecting the proper cache to reap is quite long. In the event that there are numerous caches in the system, only REAP_SCANLEN (currently defined as 10) caches are examined in each call. The last cache to be scanned is stored in the variable clock_searchp so as not to examine the same caches repeatedly. For each scanned cache, the reaper does the following:

- Check flags for SLAB_NO_REAP and skip if set;
- If the cache is growing, skip it;
- if the cache has grown recently or is current growing, DFLGS_GROWN will be set. If this flag is set, the slab is skipped but the flag is cleared so it will be a reap candidate the next time;
8.1.8 Cache Shrinking

When a cache is selected to shrink itself, the steps it takes are simple and brutal

- Delete all objects in the per CPU caches;
- Delete all slabs from slabs_free unless the growing flag gets set.

Linux is nothing, if not subtle.

Two varieties of shrink functions are provided with confusingly similar names. kmem_cache_shrink() removes all slabs from slabs_free and returns the number of pages freed as a result. This is the principal function exported for use by the slab allocator users.

The second function __kmem_cache_shrink() frees all slabs from slabs_free and then verifies that slabs_partial and slabs_full are empty. This is for internal use only and is important during cache destruction when it doesn’t matter how many pages are freed, just that the cache is empty.
When a module is unloaded, it is responsible for destroying any cache with the function `kmem_cache_destroy()`. It is important that the cache is properly destroyed as two caches of the same human-readable name are not allowed to exist. Core kernel code often does not bother to destroy its caches as their existence persists for the life of the system. The steps taken to destroy a cache are

- Delete the cache from the cache chain;
- Shrink the cache to delete all slabs;
- Free any per CPU caches (`kfree()`);
- Delete the cache descriptor from the `cache_cache`. 
8.2 Slabs

This section will describe how a slab is structured and managed. The struct which describes it is much simpler than the cache descriptor, but how the slab is arranged is considerably more complex. It is declared as follows:

```c
typedef struct slab_s {
    struct list_head list;
    unsigned long colouroff;
    void *s_mem;
    unsigned int inuse;
    kmem_bufctl_t free;
} slab_t;
```

The fields in this simple struct are as follows:

- **list**: This is the linked list the slab belongs to. This will be one of slab_full, slab_partial or slab_free from the cache manager;
- **colouroff**: This is the colour offset from the base address of the first object within the slab. The address of the first object is s_mem + colouroff;
- **s_mem**: This gives the starting address of the first object within the slab;
- **inuse**: This gives the number of active objects in the slab;
- **free**: This is an array of bufctl_s used for storing locations of free objects. See Section 8.2.3 for further details.

The reader will note that given the slab manager or an object within the slab, there does not appear to be an obvious way to determine what slab or cache they belong to. This is addressed by using the list field in the struct page that makes up the cache. `SET_PAGE_CACHE()` and `SET_PAGE_SLAB()` use the next and prev fields on the page→list to track what cache and slab an object belongs to. To get the descriptors from the page, the macros `GET_PAGE_CACHE()` and `GET_PAGE_SLAB()` are available. This set of relationships is illustrated in Figure 8.8.

The last issue is where the slab management struct is kept. Slab managers are kept either on (`CFLGS_OFF_SLAB` set in the static flags) or off-slab. Where they
8.2.1 Storing the Slab Descriptor

If the objects are larger than a threshold (512 bytes on x86), CFGS_OFF_SLAB is set in the cache flags and the slab descriptor is kept off-slab in one of the sizes cache (see Section 8.4). The selected sizes cache is large enough to contain the struct slab_t and kmem_cache_slabmgmt() allocates from it as necessary. This limits the number of objects that can be stored on the slab because there is limited space for the bufctls but that is unimportant as the objects are large and so there should not be many stored in a single slab.

Alternatively, the slab manager is reserved at the beginning of the slab. When stored on-slab, enough space is kept at the beginning of the slab to store both the slab_t and the kmem_bufctl_t which is an array of unsigned integers. The array is responsible for tracking the index of the next free object that is available for use which is discussed further in Section 8.2.3. The actual objects are stored after the kmem_bufctl_t array.

Figure 8.9 should help clarify what a slab with the descriptor on-slab looks like and Figure 8.10 illustrates how a cache uses a sizes cache to store the slab descriptor when the descriptor is kept off-slab.
8.2.2 Slab Creation

At this point, we have seen how the cache is created, but on creation, it is an empty cache with empty lists for its slab_full, slab_partial and slabs_free. New slabs are allocated to a cache by calling the function kmem_cache_grow(). This is frequently called “cache growing” and occurs when no objects are left in the slabs_partial list and there are no slabs in slabs_free. The tasks it fulfills are

- Perform basic sanity checks to guard against bad usage;
- Calculate colour offset for objects in this slab;
- Allocate memory for slab and acquire a slab descriptor;
- Link the pages used for the slab to the slab and cache descriptors described in Section 8.2;
- Initialise objects in the slab;
- Add the slab to the cache.

8.2.3 Tracking Free Objects

The slab allocator has got to have a quick and simple means of tracking where free objects are on the partially filled slabs. It achieves this by using an array of unsigned integers called kmem_bufctl_t that is associated with each slab manager as obviously it is up to the slab manager to know where its free objects are.
Historically, and according to the paper describing the slab allocator [Bon94], \texttt{kmem_bufctl_t} was a linked list of objects. In Linux 2.2.x, this struct was a union of three items, a pointer to the next free object, a pointer to the slab manager and a pointer to the object. Which it was depended on the state of the object.

Today, the slab and cache an object belongs to is determined by the \texttt{struct page} and \texttt{kmem_bufctl_t} is simply an integer array of object indices. The number of elements in the array is the same as the number of objects on the slab.

\begin{verbatim}
141 typedef unsigned int kmem_bufctl_t;

As the array is kept after the slab descriptor and there is no pointer to the first element directly, a helper macro \texttt{slab_bufctl()} is provided.

163 #define slab_bufctl(slabp) \
This seemingly cryptic macro is quite simple when broken down. The parameter
slabp is a pointer to the slab manager. The expression ((slab_t*)slabp)+1 casts
slabp to a slab_t struct and adds 1 to it. This will give a pointer to a slab_t
which is actually the beginning of the kmem_bufctl_t array. (kmem_bufctl_t *)
casts the slab_t pointer to the required type. The results in blocks of code that
contain slab_bufctl(slabp)[i]. Translated, that says “take a pointer to a slab
descriptor, offset it with slab_bufctl() to the beginning of the kmem_bufctl_t
array and return the ith element of the array”.

The index to the next free object in the slab is stored in slab_t→free eliminating
the need for a linked list to track free objects. When objects are allocated or
freed, this pointer is updated based on information in the kmem_bufctl_t array.

8.2.4 Initialising the kmem_bufctl_t Array

When a cache is grown, all the objects and the kmem_bufctl_t array on the slab
are initialised. The array is filled with the index of each object beginning with 1
and ending with the marker BUFCTL_END. For a slab with 5 objects, the elements
of the array would look like Figure 8.12.

Figure 8.12: Initialised kmem_bufctl_t Array

The value 0 is stored in slab_t→free as the 0th object is the first free object to
be used. The idea is that for a given object n, the index of the next free object will
be stored in kmem_bufctl_t[n]. Looking at the array above, the next object free
after 0 is 1. After 1, there are two and so on. As the array is used, this arrangement
will make the array act as a LIFO for free objects.

8.2.5 Finding the Next Free Object

When allocating an object, kmem_cache_alloc() performs the “real” work of up-
dating the kmem_bufctl_t() array by calling kmem_cache_alloc_one_tail(). The
field slab_t→free has the index of the first free object. The index of the next free
object is at kmem_bufctl_t[slab_t→free]. In code terms, this looks like

1253  objp = slabp->s_mem + slabp->free*cachep->objsize;
1254  slabp->free = slab_bufctl(slabp)[slabp->free];

The field slabp→s_mem is a pointer to the first object on the slab. slabp→free
is the index of the object to allocate and it has to be multiplied by the size of an
object.
8.2.6 Updating \texttt{kmem\_bufctl\_t}

The index of the next free object is stored at \texttt{kmem\_bufctl\_t[slabp\rightarrow\text{free}]}.
There is no pointer directly to the array hence the helper macro \texttt{slab\_bufctl()} is used. Note that the \texttt{kmem\_bufctl\_t} array is not changed during allocations but that the elements that are unallocated are unreachable. For example, after two allocations, index 0 and 1 of the \texttt{kmem\_bufctl\_t} array are not pointed to by any other element.

8.2.6 Updating \texttt{kmem\_bufctl\_t}

The \texttt{kmem\_bufctl\_t} list is only updated when an object is freed in the function \texttt{kmem\_cache\_free\_one()}. The array is updated with this block of code:

\begin{verbatim}
unsigned int objnr = (objp-slabp->s_mem)/cachep->objsizer;
slab_bufctl(slabp)[objnr] = slabp->free;
slabp->free = objnr;
\end{verbatim}

The pointer \texttt{objp} is the object about to be freed and \texttt{objnr} is its index. \texttt{kmem\_bufctl\_t[objnr]} is updated to point to the current value of \texttt{slabp\rightarrow\text{free}}, effectively placing the object pointed to by \texttt{free} on the pseudo linked list. \texttt{slabp\rightarrow\text{free}} is updated to the object being freed so that it will be the next one allocated.

8.2.7 Calculating the Number of Objects on a Slab

During cache creation, the function \texttt{kmem\_cache\_estimate()} is called to calculate how many objects may be stored on a single slab taking into account whether the slab descriptor must be stored on-slab or off-slab and the size of each \texttt{kmem\_bufctl\_t} needed to track if an object is free or not. It returns the number of objects that may be stored and how many bytes are wasted. The number of wasted bytes is important if cache colouring is to be used.

The calculation is quite basic and takes the following steps

- Initialise \texttt{wastage} to be the total size of the slab i.e. PAGE\_\texttt{SIZE}^{\texttt{gfp}\_order};
- Subtract the amount of space required to store the slab descriptor;
- Count up the number of objects that may be stored. Include the size of the \texttt{kmem\_bufctl\_t} if the slab descriptor is stored on the slab. Keep increasing the size of \texttt{i} until the slab is filled;
- Return the number of objects and bytes wasted.
8.2.8 Slab Destroying

When a cache is being shrunk or destroyed, the slabs will be deleted. As the objects may have destructors, these must be called, so the tasks of this function are:

- If available, call the destructor for every object in the slab;
- If debugging is enabled, check the red marking and poison pattern;
- Free the pages the slab uses.

The call graph at Figure 8.13 is very simple.

![Call Graph: kmem_slab_destroy()](image)

8.3 Objects

This section will cover how objects are managed. At this point, most of the really hard work has been completed by either the cache or slab managers.

8.3.1 Initialising Objects in a Slab

When a slab is created, all the objects in it are put in an initialised state. If a constructor is available, it is called for each object and it is expected that objects are left in an initialised state upon free. Conceptually the initialisation is very simple, cycle through all objects and call the constructor and initialise the kmem_bufctl for it. The function kmem_cache_init_objs() is responsible for initialising the objects.

8.3.2 Object Allocation

The function kmem_cache_alloc() is responsible for allocating one object to the caller which behaves slightly different in the UP and SMP cases. Figure 8.14 shows the basic call graph that is used to allocate an object in the SMP case.

There are four basic steps. The first step (kmem_cache_alloc_head()) covers basic checking to make sure the allocation is allowable. The second step is to select which slabs list to allocate from. This will be one of slabs_partial or slabs_free. If there are no slabs in slabs_free, the cache is grown (see Section 8.2.2) to create
8.3.3 Object Freeing

Figure 8.14: Call Graph: kmem_cache_alloc()

a new slab in slabs_free. The final step is to allocate the object from the selected slab.

The SMP case takes one further step. Before allocating one object, it will check to see if there is one available from the per-CPU cache and will use it if there is. If there is not, it will allocate batchcount number of objects in bulk and place them in its per-cpu cache. See Section 8.5 for more information on the per-cpu caches.

8.3.3 Object Freeing

kmem_cache_free() is used to free objects and it has a relatively simple task. Just like kmem_cache_alloc(), it behaves differently in the UP and SMP cases. The principal difference between the two cases is that in the UP case, the object is returned directly to the slab but with the SMP case, the object is returned to the per-cpu cache. In both cases, the destructor for the object will be called if one is available. The destructor is responsible for returning the object to the initialised state.

Figure 8.15: Call Graph: kmem_cache_free()
8.4 Sizes Cache

Linux keeps two sets of caches for small memory allocations for which the physical page allocator is unsuitable. One set is for use with DMA and the other is suitable for normal use. The human readable names for these caches are size-N cache and size-N(DMA) cache which are viewable from /proc/slabinfo. Information for each sized cache is stored in a struct cache_sizes, typedefed to cache_sizes_t, which is defined in mm/slab.c as:

```c
typedef struct cache_sizes {
    size_t cs_size;
    kmem_cache_t *cs_cachep;
    kmem_cache_t *cs_dmacachep;
} cache_sizes_t;
```

The fields in this struct are described as follows:

- **cs_size** The size of the memory block;
- **cs_cachep** The cache of blocks for normal memory use;
- **cs_dmacachep** The cache of blocks for use with DMA.

As there are a limited number of these caches that exist, a static array called cache_sizes is initialised at compile time beginning with 32 bytes on a 4KiB machine and 64 for greater page sizes.

```c
static cache_sizes_t cache_sizes[] = {
    #if PAGE_SIZE == 4096
    { 32, NULL, NULL},
    #endif
    { 64, NULL, NULL},
    { 128, NULL, NULL},
    { 256, NULL, NULL},
    { 512, NULL, NULL},
    { 1024, NULL, NULL},
    { 2048, NULL, NULL},
    { 4096, NULL, NULL},
    { 8192, NULL, NULL},
    { 16384, NULL, NULL},
    { 32768, NULL, NULL},
    { 65536, NULL, NULL},
    {131072, NULL, NULL},
    { 0, NULL, NULL}

    #endif
};
```

As is obvious, this is a static array that is zero terminated consisting of buffers of succeeding powers of 2 from $2^5$ to $2^{17}$. An array now exists that describes each sized cache which must be initialised with caches at system startup.
8.4.1 kmalloc()

With the existence of the sizes cache, the slab allocator is able to offer a new allocator function, kmalloc() for use when small memory buffers are required. When a request is received, the appropriate sizes cache is selected and an object assigned from it. The call graph on Figure 8.16 is therefore very simple as all the hard work is in cache allocation.

Figure 8.16: Call Graph: kmalloc()

8.4.2 kfree()

Just as there is a kmalloc() function to allocate small memory objects for use, there is a kfree() for freeing it. As with kmalloc(), the real work takes place during object freeing (See Section 8.3.3) so the call graph in Figure 8.17 is very simple.

Figure 8.17: Call Graph: kfree()

8.5 Per-CPU Object Cache

One of the tasks the slab allocator is dedicated to is improved hardware cache utilization. An aim of high performance computing [CS98] in general is to use data on the same CPU for as long as possible. Linux achieves this by trying to keep objects in the same CPU cache with a Per-CPU object cache, simply called a cpucache for each CPU in the system.
When allocating or freeing objects, they are placed in the cpucache. When there are no objects free, a batch of objects is placed into the pool. When the pool gets too large, half of them are removed and placed in the global cache. This way the hardware cache will be used for as long as possible on the same CPU.

The second major benefit of this method is that spinlocks do not have to be held when accessing the CPU pool as we are guaranteed another CPU won’t access the local data. This is important because without the caches, the spinlock would have to be acquired for every allocation and free which is unnecessarily expensive.

### 8.5.1 Describing the Per-CPU Object Cache

Each cache descriptor has a pointer to an array of cpucaches, described in the cache descriptor as

```c
231 cpucache_t *cpudata[NR_CPUS];
```

This structure is very simple

```c
typedef struct cpucache_s {
    unsigned int avail;
    unsigned int limit;
} cpucache_t;
```

The fields are as follows:

- **avail** This is the number of free objects available on this cpucache;
- **limit** This is the total number of free objects that can exist.

A helper macro `cc_data()` is provided to give the cpucache for a given cache and processor. It is defined as

```c
180 #define cc_data(cachep) \\
181  ((cachep)->cpudata[smp_processor_id()])
```

This will take a given cache descriptor (cachep) and return a pointer from the cpucache array (cpudata). The index needed is the ID of the current processor, `smp_processor_id()`.

Pointers to objects on the cpucache are placed immediately after the `cpucache_t` struct. This is very similar to how objects are stored after a slab descriptor.
8.5.2 Adding/Removing Objects from the Per-CPU Cache

To prevent fragmentation, objects are always added or removed from the end of the array. To add an object (obj) to the CPU cache (cc), the following block of code is used

\[ cc\_entry(cc)[cc->avail++] = obj; \]

To remove an object

\[ obj = cc\_entry(cc)[--cc->avail]; \]

There is a helper macro called \( cc\_entry() \) which gives a pointer to the first object in the cache. It is defined as

\[
\begin{align*}
# define cc_entry(cpucache) \\
& ((void **)(((cpucache_t*)(cpucache))+1))
\end{align*}
\]

This takes a pointer to a cache, increments the value by the size of the \( cpucache\_t \) descriptor giving the first object in the cache.

8.5.3 Enabling Per-CPU Caches

When a cache is created, its CPU cache has to be enabled and memory allocated for it using \( kmalloc() \). The function \( enable\_cpucache() \) is responsible for deciding what size to make the cache and calling \( kmem\_tune\_cpucache() \) to allocate memory for it.

Obviously a CPU cache cannot exist until after the various sizes caches have been enabled so a global variable \( g\_cpucache\_up \) is used to prevent CPU caches being enabled prematurely. The function \( enable\_all\_cpucaches() \) cycles through all caches in the cache chain and enables their cpucache.

Once the CPU cache has been setup, it can be accessed without locking as a CPU will never access the wrong cpucache so it is guaranteed safe access to it.

8.5.4 Updating Per-CPU Information

When the per-cpu caches have been created or changed, each CPU is signalled via an IPI. It is not sufficient to change all the values in the cache descriptor as that would lead to cache coherency issues and spinlocks would have to used to protect the CPU caches. Instead a \( ccupdate\_t \) struct is populated with all the information each CPU needs and each CPU swaps the new data with the old information in the cache descriptor. The struct for storing the new cpucache information is defined as follows

\[
\begin{align*}
typedef struct ccupdate_struct_s \\
869 \{
\end{align*}
\]
8.5.5 Draining a Per-CPU Cache

When a cache is being shrunk, its first step is to drain the cpucaches of any objects they might have by calling `drain_cpu_caches()`. This is so that the slab allocator will have a clearer view of what slabs can be freed or not. This is important because if just one object in a slab is placed in a per-cpu cache, that whole slab cannot be freed. If the system is tight on memory, saving a few milliseconds on allocations has a low priority.

8.6 Slab Allocator Initialisation

Here we will describe how the slab allocator initialises itself. When the slab allocator creates a new cache, it allocates the `kmem_cache_t` from the `cache_cache` or `kmem_cache` cache. This is an obvious chicken and egg problem so the `cache_cache` has to be statically initialised as

```c
357 static kmem_cache_t cache_cache = {
358   slabs_full: LIST_HEAD_INIT(cache_cache.slabs_full),
359   slabs_partial: LIST_HEAD_INIT(cache_cache.slabs_partial),
360   slabs_free: LIST_HEAD_INIT(cache_cache.slabs_free),
361   objsize: sizeof(kmem_cache_t),
362   flags: SLAB_NO_REAP,
363   spinlock: SPIN_LOCK_UNLOCKED,
364   colour_off: L1_CACHE_BYTES,
365   name: "kmem_cache",
366   };
```

This code statically initialised the `kmem_cache_t` struct as follows:

358-360 Initialise the three lists as empty lists;
361 The size of each object is the size of a cache descriptor;
362 The creation and deleting of caches is extremely rare so do not consider it for reaping ever;
Initialise the spinlock unlocked;
Align the objects to the L1 cache;
Record the human readable name.

That statically defines all the fields that can be calculated at compile time. To initialise the rest of the struct, kmem_cache_init() is called from start_kernel().

### 8.7 Interfacing with the Buddy Allocator

The slab allocator does not come with pages attached, it must ask the physical page allocator for its pages. Two APIs are provided for this task called kmem_getpages() and kmem_freepages(). They are basically wrappers around the buddy allocators API so that slab flags will be taken into account for allocations. For allocations, the default flags are taken from cachep—gfpflags and the order is taken from cachep—gfporder where cachep is the cache requesting the pages. When freeing the pages, PageClearSlab() will be called for every page being freed before calling free_pages().

### 8.8 Whats New in 2.6

The first obvious change is that the version of the /proc/slabinfo format has changed from 1.1 to 2.0 and is a lot friendlier to read. The most helpful change is that the fields now have a header negating the need to memorise what each column means.

The principal algorithms and ideas remain the same and there is no major algorithm shakeups but the implementation is quite different. Particularly, there is a greater emphasis on the use of per-cpu objects and the avoidance of locking. Secondly, there is a lot more debugging code mixed in so keep an eye out for #ifdef DEBUG blocks of code as they can be ignored when reading the code first. Lastly, some changes are purely cosmetic with function name changes but very similar behavior. For example, kmem_cache_estimate() is now called cache_estimate() even though they are identical in every other respect.

**Cache descriptor** The changes to the kmem_cache_s are minimal. First, the elements are reordered to have commonly used elements, such as the per-cpu related data, at the beginning of the struct (see Section 3.9 to for the reasoning). Secondly, the slab lists (e.g. slabs_full) and statistics related to them have been moved to a separate struct kmem_list3. Comments and the unusual use of macros indicate that there is a plan to make the structure per-node.
Cache Static Flags The flags in 2.4 still exist and their usage is the same.\texttt{CFLGS\_OPTIMIZE} no longer exists but its usage in 2.4 was non-existent. Two new flags have been introduced which are:

\begin{itemize}
  \item \texttt{SLAB\_STORE\_USER} This is a debugging only flag for recording the function that freed an object. If the object is used after it was freed, the poison bytes will not match and a kernel error message will be displayed. As the last function to use the object is known, it can simplify debugging.
  \item \texttt{SLAB\_RECLAIM\_ACCOUNT} This flag is set for caches with objects that are easily reclaimable such as inode caches. A counter is maintained in a variable called \texttt{slab\_reclaim\_pages} to record how many pages are used in slabs allocated to these caches. This counter is later used in \texttt{vm\_enough\_memory()} to help determine if the system is truly out of memory.
\end{itemize}

Cache Reaping This is one of the most interesting changes made to the slab allocator. \texttt{kmem\_cache\_reap()} no longer exists as it is very indiscriminate in how it shrinks caches when the cache user could have made a far superior selection. Users of caches can now register a “shrink cache” callback with \texttt{set\_shrinker()} for the intelligent aging and shrinking of slabs. This simple function populates a \texttt{struct shrinker} with a pointer to the callback and a “seeks” weight which indicates how difficult it is to recreate an object before placing it in a linked list called \texttt{shrinker\_list}.

During page reclaim, the function \texttt{shrink\_slab()} is called which steps through the full \texttt{shrinker\_list} and calls each shrinker callback twice. The first call passes 0 as a parameter which indicates that the callback should return how many pages it expects it could free if it was called properly. A basic heuristic is applied to determine if it is worth the cost of using the callback. If it is, it is called a second time with a parameter indicating how many objects to free.

How this mechanism accounts for the number of pages is a little tricky. Each task struct has a field called \texttt{reclaim\_state}. When the slab allocator frees pages, this field is updated with the number of pages that is freed. Before calling \texttt{shrink\_slab()}, this field is set to 0 and then read again after \texttt{shrink\_cache} returns to determine how many pages were freed.

Other changes The rest of the changes are essentially cosmetic. For example, the slab descriptor is now called \texttt{struct slab} instead of \texttt{slab\_t} which is consistent with the general trend of moving away from typedefs. Per-cpu caches remain essentially the same except the structs and APIs have new names. The same type of points applies to most of the rest of the 2.6 slab allocator implementation.
Chapter 9

High Memory Management

The kernel may only directly address memory for which it has set up a page table entry. In the most common case, the user/kernel address space split of 3GiB/1GiB implies that at best only 896MiB of memory may be directly accessed at any given time on a 32-bit machine as explained in Section 4.1. On 64-bit hardware, this is not really an issue as there is more than enough virtual address space. It is highly unlikely there will be machines running 2.4 kernels with more than terabytes of RAM.

There are many high end 32-bit machines that have more than 1GiB of memory and the inconveniently located memory cannot be simply ignored. The solution Linux uses is to temporarily map pages from high memory into the lower page tables. This will be discussed in Section 9.2.

High memory and IO have a related problem which must be addressed, as not all devices are able to address high memory or all the memory available to the CPU. This may be the case if the CPU has PAE extensions enabled, the device is limited to addresses the size of a signed 32-bit integer (2GiB) or a 32-bit device is being used on a 64-bit architecture. Asking the device to write to memory will fail at best and possibly disrupt the kernel at worst. The solution to this problem is to use a bounce buffer and this will be discussed in Section 9.4.

This chapter begins with a brief description of how the Persistent Kernel Map (PKMap) address space is managed before talking about how pages are mapped and unmapped from high memory. The subsequent section will deal with the case where the mapping must be atomic before discussing bounce buffers in depth. Finally we will talk about how emergency pools are used for when memory is very tight.

9.1 Managing the PKMap Address Space

Space is reserved at the top of the kernel page tables from PKMAP_BASE to FIXADDR_START for a PKMap. The size of the space reserved varies slightly. On the x86, PKMAP_BASE is at 0xFE000000 and the address of FIXADDR_START is a compile time constant that varies with configure options but is typically only a few pages located near the end of the linear address space. This means that there is slightly
below 32MiB of page table space for mapping pages from high memory into usable space.

For mapping pages, a single page set of PTEs is stored at the beginning of the PKMap area to allow 1024 high pages to be mapped into low memory for short periods with the function `kmap()` and unmapped with `kunmap()`. The pool seems very small but the page is only mapped by `kmap()` for a very short time. Comments in the code indicate that there was a plan to allocate contiguous page table entries to expand this area but it has remained just that, comments in the code, so a large portion of the PKMap is unused.

The page table entry for use with `kmap()` is called `pkmap_page_table` which is located at `PKMAP_BASE` and set up during system initialisation. On the x86, this takes place at the end of the `pagetable_init()` function. The pages for the PGD and PMD entries are allocated by the boot memory allocator to ensure they exist.

The current state of the page table entries is managed by a simple array called `pkmap_count` which has `LAST_PKMAP` entries in it. On an x86 system without PAE, this is 1024 and with PAE, it is 512. More accurately, albeit not expressed in code, the `LAST_PKMAP` variable is equivalent to `PTRS_PER_PTE`.

Each element is not exactly a reference count but it is very close. If the entry is 0, the page is free and has not been used since the last TLB flush. If it is 1, the slot is unused but a page is still mapped there waiting for a TLB flush. Flushes are delayed until every slot has been used at least once as a global flush is required for all CPUs when the global page tables are modified and is extremely expensive. Any higher value is a reference count of n-1 users of the page.

### 9.2 Mapping High Memory Pages

The API for mapping pages from high memory is described in Table 9.1. The main function for mapping a page is `kmap()`. For users that do not wish to block, `kmap_nonblock()` is available and interrupt users have `kmap_atomic()`. The kmap pool is quite small so it is important that users of `kmap()` call `kunmap()` as quickly as possible because the pressure on this small window grows incrementally worse as the size of high memory grows in comparison to low memory.

The `kmap()` function itself is fairly simple. It first checks to make sure an interrupt is not calling this function (as it may sleep) and calls `out_of_line_bug()` if true. An interrupt handler calling `BUG()` would panic the system so `out_of_line_bug()` prints out bug information and exits cleanly. The second check is that the page is below `highmem_start_page` as pages below this mark are already visible and do not need to be mapped.

It then checks if the page is already in low memory and simply returns the address if it is. This way, users that need `kmap()` may use it unconditionally knowing that if it is already a low memory page, the function is still safe. If it is a high page to be mapped, `kmap_high()` is called to begin the real work.

The `kmap_high()` function begins with checking the `page→virtual` field which is set if the page is already mapped. If it is NULL, `map_new_virtual()` provides a
9.2.1 Unmapping Pages

The API for unmapping pages from high memory is described in Table 9.2. The \texttt{kunmap()} function, like its complement, performs two checks. The first is an identical check to \texttt{kmap()} for usage from interrupt context. The second is that the page is below \texttt{highmem_start_page}. If it is, the page already exists in low memory and needs no further handling. Once established that it is a page to be unmapped, \texttt{kunmap_high()} is called to perform the unmapping.

Creating a new virtual mapping with \texttt{map_new_virtual()} is a simple case of linearly scanning \texttt{pkmap_count}. The scan starts at \texttt{last_pkmap_nr} instead of 0 to prevent searching over the same areas repeatedly between \texttt{kmap()}s. When \texttt{last_pkmap_nr} wraps around to 0, \texttt{flush_all_zero_pkmaps()} is called to set all entries from 1 to 0 before flushing the TLB.

If, after another scan, an entry is still not found, the process sleeps on the \texttt{pkmap_map_wait} wait queue until it is woken up after the next \texttt{kunmap()}.

Once a mapping has been created, the corresponding entry in the \texttt{pkmap_count} array is incremented and the virtual address in low memory returned.
9.3 Mapping High Memory Pages Atomically

void * kmap(struct page *page)
    Takes a struct page from high memory and maps it into low memory. The address returned is the virtual address of the mapping.

void * kmap_nonblock(struct page *page)
    This is the same as kmap() except it will not block if no slots are available and will instead return NULL. This is not the same as kmap_atomic() which uses specially reserved slots.

void * kmap_atomic(struct page *page, enum km_type type)
    There are slots maintained in the map for atomic use by interrupts (see Section 9.3). Their use is heavily discouraged and callers of this function may not sleep or schedule. This function will map a page from high memory atomically for a specific purpose.

<table>
<thead>
<tr>
<th>Table 9.1: High Memory Mapping API</th>
</tr>
</thead>
</table>

The kunmap_high() is simple in principle. It decrements the corresponding element for this page in pkmap_count. If it reaches 1 (remember this means no more users but a TLB flush is required), any process waiting on the pkmap_map_wait is woken up as a slot is now available. The page is not unmapped from the page tables then as that would require a TLB flush. It is delayed until flush_all_zero_pkmaps() is called.

void kunmap(struct page *page)
    Unmaps a struct page from low memory and frees up the page table entry mapping it

void kunmap_atomic(void *kvaddr, enum km_type type)
    Unmap a page that was mapped atomically

<table>
<thead>
<tr>
<th>Table 9.2: High Memory Unmapping API</th>
</tr>
</thead>
</table>

9.3 Mapping High Memory Pages Atomically

The use of kmap_atomic() is discouraged but slots are reserved for each CPU for when they are necessary, such as when bounce buffers, are used by devices from interrupt. There are a varying number of different requirements an architecture has for atomic high memory mapping which are enumerated by km_type. The total number of uses is KM_TYPE_NR. On the x86, there are a total of six different uses for atomic kmaps.
There are `KM_TYPE_NR` entries per processor are reserved at boot time for atomic mapping at the location `FIX_KMAP_BEGIN` and ending at `FIX_KMAP_END`. Obviously a user of an atomic kmap may not sleep or exit before calling `kunmap_atomic()` as the next process on the processor may try to use the same entry and fail.

The function `kmap_atomic()` has the very simple task of mapping the requested page to the slot set aside in the page tables for the requested type of operation and processor. The function `kunmap_atomic()` is interesting as it will only clear the PTE with `pte_clear()` if debugging is enabled. It is considered unnecessary to bother unmapping atomic pages as the next call to `kmap_atomic()` will simply replace it making TLB flushes unnecessary.

9.4 Bounce Buffers

Bounce buffers are required for devices that cannot access the full range of memory available to the CPU. An obvious example of this is when a device does not address with as many bits as the CPU, such as 32-bit devices on 64-bit architectures or recent Intel processors with PAE enabled.

The basic concept is very simple. A bounce buffer resides in memory low enough for a device to copy from and write data to. It is then copied to the desired user page in high memory. This additional copy is undesirable, but unavoidable. Pages are allocated in low memory which are used as buffer pages for DMA to and from the device. This is then copied by the kernel to the buffer page in high memory when IO completes so the bounce buffer acts as a type of bridge. There is significant overhead to this operation as at the very least it involves copying a full page but it is insignificant in comparison to swapping out pages in low memory.
9.4.1 Disk Buffering

Blocks, typically around 1KiB are packed into pages and managed by a struct buffer_head allocated by the slab allocator. Users of buffer heads have the option of registering a callback function. This function is stored in buffer_head→b_end_io() and called when IO completes. It is this mechanism that bounce buffers uses to have data copied out of the bounce buffers. The callback registered is the function bounce_end_io_write().

Any other feature of buffer heads or how they are used by the block layer is beyond the scope of this document and more the concern of the IO layer.

9.4.2 Creating Bounce Buffers

The creation of a bounce buffer is a simple affair which is started by the create_bounce() function. The principle is very simple, create a new buffer using a provided buffer head as a template. The function takes two parameters which are a read/write parameter (rw) and the template buffer head to use (bh_orig).

Figure 9.3: Call Graph: create_bounce()

A page is allocated for the buffer itself with the function alloc_bounce_page() which is a wrapper around alloc_page() with one important addition. If the allocation is unsuccessful, there is an emergency pool of pages and buffer heads available for bounce buffers. This is discussed further in Section 9.5.

The buffer head is, predictably enough, allocated with alloc_bounce_bh() which, similar in principle to alloc_bounce_page(), calls the slab allocator for a buffer_head and uses the emergency pool if one cannot be allocated. Additionally, bdflush is woken up to start flushing dirty buffers out to disk so that buffers are more likely to be freed soon.

Once the page and buffer_head have been allocated, information is copied from the template buffer_head into the new one. Since part of this operation may use kmap_atomic(), bounce buffers are only created with the IRQ safe io_request_lock held. The IO completion callbacks are changed to be either
9.4.3 Copying via bounce buffers

The most important aspect of the allocations to note is that the GFP flags specify that no IO operations involving high memory may be used. This is specified with `SLAB_NOHIGHIO` to the slab allocator and `GFP_NOHIGHIO` to the buddy allocator. This is important as bounce buffers are used for IO operations with high memory. If the allocator tries to perform high memory IO, it will recurse and eventually crash.

**9.4.3 Copying via bounce buffers**

![Figure 9.4: Call Graph: bounce_end_io_read/write()](image)

Data is copied via the bounce buffer differently depending on whether it is a read or write buffer. If the buffer is for writes to the device, the buffer is populated with the data from high memory during bounce buffer creation with the function `copy_from_high_bh()`. The callback function `bounce_end_io_write()` will complete the IO later when the device is ready for the data.

If the buffer is for reading from the device, no data transfer may take place until the device is ready. When it is, the interrupt handler for the device calls the callback function `bounce_end_io_read()` which copies the data to high memory with `copy_to_high_bh_irq()`.

In either case the buffer head and page may be reclaimed by `bounce_end_io()` once the IO has completed and the IO completion function for the template `buffer_head()` is called. If the emergency pools are not full, the resources are added to the pools otherwise they are freed back to the respective allocators.

**9.5 Emergency Pools**

Two emergency pools of `buffer_heads` and pages are maintained for the express use by bounce buffers. If memory is too tight for allocations, failing to complete IO requests is going to compound the situation as buffers from high memory cannot be
freed until low memory is available. This leads to processes halting, thus preventing the possibility of them freeing up their own memory.

The pools are initialised by `init_emergency_pool()` to contain `POOL_SIZE` entries each which is currently defined as 32. The pages are linked via the page→list field on a list headed by `emergency_pages`. Figure 9.5 illustrates how pages are stored on emergency pools and acquired when necessary.

The `buffer_heads` are very similar as they linked via the `buffer_head→inode_buffers` on a list headed by `emergency_bhs`. The number of entries left on the pages and buffer lists are recorded by two counters `nr_emergency_pages` and `nr_emergency_bhs` respectively and the two lists are protected by the `emergency_lock` spinlock.

![Figure 9.5: Acquiring Pages from Emergency Pools](image)

### 9.6 What’s New in 2.6

**Memory Pools** In 2.4, the high memory manager was the only subsystem that maintained emergency pools of pages. In 2.6, memory pools are implemented as a generic concept when a minimum amount of “stuff” needs to be reserved for when memory is tight. “Stuff” in this case can be any type of object such as pages in the case of the high memory manager or, more frequently, some object managed by the slab allocator. Pools are initialised with `mempool_create()` which takes a number of arguments. They are the minimum number of objects that should be reserved (`min_nr`), an allocator function for the object type (`alloc_fn()`), a free function (`free_fn()`) and optional private data that is passed to the allocate and free functions.
The memory pool API provides two generic allocate and free functions called `mempool_alloc_slab()` and `mempool_free_slab()`. When the generic functions are used, the private data is the slab cache that objects are to be allocated and freed from.

In the case of the high memory manager, two pools of pages are created. On page pool is for normal use and the second page pool is for use with ISA devices that must allocate from `ZONE_DMA`. The allocate function is `page_pool_alloc()` and the private data parameter passed indicates the GFP flags to use. The free function is `page_pool_free()`. The memory pools replace the emergency pool code that exists in 2.4.

To allocate or free objects from the memory pool, the memory pool API functions `mempool_alloc()` and `mempool_free()` are provided. Memory pools are destroyed with `mempool_destroy()`.

Mapping High Memory Pages In 2.4, the field `page→virtual` was used to store the address of the page within the `pkmap_count` array. Due to the number of `struct page` that exist in a high memory system, this is a very large penalty to pay for the relatively small number of pages that need to be mapped into `ZONE_NORMAL`. 2.6 still has this `pkmap_count` array but it is managed very differently.

In 2.6, a hash table called `page_address_htable` is created. This table is hashed based on the address of the `struct page` and the list is used to locate `struct page_address_slot`. This struct has two fields of interest, a `struct page` and a virtual address. When the kernel needs to find the virtual address used by a mapped page, it is located by traversing through this hash bucket. How the page is actually mapped into lower memory is essentially the same as 2.4 except now `page→virtual` is no longer required.

Performing IO The last major change is that the `struct bio` is now used instead of the `struct buffer_head` when performing IO. How bio structures work is beyond the scope of this book. However, the principle reason that bio structures were introduced is so that IO could be performed in blocks of whatever size the underlying device supports. In 2.4, all IO had to be broken up into page sized chunks regardless of the transfer rate of the underlying device.
Chapter 10
Page Frame Reclamation

A running system will eventually use all available page frames for purposes like disk buffers, dentries, inode entries, process pages and so on. Linux needs to select old pages which can be freed and invalidated for new uses before physical memory is exhausted. This chapter will focus exclusively on how Linux implements its page replacement policy and how different types of pages are invalidated.

The methods Linux uses to select pages are rather empirical in nature and the theory behind the approach is based on multiple different ideas. It has been shown to work well in practice and adjustments are made based on user feedback and benchmarks. The basics of the page replacement policy is the first item of discussion in this Chapter.

The second topic of discussion is the Page cache. All data that is read from disk is stored in the page cache to reduce the amount of disk IO that must be performed. Strictly speaking, this is not directly related to page frame reclamation, but the LRU lists and page cache are closely related. The relevant section will focus on how pages are added to the page cache and quickly located.

This will bring us to the third topic, the LRU lists. With the exception of the slab allocator, all pages in use by the system are stored on LRU lists and linked together via page→lru so they can be easily scanned for replacement. The slab pages are not stored on the LRU lists as it is considerably more difficult to age a page based on the objects used by the slab. The section will focus on how pages move through the LRU lists before they are reclaimed.

From there, we’ll cover how pages belonging to other caches, such as the dcache, and the slab allocator are reclaimed before talking about how process-mapped pages are removed. Process mapped pages are not easily swappable as there is no way to map struct pages to PTEs except to search every page table which is far too expensive. If the page cache has a large number of process-mapped pages in it, process page tables will be walked and pages swapped out by swap_out() until enough pages have been freed but this will still have trouble with shared pages. If a page is shared, a swap entry is allocated, the PTE filled with the necessary information to find the page in swap again and the reference count decremented. Only when the count reaches zero will the page be freed. Pages like this are considered to be in the Swap cache.
Finally, this chapter will cover the page replacement daemon `kswapd`, how it is implemented and what it’s responsibilities are.

## 10.1 Page Replacement Policy

During discussions the page replacement policy is frequently said to be a *Least Recently Used (LRU)*-based algorithm but this is not strictly speaking true as the lists are not strictly maintained in LRU order. The LRU in Linux consists of two lists called the `active_list` and `inactive_list`. The objective is for the `active_list` to contain the *working set* [Den70] of all processes and the `inactive_list` to contain reclaim candidates. As all reclaimable pages are contained in just two lists and pages belonging to any process may be reclaimed, rather than just those belonging to a faulting process, the replacement policy is a global one.

The lists resemble a simplified LRU 2Q [JS94] where two lists called `Am` and `A1` are maintained. With LRU 2Q, pages when first allocated are placed on a FIFO queue called `A1`. If they are referenced while on that queue, they are placed in a normal LRU managed list called `Am`. This is roughly analogous to using `lru_cache_add()` to place pages on a queue called `inactive_list (A1)` and using `mark_page_accessed()` to get moved to the `active_list (Am)`. The algorithm describes how the size of the two lists have to be tuned but Linux takes a simpler approach by using `refill_inactive()` to move pages from the bottom of the `active_list` to `inactive_list` to keep `active_list` about two thirds the size of the total page cache. Figure 10.1 illustrates how the two lists are structured, how pages are added and how pages move between the lists with `refill_inactive()`.

The lists described for 2Q presumes `Am` is an LRU list but the list in Linux closer resembles a Clock algorithm [Car84] where the hand-spread is the size of the active list. When pages reach the bottom of the list, the referenced flag is checked, if it is set, it is moved back to the top of the list and the next page checked. If it is cleared, it is moved to the `inactive_list`.

The Move-To-Front heuristic means that the lists behave in an LRU-like manner but there are too many differences between the Linux replacement policy and LRU to consider it a stack algorithm [MM87]. Even if we ignore the problem of analysing multi-programmed systems [CD80] and the fact the memory size for each process is not fixed, the policy does not satisfy the inclusion property as the location of pages in the lists depend heavily upon the size of the lists as opposed to the time of last reference. Neither is the list priority ordered as that would require list updates with every reference. As a final nail in the stack algorithm coffin, the lists are almost ignored when paging out from processes as pageout decisions are related to their location in the virtual address space of the process rather than the location within the page lists.

In summary, the algorithm does exhibit LRU-like behaviour and it has been shown by benchmarks to perform well in practice. There are only two cases where the algorithm is likely to behave really badly. The first is if the candidates for reclamation are principally anonymous pages. In this case, Linux will keep examining
Figure 10.1: Page Cache LRU Lists

a large number of pages before linearly scanning process page tables searching for pages to reclaim but this situation is fortunately rare.

The second situation is where there is a single process with many file backed resident pages in the inactive_list that are being written to frequently. Processes and kswapd may go into a loop of constantly “laundering” these pages and placing them at the top of the inactive_list without freeing anything. In this case, few pages are moved from the active_list to inactive_list as the ratio between the two lists sizes remains not change significantly.

10.2 Page Cache

The page cache is a set of data structures which contain pages that are backed by regular files, block devices or swap. There are basically four types of pages that exist in the cache:

• Pages that were faulted in as a result of reading a memory mapped file;

• Blocks read from a block device or filesystem are packed into special pages called buffer pages. The number of blocks that may fit depends on the size of the block and the page size of the architecture;
10.2.1 Page Cache Hash Table

- Anonymous pages exist in a special aspect of the page cache called the swap cache when slots are allocated in the backing storage for page-out, discussed further in Chapter 11;

- Pages belonging to shared memory regions are treated in a similar fashion to anonymous pages. The only difference is that shared pages are added to the swap cache and space reserved in backing storage immediately after the first write to the page.

The principal reason for the existence of this cache is to eliminate unnecessary disk reads. Pages read from disk are stored in a page hash table which is hashed on the struct address_space and the offset which is always searched before the disk is accessed. An API is provided that is responsible for manipulating the page cache which is listed in Table 10.1.

10.2.1 Page Cache Hash Table

There is a requirement that pages in the page cache be quickly located. To facilitate this, pages are inserted into a table page_hash_table and the fields page→next_hash and page→pprev_hash are used to handle collisions.

The table is declared as follows in mm/filemap.c:

45 atomic_t page_cache_size = ATOMIC_INIT(0);
46 unsigned int page_hash_bits;
47 struct page **page_hash_table;

The table is allocated during system initialisation by page_cache_init() which takes the number of physical pages in the system as a parameter. The desired size of the table (htable_size) is enough to hold pointers to every struct page in the system and is calculated by

\[
htable\_size = \text{num\_physpages} \times \text{sizeof(struct page *)}
\]

To allocate a table, the system begins with an order allocation large enough to contain the entire table. It calculates this value by starting at 0 and incrementing it until \(2^{\text{order}} > htable\_size\). This may be roughly expressed as the integer component of the following simple equation.

\[
\text{order} = \log_2((htable\_size \times 2) - 1))
\]

An attempt is made to allocate this order of pages with __get_free_pages(). If the allocation fails, lower orders will be tried and if no allocation is satisfied, the system panics.

The value of page_hash_bits is based on the size of the table for use with the hashing function _page_hashfn(). The value is calculated by successive divides by two but in real terms, this is equivalent to:
void add_to_page_cache(struct page * page, struct address_space * mapping, unsigned long offset)
   Adds a page to the LRU with lru_cache_add() in addition to adding it to
   the inode queue and page hash tables

void add_to_page_cache_unique(struct page * page, struct
   address_space *mapping, unsigned long offset, struct page **hash)
   This is similar to add_to_page_cache() except it checks that the page is not
   already in the page cache. This is required when the caller does not hold the
   pagecache_lock spinlock

void remove_inode_page(struct page *page)
   This function removes a page from the inode and hash queues with
   remove_page_from_inode_queue() and remove_page_from_hash_queue(), ef-
   fectively removing the page from the page cache

struct page * page_cache_alloc(struct address_space *x)
   This is a wrapper around alloc_pages() which uses x→gfp_mask as the GFP
   mask

void page_cache_get(struct page *page)
   Increases the reference count to a page already in the page cache

int page_cache_read(struct file * file, unsigned long offset)
   This function adds a page corresponding to an offset with a file if it
   is not already there. If necessary, the page will be read from disk using an
   address_space_operations→readpage function

void page_cache_release(struct page *page)
   An alias for __free_page(). The reference count is decremented and if it
   drops to 0, the page will be freed

Table 10.1: Page Cache API

\[
\text{page} \_\text{hash} \_\text{bits} = \log_2 \left\lfloor \frac{\text{PAGE} \_\text{SIZE} \times 2^{\text{order}}}{\text{sizeof(struct page \*)}} \right\rfloor
\]

This makes the table a power-of-two hash table which negates the need to use a
modulus which is a common choice for hashing functions.

10.2.2 Inode Queue

The inode queue is part of the struct address_space introduced in Section 4.4.2.
The struct contains three lists: clean_pages is a list of clean pages associated
10.2.3 Adding Pages to the Page Cache

Pages read from a file or block device are generally added to the page cache to avoid further disk IO. Most filesystems use the high level function `generic_file_read()` as their `file_operations->read()`. The shared memory filesystem, which is covered in Chapter 12, is one noteworthy exception but, in general, filesystems perform their operations through the page cache. For the purposes of this section, we’ll illustrate how `generic_file_read()` operates and how it adds pages to the page cache.

For normal IO\(^1\), `generic_file_read()` begins with a few basic checks before calling `do_generic_file_read()`. This searches the page cache, by calling `__find_page_nolock()` with the `pagecache_lock` held, to see if the page already exists in it. If it does not, a new page is allocated with `page_cache_alloc()`, which is a simple wrapper around `alloc_pages()`, and added to the page cache with `__add_to_page_cache()`. Once a page frame is present in the page cache, `generic_file_readahead()` is called which uses `page_cache_read()` to read the page from disk. It reads the page using `mapping->a_ops->readpage()`, where `mapping` is the `address_space` managing the file. `readpage()` is the filesystem specific function used to read a page on disk.

Anonymous pages are added to the swap cache when they are unmapped from a process, which will be discussed further in Section 11.4. Until an attempt is made to swap them out, they have no `address_space` acting as a mapping or any offset.

---

\(^1\)Direct IO is handled differently with `generic_file_direct_IO()`.
within a file leaving nothing to hash them into the page cache with. Note that these pages still exist on the LRU lists however. Once in the swap cache, the only real difference between anonymous pages and file backed pages is that anonymous pages will use `swapper_space` as their `struct address_space`.

Shared memory pages are added during one of two cases. The first is during `shmem_getpage_locked()` which is called when a page has to be either fetched from swap or allocated as it is the first reference. The second is when the swapout code calls `shmem_unuse()`. This occurs when a swap area is being deactivated and a page, backed by swap space, is found that does not appear to belong to any process. The inodes related to shared memory are exhaustively searched until the correct page is found. In both cases, the page is added with `add_to_page_cache()`.

![Figure 10.3: Call Graph: add_to_page_cache()](image)

### 10.3 LRU Lists

As stated in Section 10.1, the LRU lists consist of two lists called `active_list` and `inactive_list`. They are declared in `mm/page_alloc.c` and are protected by the `pagemap_lru_lock` spinlock. They, broadly speaking, store the “hot” and “cold” pages respectively, or in other words, the `active_list` contains all the working sets in the system and `inactive_list` contains reclaim candidates. The API which deals with the LRU lists that is listed in Table 10.2.

#### 10.3.1 Refilling `inactive_list`

When caches are being shrunk, pages are moved from the `active_list` to the `inactive_list` by the function `refill_inactive()`. It takes as a parameter the number of pages to move, which is calculated in `shrink_caches()` as a ratio depending on `nr_pages`, the number of pages in `active_list` and the number of pages in `inactive_list`. The number of pages to move is calculated as

\[
pages = \text{nr\_pages} \times \frac{\text{nr\_active\_pages}}{2 \times (\text{nr\_inactive\_pages} + 1)}
\]
void lru_cache_add(struct page * page)
Add a cold page to the inactive_list. Will be moved to active_list with a call to mark_page_accessed() if the page is known to be hot, such as when a page is faulted in.

void lru_cache_del(struct page * page)
Removes a page from the LRU lists by calling either del_page_from_active_list() or del_page_from_inactive_list(), whichever is appropriate.

void mark_page_accessed(struct page * page)
Mark that the page has been accessed. If it was not recently referenced (in the inactive_list and PG_referenced flag not set), the referenced flag is set. If it is referenced a second time, activate_page() is called, which marks the page hot, and the referenced flag is cleared.

void activate_page(struct page * page)
Removes a page from the inactive_list and places it on active_list. It is very rarely called directly as the caller has to know the page is on inactive_list. mark_page_accessed() should be used instead.

Table 10.2: LRU List API

This keeps the active_list about two thirds the size of the inactive_list and the number of pages to move is determined as a ratio based on how many pages we desire to swap out (nr_pages).

Pages are taken from the end of the active_list. If the PG_referenced flag is set, it is cleared and the page is put back at top of the active_list as it has been recently used and is still “hot”. This is sometimes referred to as rotating the list. If the flag is cleared, it is moved to the inactive_list and the PG_referenced flag set so that it will be quickly promoted to the active_list if necessary.

10.3.2 Reclaiming Pages from the LRU Lists

The function shrink_cache() is the part of the replacement algorithm which takes pages from the inactive_list and decides how they should be swapped out. The two starting parameters which determine how much work will be performed are nr_pages and priority. nr_pages starts out as SWAP_CLUSTER_MAX, currently defined as 32 in mm/vmscan.c. The variable priority starts as DEF_PRIORITY, currently defined as 6 in mm/vmscan.c.

Two parameters, max_scan and max_mapped determine how much work the function will do and are affected by the priority. Each time the function shrink_caches() is called without enough pages being freed, the priority will be decreased until the highest priority 1 is reached.
The variable $\text{max\_scan}$ is the maximum number of pages will be scanned by this function and is simply calculated as

$$\text{max\_scan} = \frac{\text{nr\_inactive\_pages}}{\text{priority}}$$

where $\text{nr\_inactive\_pages}$ is the number of pages in the $\text{inactive\_list}$. This means that at lowest priority 6, at most one sixth of the pages in the $\text{inactive\_list}$ will be scanned and at highest priority, all of them will be.

The second parameter is $\text{max\_mapped}$ which determines how many process pages are allowed to exist in the page cache before whole processes will be swapped out. This is calculated as the minimum of either one tenth of $\text{max\_scan}$ or

$$\text{max\_mapped} = \text{nr\_pages} \times 2^{(10 - \text{priority})}$$

In other words, at lowest priority, the maximum number of mapped pages allowed is either one tenth of $\text{max\_scan}$ or 16 times the number of pages to swap out ($\text{nr\_pages}$) whichever is the lower number. At high priority, it is either one tenth of $\text{max\_scan}$ or 512 times the number of pages to swap out.

From there, the function is basically a very large for-loop which scans at most $\text{max\_scan}$ pages to free up $\text{nr\_pages}$ pages from the end of the $\text{inactive\_list}$ or until the $\text{inactive\_list}$ is empty. After each page, it checks to see whether it should reschedule itself so that the swapper does not monopolise the CPU.

For each type of page found on the list, it makes a different decision on what to do. The different page types and actions taken are handled in this order:

1. **Page is mapped by a process.** This jumps to the $\text{page\_mapped}$ label which we will meet again in a later case. The $\text{max\_mapped}$ count is decremented. If it reaches 0, the page tables of processes will be linearly searched and swapped out by the function $\text{swap\_out()}$.

2. **Page is locked and the PG_launder bit is set.** The page is locked for IO so could be skipped over. However, if the PG_launder bit is set, it means that this is the second time the page has been found locked so it is better to wait until the IO completes and get rid of it. A reference to the page is taken with $\text{page\_cache\_get()}$ so that the page will not be freed prematurely and $\text{wait\_on\_page()}$ is called which sleeps until the IO is complete. Once it is completed, the reference count is decremented with $\text{page\_cache\_release()}$. When the count reaches zero, the page will be reclaimed.

3. **Page is dirty, is unmapped by all processes, has no buffers and belongs to a device or file mapping.** As the page belongs to a file or device mapping, it has a valid $\text{writepage()}$ function available via $\text{page\_mapping\_a\_ops\_writepage}$. The PG_dirty bit is cleared and the PG_launder bit is set as it is about to start IO. A reference is taken for the page with $\text{page\_cache\_get()}$ before calling the $\text{writepage()}$ function to synchronise the page with the backing file before dropping the reference with $\text{page\_cache\_release()}$. Be aware that this case will also synchronise anonymous pages that are part of the swap cache with the backing storage as swap cache pages use $\text{swapper\_space}$ as a $\text{page\_mapping}$. The page remains on
10.4 Shrinking all caches

The function responsible for shrinking the various caches is `shrink_caches()` which takes a few simple steps to free up some memory. The maximum number of pages that will be written to disk in any given pass is `nr_pages` which is initialised by `try_to_free_pages_zone()` to be `SWAP_CLUSTER_MAX`. The limitation is there so that if `kswapd` schedules a large number of pages to be written to disk, it will sleep occasionally to allow the IO to take place. As pages are freed, `nr_pages` is decremented to keep count.

The amount of work that will be performed also depends on the priority initialised by `try_to_free_pages_zone()` to be `DEF_PRIORITY`. For each pass that does not free up enough pages, the priority is decremented for the highest priority been 1.

The function first calls `kmem_cache_reap()` (see Section 8.1.7) which selects a slab cache to shrink. If `nr_pages` number of pages are freed, the work is complete and the function returns otherwise it will try to free `nr_pages` from other caches.

If other caches are to be affected, `refill_inactive()` will move pages from the `active_list` to the `inactive_list` before shrinking the page cache by reclaiming pages at the end of the `inactive_list` with `shrink_cache()`.

Finally, it shrinks three special caches, the `dcache` (`shrink_dcache_memory()`), the `icache` (`shrink_icache_memory()`) and the `dqcache` (`shrink_dqcache_memory()`). These objects are quite small in themselves but a cascading effect allows a lot more pages to be freed in the form of buffer and disk caches.
10.5 Swapping Out Process Pages

When max_mapped pages have been found in the page cache, swap_out() is called to start swapping out process pages. Starting from the mm_struct pointed to by swap_mm and the address mm→swap_address, the page tables are searched forward until nr_pages have been freed.

All process mapped pages are examined regardless of where they are in the lists or when they were last referenced but pages which are part of the active_list or have been recently referenced will be skipped over. The examination of hot pages is a bit costly but insignificant in comparison to linearly searching all processes for the PTEs that reference a particular struct page.

Once it has been decided to swap out pages from a process, an attempt will be
made to swap out at least SWAP_CLUSTER_MAX number of pages and the full list of mm_structs will only be examined once to avoid constant looping when no pages are available. Writing out the pages in bulk increases the chance that pages close together in the process address space will be written out to adjacent slots on disk.

The marker swap_mm is initialised to point to init_mm and the swap_address is initialised to 0 the first time it is used. A task has been fully searched when the swap_address is equal to TASK_SIZE. Once a task has been selected to swap pages from, the reference count to the mm_struct is incremented so that it will not be freed early and swap_out_mm() is called with the selected mm_struct as a parameter. This function walks each VMA the process holds and calls swap_out_vma() for it. This is to avoid having to walk the entire page table which will be largely sparse. swap_out_pgd() and swap_out_pmd() walk the page tables for given VMA until finally try_to_swap_out() is called on the actual page and PTE.

The function try_to_swap_out() first checks to make sure that the page is not part of the active_list, has been recently referenced or belongs to a zone that we are not interested in. Once it has been established this is a page to be swapped out, it is removed from the process page tables. The newly removed PTE is then checked to see if it is dirty. If it is, the struct page flags will be updated to match so that it will get synchronised with the backing storage. If the page is already a part of the swap cache, the RSS is simply updated and the reference to the page is dropped, otherwise the process is added to the swap cache. How pages are added to the swap cache and synchronised with backing storage is discussed in Chapter 11.

10.6 Pageout Daemon (kswapd)

During system startup, a kernel thread called kswapd is started from kswapd_init() which continuously executes the function kswapd() in mm/vmscan.c which usually sleeps. This daemon is responsible for reclaiming pages when memory is running low. Historically, kswapd used to wake up every 10 seconds but now it is only woken by the physical page allocator when the pages_low number of free pages in a zone is reached (see Section 2.2.1).

It is this daemon that performs most of the tasks needed to maintain the page cache correctly, shrink slab caches and swap out processes if necessary. Unlike swapout daemons such as Solaris [MM01], which are woken up with increasing frequency as there is memory pressure, kswapd keeps freeing pages until the pages_high watermark is reached. Under extreme memory pressure, processes will do the work of kswapd synchronously by calling balance_classzone() which calls try_to_free_pages_zone(). As shown in Figure 10.6, it is at try_to_free_pages_zone() where the physical page allocator synchronously performs the same task as kswapd when the zone is under heavy pressure.

When kswapd is woken up, it performs the following:

- Calls kswapd_can_sleep() which cycles through all zones checking the need_balance field in the struct zone_t. If any of them are set, it can not sleep;
10.7 What’s New in 2.6

**kswapd**  As stated in Section 2.6, there is now a `kswapd` for every memory node in the system. These daemons are still started from `kswapd()` and they all execute the same code except their work is confined to their local node. The main changes to the implementation of `kswapd` are related to the `kswapd-per-node` change.

The basic operation of `kswapd` remains the same. Once woken, it calls `balance_pgdat()` for the `pgdat` it is responsible for. `balance_pgdat()` has two modes of operation. When called with `nr_pages == 0`, it will continually try to free pages from each zone in the local `pgdat` until `pages_high` is reached. When `nr_pages` is specified, it will try and free either `nr_pages` or `MAX_CLUSTER_MAX * 8`, whichever is the smaller number of pages.

**Balancing Zones**  The two main functions called by `balance_pgdat()` to free pages are `shrink_slab()` and `shrink_zone()`. `shrink_slab()` was covered in Section 8.8 so will not be repeated here. The function `shrink_zone()` is called to free a number of pages based on how urgent it is to free pages. This function behaves

---

**Figure 10.6: Call Graph: kswapd()**

- If it cannot sleep, it is removed from the `kswapd_wait` wait queue;
- Calls the functions `kswapd_balance()`, which cycles through all zones. It will free pages in a zone with `try_to_free_pages_zone()` if `need_balance` is set and will keep freeing until the `pages_high` watermark is reached;
- The task queue for `tq_disk` is run so that pages queued will be written out;
- Add `kswapd` back to the `kswapd_wait` queue and go back to the first step.

---
very similar to how 2.4 works. refill_inactive_zone() will move a number of
pages from zone->active_list to zone->inactive_list. Remember as covered
in Section 2.6, that LRU lists are now per-zone and not global as they are in 2.4.
shrink_cache() is called to remove pages from the LRU and reclaim pages.

Pageout Pressure In 2.4, the pageout priority determined how many pages
would be scanned. In 2.6, there is a decaying average that is updated by
zone_adj_pressure(). This adjusts the zone->pressure field to indicate how
many pages should be scanned for replacement. When more pages are required, this
will be pushed up towards the highest value of DEF_PRIORITY < 10 and then decays
over time. The value of this average affects how many pages will be scanned in a
zone for replacement. The objective is to have page replacement start working and
slow gracefully rather than act in a bursty nature.

Manipulating LRU Lists In 2.4, a spinlock would be acquired when removing
pages from the LRU list. This made the lock very heavily contended so, to relieve
contention, operations involving the LRU lists take place via struct pagevec struc-
tures. This allows pages to be added or removed from the LRU lists in batches of
up to PAGEVEC_SIZE numbers of pages.

To illustrate, when refill_inactive_zone() and shrink_cache() are remov-
ing pages, they acquire the zone->lru_lock lock, remove large blocks of pages and
store them on a temporary list. Once the list of pages to remove is assembled,
shrink_list() is called to perform the actual freeing of pages which can now per-
form most of it's task without needing the zone->lru_lock spinlock.

When adding the pages back, a new page vector struct is initialised with
pagevec_init(). Pages are added to the vector with pagevec_add() and then
committed to being placed on the LRU list in bulk with pagevec_release().

There is a sizable API associated with pagevec structs which can be seen in
<linux/pagevec.h> with most of the implementation in mm/swap.c.
Chapter 11

Swap Management

Just as Linux uses free memory for purposes such as buffering data from disk, there eventually is a need to free up private or anonymous pages used by a process. These pages, unlike those backed by a file on disk, cannot be simply discarded to be read in later. Instead they have to be carefully copied to backing storage, sometimes called the swap area. This chapter details how Linux uses and manages its backing storage.

Strictly speaking, Linux does not swap as “swapping” refers to copying an entire process address space to disk and “paging” to copying out individual pages. Linux actually implements paging as modern hardware supports it, but traditionally has called it swapping in discussions and documentation. To be consistent with the Linux usage of the word, we too will refer to it as swapping.

There are two principle reasons that the existence of swap space is desirable. First, it expands the amount of memory a process may use. Virtual memory and swap space allows a large process to run even if the process is only partially resident. As “old” pages may be swapped out, the amount of memory addressed may easily exceed RAM as demand paging will ensure the pages are reloaded if necessary.

The casual reader\(^1\) may think that with a sufficient amount of memory, swap is unnecessary but this brings us to the second reason. A significant number of the pages referenced by a process early in its life may only be used for initialisation and then never used again. It is better to swap out those pages and create more disk buffers than leave them resident and unused.

It is important to note that swap is not without its drawbacks and the most important one is the most obvious one; Disk is slow, very very slow. If processes are frequently addressing a large amount of memory, no amount of swap or expensive high-performance disks will make it run within a reasonable time, only more RAM will help. This is why it is very important that the correct page be swapped out as discussed in Chapter 10, but also that related pages be stored close together in the swap space so they are likely to be swapped in at the same time while reading ahead. We will start with how Linux describes a swap area.

This chapter begins with describing the structures Linux maintains about each

\(^{1}\)Not to mention the affluent reader.
active swap area in the system and how the swap area information is organised on disk. We then cover how Linux remembers how to find pages in the swap after they have been paged out and how swap slots are allocated. After that the Swap Cache is discussed which is important for shared pages. At that point, there is enough information to begin understanding how swap areas are activated and deactivated, how pages are paged in and paged out and finally how the swap area is read and written to.

11.1 Describing the Swap Area

Each active swap area, be it a file or partition, has a struct `swap_info_struct` describing the area. All the structs in the running system are stored in a statically declared array called `swap_info` which holds `MAX_SWAPFILES`, which is statically defined as 32, entries. This means that at most 32 swap areas can exist on a running system. The `swap_info_struct` is declared as follows in `<linux/swap.h>`:

```c
64 struct swap_info_struct {
65    unsigned int flags;
66    kdev_t swap_device;
67    spinlock_t sdev_lock;
68    struct dentry * swap_file;
69    struct vfsmount * swap_vfsmnt;
70    unsigned short * swap_map;
71    unsigned int lowest_bit;
72    unsigned int highest_bit;
73    unsigned int cluster_next;
74    unsigned int cluster_nr;
75    int prio;
76    int pages;
77    unsigned long max;
78    int next;
79  };
```

Here is a small description of each of the fields in this quite sizable struct.

**flags** This is a bit field with two possible values. `SWP_USED` is set if the swap area is currently active. `SWP_WRITEOK` is defined as 3, the two lowest significant bits, including the `SWP_USED` bit. The flags is set to `SWP_WRITEOK` when Linux is ready to write to the area as it must be active to be written to;

**swap_device** The device corresponding to the partition used for this swap area is stored here. If the swap area is a file, this is NULL;

**sdev_lock** As with many structs in Linux, this one has to be protected too. `sdev_lock` is a spinlock protecting the struct, principally the `swap_map`. It is locked and unlocked with `swap_device_lock()` and `swap_device_unlock()`;
11.1 Describing the Swap Area

**swap_file** This is the dentry for the actual special file that is mounted as a swap area. This could be the dentry for a file in the /dev/ directory for example in the case a partition is mounted. This field is needed to identify the correct swap_info_struct when deactivating a swap area;

**vfs_mount** This is the vfs_mount object corresponding to where the device or file for this swap area is stored;

**swap_map** This is a large array with one entry for every swap entry, or page sized slot in the area. An entry is a reference count of the number of users of this page slot. The swap cache counts as one user and every PTE that has been paged out to the slot counts as a user. If it is equal to SWAP_MAP_MAX, the slot is allocated permanently. If equal to SWAP_MAP_BAD, the slot will never be used;

**lowest_bit** This is the lowest possible free slot available in the swap area and is used to start from when linearly scanning to reduce the search space. It is known that there are definitely no free slots below this mark;

**highest_bit** This is the highest possible free slot available in this swap area. Similar to lowest_bit, there are definitely no free slots above this mark;

**cluster_next** This is the offset of the next cluster of blocks to use. The swap area tries to have pages allocated in cluster blocks to increase the chance related pages will be stored together;

**cluster_nr** This the number of pages left to allocate in this cluster;

**prio** Each swap area has a priority which is stored in this field. Areas are arranged in order of priority and determine how likely the area is to be used. By default the priorities are arranged in order of activation but the system administrator may also specify it using the -p flag when using swapon;

**pages** As some slots on the swap file may be unusable, this field stores the number of usable pages in the swap area. This differs from max in that slots marked SWAP_MAP_BAD are not counted;

**max** This is the total number of slots in this swap area;

**next** This is the index in the swap_info array of the next swap area in the system.

The areas, though stored in an array, are also kept in a pseudo list called swap_list which is a very simple type declared as follows in <linux/swap.h>:

```c
153 struct swap_list_t {
154    int head;   /* head of priority-ordered swapfile list */
155    int next;    /* swapfile to be used next */
156};
```
11.1 Describing the Swap Area

The field `swap_list_t→head` is the swap area of the highest priority swap area in use and `swap_list_t→next` is the next swap area that should be used. This is so areas may be arranged in order of priority when searching for a suitable area but still looked up quickly in the array when necessary.

Each swap area is divided up into a number of page sized slots on disk which means that each slot is 4096 bytes on the x86 for example. The first slot is always reserved as it contains information about the swap area that should not be overwritten. The first 1 KiB of the swap area is used to store a disk label for the partition that can be picked up by userspace tools. The remaining space is used for information about the swap area which is filled when the swap area is created with the system program `mkswap`. The information is used to fill in a union `swap_header` which is declared as follows in `<linux/swap.h>`:

```c
union swap_header {
    struct {
        char reserved[PAGE_SIZE - 10];
        char magic[10];
    } magic;
    struct {
        char bootbits[1024];
        unsigned int version;
        unsigned int last_page;
        unsigned int nr_badpages;
        unsigned int padding[125];
        unsigned int badpages[1];
    } info;
};
```

A description of each of the fields follows

**magic** The magic part of the union is used just for identifying the “magic” string. The string exists to make sure there is no chance a partition that is not a swap area will be used and to decide what version of swap area is is. If the string is “SWAP-SPACE”, it is version 1 of the swap file format. If it is “SWAPSPACE2”, it is version 2. The large reserved array is just so that the magic string will be read from the end of the page;

**bootbits** This is the reserved area containing information about the partition such as the disk label;

**version** This is the version of the swap area layout;

**last_page** This is the last usable page in the area;
11.2 Mapping Page Table Entries to Swap Entries

nr_badpages The known number of bad pages that exist in the swap area are stored in this field;

padding A disk section is usually about 512 bytes in size. The three fields version, last_page and nr_badpages make up 12 bytes and the padding fills up the remaining 500 bytes to cover one sector;

badpages The remainder of the page is used to store the indices of up to MAX_SWAP_BADPAGES number of bad page slots. These slots are filled in by the mkswap system program if the -c switch is specified to check the area.

MAX_SWAP_BADPAGES is a compile time constant which varies if the struct changes but it is 637 entries in its current form as given by the simple equation;

\[
\text{MAX\_SWAP\_BADPAGES} = \frac{\text{PAGE\_SIZE} - 1024 - 512 - 10}{\text{sizeof(long)}}
\]

Where 1024 is the size of the bootblock, 512 is the size of the padding and 10 is the size of the magic string identifying the format of the swap file.

11.2 Mapping Page Table Entries to Swap Entries

When a page is swapp ed out, Linux uses the corresponding PTE to store enough information to locate the page on disk again. Obviously a PTE is not large enough in itself to store precisely where on disk the page is located, but it is more than enough to store an index into the swap_info array and an offset within the swap_map and this is precisely what Linux does.

Each PTE, regardless of architecture, is large enough to store a swp_entry_t which is declared as follows in <linux/shmem_fs.h>

16 typedef struct {
17     unsigned long val;
18 } swp_entry_t;

Two macros are provided for the translation of PTEs to swap entries and vice versa. They are pte_to_swp_entry() and swp_entry_to_pte() respectively.

Each architecture has to be able to determine if a PTE is present or swapped out. For illustration, we will show how this is implemen ted on the x86. In the swp_entry_t, two bits are always kept free. On the x86, Bit 0 is reserved for the _PAGE_PRESENT flag and Bit 7 is reserved for _PAGE_PROTNONE. The requirement for both bits is explained in Section 3.2. Bits 1-6 are for the type which is the index within the swap_info array and are returned by the SWP_TYPE() macro.

Bits 8-31 are used are to store the offset within the swap_map from the swp_entry_t. On the x86, this means 24 bits are available, “limiting” the size of the swap area to 64GiB. The macro SWP_OFFSET() is used to extract the offset.
11.3 Allocating a swap slot

All page sized slots are tracked by the array `swap_info_struct->swap_map` which is of type `unsigned short`. Each entry is a reference count of the number of users of the slot which happens in the case of a shared page and is 0 when free. If the

---

A Sun E450 could have in the region of 20 disks in it for example.
entry is `SWAP_MAP_MAX`, the page is permanently reserved for that slot. It is unlikely, if not impossible, for this condition to occur but it exists to ensure the reference count does not overflow. If the entry is `SWAP_MAP_BAD`, the slot is unusable.

![Figure 11.2: Call Graph: get_swap_page()](image)

The task of finding and allocating a swap entry is divided into two major tasks. The first performed by the high level function `get_swap_page()`. Starting with `swap_list→next`, it searches swap areas for a suitable slot. Once a slot has been found, it records what the next swap area to be used will be and returns the allocated entry.

The task of searching the map is the responsibility of `scan_swap_map()`. In principle, it is very simple as it linearly scan the array for a free slot and return. Predictably, the implementation is a bit more thorough.

Linux attempts to organise pages into clusters on disk of size `SWAPFILE_CLUSTER`. It allocates `SWAPFILE_CLUSTER` number of pages sequentially in swap keeping count of the number of sequentially allocated pages in `swap_info_struct→cluster_nr` and records the current offset in `swap_info_struct→cluster_next`. Once a sequential block has been allocated, it searches for a block of free entries of size `SWAPFILE_CLUSTER`. If a block large enough can be found, it will be used as another cluster sized sequence.

If no free clusters large enough can be found in the swap area, a simple first-free search starting from `swap_info_struct→lowest_bit` is performed. The aim is to have pages swapped out at the same time close together on the premise that pages swapped out together are related. This premise, which seems strange at first glance, is quite solid when it is considered that the page replacement algorithm will use swap space most when linearly scanning the process address space swapping out pages. Without scanning for large free blocks and using them, it is likely that the scanning would degenerate to first-free searches and never improve. With it, processes exiting are likely to free up large blocks of slots.

### 11.4 Swap Cache

Pages that are shared between many processes can not be easily swapped out because, as mentioned, there is no quick way to map a `struct page` to every PTE that
11.4 Swap Cache

references it. This leads to the race condition where a page is present for one PTE and swapped out for another gets updated without being synced to disk thereby losing the update.

To address this problem, shared pages that have a reserved slot in backing storage are considered to be part of the swap cache. The swap cache is purely conceptual as it is simply a specialisation of the page cache. The first principal difference between pages in the swap cache rather than the page cache is that pages in the swap cache always use `swapper_space` as their `address_space` in `page→mapping`. The second difference is that pages are added to the swap cache with `add_to_swap_cache()` instead of `add_to_page_cache()`.

Figure 11.3: Call Graph: `add_to_swap_cache()`

Anonymous pages are not part of the swap cache until an attempt is made to swap them out. The variable `swapper_space` is declared as follows in `swap_state.c`:

```c
struct address_space swapper_space = {
    LIST_HEAD_INIT(swapper_space.clean_pages),
    LIST_HEAD_INIT(swapper_space.dirty_pages),
    LIST_HEAD_INIT(swapper_space.locked_pages),
    0,
    &swap_aops,
};
```

A page is identified as being part of the swap cache once the `page→mapping` field has been set to `swapper_space` which is tested by the `PageSwapCache()` macro. Linux uses the exact same code for keeping pages between swap and memory in sync as it uses for keeping file-backed pages and memory in sync as they both share the page cache code, the differences are just in the functions used.

The address space for backing storage, `swapper_space` uses `swap_ops` for it's `address_space→a_ops`. The `page→index` field is then used to store the `swp_entry_t` structure instead of a file offset which is it's normal purpose. The `address_space_operations` struct `swap_aops` is declared as follows in `swap_state.c`:

```c
struct address_space_operations
```
11.4 Swap Cache

34 static struct address_space_operations swap_aops = {
35     writepage: swap_writepage,
36     sync_page: block_sync_page,
37 }; 

When a page is being added to the swap cache, a slot is allocated with get_swap_page(), added to the page cache with add_to_swap_cache() and then marked dirty. When the page is next laundered, it will actually be written to backing storage on disk as the normal page cache would operate. This process is illustrated in Figure 11.4.

Figure 11.4: Adding a Page to the Swap Cache

Subsequent swapping of the page from shared PTEs results in a call to swap_duplicate() which simply increments the reference to the slot in the swap_map. If the PTE is marked dirty by the hardware as a result of a write, the bit is cleared and the struct page is marked dirty with set_page_dirty() so that the on-disk copy will be synced before the page is dropped. This ensures that until all references to the page have been dropped, a check will be made to ensure the data on disk matches the data in the page frame.

When the reference count to the page finally reaches 0, the page is eligible to be dropped from the page cache and the swap map count will have the count of the number of PTEs the on-disk slot belongs to so that the slot will not be freed prematurely. It is laundered and finally dropped with the same LRU aging and logic described in Chapter 10.
If, on the other hand, a page fault occurs for a page that is ‘swapped out’, the logic in do_swap_page() will check to see if the page exists in the swap cache by calling lookup_swap_cache(). If it does, the PTE is updated to point to the page frame, the page reference count incremented and the swap slot decremented with swap_free().

```c
swp_entry_t get_swap_page()
This function allocates a slot in a swap_map by searching active swap areas. This is covered in greater detail in Section 11.3 but included here as it is principally used in conjunction with the swap cache.

int add_to_swap_cache(struct page *page, swp_entry_t entry)
This function adds a page to the swap cache. It first checks if it already exists by calling swap_duplicate() and if not, is adds it to the swap cache via the normal page cache interface function add_to_page_cache_unique().

struct page * lookup_swap_cache(swp_entry_t entry)
This searches the swap cache and returns the struct page corresponding to the supplied entry. It works by searching the normal page cache based on swapper_space and the swap_map offset.

int swap_duplicate(swp_entry_t entry)
This function verifies a swap entry is valid and if so, increments its swap map count.

void swap_free(swp_entry_t entry)
The complement function to swap_duplicate(). It decrements the relevant counter in the swap_map. When the count reaches zero, the slot is effectively free.
```

Table 11.1: Swap Cache API

### 11.5 Reading Pages from Backing Storage

The principal function used when reading in pages is read_swap_cache_async() which is mainly called during page faulting. The function begins by searching the swap cache with find_get_page(). Normally, swap cache searches are performed by lookup_swap_cache() but that function updates statistics on the number of searches performed and as the cache may need to be searched multiple times, find_get_page() is used instead.

The page can already exist in the swap cache if another process has the same page mapped or multiple processes are faulting on the same page at the same time. If the page does not exist in the swap cache, one must be allocated and filled with data from backing storage.
Once the page is allocated with `alloc_page()`, it is added to the swap cache with `add_to_swap_cache()` as swap cache operations may only be performed on pages in the swap cache. If the page cannot be added to the swap cache, the swap cache will be searched again to make sure another process has not put the data in the swap cache already.

To read information from backing storage, `rw_swap_page()` is called which is discussed in Section 11.7. Once the function completes, `page_cache_release()` is called to drop the reference to the page taken by `find_get_page()`.

### 11.6 Writing Pages to Backing Storage

When any page is being written to disk, the `address_space->a_ops` is consulted to find the appropriate write-out function. In the case of backing storage, the `address_space` is `swapper_space` and the swap operations are contained in `swap_aops`. The struct `swap_aops` registers `swap_writepage()` as its write-out function.

The function `swap_writepage()` behaves differently depending on whether the writing process is the last user of the swap cache page or not. It knows this by calling `remove_exclusive_swap_page()` which checks if there is any other processes using the page. This is a simple case of examining the page count with the `pagecache_lock` held. If no other process is mapping the page, it is removed from the swap cache and freed.

If `remove_exclusive_swap_page()` removed the page from the swap cache and freed it `swap_writepage()` will unlock the page as it is no longer in use. If it still exists in the swap cache, `rw_swap_page()` is called to write the data to the backing storage.
The top-level function for reading and writing to the swap area is `rw_swap_page()`. This function ensures that all operations are performed through the swap cache to prevent lost updates. `rw_swap_page_base()` is the core function which performs the real work.

It begins by checking if the operation is a read. If it is, it clears the uptodate flag with `ClearPageUptodate()` as the page is obviously not up to date if IO is required to fill it with data. This flag will be set again if the page is successfully read from disk. It then calls `get_swaphandle_info()` to acquire the device for the swap partition of the inoide for the swap file. These are required by the block layer which will be performing the actual IO.

The core function can work with either swap partition or files as it uses the block layer function `brw_page()` to perform the actual disk IO. If the swap area is a file, `bmap()` is used to fill a local array with a list of all blocks in the filesystem which contain the page data. Remember that filesystems may have their own method of storing files and disk and it is not as simple as the swap partition where information may be written directly to disk. If the backing storage is a partition, then only one page-sized block requires IO and as there is no filesystem involved, `bmap()` is unnecessary.

Once it is known what blocks must be read or written, a normal block IO operation takes place with `brw_page()`. All IO that is performed is asynchronous so the function returns quickly. Once the IO is complete, the block layer will unlock the page and any waiting process will wake up.
11.8 Activating a Swap Area

As it has now been covered what swap areas are, how they are represented and how pages are tracked, it is time to see how they all tie together to activate an area. Activating an area is conceptually quite simple; Open the file, load the header information from disk, populate a swap_info_struct and add it to the swap list.

The function responsible for the activation of a swap area is sys_swapon() and it takes two parameters, the path to the special file for the swap area and a set of flags. While swap is been activated, the Big Kernel Lock (BKL) is held which prevents any application entering kernel space while this operation is been performed. The function is quite large but can be broken down into the following simple steps;

- Find a free swap_info_struct in the swap_info array and initialise it with default values
- Call user_path_walk() which traverses the directory tree for the supplied special file and populates a namidata structure with the available data on the file, such as the dentry and the filesystem information for where it is stored (vfsmount)
- Populate swap_info_struct fields pertaining to the dimensions of the swap area and how to find it. If the swap area is a partition, the block size will be configured to the \texttt{PAGE\_SIZE} before calculating the size. If it is a file, the information is obtained directly from the inode
- Ensure the area is not already activated. If not, allocate a page from memory and read the first page sized slot from the swap area. This page contains information such as the number of good slots and how to populate the swap_info_struct→swap_map with the bad entries
- Allocate memory with vmalloc() for swap_info_struct→swap_map and initialise each entry with 0 for good slots and \texttt{SWAP\_MAP\_BAD} otherwise. Ideally the header information will be a version 2 file format as version 1 was limited to swap areas of just under 128MiB for architectures with 4KiB page sizes like the x86\footnote{See the Code Commentary for the comprehensive reason for this.}
- After ensuring the information indicated in the header matches the actual swap area, fill in the remaining information in the swap_info_struct such as the maximum number of pages and the available good pages. Update the global statistics for nr_swap_pages and total_swap_pages
- The swap area is now fully active and initialised and so it is inserted into the swap list in the correct position based on priority of the newly activated area

At the end of the function, the BKL is released and the system now has a new swap area available for paging to.
11.9 Deactivating a Swap Area

In comparison to activating a swap area, deactivation is incredibly expensive. The principal problem is that the area cannot be simply removed, every page that is swapped out must now be swapped back in again. Just as there is no quick way of mapping a `struct page` to every PTE that references it, there is no quick way to map a swap entry to a PTE either. This requires that all process page tables be traversed to find PTEs which reference the swap area to be deactivated and swap them in. This of course means that swap deactivation will fail if the physical memory is not available.

The function responsible for deactivating an area is, predictably enough, called `sys_swapoff()`. This function is mainly concerned with updating the `swap_info_struct`. The major task of paging in each paged-out page is the responsibility of `try_to_unuse()` which is extremely expensive. For each slot used in the `swap_map`, the page tables for processes have to be traversed searching for it. In the worst case, all page tables belonging to all `mm_structs` may have to be traversed. Therefore, the tasks taken for deactivating an area are broadly speaking;

- Call `user_path_walk()` to acquire the information about the special file to be deactivated and then take the BKL

- Remove the `swap_info_struct` from the swap list and update the global statistics on the number of swap pages available (`nr_swap_pages`) and the total number of swap entries (`total_swap_pages`). Once this is acquired, the BKL can be released again

- Call `try_to_unuse()` which will page in all pages from the swap area to be deactivated. This function loops through the swap map using `find_next_to_unuse()` to locate the next used swap slot. For each used slot it finds, it performs the following;

  - Call `read_swap_cache_async()` to allocate a page for the slot saved on disk. Ideally it exists in the swap cache already but the page allocator will be called if it is not

  - Wait on the page to be fully paged in and lock it. Once locked, call `unuse_process()` for every process that has a PTE referencing the page. This function traverses the page table searching for the relevant PTE and then updates it to point to the `struct page`. If the page is a shared memory page with no remaining reference, `shmem_unuse()` is called instead

  - Free all slots that were permanently mapped. It is believed that slots will never become permanently reserved so the risk is taken.

  - Delete the page from the swap cache to prevent `try_to_swap_out()` referencing a page in the event it still somehow has a reference in swap map
• If there was not enough available memory to page in all the entries, the swap area is reinserted back into the running system as it cannot be simply dropped. If it succeeded, the swap_info_struct is placed into an uninitialised state and the swap_map memory freed with vfree().

11.10 What's New in 2.6

The most important addition to the struct swap_info_struct is the addition of a linked list called extent_list and a cache field called curr_swap_extent for the implementation of extents.

Extents, which are represented by a struct swap_extent, map a contiguous range of pages in the swap area into a contiguous range of disk blocks. These extents are setup at swap on time by the function setup_swap_extents(). For block devices, there will only be one swap extent and it will not improve performance but the extent it setup so that swap areas backed by block devices or regular files can be treated the same.

It can make a large difference with swap files which will have multiple extents representing ranges of pages clustered together in blocks. When searching for the page at a particular offset, the extent list will be traversed. To improve search times, the last extent that was searched will be cached in swap_extent→curr_swap_extent.
Chapter 12

Shared Memory Virtual Filesystem

Sharing a region of memory backed by a file or device is simply a case of calling `mmap()` with the `MAP_SHARED` flag. However, there are two important cases where an anonymous region needs to be shared between processes. The first is when `mmap()` with `MAP_SHARED` but no file backing. These regions will be shared between a parent and child process after a `fork()` is executed. The second is when a region is explicitly setting them up with `shmget()` and attached to the virtual address space with `shmat()`.

When pages within a VMA are backed by a file on disk, the interface used is straightforward. To read a page during a page fault, the required `nopage()` function is found via `vm_area_struct→vm_ops`. To write a page to backing storage, the appropriate `writepage()` function is found in the `address_space_operations` via `inode→i_mapping→a_ops` or alternatively via `page→mapping→a_ops`. When normal file operations are taking place such as `mmap()`, `read()` and `write()`, the `struct file_operations` with the appropriate functions is found via `inode→i_fop` and so on. These relationships were illustrated in Figure 4.2.

This is a very clean interface that is conceptually easy to understand but it does not help anonymous pages as there is no file backing. To keep this nice interface, Linux creates an artificial file-backing for anonymous pages using a RAM-based filesystem where each VMA is backed by a “file” in this filesystem. Every inode in the filesystem is placed on a linked list called `shmem_inodes` so that they may always be easily located. This allows the same file-based interface to be used without treating anonymous pages as a special case.

The filesystem comes in two variations called `shm` and `tmpfs`. They both share core functionality and mainly differ in what they are used for. `shm` is for use by the kernel for creating file backings for anonymous pages and for backing regions created by `shmget()`. This filesystem is mounted by `kern_mount()` so that it is mounted internally and not visible to users. `tmpfs` is a temporary filesystem that may be optionally mounted on `/tmp/` to have a fast RAM-based temporary filesystem. A secondary use for `tmpfs` is to mount it on `/dev/shm/`. Processes that `mmap()` files in the `tmpfs` filesystem will be able to share information between them as an alternative to System V IPC mechanisms. Regardless of the type of use, `tmpfs` must be explicitly mounted by the system administrator.
This chapter begins with a description of how the virtual filesystem is implemented. From there we will discuss how shared regions are setup and destroyed before talking about how the tools are used to implement System V IPC mechanisms.

12.1 Initialising the Virtual Filesystem

The virtual filesystem is initialised by the function `init_tmpfs()` during either system start or when the module is begin loaded. This function registers the two filesystems, `tmpfs` and `shm`, mounts `shm` as an internal filesystem with `kern_mount()`. It then calculates the maximum number of blocks and inodes that can exist in the filesystems. As part of the registration, the function `shmem_read_super()` is used as a callback to populate a `struct super_block` with more information about the filesystems such as making the block size equal to the page size.

Figure 12.1: Call Graph: `init_tmpfs()`

Every inode created in the filesystem will have a `struct shmem_inode_info` associated with it which contains private information specific to the filesystem. The function `SHMEM_I()` takes an inode as a parameter and returns a pointer to a struct of this type. It is declared as follows in `<linux/shmem_fs.h>`:

```c
struct shmem_inode_info {
    spinlock_t lock;
    unsigned long next_index;
    swp_entry_t i_direct[SHMEM_NR_DIRECT];
    void **i_indirect;
    unsigned long swapped;
    unsigned long flags;
    struct list_head list;
    struct inode *inode;
};
```

The fields are:
lock is a spinlock protecting the inode information from concurrent accesses.

next_index is an index of the last page being used in the file. This will be different from inode→i_size while a file is being truncated.

i_direct is a direct block containing the first SHMEM_NR_DIRECT swap vectors in use by the file. See Section 12.4.1.

i_indirect is a pointer to the first indirect block. See Section 12.4.1.

swapped is a count of the number of pages belonging to the file that are currently swapped out.

flags is currently only used to remember if the file belongs to a shared region setup by shmget(). It is set by specifying SHM_LOCK with shmctl() and unlocked by specifying SHM_UNLOCK.

list is a list of all inodes used by the filesystem.

inode is a pointer to the parent inode.

12.2 Using shmem Functions

Different structs contain pointers for shmem specific functions. In all cases, tmpfs and shm share the same structs.

For faulting in pages and writing them to backing storage, two structs called shmem_aops and shmem_vm_ops of type struct address_space_operations and struct vm_operations_struct respectively are declared.

The address space operations struct shmem_aops contains pointers to a small number of functions of which the most important one is shmem_writepage() which is called when a page is moved from the page cache to the swap cache. shmem_removepage() is called when a page is removed from the page cache so that the block can be reclaimed. shmem_readpage() is not used by tmpfs but is provided so that the sendfile() system call my be used with tmpfs files. shmem_prepare_write() and shmem_commit_write() are also unused, but are provided so that tmpfs can be used with the loopback device. shmem_aops is declared as follows in mm/shmem.c

1500 static struct address_space_operations shmem_aops = {
1501          removepage: shmem_removepage,
1502          writepage:  shmem_writepage,
1503 #ifdef CONFIG_TMPFS
1504          readpage:  shmem_readpage,
1505          prepare_write:  shmem_prepare_write,
1506          commit_write:  shmem_commit_write,
1507 #endif
1508     };
Anonymous VMAs use `shmem_vm_ops` as it's `vm_operations_struct` so that `shmem_nopage()` is called when a new page is being faulted in. It is declared as follows:

```c
static struct vm_operations_struct shmem_vm_ops = {
    nopage: shmem_nopage,
};
```

To perform operations on files and inodes, two structs, `file_operations` and `inode_operations` are required. The `file_operations`, called `shmem_file_operations`, provides functions which implement `mmap()`, `read()`, `write()` and `fsync()`. It is declared as follows:

```c
static struct file_operations shmem_file_operations = {
    mmap: shmem_mmap,
    #ifdef CONFIG_TMPFS
    read: shmem_file_read,
    write: shmem_file_write,
    fsync: shmem_sync_file,
    #endif
};
```

Three sets of `inode_operations` are provided. The first is `shmem_inode_operations` which is used for file inodes. The second, called `shmem_dir_inode_operations` is for directories. The last pair, called `shmem_symlink_inline_operations` and `shmem_symlink_inode_operations` is for use with symbolic links.

The two file operations supported are `truncate()` and `setattr()` which are stored in a struct `inode_operations` called `shmem_inode_operations`. `shmem_truncate()` is used to truncate a file. `shmem_notify_change()` is called when the file attributes change. This allows, among other things, to allow a file to be grown with `truncate()` and use the global zero page as the data page. `shmem_inode_operations` is declared as follows:

```c
static struct inode_operations shmem_inode_operations = {
    truncate: shmem_truncate,
    setattr: shmem_notify_change,
};
```

The directory `inode_operations` provides functions such as `create()`, `link()` and `mkdir()`. They are declared as follows:
12.2 Using shmem Functions

```c
static struct inode_operations shmem_dir_inode_operations = {
    #ifdef CONFIG_TMPFS
    create: shmem_create,
    lookup: shmem_lookup,
    link: shmem_link,
    unlink: shmem_unlink,
    symlink: shmem_symlink,
    mkdir: shmem_mkdir,
    rmdir: shmem_rmdir,
    mknod: shmem_mknod,
    rename: shmem_rename,
    #endif
};
```

The last pair of operations are for use with symlinks. They are declared as:

```c
static struct inode_operations shmem_symlink_inline_operations = {
    readlink: shmem_readlink_inline,
    follow_link: shmem_follow_link_inline,
};
```

```c
static struct inode_operations shmem_symlink_inode_operations = {
    truncate: shmem_truncate,
    readlink: shmem_readlink,
    follow_link: shmem_follow_link,
};
```

The difference between the two readlink() and follow_link() functions is related to where the link information is stored. A symlink inode does not require the private inode information struct shmem_inode_information. If the length of the symbolic link name is smaller than this struct, the space in the inode is used to store the name and shmem_symlink_inline_operations becomes the inode operations struct. Otherwise a page is allocated with shmem_getpage(), the symbolic link is copied to it and shmem_symlink_inode_operations is used. The second struct includes a truncate() function so that the page will be reclaimed when the file is deleted.

These various structs ensure that the shmem equivalent of inode related operations will be used when regions are backed by virtual files. When they are used, the majority of the VM sees no difference between pages backed by a real file and ones backed by virtual files.
12.3 Creating Files in tmpfs

As tmpfs is mounted as a proper filesystem that is visible to the user, it must support directory inode operations such as `open()`, `mkdir()` and `link()`. Pointers to functions which implement these for tmpfs are provided in `shmem_dir_inode_operations` which was shown in Section 12.2.

The implementations of most of these functions are quite small and, at some level, they are all interconnected as can be seen from Figure 12.2. All of them share the same basic principal of performing some work with inodes in the virtual filesystem and the majority of the inode fields are filled in by `shmem_get_inode()`.

Figure 12.2: Call Graph: `shmem_create()`

When creating a new file, the top-level function called is `shmem_create()`. This small function calls `shmem_mknod()` with the `S_IFREG` flag added so that a regular file will be created. `shmem_mknod()` is little more than a wrapper
12.4 Page Faulting within a Virtual File

When a page fault occurs, `do_no_page()` will call `vma→vm_ops→nopage` if it exists. In the case of the virtual filesystem, this means the function `shmem_nopage()` , whose call graph is shown in Figure 12.3, will be called when a page fault occurs.

![Figure 12.3: Call Graph: shmem_nopage()](image)

The core function in this case is `shmem_getpage()` which is responsible for either allocating a new page or finding it in swap. This overloading of fault types is unusual as `do_swap_page()` is normally responsible for locating pages that have been moved to the swap cache or backing storage using information encoded within the PTE. In this case, pages backed by virtual files have their PTE set to 0 when they are moved to the swap cache. The inode’s private filesystem data stores direct and indirect block information which is used to locate the pages later. This operation is very similar in many respects to normal page faulting.

12.4.1 Locating Swapped Pages

When a page has been swapped out, a `swp_entry_t` will contain information needed to locate the page again. Instead of using the PTEs for this task, the information is stored within the filesystem-specific private information in the inode.

When faulting, the function called to locate the swap entry is `shmem_alloc_entry()`. It’s basic task is to perform basic checks and ensure that `shmem_inode_info→next_index` always points to the page index at the end of the virtual file. It’s principal task is to call `shmem_swp_entry()` which searches for the swap vector within the inode information with `shmem_swp_entry()` and allocate new pages as necessary to store swap vectors.

The first `SHMEM_NR_DIRECT` entries are stored in `inode→i_direct`. This means that for the x86, files that are smaller than 64KiB (`SHMEM_NR_DIRECT * PAGE_SIZE`)
will not need to use indirect blocks. Larger files must use indirect blocks starting with the one located at \texttt{inode}→\texttt{i\_indirect}.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure12_4.png}
\caption{Traversing Indirect Blocks in a Virtual File}
\end{figure}

The initial indirect block (\texttt{inode}→\texttt{i\_indirect}) is broken into two halves. The first half contains pointers to doubly indirect blocks and the second half contains pointers to triply indirect blocks. The doubly indirect blocks are pages containing swap vectors (\texttt{swp\_entry\_t}). The triple indirect blocks contain pointers to pages which in turn are filled with swap vectors. The relationship between the different levels of indirect blocks is illustrated in Figure 12.4. The relationship means that the maximum number of pages in a virtual file (\texttt{SHMEM\_MAX\_INDEX}) is defined as follows in \texttt{mm/shmem.c}:

\begin{verbatim}
44 #define SHMEM_MAX_INDEX  
    (SHMEM_NR_DIRECT + 
      (ENTRIES_PER_PAGE/2) * 
      (ENTRIES_PER_PAGE+1))
\end{verbatim}

\subsection{12.4.2 Writing Pages to Swap}

The function \texttt{shmem\_writepage()} is the registered function in the filesystems \texttt{address\_space\_operations} for writing pages to swap. The function is responsible for simply moving the page from the page cache to the swap cache. This is implemented with a few simple steps:
12.5 File Operations in tmpfs

Four operations, `mmap()`, `read()`, `write()` and `fsync()` are supported with virtual files. Pointers to the functions are stored in `shmem_file_operations` which was shown in Section 12.2.

There is little that is unusual in the implementation of these operations and they are covered in detail in the Code Commentary. The `mmap()` operation is implemented by `shmem_mmap()` and it simply updates the VMA that is managing the mapped region. `read()`, implemented by `shmem_read()`, performs the operation of copying bytes from the virtual file to a userspace buffer, faulting in pages as necessary. `write()`, implemented by `shmem_write()` is essentially the same. The `fsync()` operation is implemented by `shmem_file_sync()` but is essentially a NULL operation as it performs no task and simply returns 0 for success. As the files only exist in RAM, they do not need to be synchronised with any disk.

12.6 Inode Operations in tmpfs

The most complex operation that is supported for inodes is truncation and involves four distinct stages. The first, in `shmem_truncate()` will truncate the a partial page at the end of the file and continually calls `shmem_truncate_indirect()` until the file is truncated to the proper size. Each call to `shmem_truncate_indirect()` will only process one indirect block at each pass which is why it may need to be called multiple times.

The second stage, in `shmem_truncate_indirect()`, understands both doubly and triply indirect blocks. It finds the next indirect block that needs to be truncated. This indirect block, which is passed to the third stage, will contain pointers to pages which in turn contain swap vectors.

The third stage in `shmem_truncate_direct()` works with pages that contain swap vectors. It selects a range that needs to be truncated and passes the range to the last stage `shmem_swp_free()`. The last stage frees entries with `free_swap_and_cache()` which frees both the swap entry and the page containing data.

The linking and unlinking of files is very simple as most of the work is performed by the filesystem layer. To link a file, the directory inode size is incremented, the
ctime and mtime of the affected inodes is updated and the number of links to the
inode being linked to is incremented. A reference to the new dentry is then taken
with dget() before instantiating the new dentry with d_instantiate(). Unlinking
updates the same inode statistics before decrementing the reference to the dentry
with dput(). dput() will also call iput() which will clear up the inode when it’s
reference count hits zero.

Creating a directory will use shmem_mkdir() to perform the task. It simply
uses shmem_mknod() with the S_IFDIR flag before incrementing the parent directory
inode’s i_nlink counter. The function shmem_rmdir() will delete a directory by first
ensuring it is empty with shmem_empty(). If it is, the function then decrementing
the parent directory inode’s i_nlink count and calls shmem_unlink() to remove
the requested directory.

12.7 Setting up Shared Regions

A shared region is backed by a file created in shm. There are two cases where a new
file will be created, during the setup of a shared region with shmget() and when an
anonymous region is setup with mmap() with the MAP_SHARED flag. Both functions
use the core function shmem_file_setup() to create a file.

Figure 12.5: Call Graph: shmem_zero_setup()

As the filesystem is internal, the names of the files created do not have
to be unique as the files are always located by inode, not name. Therefore,
shmem_zero_setup() always says to create a file called dev/zero which is how it
shows up in the file /proc/pid/maps. Files created by shmget() are called SYSVNN
where the NN is the key that is passed as a parameter to shmget().

The core function shmem_file_setup() simply creates a new dentry and inode,
fills in the relevant fields and instantiates them.
12.8 System V IPC

The full internals of the IPC implementation is beyond the scope of this book. This section will focus just on the implementations of `shmget()` and `shmat()` and how they are affected by the VM. The system call `shmget()` is implemented by `sys_shmget()`. It performs basic checks to the parameters and sets up the IPC related data structures. To create the segment, it calls `newseg()`. This is the function that creates the file in `shmfs` with `shmem_file_setup()` as discussed in the previous section.

The system call `shmat()` is implemented by `sys_shmat()`. There is little remarkable about the function. It acquires the appropriate descriptor and makes sure all the parameters are valid before calling `do_mmap()` to map the shared region into the process address space. There are only two points of note in the function.

The first is that it is responsible for ensuring that VMAs will not overlap if the caller specifies the address. The second is that the `shp->shm_nattch` counter is maintained by a `vm_operations_struct` called `shm_vm_ops`. It registers `open()` and `close()` callbacks called `shm_open()` and `shm_close()` respectively. The `shm_close()` callback is also responsible for destroyed shared regions if the `SHM_DEST` flag is specified and the `shm_nattch` counter reaches zero.

12.9 What’s New in 2.6

The core concept and functionality of the filesystem remains the same and the changes are either optimisations or extensions to the filesystem’s functionality. If the reader understands the 2.4 implementation well, the 2.6 implementation will not present much trouble\(^1\).

A new field has been added to the `shmem_inode_info` called `allocated`. The `allocated` field stores how many data pages are allocated to the file which had to be calculated on the fly in 2.4 based on `inode->i_blocks`. It both saves a few clock cycles on a common operation as well as making the code a bit more readable.

\(^1\)I find that saying “How hard could it possibly be” always helps.
The flags field now uses the VM_ACCOUNT flag as well as the VM_LOCKED flag. The VM_ACCOUNT, always set, means that the VM will carefully account for the amount of memory used to make sure that allocations will not fail.

Extensions to the file operations are the ability to seek with the system call _llseek(), implemented by generic_file_llseek() and to use sendfile() with virtual files, implemented by shmem_file_sendfile(). An extension has been added to the VMA operations to allow non-linear mappings, implemented by shmem_populate().

The last major change is that the filesystem is responsible for the allocation and destruction of its own inodes which are two new callbacks in struct super_operations. It is simply implemented by the creation of a slab cache called shmem_inode_cache. A constructor function init_once() is registered for the slab allocator to use for initialising each new inode.
Chapter 13

Out Of Memory Management

The last aspect of the VM we are going to discuss is the Out Of Memory (OOM) manager. This intentionally is a very short chapter as it has one simple task; check if there is enough available memory to satisfy, verify that the system is truly out of memory and if so, select a process to kill. This is a controversial part of the VM and it has been suggested that it be removed on many occasions. Regardless of whether it exists in the latest kernel, it still is a useful system to examine as it touches off a number of other subsystems.

13.1 Checking Available Memory

For certain operations, such as expanding the heap with brk() or remapping an address space with mremap(), the system will check if there is enough available memory to satisfy a request. Note that this is separate to the out_of_memory() path that is covered in the next section. This path is used to avoid the system being in a state of OOM if at all possible.

When checking available memory, the number of required pages is passed as a parameter to vm_enough_memory(). Unless the system administrator has specified that the system should overcommit memory, the amount of available memory will be checked. To determine how many pages are potentially available, Linux sums up the following bits of data:

- **Total page cache** as page cache is easily reclaimed
- **Total free pages** because they are already available
- **Total free swap pages** as userspace pages may be paged out
- **Total pages managed by swapper_space** although this double-counts the free swap pages. This is balanced by the fact that slots are sometimes reserved but not used
- **Total pages used by the dentry cache** as they are easily reclaimed
13.2 Determining OOM Status

When the machine is low on memory, old page frames will be reclaimed (see Chapter 10) but despite reclaiming pages is may find that it was unable to free enough pages to satisfy a request even when scanning at highest priority. If it does fail to free page frames, out_of_memory() is called to see if the system is out of memory and needs to kill a process.

Unfortunately, it is possible that the system is not out memory and simply needs to wait for IO to complete or for pages to be swapped to backing storage. This is unfortunate, not because the system has memory, but because the function is being called unnecessarily opening the possibly of processes being unnecessarily killed. Before deciding to kill a process, it goes through the following checklist.

- Is there enough swap space left (nr_swap_pages > 0) ? If yes, not OOM
• Has it been more than 5 seconds since the last failure? If yes, not OOM
• Have we failed within the last second? If no, not OOM
• If there hasn’t been 10 failures at least in the last 5 seconds, we’re not OOM
• Has a process been killed within the last 5 seconds? If yes, not OOM

It is only if the above tests are passed that oom_kill() is called to select a process to kill.

13.3 Selecting a Process

The function select_bad_process() is responsible for choosing a process to kill. It decides by stepping through each running task and calculating how suitable it is for killing with the function badness(). The badness is calculated as follows, note that the square roots are integer approximations calculated with int_sqrt();

\[
\text{badness}_{\text{for_task}} = \frac{\text{total_vm}_{\text{for_task}}}{\sqrt{\text{cpu_time_{in_seconds}}} \times \sqrt{\text{cpu_time_{in_minutes}}}}
\]

This has been chosen to select a process that is using a large amount of memory but is not that long lived. Processes which have been running a long time are unlikely to be the cause of memory shortage so this calculation is likely to select a process that uses a lot of memory but has not been running long. If the process is a root process or has CAP_SYS_ADMIN capabilities, the points are divided by four as it is assumed that root privilege processes are well behaved. Similarly, if it has CAP_SYS_RAWIO capabilities (access to raw devices) privileges, the points are further divided by 4 as it is undesirable to kill a process that has direct access to hardware.

13.4 Killing the Selected Process

Once a task is selected, the list is walked again and each process that shares the same mm_struct as the selected process (i.e. they are threads) is sent a signal. If the process has CAP_SYS_RAWIO capabilities, a SIGTERM is sent to give the process a chance of exiting cleanly, otherwise a SIGKILL is sent.

13.5 Is That It?

Yes, that’s it, out of memory management touches a lot of subsystems otherwise, there is not much to it.
13.6 What’s New in 2.6

The majority of OOM management remains essentially the same for 2.6 except for the introduction of VM accounted objects. These are VMAs that are flagged with the `VM_ACCOUNT` flag, first mentioned in Section 4.8. Additional checks will be made to ensure there is memory available when performing operations on VMAs with this flag set. The principal incentive for this complexity is to avoid the need of an OOM killer.

Some regions which always have the `VM_ACCOUNT` flag set are the process stack, the process heap, regions `mmap()`ed with `MAP_SHARED`, private regions that are writable and regions set up `shmget()`. In other words, most userspace mappings have the `VM_ACCOUNT` flag set.

Linux accounts for the amount of memory that is committed to these VMAs with `vm_acct_memory()` which increments a variable called `committed_space`. When the VMA is freed, the committed space is decremented with `vm_unacct_memory()`. This is a fairly simple mechanism, but it allows Linux to remember how much memory it has already committed to userspace when deciding if it should commit more.

The checks are performed by calling `security_vm_enough_memory()` which introduces us to another new feature. 2.6 has a feature available which allows security related kernel modules to override certain kernel functions. The full list of hooks available is stored in a `struct security_operations` called `security_ops`. There are a number of dummy, or default, functions that may be used which are all listed in `security/dummy.c` but the majority do nothing except return. If there are no security modules loaded, the `security_operations` struct used is called `dummy_security_ops` which uses all the default function.

By default, `security_vm_enough_memory()` calls `dummy_vm_enough_memory()` which is declared in `security/dummy.c` and is very similar to 2.4’s `vm_enough_memory()` function. The new version adds the following pieces of information together to determine available memory:

- **Total page cache** as page cache is easily reclaimed
- **Total free pages** because they are already available
- **Total free swap pages** as userspace pages may be paged out
- **Slab pages with SLAB_RECLAIM_ACCOUNT set** as they are easily reclaimed

These pages, minus a 3% reserve for root processes, is the total amount of memory that is available for the request. If the memory is available, it makes a check to ensure the total amount of committed memory does not exceed the allowed threshold. The allowed threshold is \( \text{TotalRam} \times (\text{OverCommitRatio}/100) + \text{TotalSwapPage} \), where \( \text{OverCommitRatio} \) is set by the system administrator. If the total amount of committed space is not too high, 1 will be returned so that the allocation can proceed.
Chapter 14

The Final Word

Make no mistake, memory management is a large, complex and time consuming field to research and difficult to apply to practical implementations. As it is very difficult to model how systems behave in real multi-programmed systems [CD80], developers often rely on intuition to guide them and examination of virtual memory algorithms depends on simulations of specific workloads. Simulations are necessary as modeling how scheduling, paging behaviour and multiple processes interact presents a considerable challenge. Page replacement policies, a field that has been the focus of considerable amounts of research, is a good example as it is only ever shown to work well for specified workloads. The problem of adjusting algorithms and policies to different workloads is addressed by having administrators tune systems as much as by research and algorithms.

The Linux kernel is also large, complex and fully understood by a relatively small core group of people. It’s development is the result of contributions of thousands of programmers with a varying range of specialties, backgrounds and spare time. The first implementations are developed based on the all-important foundation that theory provides. Contributors built upon this framework with changes based on real world observations.

It has been asserted on the Linux Memory Management mailing list that the VM is poorly documented and difficult to pick up as “the implementation is a nightmare to follow” and the lack of documentation on practical VMs is not just confined to Linux. Matt Dillon, one of the principal developers of the FreeBSD VM and considered a “VM Guru” stated in an interview that documentation can be “hard to come by”. One of the principal difficulties with deciphering the implementation is the fact the developer must have a background in memory management theory to see why implementation decisions were made as a pure understanding of the code is insufficient for any purpose other than micro-optimisations.

This book attempted to bridge the gap between memory management theory and the practical implementation in Linux and tie both fields together in a single

\[1\text{http://mail.nl.linux.org/linux-mm/2002-05/msg00035.html}\]
\[2\text{His past involvement with the Linux VM is evident from http://mail.nl.linux.org/linux-mm/2000-05/msg00419.html}\]
\[3\text{http://kerneltrap.com/node.php?id=8}\]
place. It tried to describe what life is like in Linux as a memory manager in a manner that was relatively independent of hardware architecture considerations. I hope after reading this, and progressing onto the code commentary, that you, the reader feels a lot more comfortable with tackling the VM subsystem. As a final parting shot, Figure 14.1 broadly illustrates how of the sub-systems we discussed in detail interact with each other.

On a final personal note, I hope that this book encourages other people to produce similar works for other areas of the kernel. I know I'll buy them!

Figure 14.1: Broad Overview on how VM Sub-Systems Interact
Appendix A

Introduction

Welcome to the code commentary section of the book. If you are reading this, you are looking for a heavily detailed tour of the code. The commentary presumes you have read the equivalent section in the main part of the book so if you just started reading here, you’re probably in the wrong place.

Each appendix section corresponds to the order and structure as the book. The order the functions are presented is the same order as displayed in the call graphs which are referenced throughout the commentary. At the beginning of each appendix and subsection, there is a mini table of contents to help navigate your way through the commentary. The code coverage is not 100% but all the principal code patterns that are found throughout the VM may be found. If the function you are interested in is not commented on, try and find a similar function to it.

Some of the code has been reformatted slightly for presentation but the actual code is not changed. It is recommended you use the companion CD while reading the code commentary. In particular use LXR to browse through the source code so you get a “feel” for reading the code with and without the aid of the commentary.

Good Luck!
Appendix B

Describing Physical Memory

Contents

B.1 Initialising Zones .................................................. 202
  B.1.1 Function: setup_memory() ................................... 202
  B.1.2 Function: zone_sizes_init() ............................... 205
  B.1.3 Function: free_area_init() ................................. 206
  B.1.4 Function: free_area_init_node() ......................... 206
  B.1.5 Function: free_area_init_core() ......................... 208
  B.1.6 Function: build_zonelists() ............................... 214
B.2 Page Operations .................................................. 216
  B.2.1 Locking Pages ............................................... 216
    B.2.1.1 Function: lock_page() ................................ 216
    B.2.1.2 Function: __lock_page() ............................. 216
    B.2.1.3 Function: sync_page() ................................ 217
  B.2.2 Unlocking Pages ............................................. 218
    B.2.2.1 Function: unlock_page() ............................. 218
  B.2.3 Waiting on Pages ............................................ 219
    B.2.3.1 Function: wait_on_page() ............................ 219
    B.2.3.2 Function: __wait_on_page() .......................... 219
B.1 Initialising Zones

Contents

B.1 Initialising Zones 202
B.1.1 Function: setup_memory() 202
B.1.2 Function: zone_sizes_init() 205
B.1.3 Function: free_area_init() 206
B.1.4 Function: free_area_init_node() 206
B.1.5 Function: free_area_init_core() 208
B.1.6 Function: build_zonelists() 214

B.1.1 Function: setup_memory() (arch/i386/kernel/setup.c)

The call graph for this function is shown in Figure 2.3. This function gets the necessary information to give to the boot memory allocator to initialise itself. It is broken up into a number of different tasks.

- Find the start and ending PFN for low memory (min_low_pfn, max_low_pfn), the start and end PFN for high memory (highstart_pfn, highend_pfn) and the PFN for the last page in the system (max_pfn).

- Initialise the bootmem_data structure and declare which pages may be used by the boot memory allocator.

- Mark all pages usable by the system as “free” and then reserve the pages used by the bitmap representing the pages.

- Reserve pages used by the SMP config or the initrd image if one exists.

```c
static unsigned long __init setup_memory(void)
{
    unsigned long bootstrap_size, start_pfn, max_low_pfn;
    /* partially used pages are not usable - thus
    * we are rounding upwards:
    */
    start_pfn = PFN_UP(__pa(&_end));
    find_max_pfn();
    max_low_pfn = find_max_low_pfn();

    #ifdef CONFIG_HIGHMEM
    highstart_pfn = highend_pfn = max_pfn;
    if (max_pfn > max_low_pfn) {
        highstart_pfn = max_low_pfn;
    }
    ```
B.1 Initialising Zones (setup_memory())

```c
B.1 Initialising Zones (setup_memory())

1009     }
1010     printk(KERN_NOTICE "%ldMB HIGHMEM available.
",  
1011     pages_to_mb(highend_pfn - highstart_pfn));
1012 #endif
1013     printk(KERN_NOTICE "%ldMB LOWMEM available.
",  
1014     pages_to_mb(max_low_pfn));

PFN_UP() takes a physical address, rounds it up to the next page and returns  
the page frame number. _end is the address of the end of the loaded kernel  
image so start_pfn is now the offset of the first physical page frame that may  
be used

1001 find_max_pfn() loops through the e820 map searching for the highest avail-  
able pfn

1003 find_max_low_pfn() finds the highest page frame addressable in ZONE_NORMAL  

1005-1011 If high memory is enabled, start with a high memory region of 0. If it  
turns out there is memory after max_low_pfn, put the start of high memory  
(highstart_pfn) there and the end of high memory at max_pfn. Print out an  
informational message on the availability of high memory

1013-1014 Print out an informational message on the amount of low memory

1018     bootstrap_size = init_bootmem(start_pfn, max_low_pfn);
1019     register_bootmem_low_pages(max_low_pfn);
1020     reserve_bootmem(HIGH_MEMORY, (PFN_PHYS(start_pfn) +  
1021     bootstrap_size + PAGE_SIZE-1) - (HIGH_MEMORY));
1028     reserve_bootmem(0, PAGE_SIZE);
1036 #ifdef CONFIG_SMP
1043     reserve_bootmem(PAGE_SIZE, PAGE_SIZE);
1044 #endif
1045 #ifdef CONFIG_ACPI_SLEEP
1046     /*
1047     * Reserve low memory region for sleep support.
1048     */
1049     acpi_reserve_bootmem();
1050 #endif

1018 init_bootmem() (See Section E.1.1) initialises the bootmem_data struct for  
the config_page_data node. It sets where physical memory begins and ends  
for the node, allocates a bitmap representing the pages and sets all pages as  
reserved initially
```
register_bootmem_low_pages() reads the e820 map and calls free_bootmem() (See Section E.3.1) for all usable pages in the running system. This is what marks the pages marked as reserved during initialisation as free.

Reserve the pages that are being used to store the bitmap representing the pages.

Reserve page 0 as it is often a special page used by the bios.

Reserve an extra page which is required by the trampoline code. The trampoline code deals with how userspace enters kernel space.

If sleep support is added, reserve memory required for it. This is only of interest to laptops interested in suspending and beyond the scope of this book.

This function reserves memory that stores config information about the SMP setup.
If initrd is enabled, the memory containing its image will be reserved. initrd provides a tiny filesystem image which is used to boot the system.

Return the upper limit of addressable memory in ZONE_NORMAL.

This is the top-level function which is used to initialise each of the zones. The size of the zones in PFNs was discovered during setup_memory() (See Section B.1.1). This function populates an array of zone sizes for passing to free_area_init().

Initialise the sizes to 0

Calculate the PFN for the maximum possible DMA address. This doubles up as the largest number of pages that may exist in ZONE_DMA.

max_low_pfn is the highest PFN available to ZONE_NORMAL.

highend_pfn is the highest PFN available to ZONE_HIGHMEM.

If the highest PFN in ZONE_NORMAL is below MAX_DMA_ADDRESS, then just set the size of ZONE_DMA to it. The other zones remain at 0.

Set the number of pages in ZONE_DMA.

The size of ZONE_NORMAL is max_low_pfn minus the number of pages in ZONE_DMA.
B.1 Initialising Zones \((\text{zone\_sizes\_init})\)

338 The size of \text{ZONE\_HIGHMEM} is the highest possible PFN minus the highest possible PFN in \text{ZONE\_NORMAL} \((\text{max\_low\_pfm})\)

B.1.3 Function: \text{free\_area\_init()} \((\text{mm/page\_alloc.c})\)

This is the architecture independent function for setting up a UMA architecture. It simply calls the core function passing the static \text{contig\_page\_data} as the node. NUMA architectures will use \text{free\_area\_init\_node()} instead.

838 void \_\_init free\_area\_init(unsigned long *zones\_size)
839 {
840 free\_area\_init\_core(0, &contig\_page\_data, &mem\_map, zones\_size, 0, 0, 0);
841 }

838 The parameters passed to \text{free\_area\_init\_core()} are

\(0\) is the Node Identifier for the node, which is 0

\text{contig\_page\_data} is the static global \text{pg\_data\_t}

\text{mem\_map} is the global \text{mem\_map} used for tracking \text{struct pages}. The function \text{free\_area\_init\_core()} will allocate memory for this array

\text{zones\_sizes} is the array of zone sizes filled by \text{zone\_sizes\_init()}

\(0\) This zero is the starting physical address

\(0\) The second zero is an array of memory hole sizes which doesn’t apply to UMA architectures

\(0\) The last 0 is a pointer to a local \text{mem\_map} for this node which is used by NUMA architectures

B.1.4 Function: \text{free\_area\_init\_node()} \((\text{mm/numa.c})\)

There are two versions of this function. The first is almost identical to \text{free\_area\_init()} except it uses a different starting physical address. There is for architectures that have only one node (so they use \text{contig\_page\_data}) but whose physical address is not at 0.

This version of the function, called after the pagetable initialisation, if for initialisation each pgdat in the system. The caller has the option of allocating their own local portion of the \text{mem\_map} and passing it in as a parameter if they want to optimise it’s location for the architecture. If they choose not to, it will be allocated later by \text{free\_area\_init\_core()}.

61 void \_\_init free\_area\_init\_node(int nid,
\hspace{1em} pg\_data\_t *pgdat, struct page *pmap,
62 unsigned long *zones\_size, unsigned long zone\_start\_paddr,
63 unsigned long *zholes\_size)
64 {

The parameters to the function are:

nid is the Node Identifier (NID) of the pgdat passed in

pgdat is the node to be initialised

pmap is a pointer to the portion of the mem_map for this node to use, frequently passed as NULL and allocated later

zones_size is an array of zone sizes in this node

zone_start_paddr is the starting physical address for the node

zholes_size is an array of hole sizes in each zone

68-69 If the global mem_map has not been set, set it to the beginning of the kernel portion of the linear address space. Remember that with NUMA, mem_map is a virtual array with portions filled in by local maps used by each node

71 Call free_area_init_core(). Note that discard is passed in as the third parameter as no global mem_map needs to be set for NUMA

73 Record the pgdat's NID

78-79 Calculate the total size of the node

80 Recalculate size as the number of bits requires to have one bit for every byte of the size
Allocate a bitmap to represent where valid areas exist in the node. In reality, this is only used by the sparc architecture so it is unfortunate to waste the memory every other architecture

Initially, all areas are invalid. Valid regions are marked later in the mem_init() functions for the sparc. Other architectures just ignore the bitmap

**Function: free_area_init_core()** *(mm/page_alloc.c)*

This function is responsible for initialising all zones and allocating their local lmem_map within a node. In UMA architectures, this function is called in a way that will initialise the global mem_map array. In NUMA architectures, the array is treated as a virtual array that is sparsely populated.

```c
void __init free_area_init_core(int nid, pg_data_t *pgdat, struct page **gmap,
unsigned long *zones_size, unsigned long zone_start_paddr,
unsigned long *zholes_size, struct page *lmem_map)
{
unsigned long i, j;
unsigned long map_size;
unsigned long totalpages, offset, realtotalpages;
const unsigned long zone_required_alignment =
1UL << (MAX_ORDER-1);
if (zone_start_paddr & ~PAGE_MASK)
    BUG();
totalpages = 0;
for (i = 0; i < MAX_NR_ZONES; i++) {
    unsigned long size = zones_size[i];
    totalpages += size;
}
realtotalpages = totalpages;
if (zholes_size)
    for (i = 0; i < MAX_NR_ZONES; i++)
        realtotalpages -= zholes_size[i];
printk("On node %d totalpages: %lu\n", nid, realtotalpages);
```

This block is mainly responsible for calculating the size of each zone.

The zone must be aligned against the maximum sized block that can be allocated by the buddy allocator for bitwise operations to work

It is a bug if the physical address is not page aligned
B.1 Initialising Zones (*free_area_init_core()*)

696 Initialise the totalpages count for this node to 0

697-700 Calculate the total size of the node by iterating through zone_sizes

701-704 Calculate the real amount of memory by subtracting the size of the holes in zholes_size

706 Print an informational message for the user on how much memory is available in this node

707-714 Some architectures (with lots of mem and discontinuous memory maps) have to search for a good mem_map area:

715-727 For discontigmem, the conceptual mem map array starts from PAGE_OFFSET, we need to align the actual array onto a mem map boundary, so that MAP_NR works.

715 map_size = (totalpages + 1)*sizeof(struct page);
716 if (lmem_map == (struct page *)0) {
    lmem_map = (struct page *) alloc_bootmem_node(pgdat, map_size);
    lmem_map = (struct page *)(PAGE_OFFSET + MAP_ALIGN((unsigned long)lmem_map - PAGE_OFFSET));

    *gmap = pgdat->node_mem_map = lmem_map;
    pgdat->node_size = totalpages;
    pgdat->node_start_paddr = zone_start_paddr;
    pgdat->node_start_mapnr = (lmem_map - mem_map);
    pgdat->nr_zones = 0;

    offset = lmem_map - mem_map;
}

This block allocates the local lmem_map if necessary and sets the gmap. In UMA architectures, gmap is actually mem_map and so this is where the memory for it is allocated

715 Calculate the amount of memory required for the array. It is the total number of pages multiplied by the size of a struct page

716 If the map has not already been allocated, allocate it

717 Allocate the memory from the boot memory allocator

718 MAP_ALIGN() will align the array on a struct page sized boundary for calculations that locate offsets within the mem_map based on the physical address with the MAP_NR() macro

721 Set the gmap and pgdat->node_mem_map variables to the allocated lmem_map.
In UMA architectures, this just set mem_map
B.1 Initialising Zones (*free_area_init_core()*)

722 Record the size of the node

723 Record the starting physical address

724 Record what the offset within mem_map this node occupies

725 Initialise the zone count to 0. This will be set later in the function

727 offset is now the offset within mem_map that the local portion lmem_map begins at

728 for (j = 0; j < MAX_NR_ZONES; j++) {
729   zone_t *zone = pgdat->node_zones + j;
730   unsigned long mask;
731   unsigned long size, realsize;
732   zone_table[nid * MAX_NR_ZONES + j] = zone;
733   realsize = size = zones_size[j];
734   if (zholes_size)
735     realsize -= zholes_size[j];
736   printk("zone(%lu): %lu pages.\n", j, size);
737   zone->size = size;
738   zone->name = zone_names[j];
739   zone->lock = SPIN_LOCK_UNLOCKED;
740   zone->zone_pgdat = pgdat;
741   zone->free_pages = 0;
742   zone->need_balance = 0;
743   if (!size)
744     continue;
745   zone->size = realsize;
746   zone_table[nid * MAX_NR_ZONES + j] = zone;
747   zone->size = size;
748   zone->name = zone_names[j];
749   zone->lock = SPIN_LOCK_UNLOCKED;
750   zone->zone_pgdat = pgdat;
751   zone->free_pages = 0;
752   zone->need_balance = 0;
753   if (!size)
754     continue;

This block starts a loop which initialises every zone_t within the node. The initialisation starts with the setting of the simpler fields that values already exist for.

728 Loop through all zones in the node

733 Record a pointer to this zone in the zone_table. See Section 2.4.1

734-736 Calculate the real size of the zone based on the full size in zones_size minus the size of the holes in zholes_size

738 Print an informational message saying how many pages are in this zone

739 Record the size of the zone

740 zone_names is the string name of the zone for printing purposes

741-744 Initialise some other fields for the zone such as it’s parent pgdat
B.1 Initialising Zones (free_area_init_core())

745-746 If the zone has no memory, continue to the next zone as nothing further is required.

752-754 Calculate the shift for the hashing algorithm.

755-758 Allocate a table of wait_queue_head_t that can hold zone->wait_table_size entries.

759-760 Initialise all of the wait queues.

773-774 Check for correct zone alignment.

778-779 Print a warning message if the alignment is incorrect.

779

Calculate the watermarks for the zone and record the location of the zone. The watermarks are calculated as ratios of the zone size.

First, as a new zone is active, update the number of zones in this node

Calculate the mask (which will be used as the \texttt{pages\_min} watermark) as the size of the zone divided by the balance ratio for this zone. The balance ratio is 128 for all zones as declared at the top of \texttt{mm/page\_alloc.c}

The \texttt{zone\_balance\_min} ratios are 20 for all zones so this means that \texttt{pages\_min} will never be below 20

Similarly, the \texttt{zone\_balance\_max} ratios are all 255 so \texttt{pages\_min} will never be over 255

\texttt{pages\_min} is set to \texttt{mask}

\texttt{pages\_low} is twice the number of pages as \texttt{pages\_min}

\texttt{pages\_high} is three times the number of pages as \texttt{pages\_min}

Record where the first \texttt{struct page} for this zone is located within \texttt{mem\_map}

Record the index within \texttt{mem\_map} this zone begins at

Record the starting physical address

Ensure that the zone is correctly aligned for use with the buddy allocator otherwise the bitwise operations used for the buddy allocator will break

\begin{verbatim}
/*
 * Initially all pages are reserved - free ones are freed
 * up by free\_all\_bootmem() once the early boot process is
 * done. Non-atomic initialization, single-pass.
 */
for (i = 0; i < size; i++) {
    struct page *page = mem_map + offset + i;
    set_page_zone(page, nid * MAX_NR_zones + j);
    set_page_count(page, 0);
    SetPageReserved(page);
    INIT\_LIST\_HEAD(&page->list);
    if (j != ZONE\_HIGH\_MEM)
        set_page_address(page, __va(zone_start_paddr));
    zone_start_paddr += PAGE\_SIZE;
}
\end{verbatim}
B.1 Initialising Zones (*free_area_init_core()*)

785-794 Initially, all pages in the zone are marked as reserved as there is no way to know which ones are in use by the boot memory allocator. When the boot memory allocator is retiring in *free_all_bootmem()*, the unused pages will have their `PG_reserved` bit cleared.

786 Get the page for this offset.

787 The zone the page belongs to is encoded with the page flags. See Section 2.4.1

788 Set the count to 0 as no one is using it.

789 Set the reserved flag. Later, the boot memory allocator will clear this bit if the page is no longer in use.

790 Initialise the list head for the page.

791-792 Set the `page->virtual` field if it is available and the page is in low memory.

793 Increment `zone_start_paddr` by a page size as this variable will be used to record the beginning of the next zone.

```
796    offset += size;
797    for (i = 0; ; i++) {
798         unsigned long bitmap_size;
799
800        INIT_LIST_HEAD(&zone->free_area[i].free_list);
801        if (i == MAX_ORDER-1) {
802            zone->free_area[i].map = NULL;
803            break;
804        }
805
829        bitmap_size = (size-1) >> (i+4);
830        bitmap_size = LONG_ALIGN(bitmap_size+1);
831        zone->free_area[i].map =
832            (unsigned long *) alloc_bootmem_node(pgd, bitmap_size);
833    }
834 }
835 build_zonelists(pgd);
836 }
```

This block initialises the free lists for the zone and allocates the bitmap used by the buddy allocator to record the state of page buddies.

797 This will loop from 0 to `MAX_ORDER-1`.

800 Initialise the linked list for the `free_list` of the current order `i`. 
801-804 If this is the last order, then set the free area map to NULL as this is what marks the end of the free lists.

829 Calculate the bitmap_size to be the number of bytes required to hold a bitmap where each bit represents on pair of buddies that are \(2^i\) number of pages.

830 Align the size to a long with \texttt{LONG\_ALIGN()} as all bitwise operations are on longs.

831-832 Allocate the memory for the map.

834 This loops back to move to the next zone.

835 Build the zone fallback lists for this node with \texttt{build_zonelists()}.

\textbf{B.1.6 Function: build_zonelists()} \texttt{(mm/page_alloc.c)}

This builds the list of fallback zones for each zone in the requested node. This is for when an allocation cannot be satisfied and another zone is consulted. When this is finished, allocations from \texttt{ZONE\_HIGHMEM} will fallback to \texttt{ZONE\_NORMAL}. Allocations from \texttt{ZONE\_NORMAL} will fall back to \texttt{ZONE\_DMA} which in turn has nothing to fall back on.

589 static inline void build_zonelists(pg_data_t *pgdat)
590 {
591    int i, j, k;
592
593    for (i = 0; i <= GFP\_ZONEMASK; i++) {
594        zonelist_t *zonelist;
595        zone_t *zone;
596
597        zonelist = pgdat->node_zonelists + i;
598        memset(zonelist, 0, sizeof(*zonelist));
599
600        j = 0;
601        k = ZONE\_NORMAL;
602        if (i & __GFP\_HIGHMEM)
603            k = ZONE\_HIGHMEM;
604        if (i & __GFP\_DMA)
605            k = ZONE\_DMA;
606
607        switch (k) {
608            default:
609                BUG();
610            /*
611                * fallthrough:

This looks through the maximum possible number of zones

Get the zonelist for this zone and zero it

Start \( j \) at 0 which corresponds to ZONE_DMA

Set \( k \) to be the type of zone currently being examined

Get the ZONE_HIGHMEM

If the zone has memory, then ZONE_HIGHMEM is the preferred zone to allocate from for high memory allocations. If ZONE_HIGHMEM has no memory, then ZONE_NORMAL will become the preferred zone when the next case is fallen through to as \( j \) is not incremented for an empty zone

Set the next preferred zone to allocate from to be ZONE_NORMAL. Again, do not use it if the zone has no memory

Set the final fallback zone to be ZONE_DMA. The check is still made for ZONE_DMA having memory as in a NUMA architecture, not all nodes will have a ZONE_DMA
B.2 Page Operations

B.2.1 Locking Pages

B.2.1.1 Function: lock_page() (mm/filemap.c)
This function tries to lock a page. If the page cannot be locked, it will cause the process to sleep until the page is available.

```c
void lock_page(struct page *page)
{
    if (TryLockPage(page))
        __lock_page(page);
}
```

TryLockPage() is just a wrapper around test_and_set_bit() for the PG_locked bit in page->flags. If the bit was previously clear, the function returns immediately as the page is now locked.

Otherwise call __lock_page() (See Section B.2.1.2) to put the process to sleep.

B.2.1.2 Function: __lock_page() (mm/filemap.c)
This is called after a TryLockPage() failed. It will locate the waitqueue for this page and sleep on it until the lock can be acquired.

```c
static void __lock_page(struct page *page)
{
    wait_queue_head_t *waitqueue = page_waitqueue(page);
    struct task_struct *tsk = current;
    DECLARE_WAITQUEUE(wait, tsk);
    add_wait_queue_exclusive(waitqueue, &wait);
    for (;;)
    {
        set_task_state(tsk, TASK_UNINTERRUPTIBLE);
    }
```
B.2.1 Locking Pages (_lock_page())

906  if (PageLocked(page)) {
907      sync_page(page);
908      schedule();
909  }
910  if (!TryLockPage(page))
911      break;
912 }
913  __set_task_state(tsk, TASK_RUNNING);
914  remove_wait_queue(waitqueue, &wait);
915 }

899 page_waitqueue() is the implementation of the hash algorithm which determines which wait queue this page belongs to in the table zone->wait_table

900-901 Initialise the waitqueue for this task

903 Add this process to the waitqueue returned by page_waitqueue()

904-912 Loop here until the lock is acquired

905 Set the process states as being in uninterruptible sleep. When schedule() is called, the process will be put to sleep and will not wake again until the queue is explicitly woken up

906 If the page is still locked then call sync_page() function to schedule the page to be synchronised with it’s backing storage. Call schedule() to sleep until the queue is woken up such as when IO on the page completes

910-1001 Try and lock the page again. If we succeed, exit the loop, otherwise sleep on the queue again

913-914 The lock is now acquired so set the process state to TASK_RUNNING and remove it from the wait queue. The function now returns with the lock acquired

B.2.1.3 Function: sync_page() (mm/filemap.c)

This calls the filesystem-specific sync_page() to synchronise the page with it’s backing storage.

140 static inline int sync_page(struct page *page)
141 {
142    struct address_space *mapping = page->mapping;
143
144    if (mapping && mapping->a_ops && mapping->a_ops->sync_page)
145      return mapping->a_ops->sync_page(page);
146  return 0;
147 }
B.2.2 Unlocking Pages

B.2.2.1 Function: unlock_page()  (mm/filemap.c)

This function unlocks a page and wakes up any processes that may be waiting on it.

```c
874 void unlock_page(struct page *page)
875 {
876   wait_queue_head_t *waitqueue = page_waitqueue(page);
877   ClearPageLaunder(page);
878   smp_mb__before_clear_bit();
879   if (!test_and_clear_bit(PG_locked, &(page)->flags))
880     BUG();
881   smp_mb__after_clear_bit();
882
883   /*
884    * Although the default semantics of wake_up() are
885    * to wake all, here the specific function is used
886    * to make it even more explicit that a number of
887    * pages are being waited on here.
888    */
889   if (waitqueue_active(waitqueue))
890     wake_up_all(waitqueue);
891 }
```

**page_waitqueue**() is the implementation of the hash algorithm which determines which wait queue this page belongs to in the table zone→wait_table.

877 Clear the launder bit as IO has now completed on the page.

878 This is a memory block operations which must be called before performing bit operations that may be seen by multiple processors.

879-880 Clear the PG_locked bit. It is a **BUG()** if the bit was already cleared.

881 Complete the SMP memory block operation.

889-890 If there are processes waiting on the page queue for this page, wake them.
B.2.3 Waiting on Pages

B.2.3.1 Function: wait_on_page()  (include/linux/pagemap.h)

94 static inline void wait_on_page(struct page * page)  
95 {  
96     if (PageLocked(page))  
97         ___wait_on_page(page);  
98 }

96-97 If the page is currently locked, then call ___wait_on_page() to sleep until it is unlocked

B.2.3.2 Function: ___wait_on_page()  (mm/lemap.c)

This function is called after PageLocked() has been used to determine the page is locked. The calling process will probably sleep until the page is unlocked.

849 void ___wait_on_page(struct page *page)  
850 {  
851     wait_queue_head_t *waitqueue = page_waitqueue(page);  
852     struct task_struct *tsk = current;  
853     DECLARE_WAITQUEUE(wait, tsk);  
854     
855     add_wait_queue(waitqueue, &wait);  
856     do {  
857         set_task_state(tsk, TASK_UNINTERRUPTIBLE);  
858         if (!PageLocked(page))  
859             break;  
860         sync_page(page);  
861         schedule();  
862     } while (PageLocked(page));  
863     __set_task_state(tsk, TASK_RUNNING);  
864     remove_wait_queue(waitqueue, &wait);  
865 }

851 page_waitqueue() is the implementation of the hash algorithm which determines which wait queue this page belongs to in the table zone->wait_table

852-853 Initialise the waitqueue for the current task

855 Add this task to the waitqueue returned by page_waitqueue()

857 Set the process state to be in uninterruptible sleep. When schedule() is called, the process will sleep

858-859 Check to make sure the page was not unlocked since we last checked
860 Call `sync_page()` (See Section B.2.1.3) to call the filesystem-specific function to synchronise the page with it’s backing storage.

861 Call `schedule()` to go to sleep. The process will be woken when the page is unlocked.

862 Check if the page is still locked. Remember that multiple pages could be using this wait queue and there could be processes sleeping that wish to lock this page.

863-864 The page has been unlocked. Set the process to be in the `TASK_RUNNING` state and remove the process from the waitqueue.
Appendix C

Page Table Management

Contents

C.1 Page Table Initialisation ........................................ 222
  C.1.1 Function: paging_init() .................................. 222
  C.1.2 Function: pagetable_init() .............................. 223
  C.1.3 Function: fixrange_init() ................................ 227
  C.1.4 Function: kmap_init() .................................... 228
C.2 Page Table Walking ............................................. 230
  C.2.1 Function: follow_page() .................................. 230
C.1 Page Table Initialisation

Contents

C.1 Page Table Initialisation 222
  C.1.1 Function: paging_init() 222
  C.1.2 Function: pagetable_init() 223
  C.1.3 Function: fixrange_init() 227
  C.1.4 Function: kmap_init() 228

C.1.1 Function: paging_init() (arch/i386/mm/init.c)

This is the top-level function called from setup_arch(). When this function returns, the page tables have been fully setup. Be aware that this is all x86 specific.

```c
void __init paging_init(void)
{
    pagetable_init();
    load_cr3(swapper_pg_dir);

    #if CONFIG_X86_PAE
    if (cpu_has_pae)
        set_in_cr4(X86_CR4_PAE);
    #endif
    __flush_tlb_all();

    #ifdef CONFIG_HIGHMEM
    kmap_init();
    #endif
    zone_sizes_init();
}
```

351-362 paging_init() is responsible for setting up a static page table using swapper_pg_dir as the PGD

355 Load the initialised swapper_pg_dir into the CR3 register so that the CPU will be able to use it

362-363 If PAE is enabled, set the appropriate bit in the CR4 register

366 Flush all TLBs, including the global kernel ones

369 kmap_init() initialises the region of pagetables reserved for use with kmap()

371 zone_sizes_init() (See Section B.1.2) records the size of each of the zones before calling free_area_init() (See Section B.1.3) to initialise each zone
C.1.2 Function: pagetable_init() (arch/i386/mm/init.c)

This function is responsible for statically initializing a pagetable starting with a statically defined PGD called swapper_pg_dir. At the very least, a PTE will be available that points to every page frame in ZONE_NORMAL.

205 static void __init pagetable_init (void)
206 {
207     unsigned long vaddr, end;
208     pgd_t *pgd, *pgd_base;
209     int i, j, k;
210     pmd_t *pmd;
211     pte_t *pte, *pte_base;
212
213     /*
214     * This can be zero as well - no problem, in that case we exit
215     * the loops anyway due to the PTRS_PER_* conditions.
216     */
217     end = (unsigned long)__va(max_low_pfn*PAGE_SIZE);
218
219     pgd_base = swapper_pg_dir;
220     #if CONFIG_X86_PAE
221     for (i = 0; i < PTRS_PER_PGD; i++)
222         set_pgd(pgd_base + i, __pgd(1 + __pa(empty_zero_page)));
223     #endif
224     i = __pgd_offset(PAGE_OFFSET);
225     pgd = pgd_base + i;

This first block initializes the PGD. It does this by pointing each entry to the global zero page. Entries needed to reference available memory in ZONE_NORMAL will be allocated later.

217 The variable end marks the end of physical memory in ZONE_NORMAL

219 pgd_base is set to the beginning of the statically declared PGD

220-223 If PAE is enabled, it is insufficient to leave each entry as simply 0 (which in effect points each entry to the global zero page) as each pgd_t is a struct. Instead, set_pgd must be called for each pgd_t to point the entry to the global zero page

224 i is initialised as the offset within the PGD that corresponds to PAGE_OFFSET. In other words, this function will only be initializing the kernel portion of the linear address space, the userspace portion is left alone

225 pgd is initialised to the pgd_t corresponding to the beginning of the kernel portion of the linear address space
C.1 Page Table Initialisation (pagetable_init())

227 for (; i < PTRS_PER_PGD; pgd++, i++) {
228     vaddr = i*PGDIR_SIZE;
229     if (end && (vaddr >= end))
230         break;
231 #if CONFIG_X86_PAE
232     pmd = (pmd_t *) alloc_bootmem_low_pages(PAGE_SIZE);
233     set_pgd(pgd, __pgd(__pa(pmd) + 0x1));
234 #else
235     pmd = (pmd_t *)pgd;
236 #endif
237     if (pmd != pmd_offset(pgd, 0))
238         BUG();

This loop begins setting up valid PMD entries to point to. In the PAE case, pages are allocated with alloc_bootmem_low_pages() and the PGD is set appropriately. Without PAE, there is no middle directory, so it is just “folded” back onto the PGD to preserve the illusion of a 3-level pagetable.

227 i is already initialised to the beginning of the kernel portion of the linear address space so keep looping until the last pgd_t at PTRS_PER_PGD is reached

228 Calculate the virtual address for this PGD

229-230 If the end of ZONE_NORMAL is reached, exit the loop as further page table entries are not needed

231-234 If PAE is enabled, allocate a page for the PMD and it with set_pgd()

235 If PAE is not available, just set pmd to the current pgd_t. This is the “folding back” trick for emulating 3-level pagetables

237-238 Sanity check to make sure the PMD is valid

239     for (j = 0; j < PTRS_PER_PMD; pmd++, j++) {
240         vaddr = i*PGDIR_SIZE + j*PMD_SIZE;
241         if (end && (vaddr >= end))
242             break;
243         if (cpu_has_pse) {
244             unsigned long __pe;
245             set_in_cr4(X86_CR4_PSE);
246             boot_cpu_data.wp_works_ok = 1;
247             __pe = _KERNPG_TABLE + _PAGE_PSE + __pa(vaddr);
248             /* Make it "global" too if supported */
249             if (cpu_has_pge) {
250                 set_in_cr4(X86_CR4_PGE);
251                 __pe += _PAGE_GLOBAL;
252             
253             }
C.1 Page Table Initialisation (**pagetable_init()**)

253     
254     set_pmd(pmd, __pmd(__pe));
255     continue;
256     
257     
258     pte_base = pte =
259             (pte_t *) alloc_bootmem_low_pages(PAGE_SIZE);
260 

Initialise each entry in the PMD. This loop will only execute unless PAE is enabled. Remember that without PAE, **PTRS_PER_PMD** is 1.

240 Calculate the virtual address for this PMD

241-242 If the end of **ZONE_NORMAL** is reached, finish

243-248 If the CPU support PSE, then use large TLB entries. This means that for kernel pages, a TLB entry will map 4MiB instead of the normal 4KiB and the third level of PTEs is unnecessary

258 __pe is set as the flags for a kernel pagetable (**_KERNPG_TABLE**), the flag to indicate that this is an entry mapping 4MiB (**_PAGE_PSE**) and then set to the physical address for this virtual address with **__pa()**. Note that this means that 4MiB of physical memory is not being mapped by the pagetables

250-253 If the CPU supports PGE, then set it for this page table entry. This marks the entry as being “global” and visible to all processes

254-255 As the third level is not required because of PSE, set the PMD now with **set_pmd()** and continue to the next PMD

258 Else, PSE is not support and PTEs are required so allocate a page for them

260 for (k = 0; k < PTRS_PER_PTE; pte++, k++) {
261     vaddr = i*PGDIR_SIZE + j*PMD_SIZE + k*PAGE_SIZE;
262     if (end && (vaddr >= end))
263         break;
264     *pte = mk_pste_phys(__pa(vaddr), PAGE_KERNEL);
265 
266     set_pmd(pmd, __pmd(_KERNPG_TABLE + __pa(pte_base)));
267     if (pte_base != pte_offset(pmd, 0))
268         BUG();
269 
270 
271     }

Initialise the PTEs.
For each pte_t, calculate the virtual address currently being examined and create a PTE that points to the appropriate physical page frame.

The PTEs have been initialised so set the PMD to point to it.

Make sure that the entry was established correctly.

The fixed address space is considered to start at FIXADDR_TOP and "finish" earlier in the address space. __fix_to_virt() takes an index as a parameter and returns the index’th page frame backwards (starting from FIXADDR_TOP).
within the the fixed virtual address space. __end_of_fixed_addresses is the last index used by the fixed virtual address space. In other words, this line returns the virtual address of the PMD that corresponds to the beginning of the fixed virtual address space.

By passing 0 as the “end” to fixrange_init(), the function will start at vaddr and build valid PGDs and PMDs until the end of the virtual address space. PTEs are not needed for these addresses.

Set up page tables for use with kmap(). Get the PTE corresponding to the beginning of the region for use with kmap(). This sets up a temporary identity mapping between the virtual address 0 and the physical address 0.

C.1.3 Function: fixrange_init() (arch/i386/mm/init.c)

This function creates valid PGDs and PMDs for fixed virtual address mappings.

```c
167 static void __init fixrange_init (unsigned long start,
                                           unsigned long end,
                                           pgd_t *pgd_base)
{
  pgd_t *pgd;
  pmd_t *pmd;
  pte_t *pte;
  int i, j;
  unsigned long vaddr;
  vaddr = start;
  i = __pgd_offset(vaddr);
  j = __pmd_offset(vaddr);
  pgd = pgd_base + i;
  for ( ; (i < PTRS_PER_PGD) && (vaddr != end); pgd++, i++) {
    #if CONFIG_X86_PAE
      if (pgd_none(*pgd)) {
        pmd = (pmd_t *) alloc_bootmem_low_pages(PAGE_SIZE);
        set_pgd(pgd, __pgd(__pa(pmd) + 0x1));
        if (pmd != pmd_offset(pgd, 0))
          printk("PAE BUG #02!\n");
      }
    #else
      pmd = (pmd_t *)pgd;
    #endif
  }
```

C.1 Page Table Initialisation (fixrange_init())

```c
#endif
for (; (j < PTRS_PER_PMD) && (vaddr != end); pmd++, j++) {
  if (pmd_none(*pmd)) {
    pte = (pte_t *) alloc_bootmem_low_pages(PAGE_SIZE);
    set_pmd(pmd, __pmd(_KERNPG_TABLE + __pa(pte)));
    if (pte != pte_offset(pmd, 0))
      BUG();
  }
  vaddr += PMD_SIZE;
  j = 0;
}
```

175 Set the starting virtual address (vaddr) to the requested starting address provided as the parameter.

176 Get the index within the PGD corresponding to vaddr.

177 Get the index within the PMD corresponding to vaddr.

178 Get the starting pgd_t.

180 Keep cycling until end is reached. When pagetable_init() passes in 0, this loop will continue until the end of the PGD.

182-187 In the case of PAE, allocate a page for the PMD if one has not already been allocated.

190 Without PAE, there is no PMD so treat the pgd_t as the pmd_t.

192-200 For each entry in the PMD, allocate a page for the pte_t entries and set it within the pagetables. Note that vaddr is incremented in PMD-sized strides.

C.1.4 Function: kmap_init() (arch/i386/mm/init.c)

This function only exists if CONFIG_HIGHMEM is set during compile time. It is responsible for caching where the beginning of the kmap region is, the PTE referencing it and the protection for the page tables. This means the PGD will not have to be checked every time kmap() is used.

```c
#define kmap_get_fixmap_pte(vaddr) pte_offset(pmd_offset(pgd_offset_k(vaddr), (vaddr)), (vaddr))
```
81 void __init kmap_init(void)
82 {
83     unsigned long kmap_vstart;
84
85     /* cache the first kmap pte */
86     kmap_vstart = __fix_to_virt(FIX_KMAP_BEGIN);
87     kmap_pte = kmap_get_fixmap_pte(kmap_vstart);
88
89     kmap_prot = PAGE_KERNEL;
90 }
91 #endif /* CONFIG_HIGHMEM */

78-79 As fixrange_init() has already set up valid PGDs and PMDs, there is no need to double check them so kmap_get_fixmap_pte() is responsible for quickly traversing the page table

86 Cache the virtual address for the kmap region in kmap_vstart

87 Cache the PTE for the start of the kmap region in kmap_pte

89 Cache the protection for the page table entries with kmap_prot
C.2 Page Table Walking

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.2 Page Table Walking</td>
<td>230</td>
</tr>
<tr>
<td>C.2.1 Function: follow_page()</td>
<td>230</td>
</tr>
</tbody>
</table>

C.2.1 Function: follow_page() (mm/memory.c)

This function returns the struct page used by the PTE at address in mm’s page tables.

405 static struct page * follow_page(struct mm_struct *mm, unsigned long address, int write)

406 {
407     pgd_t *pgd;
408     pmd_t *pmd;
409     pte_t *ptep, pte;
410
411     pgd = pgd_offset(mm, address);
412     if (pgd_none(*pgd) || pgd_bad(*pgd))
413         goto out;
414
415     pmd = pmd_offset(pgd, address);
416     if (pmd_none(*pmd) || pmd_bad(*pmd))
417         goto out;
418
419     ptep = pte_offset(pmd, address);
420     if (!ptep)
421         goto out;
422
423     pte = *ptep;
424     if (pte_present(pte)) {
425         if (!write ||
426             (pte_write(pte) && pte_dirty(pte)))
427             return pte_page(pte);
428     }
429
430     out:
431     return 0;
432 }

405 The parameters are the mm whose page tables that is about to be walked, the address whose struct page is of interest and write which indicates if the page is about to be written to.

411 Get the PGD for the address and make sure it is present and valid.
415-417 Get the PMD for the address and make sure it is present and valid

419 Get the PTE for the address and make sure it exists

424 If the PTE is currently present, then we have something to return

425-426 If the caller has indicated a write is about to take place, check to make sure that the PTE has write permissions set and if so, make the PTE dirty

427 If the PTE is present and the permissions are fine, return the struct page mapped by the PTE

431 Return 0 indicating that the address has no associated struct page
# Appendix D

## Process Address Space

### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>D.1</strong> Process Memory Descriptors</td>
<td>236</td>
</tr>
<tr>
<td>D.1.1 Initialising a Descriptor</td>
<td>236</td>
</tr>
<tr>
<td>D.1.2 Copying a Descriptor</td>
<td>236</td>
</tr>
<tr>
<td>D.1.2.1 Function: <code>copy_mm()</code></td>
<td>236</td>
</tr>
<tr>
<td>D.1.2.2 Function: <code>mm_init()</code></td>
<td>239</td>
</tr>
<tr>
<td>D.1.3 Allocating a Descriptor</td>
<td>239</td>
</tr>
<tr>
<td>D.1.3.1 Function: <code>allocate_mm()</code></td>
<td>239</td>
</tr>
<tr>
<td>D.1.3.2 Function: <code>mm_alloc()</code></td>
<td>240</td>
</tr>
<tr>
<td>D.1.4 Destroying a Descriptor</td>
<td>240</td>
</tr>
<tr>
<td>D.1.4.1 Function: <code>mmput()</code></td>
<td>240</td>
</tr>
<tr>
<td>D.1.4.2 Function: <code>mmdrop()</code></td>
<td>241</td>
</tr>
<tr>
<td>D.1.4.3 Function: <code>__mmdrop()</code></td>
<td>242</td>
</tr>
<tr>
<td><strong>D.2</strong> Creating Memory Regions</td>
<td>243</td>
</tr>
<tr>
<td>D.2.1 Creating A Memory Region</td>
<td>243</td>
</tr>
<tr>
<td>D.2.1.1 Function: <code>do_mmap()</code></td>
<td>243</td>
</tr>
<tr>
<td>D.2.1.2 Function: <code>do_mmap_pgoff()</code></td>
<td>244</td>
</tr>
<tr>
<td>D.2.2 Inserting a Memory Region</td>
<td>252</td>
</tr>
<tr>
<td>D.2.2.1 Function: <code>__insert_vm_struct()</code></td>
<td>252</td>
</tr>
<tr>
<td>D.2.2.2 Function: <code>find_vma_prepare()</code></td>
<td>253</td>
</tr>
<tr>
<td>D.2.2.3 Function: <code>vma_link()</code></td>
<td>255</td>
</tr>
<tr>
<td>D.2.2.4 Function: <code>__vma_link()</code></td>
<td>256</td>
</tr>
<tr>
<td>D.2.2.5 Function: <code>__vma_link_list()</code></td>
<td>256</td>
</tr>
<tr>
<td>D.2.2.6 Function: <code>__vma_link_rb()</code></td>
<td>257</td>
</tr>
<tr>
<td>D.2.2.7 Function: <code>__vma_link_file()</code></td>
<td>257</td>
</tr>
</tbody>
</table>
APPENDIX D. PROCESS ADDRESS SPACE

D.2.3 Merging Contiguous Regions

D.2.3.1 Function: vma_merge() 258
D.2.3.2 Function: can_vma_merge() 260

D.2.4 Remapping and Moving a Memory Region

D.2.4.1 Function: sys_mremap() 261
D.2.4.2 Function: do_mremap() 261
D.2.4.3 Function: move_vma() 267
D.2.4.4 Function: make_pages_present() 271
D.2.4.5 Function: get_user_pages() 272
D.2.4.6 Function: move_page_tables() 276
D.2.4.7 Function: move_one_page() 277
D.2.4.8 Function: get_one_pde() 277
D.2.4.9 Function: alloc_one_pde() 278
D.2.4.10 Function: copy_one_pde() 279

D.2.5 Deleting a memory region

D.2.5.1 Function: do_munmap() 280
D.2.5.2 Function: unmap_fixup() 284

D.2.6 Deleting all memory regions

D.2.6.1 Function: exit_mmap() 287
D.2.6.2 Function: clear_page_tables() 290
D.2.6.3 Function: free_one_pgd() 290
D.2.6.4 Function: free_one_pmd() 291

D.3 Searching Memory Regions

D.3.1 Finding a Mapped Memory Region

D.3.1.1 Function: find_vma() 293
D.3.1.2 Function: find_vma_prev() 294
D.3.1.3 Function: find_vma_intersection() 296

D.3.2 Finding a Free Memory Region

D.3.2.1 Function: get_unmapped_area() 296
D.3.2.2 Function: arch_get_unmapped_area() 297

D.4 Locking and Unlocking Memory Regions

D.4.1 Locking a Memory Region

D.4.1.1 Function: sys_mlock() 299
D.4.1.2 Function: sys_mlockall() 300
D.4.1.3 Function: do_mlockall() 302
D.4.1.4 Function: do_mlock() 303
APPENDIX D. PROCESS ADDRESS SPACE

D.4.2 Unlocking the region ........................................ 305
  D.4.2.1 Function: sys_munlock() ............................. 305
  D.4.2.2 Function: sys_munlockall() ......................... 306

D.4.3 Fixing up regions after locking/unlocking ............. 306
  D.4.3.1 Function: mlock_fixup() .......................... 306
  D.4.3.2 Function: mlock_fixup_all() ....................... 308
  D.4.3.3 Function: mlock_fixup_start() ..................... 308
  D.4.3.4 Function: mlock_fixup_end() ....................... 309
  D.4.3.5 Function: mlock_fixup_middle() .................... 310

D.5 Page Faulting ............................................. 313
  D.5.1 x86 Page Fault Handler ............................... 313
    D.5.1.1 Function: do_page_fault() ....................... 313
  D.5.2 Expanding the Stack .................................. 323
    D.5.2.1 Function: expand_stack() ....................... 323
  D.5.3 Architecture Independent Page Fault Handler ......... 324
    D.5.3.1 Function: handle_mm_fault() .................... 324
    D.5.3.2 Function: handle_pte_fault() .................... 326
  D.5.4 Demand Allocation .................................... 327
    D.5.4.1 Function: do_no_page() .......................... 327
    D.5.4.2 Function: do_anonymous_page() .................. 330
  D.5.5 Demand Paging ......................................... 332
    D.5.5.1 Function: do_swap_page() ....................... 332
    D.5.5.2 Function: can_share_swap_page() ................ 336
    D.5.5.3 Function: exclusive_swap_page() ................. 337
  D.5.6 Copy On Write (COW) Pages .......................... 338
    D.5.6.1 Function: do_wp_page() .......................... 338

D.6 Page-Related Disk IO ...................................... 341
  D.6.1 Generic File Reading ................................... 341
    D.6.1.1 Function: generic_file_read() ................... 341
    D.6.1.2 Function: do_generic_file_read() ................. 344
    D.6.1.3 Function: generic_file_readahead() ............... 351
  D.6.2 Generic File mmap() .................................. 355
    D.6.2.1 Function: generic_file_mmap() ................... 355
  D.6.3 Generic File Truncation ............................... 356
    D.6.3.1 Function: vmtruncate() .......................... 356
    D.6.3.2 Function: vmtruncate_list() ..................... 358
D.6.3.3 Function: zap_page_range() .................. 359
D.6.3.4 Function: zap_pmd_range() .................. 361
D.6.3.5 Function: zap_pte_range() .................. 362
D.6.3.6 Function: truncate_inode_pages() ............ 364
D.6.3.7 Function: truncate_list_pages() ............... 365
D.6.3.8 Function: truncate_complete_page() .......... 367
D.6.3.9 Function: do_flushpage() .................... 368
D.6.3.10 Function: truncate_partial_page() .......... 368
D.6.4 Reading Pages for the Page Cache ............... 369
D.6.4.1 Function: filemap_nopage() .................. 369
D.6.4.2 Function: page_cache_read() ................ 374
D.6.5 File Readahead for nopage() ................... 375
D.6.5.1 Function: nopage_sequential_readahead() .... 375
D.6.5.2 Function: read_cluster_nonblocking() ....... 377
D.6.6 Swap Related Read-Ahead ....................... 378
D.6.6.1 Function: swapin_readahead() ............... 378
D.6.6.2 Function: valid_swaphandles() ............... 379
D.1 Process Memory Descriptors

Contents

D.1 Process Memory Descriptors 236
D.1.1 Initialising a Descriptor 236
D.1.2 Copying a Descriptor 236
  D.1.2.1 Function: copy_mm() 236
  D.1.2.2 Function: mm_init() 239
D.1.3 Allocating a Descriptor 239
  D.1.3.1 Function: allocate_mm() 239
  D.1.3.2 Function: mm_alloc() 240
D.1.4 Destroying a Descriptor 240
  D.1.4.1 Function: mmput() 240
  D.1.4.2 Function: mmdrop() 241
  D.1.4.3 Function: __mmdrop() 242

This section covers the functions used to allocate, initialise, copy and destroy
memory descriptors.

D.1.1 Initialising a Descriptor

The initial mm_struct in the system is called init_mm and is statically initialised at
compile time using the macro INIT_MM().

238  #define INIT_MM(name) \  239   {  
240    mm_rb: RB_ROOT, \  241    pgd: swapper_pg_dir, \  242    mm_users: ATOMIC_INIT(2), \  243    mm_count: ATOMIC_INIT(1), \  244    mmap_sem: __RWSEM_INITIALIZER(name.mmap_sem),\  245    page_table_lock: SPIN_LOCK_UNLOCKED, \  246    mmlist: LIST_HEAD_INIT(name.mmlist), \  247   }

Once it is established, new mm_structs are copies of their parent mm_struct
and are copied using copy_mm() with the process specific fields initialised with
init_mm().

D.1.2 Copying a Descriptor

D.1.2.1 Function: copy_mm() (kernel/fork.c)

This function makes a copy of the mm_struct for the given task. This is only
called from do_fork() after a new process has been created and needs its own
mm_struct.
D.1.2 Copying a Descriptor (copy_mm())

315 static int copy_mm(unsigned long clone_flags,  
struct task_struct * tsk)
316 {
317 struct mm_struct * mm, *oldmm;
318 int retval;
319
320 tsk->min_flt = tsk->maj_flt = 0;
321 tsk->cmin_flt = tsk->cmaj_flt = 0;
322 tsk->nswap = tsk->cnswap = 0;
323
324 tsk->mm = NULL;
325 tsk->active_mm = NULL;
326
327 /*
328 * Are we cloning a kernel thread?
329 * We need to steal a active VM for that..
330 */
331 oldmm = current->mm;
332 if (!oldmm)
333 if (clone_flags & CLONE_VM) {
334 atomic_inc(&oldmm->mm_users);
335 mm = oldmm;
336 goto good_mm;
337 }
338 }
339
340}

Reset fields that are not inherited by a child mm_struct and find a mm to copy from.

The parameters are the flags passed for clone and the task that is creating a copy of the mm_struct

Initialise the task_struct fields related to memory management

Borrow the mm of the current running process to copy from

A kernel thread has no mm so it can return immediately

If the CLONE_VM flag is set, the child process is to share the mm with the parent process. This is required by users like pthreads. The mm_users field is incremented so the mm is not destroyed prematurely later. The good_mm label sets tsk—mm and tsk—active_mm and returns success

retval = -ENOMEM;
mm = allocate_mm();
if (!mm)
Allocate a new mm

Copy the parent mm and initialise the process specific mm fields with init_mm()

Initialise the MMU context for architectures that do not automatically manage their MMU

Call dup_mmap() which is responsible for copying all the VMAs regions in use by the parent process

if (retval)
    goto free_pt;

/*
 * child gets a private LDT (if there was an LDT in the parent)
 */

good_mm:
    tsk->mm = mm;
    tsk->active_mm = mm;
    return 0;

good_mm:

free_pt:
    mmput(mm);

fail_nomem:
    return retval;

}
D.1.2 Copying a Descriptor \((\text{copy_mm}())\)

This copies the LDT for the new process based on the parent process.

D.1.2.2 Function: \text{mm_init()}  \((\text{kernel/fork.c})\)

This function initialises process specific mm fields.

```c
static struct mm_struct * mm_init(struct mm_struct * mm)
{
    atomic_set(&mm->mm_users, 1);
    atomic_set(&mm->mm_count, 1);
    init_rwsem(&mm->mmap_sem);
    mm->page_table_lock = SPIN_LOCK_UNLOCKED;
    mm->pgd = pgd_alloc(mm);
    mm->def_flags = 0;
    if (mm->pgd)
        return mm;
    free_mm(mm);
    return NULL;
}
```

232 Set the number of users to 1
233 Set the reference count of the mm to 1
234 Initialise the semaphore protecting the VMA list
235 Initialise the spinlock protecting write access to it
236 Allocate a new PGD for the struct
237 By default, pages used by the process are not locked in memory
238 If a PGD exists, return the initialised struct
239 Initialisation failed, delete the \text{mm_struct} and return

D.1.3 Allocating a Descriptor

Two functions are provided allocating a \text{mm_struct}. To be slightly confusing, they are essentially the name. \text{allocate_mm()} will allocate a \text{mm_struct} from the slab allocator. \text{mm_alloc()} will allocate the struct and then call the function \text{mm_init()} to initialise it.

D.1.3.1 Function: \text{allocate_mm()}  \((\text{kernel/fork.c})\)

```c
#define allocate_mm() (kmem_cache_alloc(mm_cachep, SLAB_KERNEL))
```

227 Allocate a \text{mm_struct} from the slab allocator
D.1.3.2 Function: mm_alloc()  
(kernel/fork.c)

248 struct mm_struct * mm_alloc(void)
249 {  
250    struct mm_struct * mm;
251    mm = allocate_mm();
252    if (mm) {
253        memset(mm, 0, sizeof(*mm));
254        return mm_init(mm);
255    }
256    return NULL;
257 }

252 Allocate a mm_struct from the slab allocator
254 Zero out all contents of the struct
255 Perform basic initialisation

D.1.4 Destroying a Descriptor

A new user to an mm increments the usage count with a simple call,
atomic_inc(&mm->mm_users);

It is decremented with a call to mmput(). If the mm_users count reaches zero, all the
mapped regions are deleted with exit_mmap() and the page tables destroyed as there is no longer any users of the userspace portions. The mm_count count is
decremented with mmdrop() as all the users of the page tables and VMAs are
counted as one mm_struct user. When mm_count reaches zero, the mm_struct will
be destroyed.

D.1.4.1 Function: mmput()  (kernel/fork.c)

276 void mmput(struct mm_struct *mm)
277 {  
278     if (atomic_dec_and_lock(&mm->mm_users, &mmlist_lock)) {
279         extern struct mm_struct *swap_mm;
280         if (swap_mm == mm)
281             swap_mm = list_entry(mm->mmlist.next,
282             struct mm_struct, mmlist);
283             list_del(&mm->mmlist);
284             mmlist_nr--;
285             spin_unlock(&mmlist_lock);
286             exit_mmap(mm);
287             mmdrop(mm);
288         }
289     }

278 Atomically decrement the mm_users field while holding the mmlist_lock lock. Return with the lock held if the count reaches zero

279-286 If the usage count reaches zero, the mm and associated structures need to be removed

279-281 The swap_mm is the last mm that was swapped out by the vmscan code. If the current process was the last mm swapped, move to the next entry in the list

282 Remove this mm from the list

283-284 Reduce the count of mms in the list and release the mmlist lock

285 Remove all associated mappings

286 Delete the mm

D.1.4.2 Function: mmdrop() (include/linux/sched.h)

765 static inline void mmdrop(struct mm_struct * mm)
766 {
767     if (atomic_dec_and_test(&mm->mm_count))
768         _mmdrop(mm);
769 }

767 Atomically decrement the reference count. The reference count could be higher if the mm was been used by lazy tlb switching tasks

768 If the reference count reaches zero, call _mmdrop()
D.1.4.3 Function: __mmdrop() (kernel/fork.c)

265 inline void __mmdrop(struct mm_struct *mm) {
266     BUG_ON(mm == &init_mm);
267     pgd_free(mm->pgd);
268     destroy_context(mm);
269     free_mm(mm);
270 }

267 Make sure the init_mm is not destroyed
268 Delete the PGD entry
269 Delete the LDT
270 Call kmem_cache_free() for the mm freeing it with the slab allocator
## D.2 Creating Memory Regions

### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.2 Creating Memory Regions</td>
<td>243</td>
</tr>
<tr>
<td>D.2.1 Creating A Memory Region</td>
<td>243</td>
</tr>
<tr>
<td>D.2.1.1 Function: do_mmap()</td>
<td>243</td>
</tr>
<tr>
<td>D.2.1.2 Function: do_mmap_pgoff()</td>
<td>244</td>
</tr>
<tr>
<td>D.2.2 Inserting a Memory Region</td>
<td>252</td>
</tr>
<tr>
<td>D.2.2.1 Function: __insert_vma_struct()</td>
<td>252</td>
</tr>
<tr>
<td>D.2.2.2 Function: find_vma_prepare()</td>
<td>253</td>
</tr>
<tr>
<td>D.2.2.3 Function: vma_link()</td>
<td>255</td>
</tr>
<tr>
<td>D.2.2.4 Function: __vma_link()</td>
<td>256</td>
</tr>
<tr>
<td>D.2.2.5 Function: __vma_link_list()</td>
<td>256</td>
</tr>
<tr>
<td>D.2.2.6 Function: __vma_link_rb()</td>
<td>257</td>
</tr>
<tr>
<td>D.2.2.7 Function: __vma_link_file()</td>
<td>257</td>
</tr>
<tr>
<td>D.2.3 Merging Contiguous Regions</td>
<td>258</td>
</tr>
<tr>
<td>D.2.3.1 Function: vma_merge()</td>
<td>258</td>
</tr>
<tr>
<td>D.2.3.2 Function: can_vma_merge()</td>
<td>260</td>
</tr>
<tr>
<td>D.2.4 Remapping and Moving a Memory Region</td>
<td>261</td>
</tr>
<tr>
<td>D.2.4.1 Function: sys_mremap()</td>
<td>261</td>
</tr>
<tr>
<td>D.2.4.2 Function: do_mremap()</td>
<td>261</td>
</tr>
<tr>
<td>D.2.4.3 Function: move_vma()</td>
<td>267</td>
</tr>
<tr>
<td>D.2.4.4 Function: make_pages_present()</td>
<td>271</td>
</tr>
<tr>
<td>D.2.4.5 Function: get_user_pages()</td>
<td>272</td>
</tr>
<tr>
<td>D.2.4.6 Function: move_page_tables()</td>
<td>276</td>
</tr>
<tr>
<td>D.2.4.7 Function: move_one_page()</td>
<td>277</td>
</tr>
<tr>
<td>D.2.4.8 Function: get_one_pte()</td>
<td>277</td>
</tr>
<tr>
<td>D.2.4.9 Function: alloc_one_pte()</td>
<td>278</td>
</tr>
<tr>
<td>D.2.4.10 Function: copy_one_pte()</td>
<td>279</td>
</tr>
<tr>
<td>D.2.5 Deleting a memory region</td>
<td>280</td>
</tr>
<tr>
<td>D.2.5.1 Function: do_munmap()</td>
<td>280</td>
</tr>
<tr>
<td>D.2.5.2 Function: unmap_fixup()</td>
<td>284</td>
</tr>
<tr>
<td>D.2.6 Deleting all memory regions</td>
<td>287</td>
</tr>
<tr>
<td>D.2.6.1 Function: exit_mmap()</td>
<td>287</td>
</tr>
<tr>
<td>D.2.6.2 Function: clear_page_tables()</td>
<td>290</td>
</tr>
<tr>
<td>D.2.6.3 Function: free_one_pgd()</td>
<td>290</td>
</tr>
<tr>
<td>D.2.6.4 Function: free_one_pmd()</td>
<td>291</td>
</tr>
</tbody>
</table>

This large section deals with the creation, deletion and manipulation of memory regions.

### D.2.1 Creating A Memory Region

The main call graph for creating a memory region is shown in Figure 4.4.

#### D.2.1.1 Function: do_mmap() \( \text{(include/linux/mm.h)} \)

This is a very simply wrapper function around do_mmap_pgoff() which performs most of the work.
557 static inline unsigned long do_mmap(struct file *file,
558     unsigned long addr,
559     unsigned long len, unsigned long prot,
560     unsigned long flag, unsigned long offset)
561 {
562     unsigned long ret = -EINVAL;
563     if ((offset + PAGE_ALIGN(len)) < offset)
564         goto out;
565     if (!(offset & ~PAGE_MASK))
566         ret = do_mmap_pgoff(file, addr, len, prot, flag,
567             offset >> PAGE_SHIFT);
568 out:
569     return ret;
570 }

561 By default, return -EINVAL
562-563 Make sure that the size of the region will not overflow the total size of
the address space
564-565 Page align the offset and call do_mmap_pgoff() to map the region

D.2.1.2 Function: do_mmap_pgoff()  (mm/mmap.c)
This function is very large and so is broken up into a number of sections. Broadly
speaking the sections are

• Sanity check the parameters
• Find a free linear address space large enough for the memory mapping. If
a filesystem or device specific get_unmapped_area() function is provided, it
will be used otherwise arch_get_unmapped_area() is called
• Calculate the VM flags and check them against the file access permissions
• If an old area exists where the mapping is to take place, fix it up so it is
suitable for the new mapping
• Allocate a vm_area_struct from the slab allocator and fill in its entries
• Link in the new VMA
• Call the filesystem or device specific mmap() function
• Update statistics and exit
The parameters which correspond directly to the parameters to the \texttt{mmap} system call are:

- \texttt{file} the struct file to \texttt{mmap} if this is a file backed mapping
- \texttt{addr} the requested address to map
- \texttt{len} the length in bytes to \texttt{mmap}
- \texttt{prot} is the permissions on the area
- \texttt{flags} are the flags for the mapping
- \texttt{pgoff} is the offset within the file to begin the \texttt{mmap} at
D.2.1 Creating A Memory Region (do_mmap_pgoff())

403-404 If a file or device is been mapped, make sure a filesystem or device specific mmap function is provided. For most filesystems, this will call generic_file_mmap() (See Section D.6.2.1)

406-407 Make sure a zero length mmap() is not requested

409 Ensure that the mapping is confined to the userspace portion of the address space. On the x86, kernel space begins at PAGE_OFFSET (3GiB)

415-416 Ensure the mapping will not overflow the end of the largest possible file size

419-490 Only max_map_count number of mappings are allowed. By default this value is DEFAULT_MAX_MAP_COUNT or 65536 mappings

422 /* Obtain the address to map to. we verify (or select) it and
423 * ensure that it represents a valid section of the address space.
424 */
425 addr = get_unmapped_area(file, addr, len, pgoff, flags);
426 if (addr & ~PAGE_MASK)
427 return addr;
428
429 After basic sanity checks, this function will call the device or file specific get_unmapped_area() function. If a device specific one is unavailable, arch_get_unmapped_area() is called. This function is discussed in Section D.3.2.2

429 /* Do simple checking here so the lower-level routines won’t have
430 * to. we assume access permissions have been handled by the open
431 * of the memory object, so we don’t do any here.
432 */
433 vm_flags = calc_vm_flags(prot,flags) | mm->def_flags
434 | VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;
435
435 /* mlock MCL_FUTURE? */
436 if (vm_flags & VM_LOCKED) {
437 unsigned long locked = mm->locked_vm << PAGE_SHIFT;
438 locked += len;
439 if (locked > current->rlim[RLIMIT_MEMLOCK].rlim_cur)
440 return -EAGAIN;
441 }
442
calc_vm_flags() translates the prot and flags from userspace and translates them to their VM.equivalents

Check if it has been requested that all future mappings be locked in memory. If yes, make sure the process isn’t locking more memory than it is allowed to. If it is, return -EAGAIN

if (file) {
    switch (flags & MAP_TYPE) {
    case MAP_SHARED:
        if ((prot & PROT_WRITE) &&
            (!(file->f_mode & FMODE_WRITE))
            return -EACCES;

        /* Make sure we don’t allow writing to
         * an append-only file.. */
        if (IS_APPEND(file->f_dentry->d_inode) &&
            (file->f_mode & FMODE_WRITE))
            return -EACCES;

        /* make sure there are no mandatory
         * locks on the file. */
        if (locks_verify_locked(file->f_dentry->d_inode))
            return -EAGAIN;

        vm_flags |= VM_SHARED | VM_MAYSHARE;
        if (!(file->f_mode & FMODE_WRITE))
            vm_flags &= ~(VM_MAYWRITE | VM_SHARED);

        /* fall through */
    case MAP_PRIVATE:
        if (!(file->f_mode & FMODE_READ))
            return -EACCES;

        break;

    default:
        return -EINVAL;
    }

If a file is been memory mapped, check the files access permissions

If write access is requested, make sure the file is opened for write

Similarly, if the file is opened for append, make sure it cannot be written to. The prot field is not checked because the prot field applies only to the mapping where as we need to check the opened file
If the file is mandatory locked, return -EAGAIN so the caller will try a second type

Fix up the flags to be consistent with the file flags

Make sure the file can be read before mmapping it

If the file is been mapped for anonymous use, fix up the flags if the requested mapping is MAP_PRIVATE to make sure the flags are consistent

/* Clear old maps */
munmap_back:

find_vma_prepare(See Section D.2.2.2) steps through the RB tree for the VMA corresponding to a given address

If a VMA was found and it is part of the new mmapping, remove the old mapping as the new one will cover both
D.2.1 Creating A Memory Region (\texttt{do_mmap_pgoff()})

491 /* Check against address space limit. */
492 if ((mm->total_vm << PAGE_SHIFT) + len > current->rlim[RLIMIT_AS].rlim_cur)
493 return -ENOMEM;
494
495 /* Private writable mapping? Check memory availability.. */
496 if ((vm_flags & (VM_SHARED | VM_WRITE)) == VM_WRITE &&
497 !(flags & MAP_NORESERVE) &&
498 !vm_enough_memory(len >> PAGE_SHIFT))
499 return -ENOMEM;
500
501 /* Can we just expand an old anonymous mapping? */
502 if (!file && !(vm_flags & VM_SHARED) && rb_parent)
503 if (vma_merge(mm, prev, rb_parent,
504 addr, addr + len, vm_flags))
505 goto out;
506
507 493-495 Make sure the new mapping will not exceed the total VM a process is allowed to have. It is unclear why this check is not made earlier.

498-501 If the caller does not specifically request that free space is not checked with MAP_NORESERVE and it is a private mapping, make sure enough memory is available to satisfy the mapping under current conditions.

504-506 If two adjacent memory mappings are anonymous and can be treated as one, expand the old mapping rather than creating a new one.

508 /* Determine the object being mapped and call the appropriate
509 * specific mapper. the address has already been validated, but
510 * not unmapped, but the maps are removed from the list.
511 */
512 vma = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
513 if (!vma)
514 return -ENOMEM;
515
516 vma->vm_mm = mm;
517 vma->vm_start = addr;
518 vma->vm_end = addr + len;
519 vma->vm_flags = vm_flags;
520 vma->vm_page_prot = protection_map[vm_flags & 0x0f];
521 vma->vm_ops = NULL;
522 vma->vm_pgoff = pgoff;
Allocate a vm_area_struct from the slab allocator

Fill in the basic vm_area_struct fields

Fill in the file related fields if this is a file been mapped

These are both invalid flags for a file mapping so free the vm_area_struct and return

This flag is cleared by the system call mmap() but is still cleared for kernel modules that call this function directly. Historically, -ETXTBSY was returned to the calling process if the underlying file was been written to

Fill in the vm_file field

This increments the file usage count

Call the filesystem or device specific mmap() function. In many filesystem cases, this will call generic_file_mmap() (See Section D.6.2.1)

If an error called, goto unmap_and_free_vma to clean up and return the error
If this is an anonymous shared mapping, the region is created and setup by `shmem_zero_setup()` (See Section L.7.1). Anonymous shared pages are backed by a virtual tmpfs filesystem so that they can be synchronised properly with swap. The writeback function is `shmem_writepage()` (See Section L.6.1).

If the address has changed, it means the device specific `mmap` operation moved the VMA address to somewhere else. The function `find_vma_prepare()` (See Section D.2.2.2) is used to find where the VMA was moved to.
D.2.2 Inserting a Memory Region

Update statistics for the process mm_struct and return the new address

This is reached if the file has been partially mapped before failing.

The write statistics are updated and then all user pages are removed with zap_page_range().

This goto is used if the mapping failed immediately after the vm_area_struct is created. It is freed back to the slab allocator before the error is returned.

D.2.2 Inserting a Memory Region

The call graph for insert_vm_struct() is shown in Figure 4.6.

D.2.2.1 Function: __insert_vm_struct() (mm/mmap.c)

This is the top level function for inserting a new vma into an address space. There is a second function like it called simply insert_vm_struct() that is not described in detail here as the only difference is the one line of code increasing the map_count.
D.2.2 Inserting a Memory Region (__insert_vm_struct())

1174 void __insert_vm_struct(struct mm_struct * mm,
                              struct vm_area_struct * vma)
1175 {
1176     struct vm_area_struct * __vma, * prev;
1177     rb_node_t ** rb_link, * rb_parent;
1178
1179     __vma = find_vma_prepare(mm, vma->vm_start, &prev,
                                 &rb_link, &rb_parent);
1180     if (__vma && __vma->vm_start < vma->vm_end)
1181         BUG();
1182     __vma_link(mm, vma, prev, rb_link, rb_parent);
1183     mm->map_count++;
1184     validate_mm(mm);
1185 }

The arguments are the mm_struct that represents the linear address space and the vm_area_struct that is to be inserted.

find_vma_prepare() (See Section D.2.2.2) locates where the new VMA can be inserted. It will be inserted between prev and __vma and the required nodes for the red-black tree are also returned.

If this condition occurs without manually inserting bogus VMAs into the address space.

This function does the actual work of linking the vma struct into the linear linked list and the red-black tree.

Increase the map_count to show a new mapping has been added. This line is not present in insert_vm_struct().

validate_mm() is a debugging macro for red-black trees. If DEBUG_MM_RB is set, the linear list of VMAs and the tree will be traversed to make sure it is valid. The tree traversal is a recursive function so it is very important that that it is used only if really necessary as a large number of mappings could cause a stack overflow. If it is not set, validate_mm() does nothing at all.

D.2.2.2 Function: find_vma_prepare() (mm/mmap.c)

This is responsible for finding the correct places to insert a VMA at the supplied address. It returns a number of pieces of information via the actual return and the function arguments. The forward VMA to link to is returned with return. pprev is the previous node which is required because the list is a singly linked list. rb_link and rb_parent are the parent and leaf node the new VMA will be inserted between.
D.2.2 Inserting a Memory Region (find_vma_prepare())

246 static struct vm_area_struct * find_vma_prepare(
    struct mm_struct * mm,
    unsigned long addr,
    struct vm_area_struct ** pprev,
    struct vm_area_struct ** pprev,
    rb_node_t *** rb_link,
    rb_node_t ** rb_parent)
249 {
    struct vm_area_struct * vma;
    rb_node_t ** __rb_link, * __rb_parent, * rb_prev;
252 __rb_link = &mm->mm_rb.rb_node;
    __rb_parent = &mm->mm_rb.rb_node;
    rb_prev = __rb_parent = NULL;
255 vma = NULL;
256 while (*__rb_link) {
    struct vm_area_struct *vma_tmp;
    __rb_parent = *__rb_link;
    vma_tmp = rb_entry(__rb_parent,
        struct vm_area_struct, vm_rb);
262 if (vma_tmp->vm_end > addr) {
        vma = vma_tmp;
        if (vma_tmp->vm_start <= addr)
            return vma;
267 __rb_link = &__rb_parent->rb_left;
    } else {
46         __rb_link = &__rb_parent->rb_right;
271 }
272 }
273 *pprev = NULL;
275 if (rb_prev)
276     *pprev = rb_entry(rb_prev, struct vm_area_struct, vm_rb);
277 *rb_link = __rb_link;
278 *rb_parent = __rb_parent;
279 return vma;
280 }

The function arguments are described above

253-255 Initialise the search
D.2.2 Inserting a Memory Region (find_vma_prepare())

263-272 This is a similar tree walk to what was described for find_vma(). The only real difference is the nodes last traversed are remembered with the __rb_link and __rb_parent variables.

275-276 Get the back linking VMA via the red-black tree.

279 Return the forward linking VMA.

D.2.2.3 Function: vma_link() (mm/mmap.c)

This is the top-level function for linking a VMA into the proper lists. It is responsible for acquiring the necessary locks to make a safe insertion.

337 static inline void vma_link(struct mm_struct * mm, 
    struct vm_area_struct * vma, 
    struct vm_area_struct * prev, 
    rb_node_t ** rb_link, rb_node_t * rb_parent)
339 {
340    lock_vma_mappings(vma);
341    spin_lock(&mm->page_table_lock);
342    __vma_link(mm, vma, prev, rb_link, rb_parent);
343    spin_unlock(&mm->page_table_lock);
344    unlock_vma_mappings(vma);
345    mm->map_count++;
347    validate_mm(mm);
348 }

337 mm is the address space the VMA is to be inserted into. prev is the backwards linked VMA for the linear linked list of VMAs. rb_link and rb_parent are the nodes required to make the rb insertion.

340 This function acquires the spinlock protecting the address_space representing the file that is been memory mapped.

341 Acquire the page table lock which protects the whole mm_struct.

342 Insert the VMA.

343 Free the lock protecting the mm_struct.

345 Unlock the address_space for the file.

346 Increase the number of mappings in this mm.

347 If DEBUG_MM_RB is set, the RB trees and linked lists will be checked to make sure they are still valid.
D.2.2.4 Function: `__vma_link()`  
This simply calls three helper functions which are responsible for linking the VMA into the three linked lists that link VMAs together.

```c
329 static void __vma_link(struct mm_struct * mm,
    struct vm_area_struct * vma,
    struct vm_area_struct * prev,
    rb_node_t ** rb_link, rb_node_t * rb_parent)
330 {
331    __vma_link_list(mm, vma, prev, rb_parent);
332    __vma_link_rb(mm, vma, rb_link, rb_parent);
333    __vma_link_file(vma);
334 }
```

332 This links the VMA into the linear linked lists of VMAs in this mm via the `vm_next` field.

333 This links the VMA into the red-black tree of VMAs in this mm whose root is stored in the `vm_rb` field.

334 This links the VMA into the shared mapping VMA links. Memory mapped files are linked together over potentially many mms by this function via the `vm_next_share` and `vm_pprev_share` fields.

D.2.2.5 Function: `__vma_link_list()`  

```c
282 static inline void __vma_link_list(struct mm_struct * mm,
    struct vm_area_struct * vma,
    struct vm_area_struct * prev,
    rb_node_t * rb_parent)
283 {
284     if (prev) {
285         vma->vm_next = prev->vm_next;
286         prev->vm_next = vma;
287     } else {
288         mm->mmap = vma;
289         if (rb_parent)
290             vma->vm_next = rb_entry(rb_parent,
291                                         struct vm_area_struct,
292                                         vm_rb);
293         else
294             vma->vm_next = NULL;
295     }
296 }
```

285 If `prev` is not null, the vma is simply inserted into the list.
D.2.2 Inserting a Memory Region (__vma_link_list())

Else this is the first mapping and the first element of the list has to be stored in the mm_struct

The VMA is stored as the parent node

D.2.2.6 Function: __vma_link_rb() (mm/mmap.c)

The principal workings of this function are stored within <linux/rbtree.h> and will not be discussed in detail in this book.

D.2.2.7 Function: __vma_link_file() (mm/mmap.c)

This function links the VMA into a linked list of shared file mappings.
D.2.3 Merging Contiguous Regions

D.2.3.1 Function: vma_merge() (mm/mmap.c)

This function checks to see if a region pointed to by prev may be expanded forwards to cover the area from addr to end instead of allocating a new VMA. If it cannot, the VMA ahead is checked to see can it be expanded backwards instead.

```c
static int vma_merge(struct mm_struct * mm,
                     struct vm_area_struct * prev,
                     rb_node_t * rb_parent,
                     unsigned long addr, unsigned long end,
                     unsigned long vm_flags)
```

The parameters are as follows;

- **mm** The mm the VMAs belong to
- **prev** The VMA before the address we are interested in
- **rb_parent** The parent RB node as returned by find_vma_prepare()
- **addr** The starting address of the region to be merged
- **end** The end of the region to be merged
- **vm_flags** The permission flags of the region to be merged

353 This is the lock to the mm
If `prev` is not passed it, it is taken to mean that the VMA being tested for merging is in front of the region from `addr` to `end`. The entry for that VMA is extracted from the `rb_parent`

```c
if (prev->vm_end == addr && can_vma_merge(prev, vm_flags)) {
    struct vm_area_struct * next;

    spin_lock(lock);
    prev->vm_end = end;
    next = prev->vm_next;
    if (next && prev->vm_end == next->vm_start &&
        can_vma_merge(next, vm_flags)) {
        prev->vm_end = next->vm_end;
        __vma_unlink(mm, next, prev);
        spin_unlock(lock);

        mm->map_count--;
        kmem_cache_free(vm_area_cachep, next);
        return 1;
    }
    spin_unlock(lock);
    return 1;
}

prev = prev->vm_next;
if (prev) {
    merge_next:
    if (!can_vma_merge(prev, vm_flags))
        return 0;
    if (end == prev->vm_start) {
        spin_lock(lock);
        prev->vm_start = addr;
        spin_unlock(lock);
        return 1;
    }
}
return 0;
```

**358-375** Check to see can the region pointed to by `prev` may be expanded to cover the current region

**358** The function `can_vma_merge()` checks the permissions of `prev` with those in `vm_flags` and that the VMA has no file mappings (i.e. it is anonymous). If it is true, the area at `prev` may be expanded
D.2.3 Merging Contiguous Regions (\texttt{vma_merge()})

361 Lock the mm

362 Expand the end of the VMA region (\texttt{vm_end}) to the end of the new mapping (\texttt{end})

363 \texttt{next} is now the VMA in front of the newly expanded VMA

364 Check if the expanded region can be merged with the VMA in front of it

365 If it can, continue to expand the region to cover the next VMA

366 As a VMA has been merged, one region is now defunct and may be unlinked

367 No further adjustments are made to the mm struct so the lock is released

369 There is one less mapped region to reduce the \texttt{map_count}

370 Delete the struct describing the merged VMA

371 Return success

377 If this line is reached it means the region pointed to by \texttt{prev} could not be expanded forward so a check is made to see if the region ahead can be merged backwards instead

382-388 Same idea as the above block except instead of adjusted \texttt{vm_end} to cover \texttt{end}, \texttt{vm_start} is expanded to cover \texttt{addr}

D.2.3.2 Function: \texttt{can_vma_merge()} \texttt{(include/linux/mm.h)}

This trivial function checks to see if the permissions of the supplied VMA match the permissions in \texttt{vm_flags}

582 \texttt{static inline int can_vma_merge(struct vm_area_struct * vma, unsigned long vm_flags) }

583 \{
584 \hspace{1em} \texttt{if (!vma->vm_file && vma->vm_flags == vm_flags) } \hspace{1em}
585 \hspace{2em} \texttt{return 1; } \\
586 \hspace{1em} \texttt{else } \hspace{2em}
587 \hspace{2em} \texttt{return 0; } \\
588 \}

584 Self explanatory. Return true if there is no file/device mapping (i.e. it is anonymous) and the VMA flags for both regions match
D.2.4 Remapping and Moving a Memory Region

D.2.4.1 Function: sys_mremap()  
_mm/mremap.c

The call graph for this function is shown in Figure 4.7. This is the system service call to remap a memory region.

```c
asmlinkage unsigned long sys_mremap(unsigned long addr,
           unsigned long old_len, unsigned long new_len,
           unsigned long flags, unsigned long new_addr)
{
    unsigned long ret;
    down_write(&current->mm->mmap_sem);
    ret = do_mremap(addr, old_len, new_len, flags, new_addr);
    up_write(&current->mm->mmap_sem);
    return ret;
}
```

347-349 The parameters are the same as those described in the mremap() man page.
353 Acquire the mm semaphore.
354 do_mremap() (See Section D.2.4.2) is the top level function for remapping a region.
355 Release the mm semaphore.
356 Return the status of the remapping.

D.2.4.2 Function: do_mremap()  
_mm/mremap.c

This function does most of the actual “work” required to remap, resize and move a memory region. It is quite long but can be broken up into distinct parts which will be dealt with separately here. The tasks are broadly speaking:

- Check usage flags and page align lengths
- Handle the condition where MAP_FIXED is set and the region is been moved to a new location.
- If a region is shrinking, allow it to happen unconditionally
- If the region is growing or moving, perform a number of checks in advance to make sure the move is allowed and safe
- Handle the case where the region is been expanded and cannot be moved
- Finally handle the case where the region has to be resized and moved
D.2.4 Remapping and Moving a Memory Region (do_mremap())

219 unsigned long do_mremap(unsigned long addr,
220 unsigned long old_len, unsigned long new_len,
221 unsigned long flags, unsigned long new_addr)
222 {
223     struct vm_area_struct *vma;
224     unsigned long ret = -EINVAL;
225
226     if (flags & ~(MREMAP_FIXED | MREMAP_MAYMOVE))
227         goto out;
228     if (addr & ~PAGE_MASK)
229         goto out;
230     old_len = PAGE_ALIGN(old_len);
231     new_len = PAGE_ALIGN(new_len);

219 The parameters of the function are

    addr is the old starting address
    old_len is the old region length
    new_len is the new region length
    flags is the option flags passed. If MREMAP_MAYMOVE is specified, it means that
    the region is allowed to move if there is not enough linear address space
    at the current space. If MREMAP_FIXED is specified, it means that the
    whole region is to move to the specified new_addr with the new length.
    The area from new_addr to new_addr+new_len will be unmapped with
    do_munmap().

    new_addr is the address of the new region if it is moved

224 At this point, the default return is -EINVAL for invalid arguments

226-227 Make sure flags other than the two allowed flags are not used

229-230 The address passed in must be page aligned

232-233 Page align the passed region lengths

236     if (flags & MREMAP_FIXED) {
237         if (new_addr & ~PAGE_MASK)
238             goto out;
239         if (!flags & MREMAP_MAYMOVE))
240             goto out;
if (new_len > TASK_SIZE || new_addr > TASK_SIZE - new_len)
    goto out;

/* Check if the location we’re moving into overlaps the
 * old location at all, and fail if it does. */
if ((new_addr <= addr) && (new_addr+new_len) > addr)
    goto out;

if ((addr <= new_addr) && (addr+old_len) > new_addr)
    goto out;

do_munmap(current->mm, new_addr, new_len);

This block handles the condition where the region location is fixed and must be
fully moved. It ensures the area been moved to is safe and definitely unmapped.

MREMAP_FIXED is the flag which indicates the location is fixed

The specified new_addr must be be page aligned

If MREMAP_FIXED is specified, then the MAYMOVE flag must be used as well

Make sure the resized region does not exceed TASK_SIZE

Just as the comments indicate, the two regions been used for the move
may not overlap

Unmap the region that is about to be used. It is presumed the caller ensures
that the region is not in use for anything important

ret = addr;
if (old_len >= new_len) {
do_munmap(current->mm, addr+new_len, old_len - new_len);
if (!((flags & MREMAP_FIXED) || (new_addr == addr)))
    goto out;
}

At this point, the address of the resized region is the return value

If the old length is larger than the new length, then the region is shrinking

Unmap the unused region

If the region is not to be moved, either because MREMAP_FIXED is not used
or the new address matches the old address, goto out which will return the
address
D.2.4 Remapping and Moving a Memory Region (do_mremap())

```
ret = -EFAULT;
vma = find_vma(current->mm, addr);
if (!vma || vma->vm_start > addr)
goto out;
/* We can’t remap across vm area boundaries */
if (old_len > vma->vm_end - addr)
goto out;
if (vma->vm_flags & VM_DONTEXPAND) {
    if (new_len > old_len)
goto out;
}
if (vma->vm_flags & VM_LOCKED) {
    unsigned long locked = current->mm->locked_vm << PAGE_SHIFT;
    locked += new_len - old_len;
    ret = -EAGAIN;
    if (locked > current->rlim[RLIMIT_MEMLOCK].rlim_cur)
goto out;
ret = -ENOMEM;
if ((current->mm->total_vm << PAGE_SHIFT) + (new_len - old_len)
> current->rlim[RLIMIT_AS].rlim_cur)
goto out;
/* Private writable mapping? Check memory availability.. */
if (((vma->vm_flags & (VM_SHARED | VM_WRITE)) == VM_WRITE &&
!flags & MAP_NORESERVE) &&
!vm_enough_memory((new_len - old_len) >> PAGE_SHIFT))
goto out;
```

Do a number of checks to make sure it is safe to grow or move the region

271 At this point, the default action is to return -EFAULT causing a segmentation fault as the ranges of memory been used are invalid

272 Find the VMA responsible for the requested address

273 If the returned VMA is not responsible for this address, then an invalid address was used so return a fault

276-277 If the old_len passed in exceeds the length of the VMA, it means the user is trying to remap multiple regions which is not allowed

278-281 If the VMA has been explicitly marked as non-resizable, raise a fault

282-283 If the pages for this VMA must be locked in memory, recalculate the number of locked pages that will be kept in memory. If the number of pages exceed the ulimit set for this resource, return EAGAIN indicating to the caller that the region is locked and cannot be resized
The default return at this point is to indicate there is not enough memory.

Ensure that the user will not exist their allowed allocation of memory.

Ensure that there is enough memory to satisfy the request after the resizing with `vm_enough_memory()` (See Section M.1.1)

```c
if (old_len == vma->vm_end - addr &&
    !((flags & MREMAP_FIXED) && (addr != new_addr)) &&
    (old_len != new_len || !(flags & MREMAP_MAYMOVE))) {
    unsigned long max_addr = TASK_SIZE;
    if (vma->vm_next)
        max_addr = vma->vm_next->vm_start;
    /* can we just expand the current mapping? */
    if (max_addr - addr >= new_len) {
        int pages = (new_len - old_len) >> PAGE_SHIFT;
        spin_lock(&vma->vm_mm->page_table_lock);
        vma->vm_end = addr + new_len;
        spin_unlock(&vma->vm_mm->page_table_lock);
        current->mm->total_vm += pages;
        if (vma->vm_flags & VM_LOCKED) {
            current->mm->locked_vm += pages;
            make_pages_present(addr + old_len,
                                addr + new_len);
        }
        ret = addr;
        goto out;
    }
}
```

Handle the case where the region is been expanded and cannot be moved

If it is the full region that is been remapped and ...

The region is definitely not been moved and ...

The region is been expanded and cannot be moved then ...

Set the maximum address that can be used to TASK_SIZE, 3GiB on an x86

If there is another region, set the max address to be the start of the next region

Only allow the expansion if the newly sized region does not overlap with the next VMA

Calculate the number of extra pages that will be required
311 Lock the mm spinlock
312 Expand the VMA
313 Free the mm spinlock
314 Update the statistics for the mm
315-319 If the pages for this region are locked in memory, make them present now
320-321 Return the address of the resized region

```
ret = -ENOMEM;
if (flags & MREMAP_MAYMOVE) {
  if (!(flags & MREMAP_FIXED)) {
    unsigned long map_flags = 0;
    if (vma->vm_flags & VM_SHARED)
      map_flags |= MAP_SHARED;
    new_addr = get_unmapped_area(vma->vm_file, 0,
      new_len, vma->vm_pgoff, map_flags);
    ret = new_addr;
    if (new_addr & ~PAGE_MASK)
      goto out;
  }
  ret = move_vma(vma, addr, old_len, new_len, new_addr);
} goto out;
return ret;
```

To expand the region, a new one has to be allocated and the old one moved to it

329 The default action is to return saying no memory is available
330 Check to make sure the region is allowed to move
331 If MREMAP_FIXED is not specified, it means the new location was not supplied so one must be found
333-334 Preserve the MAP_SHARED option
336 Find an unmapped region of memory large enough for the expansion
337 The return value is the address of the new region
338-339 For the returned address to be not page aligned, get_unmapped_area() would need to be broken. This could possibly be the case with a buggy device driver implementing get_unmapped_area() incorrectly
D.2.4 Remapping and Moving a Memory Region (do_mremap())

341 Call move_vma to move the region

343-344 Return the address if successful and the error code otherwise

D.2.4.3 Function: move_vma() (mm/mremap.c)

The call graph for this function is shown in Figure 4.8. This function is responsible for moving all the page table entries from one VMA to another region. If necessary a new VMA will be allocated for the region being moved to. Just like the function above, it is very long but may be broken up into the following distinct parts.

- Function preamble, find the VMA preceding the area about to be moved to and the VMA in front of the region to be mapped
- Handle the case where the new location is between two existing VMAs. See if the preceding region can be expanded forward or the next region expanded backwards to cover the new mapped region
- Handle the case where the new location is going to be the last VMA on the list. See if the preceding region can be expanded forward
- If a region could not be expanded, allocate a new VMA from the slab allocator
- Call move_page_tables(), fill in the new VMA details if a new one was allocated and update statistics before returning

125 static inline unsigned long move_vma(struct vm_area_struct * vma,
126         unsigned long addr, unsigned long old_len, unsigned long new_len,
127         unsigned long new_addr)
128 {
129     struct mm_struct * mm = vma->vm_mm;
130     struct vm_area_struct * new_vma, * next, * prev;
131     int allocated_vma;
132
133     new_vma = NULL;
134     next = find_vma_prev(mm, new_addr, &prev);

125-127 The parameters are

vma The VMA that the address been moved belongs to
addr The starting address of the moving region
old_len The old length of the region to move
new_len The new length of the region moved
new_addr The new address to relocate to
D.2.4 Remapping and Moving a Memory Region (move_vma())

Find the VMA preceding the address been moved to indicated by prev and return the region after the new mapping as next

```c
if (next) {
    if (prev && prev->vm_end == new_addr &&
        can_vma_merge(prev, vma->vm_flags) &&
        !vma->vm_file && !(vma->vm_flags & VM_SHARED)) {
        spin_lock(&mm->page_table_lock);
        prev->vm_end = new_addr + new_len;
        spin_unlock(&mm->page_table_lock);
        new_vma = prev;
        if (next != prev->vm_next)
            BUG();
        if (prev->vm_end == next->vm_start &&
            can_vma_merge(next, prev->vm_flags)) {
            spin_lock(&mm->page_table_lock);
            prev->vm_end = next->vm_end;
            __vma_unlink(mm, next, prev);
            spin_unlock(&mm->page_table_lock);
            mm->map_count--;
            kmem_cache_free(vm_area_cachep, next);
        } else if (next->vm_start == new_addr + new_len &&
            can_vma_merge(next, vma->vm_flags) &&
            !vma->vm_file && !(vma->vm_flags & VM_SHARED)) {
            spin_lock(&mm->page_table_lock);
            next->vm_start = new_addr;
            spin_unlock(&mm->page_table_lock);
            new_vma = next;
        }
    } else if (next->vm_start == new_addr + new_len &
        can_vma_merge(next, vma->vm_flags) &
        !vma->vm_file && !(vma->vm_flags & VM_SHARED)) {
        spin_lock(&mm->page_table_lock);
        next->vm_start = new_addr;
        spin_unlock(&mm->page_table_lock);
        new_vma = next;
    } else {
        In this block, the new location is between two existing VMAs. Checks are made to see can be preceding region be expanded to cover the new mapping and then if it can be expanded to cover the next VMA as well. If it cannot be expanded, the next region is checked to see if it can be expanded backwards.

136-137 If the preceding region touches the address to be mapped to and may be merged then enter this block which will attempt to expand regions

138 Lock the mm

139 Expand the preceding region to cover the new location

140 Unlock the mm
```
The new VMA is now the preceding VMA which was just expanded.

Make sure the VMA linked list is intact. It would require a device driver with severe brain damage to cause this situation to occur.

Check if the region can be expanded forward to encompass the next region.

If it can, then lock the mm.

Expand the VMA further to cover the next VMA.

There is now an extra VMA so unlink it.

Unlock the mm.

There is one less mapping now so update the map_count.

Free the memory used by the memory mapping.

Else the prev region could not be expanded forward so check if the region pointed to by next may be expanded backwards to cover the new mapping instead.

If it can, lock the mm.

Expand the mapping backwards.

Unlock the mm.

The VMA representing the new mapping is now next.

This block is for the case where the newly mapped region is the last VMA (next is NULL) so a check is made to see can the preceding region be expanded.

Get the previously mapped region.

Check if the regions may be mapped.

Lock the mm.
D.2.4 Remapping and Moving a Memory Region (move_vma())

Expand the preceding region to cover the new mapping

Lock the mm

The VMA representing the new mapping is now prev

Set a flag indicating if a new VMA was not allocated

If a VMA has not been expanded to cover the new mapping then...

Allocate a new VMA from the slab allocator

If it could not be allocated, goto out to return failure

Set the flag indicated a new VMA was allocated

if (!move_page_tables(current->mm, new_addr, addr, old_len)) {
    unsigned long vm_locked = vma->vm_flags & VM_LOCKED;
    if (allocated_vma) {
        *new_vma = *vma;
        new_vma->vm_start = new_addr;
        new_vma->vm_end = new_addr+new_len;
        new_vma->vm_pgoff += (addr-vma->vm_start) >> PAGE_SHIFT;
        new_vma->vm_raend = 0;
        if (new_vma->vm_file)
            get_file(new_vma->vm_file);
        if (new_vma->vm_ops & new_vma->vm_ops->open)
            new_vma->vm_ops->open(new_vma);
        insert_vm_struct(current->mm, new_vma);
    }
}
do_munmap(current->mm, addr, old_len);

current->mm->total_vm += new_len >> PAGE_SHIFT;
if (new_vma->vm_flags & VM_LOCKED) {

D.2.4 Remapping and Moving a Memory Region (move_vma())

```c
199     current->mm->locked_vm += new_len >> PAGE_SHIFT;
200     make_pages_present(new_vma->vm_start, new_vma->vm_end);
202 }
204 } }
205 if (allocated_vma)
206     kmem_cache_free(vm_area_cachep, new_vma);
207 out:
208     return -ENOMEM;
209 }
```

179 move_page_tables() (See Section D.2.4.6) is responsible for copying all the page table entries. It returns 0 on success.

182-193 If a new VMA was allocated, fill in all the relevant details, including the file/device entries and insert it into the various VMA linked lists with insert_vm_struct() (See Section D.2.2.1)

194 Unmap the old region as it is no longer required

197 Update the total_vm size for this process. The size of the old region is not important as it is handled within do_munmap()

198-202 If the VMA has the VM_LOCKED flag, all the pages within the region are made present with mark_pages_present()

203 Return the address of the new region

205-206 This is the error path. If a VMA was allocated, delete it

208 Return an out of memory error

D.2.4.4 Function: make_pages_present() (mm/memory.c)

This function makes all pages between addr and end present. It assumes that the two addresses are within the same VMA.

```c
1460 int make_pages_present(unsigned long addr, unsigned long end)
1461 {
1462     int ret, len, write;
1463     struct vm_area_struct * vma;
1464     vma = find_vma(current->mm, addr);
1466     write = (vma->vm_flags & VM_WRITE) != 0;
1467     if (addr >= end)  
1468         BUG();
1469     if (end > vma->vm_end)  
```
D.2.4 Remapping and Moving a Memory Region (make_pages_present())

1470 BUG();
1471 len = (end+PAGE_SIZE-1)/PAGE_SIZE-addr/PAGE_SIZE;
1472 ret = get_user_pages(current, current->mm, addr,
1473 len, write, 0, NULL, NULL);
1474 return ret == len ? 0 : -1;
1475 }

1465 Find the VMA with find_vma() (See Section D.3.1.1) that contains the starting address
1466 Record if write-access is allowed in write
1467-1468 If the starting address is after the end address, then BUG()
1469-1470 If the range spans more than one VMA its a bug
1471 Calculate the length of the region to fault in
1472 Call get_user_pages() to fault in all the pages in the requested region. It returns the number of pages that were faulted in
1474 Return true if all the requested pages were successfully faulted in

D.2.4.5 Function: get_user_pages() (mm/memory.c)
This function is used to fault in user pages and may be used to fault in pages belonging to another process, which is required by ptrace() for example.

454 int get_user_pages(struct task_struct *tsk, struct mm_struct *mm,
455 unsigned long start,
456 int len, int write, int force, struct page **pages,
457 struct vm_area_struct **vmas)
458 {
459 int i;
460 unsigned int flags;
461 /*
462 * Require read or write permissions.
463 * If 'force' is set, we only require the "MAY" flags.
464 */
465 flags = write ? (VM_WRITE | VM_MAYWRITE) : (VM_READ | VM_MAYREAD);
466 flags &= force ? (VM_MAYREAD | VM_MAYWRITE) : (VM_READ | VM_WRITE);
467 i = 0;
468
454 The parameters are:

tsk is the process that pages are been faulted for
D.2.4 Remapping and Moving a Memory Region (get_user_pages())

- **mm** is the `mm_struct` managing the address space being faulted.
- **start** is where to start faulting.
- **len** is the length of the region, in pages, to fault.
- **write** indicates if the pages are being faulted for writing.
- **force** indicates that the pages should be faulted even if the region only has the `VM_MAYREAD` or `VM_MAYWRITE` flags.
- **pages** is an array of struct pages which may be NULL. If supplied, the array will be filled with `struct pages` that were faulted in.
- **vmases** is similar to the **pages** array. If supplied, it will be filled with VMAs that were affected by the faults.

464 Set the required flags to `VM_WRITE` and `VM_MAYWRITE` flags if the parameter **write** is set to 1. Otherwise use the read equivalents.

465 If force is specified, only require the **MAY** flags.

```c
    do {
        struct vm_area_struct * vma;
        vma = find_extend_vma(mm, start);
        if ( vma ||
            (pages && vma->vm_flags & VM_IO) ||
            !(flags & vma->vm_flags) )
            return i ? : -EFAULT;
        spin_lock(&mm->page_table_lock);
        do {
            struct page *map;
            while (!((map = follow_page(mm, start, write))) { 
                spin_unlock(&mm->page_table_lock);
                switch (handle_mm_fault(mm, vma, start, write)) { 
                    case 1:
                        tsk->min_flt++;
                        break;
                    case 2:
                        tsk->maj_flt++;
                        break;
                    case 0:
                        if (i) return i;
                        return -EFAULT;
                    default:
                        if (i) return i;
                        return -ENOMEM;
                }
            }
        }
    }
```
D.2.4 Remapping and Moving a Memory Region (get_user_pages())

This outer loop will move through every VMA affected by the faults.

Find the VMA affected by the current value of start. This variable is incremented in PAYE_SIZE1 strides.

If a VMA does not exist for the address, or the caller has requested struct pages for a region that is IO mapped (and therefore not backed by physical memory) or that the VMA does not have the required flags, then return -EFAULT.

Lock the page table spinlock.

follow_page() (See Section C.2.1) walks the page tables and returns the struct page representing the frame mapped at start. This loop will only be entered if the PTE is not present and will keep looping until the PTE is known to be present with the page table spinlock held.

Unlock the page table spinlock as handle_mm_fault() is likely to sleep.

If the page is not present, fault it in with handle_mm_fault() (See Section D.5.3.1).

Update the task_struct statistics indicating if a major or minor fault occurred.

If the faulted address is invalid, return.
491-493 If the system is out of memory, return -ENOMEM

495 Relock the page tables. The loop will check to make sure the page is actually present

507-505 If the caller requested it, populate the pages array with struct pages affected by this function. Each struct will have a reference to it taken with page_cache_get()

506-507 Similarly, record VMAs affected

508 Increment i which is a counter for the number of pages present in the requested region

509 Increment start in a page-sized stride

510 Decrement the number of pages that must be faulted in

511 Keep moving through the VMAs until the requested pages have been faulted in

512 Release the page table spinlock

515 Return the number of pages known to be present in the region

516
517  /*
518   * We found an invalid page in the VMA. Release all we have
519   * so far and fail.
520   */
521 bad_page:
522   spin_unlock(&mm->page_table_lock);
523   while (i--)
524     page_cache_release(pages[i]);
525   i = -EFAULT;
526   goto out;
527 }

521 This will only be reached if a struct page is found which represents a non-existent page frame

523-524 If one if found, release references to all pages stored in the pages array

525-526 Return -EFAULT
D.2.4.6 Function: move_page_tables() (mm/remap.c)

The call graph for this function is shown in Figure 4.9. This function is responsible for copying all the page table entries from the region pointed to be `old_addr` to `new_addr`. It works by literally copying page table entries one at a time. When it is finished, it deletes all the entries from the old area. This is not the most efficient way to perform the operation, but it is very easy to error recover.

```c
90 static int move_page_tables(struct mm_struct * mm,
91     unsigned long new_addr, unsigned long old_addr,
92     unsigned long len)
93 {
94     unsigned long offset = len;
95     flush_cache_range(mm, old_addr, old_addr + len);
96     while (offset) {
97         offset -= PAGE_SIZE;
98         if (move_one_page(mm, old_addr + offset, new_addr +
99             offset))
100             goto oops_we_failed;
101     }
102     flush_tlb_range(mm, old_addr, old_addr + len);
103     return 0;
104
112     oops_we_failed:
113     flush_cache_range(mm, new_addr, new_addr + len);
114     while ((offset += PAGE_SIZE) < len)
115         move_one_page(mm, new_addr + offset, old_addr + offset);
116     zap_page_range(mm, new_addr, len);
117     return -1;
118 }
```

90 The parameters are the mm for the process, the new location, the old location and the length of the region to move entries for.

95 `flush_cache_range()` will flush all CPU caches for this range. It must be called first as some architectures, notably Sparc’s require that a virtual to physical mapping exist before flushing the TLB.

102-106 This loops through each page in the region and moves the PTE with `move_one_pte()` (See Section D.2.4.7). This translates to a lot of page table walking and could be performed much better but it is a rare operation.

107 Flush the TLB for the old region.

108 Return success.
D.2.4 Remapping and Moving a Memory Region (move_page_tables())

118-120 This block moves all the PTEs back. A flush_tlb_range() is not necessary as there is no way the region could have been used yet so no TLB entries should exist.

121 Zap any pages that were allocated for the move

122 Return failure

D.2.4.7 Function: move_one_page() (mm/mremap.c)

This function is responsible for acquiring the spinlock before finding the correct PTE with get_one_pte() and copying it with copy_one_pte()

77 static int move_one_page(struct mm_struct *mm, unsigned long old_addr, unsigned long new_addr)
78 {
79   int error = 0;
80   pte_t * src;
81
82   spin_lock(&mm->page_table_lock);
83   src = get_one_pte(mm, old_addr);
84   if (src)
85     error = copy_one_pte(mm, src, alloc_one_pte(mm, new_addr));
86   spin_unlock(&mm->page_table_lock);
87   return error;
88 }

82 Acquire the mm lock

83 Call get_one_pte() (See Section D.2.4.8) which walks the page tables to get the correct PTE

84-85 If the PTE exists, allocate a PTE for the destination and copy the PTEs with copy_one_pte() (See Section D.2.4.10)

86 Release the lock

87 Return whatever copy_one_pte() returned. It will only return an error if alloc_one_pte() (See Section D.2.4.9) failed on line 85

D.2.4.8 Function: get_one_pte() (mm/mremap.c)

This is a very simple page table walk.

18 static inline pte_t * get_one_pte(struct mm_struct *mm, unsigned long addr)
19 {
20   pgd_t * pgd;
21   pmd_t * pmd;

D.2.4 Remapping and Moving a Memory Region (get_one_pte())

22 pte_t * pte = NULL;
23
24 pgd = pgd_offset(mm, addr);
25 if (pgd_none(*pgd))
26     goto end;
27 if (pgd_bad(*pgd)) {
28     pgd_ERROR(*pgd);
29     pgd_clear(pgd);
30     goto end;
31 }
32
33 pmd = pmd_offset(pgd, addr);
34 if (pmd_none(*pmd))
35     goto end;
36 if (pmd_bad(*pmd)) {
37     pmd_ERROR(*pmd);
38     pmd_clear(pmd);
39     goto end;
40 }
41
42 pte = pte_offset(pmd, addr);
43 if (pte_none(*pte))
44     pte = NULL;
45 end:
46 return pte;
47 }

Get the PGD for this address
25-26 If no PGD exists, return NULL as no PTE will exist either
27-31 If the PGD is bad, mark that an error occurred in the region, clear its contents and return NULL
33-40 Acquire the correct PMD in the same fashion as for the PGD
42 Acquire the PTE so it may be returned if it exists

D.2.4.9 Function: alloc_one_pte() (mm/mremap.c)

Trivial function to allocate what is necessary for one PTE in a region.

49 static inline pte_t *alloc_one_pte(struct mm_struct *mm,
        unsigned long addr)
50 {
51     pmd_t * pmd;
52     pte_t * pte = NULL;
D.2.4 Remapping and Moving a Memory Region \textit{(alloc\_one\_pte())} \hspace{1cm} 279

53 pmd = pmd\_alloc(mm, pgd\_offset(mm, addr), addr);
54 if (pmd)
55 pte = pte\_alloc(mm, pmd, addr);
56 return pte;
57 }

54 If a PMD entry does not exist, allocate it
55-56 If the PMD exists, allocate a PTE entry. The check to make sure it succeeded
is performed later in the function copy\_one\_pte()

D.2.4.10 \textbf{Function:} \textit{copy\_one\_pte()} (\texttt{mm/mremap.c})

Copies the contents of one PTE to another.

60 static inline int copy\_one\_pte(struct mm\_struct *mm,
pte\_t * src, pte\_t * dst)
61 {
62 int error = 0;
63 pte\_t pte;
64
65 if (!pte\_none(*src)) {
66 pte = ptep\_get\_and\_clear(src);
67 if (!dst) {
68 /* No dest? We must put it back. */
69 dst = src;
70 error++;
71 }
72 set\_pte(dst, pte);
73 }
74 return error;
75 }

65 If the source PTE does not exist, just return 0 to say the copy was successful
66 Get the PTE and remove it from its old location
67-71 If the \texttt{dst} does not exist, it means the call to \texttt{alloc\_one\_pte()} failed and
the copy operation has failed and must be aborted
72 Move the PTE to its new location
74 Return an error if one occurred
D.2.5 Deleting a memory region

D.2.5.1 Function: do_munmap() (mm/mmap.c)

The call graph for this function is shown in Figure 4.11. This function is responsible for unmapping a region. If necessary, the unmapping can span multiple VMAs and it can partially unmap one if necessary. Hence the full unmapping operation is divided into two major operations. This function is responsible for finding what VMAs are affected and unmap_fixup() is responsible for fixing up the remaining VMAs.

This function is divided up in a number of small sections will be dealt with in turn. The are broadly speaking:

- Function preamble and find the VMA to start working from
- Take all VMAs affected by the unmapping out of the mm and place them on a linked list headed by the variable free
- Cycle through the list headed by free, unmap all the pages in the region to be unmapped and call unmap_fixup() to fix up the mappings
- Validate the mm and free memory associated with the unmapping

```c
924 int do_munmap(struct mm_struct *mm, unsigned long addr, size_t len)
925 {
926     struct vm_area_struct *mpnt, *prev, **npp, *free, *extra;
927     if ((addr & ~PAGE_MASK) || addr > TASK_SIZE ||
928          len > TASK_SIZE-addr)
929         return -EINVAL;
930     if ((len = PAGE_ALIGN(len)) == 0)
931         return -EINVAL;
932     mpnt = find_vma_prev(mm, addr, &prev);
933     if (!mpnt)
934         return 0;
935     /* we have addr < mpnt->vm_end */
936     if (mpnt->vm_start >= addr+len)
937         return 0;
938     if ((mpnt->vm_start < addr && mpnt->vm_end > addr+len) && mm->map_count >= max_map_count)
939         return -ENOMEM;
940     extra = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
```
D.2.5 Deleting a memory region (do_munmap())

924 The parameters are as follows;

mm The mm for the processes performing the unmap operation
addr The starting address of the region to unmap
len The length of the region

928-929 Ensure the address is page aligned and that the area to be unmapped is not in the kernel virtual address space

931-932 Make sure the region size to unmap is page aligned

939 Find the VMA that contains the starting address and the preceding VMA so it can be easily unlinked later

940-941 If no mpnt was returned, it means the address must be past the last used VMA so the address space is unused, just return

944-945 If the returned VMA starts past the region we are trying to unmap, then the region in unused, just return

948-950 The first part of the check sees if the VMA is just been partially un- mapped, if it is, another VMA will be created later to deal with a region being broken into so to the map_count has to be checked to make sure it is not too large

956-958 In case a new mapping is required, it is allocated now as later it will be much more difficult to back out in event of an error

960 npp = (prev ? &prev->vm_next : &mm->mmap);
961 free = NULL;
962 spin_lock(&mm->page_table_lock);
963 for (; mpnt && mpnt->vm_start < addr+len; mpnt = *npp) {
964    *npp = mpnt->vm_next;
965    mpnt->vm_next = free;
966    free = mpnt;
967    rb_erase(&mpnt->vm_rb, &mm->mm_rb);
968 }
969 mm->mmap_cache = NULL; /* Kill the cache. */
970 spin_unlock(&mm->page_table_lock);

This section takes all the VMAs affected by the unmapping and places them on a separate linked list headed by a variable called free. This makes the fix up of the regions much easier.
D.2.5 Deleting a memory region (do_munmap())

npp becomes the next VMA in the list during the for loop following below. To initialise it, it is either the current VMA (mpnt) or else it becomes the first VMA in the list

free is the head of a linked list of VMAs that are affected by the unmapping

Lock the mm

Cycle through the list until the start of the current VMA is past the end of the region to be unmapped

npp becomes the next VMA in the list

Remove the current VMA from the linear linked list within the mm and place it on a linked list headed by free. The current mpnt becomes the head of the free linked list

Delete mpnt from the red-black tree

Remove the cached result in case the last looked up result is one of the regions to be unmapped

Free the mm

/* Ok - we have the memory areas we should free on the
 * 'free' list, so release them, and unmap the page range..
 * If the one of the segments is only being partially unmapped,
 * it will put new vm_area_struct(s) into the address space.
 * In that case we have to be careful with VM_DENYWRITE.
 */
while ((mpnt = free) != NULL) {
    unsigned long st, end, size;
    struct file *file = NULL;

    free = free->vm_next;

    st = addr < mpnt->vm_start ? mpnt->vm_start : addr;
    end = addr+len;
    end = end > mpnt->vm_end ? mpnt->vm_end : end;
    size = end - st;

    if (mpnt->vm_flags & VM_DENYWRITE &&
        (st != mpnt->vm_start || end != mpnt->vm_end) &&
        (file = mpnt->vm_file) != NULL) {
        atomic_dec(&file->f_dentry->d_inode->i_writecount);
    }
    remove_shared_vm_struct(mpnt);
D.2.5 Deleting a memory region (do_munmap())

```c
995    mm->map_count--;  
996    zap_page_range(mm, st, size);  
997    /* Fix the mapping, and free the old area  
* if it wasn't reused. */  
999    extra = unmap_fixup(mm, mpnt, st, size, extra);  
1000   if (file)  
1001      atomic_inc(&file->f_dentry->d_inode->i_writecount);  
1002    }

978 Keep stepping through the list until no VMAs are left
982 Move free to the next element in the list leaving mpnt as the head about to be removed
984 st is the start of the region to be unmapped. If the addr is before the start of the VMA, the starting point is mpnt->vm_start, otherwise it is the supplied address
985-986 Calculate the end of the region to map in a similar fashion
987 Calculate the size of the region to be unmapped in this pass
989-993 If the VM_DENYWRITE flag is specified, a hole will be created by this un-mapping and a file is mapped then the i_writecount is decremented. When this field is negative, it counts how many users there is protecting this file from being opened for writing
994 Remove the file mapping. If the file is still partially mapped, it will be acquired again during unmap_fixup() (See Section D.2.5.2)
995 Reduce the map count
997 Remove all pages within this region
1002 Call unmap_fixup() (See Section D.2.5.2) to fix up the regions after this one is deleted
1003-1004 Increment the writecount to the file as the region has been unmapped. If it was just partially unmapped, this call will simply balance out the decrement at line 987
1006    validate_mm(mm);  
1007    /* Release the extra vma struct if it wasn’t used */
```
D.2.5 Deleting a memory region (do_munmap())

1009 if (extra)
1010     kmem_cache_free(vm_area_cachep, extra);
1011 free_pgtables(mm, prev, addr, addr+len);
1013 return 0;
1015 }

validate_mm() is a debugging function. If enabled, it will ensure the VMA tree for this mm is still valid.

1009-1010 If extra VMA was not required, delete it.
1012 Free all the page tables that were used for the unmapped region.
1014 Return success.

D.2.5.2 Function: unmap_fixup() (mm/mmap.c)

This function fixes up the regions after a block has been unmapped. It is passed a list of VMAs that are affected by the unmapping, the region and length to be unmapped and a spare VMA that may be required to fix up the region if a whole is created. There is four principle cases it handles; The unmapping of a region, partial unmapping from the start to somewhere in the middle, partial unmapping from somewhere in the middle to the end and the creation of a hole in the middle of the region. Each case will be taken in turn.

787 static struct vm_area_struct * unmap_fixup(struct mm_struct *mm,
788       struct vm_area_struct *area, unsigned long addr, size_t len,
789       struct vm_area_struct *extra)
790 {
791     struct vm_area_struct *mpnt;
792     unsigned long end = addr + len;
793     area->vm_mm->total_vm -= len >> PAGE_SHIFT;
794     if (area->vm_flags & VM_LOCKED)
795         area->vm_mm->locked_vm -= len >> PAGE_SHIFT;
797
Function preamble.

787 The parameters to the function are;

    mm   is the mm the unmapped region belongs to
    area is the head of the linked list of VMAs affected by the unmapping
    addr is the starting address of the unmapping
    len  is the length of the region to be unmapped
D.2.5 Deleting a memory region (`unmap_fixup()`)  

extra is a spare VMA passed in for when a hole in the middle is created.

792 Calculate the end address of the region being unmapped.

794 Reduce the count of the number of pages used by the process.

795-796 If the pages were locked in memory, reduce the locked page count.

798-799 The first, and easiest, case is where the full region is being unmapped.

799 The full region is unmapped if the addr is the start of the VMA and the end is the end of the VMA. This is interesting because if the unmapping is spanning regions, it is possible the end is beyond the end of the VMA but the full of this VMA is still being unmapped.

800-801 If a close operation is supplied by the VMA, call it.

802-803 If a file or device is mapped, call `fput()` which decrements the usage count and releases it if the count falls to 0.

804 Free the memory for the VMA back to the slab allocator.

805 Return the extra VMA as it was unused.

809 If (end == area->vm_end) {
810     /*
811        * here area isn’t visible to the semaphore-less readers
812        * so we don’t need to update it under the spinlock.
813     */
814     area->vm_end = addr;
815     lock_vma_mappings(area);
816     spin_lock(&mm->page_table_lock);
817 }

Handle the case where the middle of the region to the end is been unmapped.

814 Truncate the VMA back to addr. At this point, the pages for the region have already freed and the page table entries will be freed later so no further work is required.
If a file/device is being mapped, the lock protecting shared access to it is taken in the function `lock_vm_mappings()`

Lock the mm. Later in the function, the remaining VMA will be reinserted into the mm

Else if (addr == area->vm_start) {
    area->vm_pgoff += (end - area->vm_start) >> PAGE_SHIFT;
    /* same locking considerations of the above case */
    area->vm_start = end;
    lock_vma_mappings(area);
    spin_lock(&mm->page_table_lock);
} else {

    Handle the case where the VMA is been unmapped from the start to some part in the middle

    Increase the offset within the file/device mapped by the number of pages this unmapping represents

    Move the start of the VMA to the end of the region being unmapped

    Lock the file/device and mm as above

} else {
    /* Add end mapping -- leave beginning for below */
    mpnt = extra;
    extra = NULL;

    mpnt->vm_mm = area->vm_mm;
    mpnt->vm_start = end;
    mpnt->vm_end = area->vm_end;
    mpnt->vm_page_prot = area->vm_page_prot;
    mpnt->vm_flags = area->vm_flags;
    mpnt->vm_raend = 0;
    mpnt->vm_ops = area->vm_ops;
    mpnt->vm_pgoff = area->vm_pgoff +
        ((end - area->vm_start) >> PAGE_SHIFT);
    mpnt->vm_file = area->vm_file;
    mpnt->vm_private_data = area->vm_private_data;
    if (mpnt->vm_file)
        get_file(mpnt->vm_file);
    if (mpnt->vm_ops && mpnt->vm_ops->open)
        mpnt->vm_ops->open(mpnt);
    area->vm_end = addr; /* Truncate area */
}
D.2.6 Deleting all memory regions

D.2.6.1 Function: exit_mmap()  (mm/mmap.c)

This function simply steps through all VMAs associated with the supplied mm and unmaps them.
D.2.6 Deleting all memory regions (exit_mmap())

```c
void exit_mmap(struct mm_struct * mm) {
    struct vm_area_struct * mpnt;
    release_segments(mm);
    spin_lock(&mm->page_table_lock);
    mpnt = mm->mmap;
    mm->mmap = mm->mmap_cache = NULL;
    mm->mm_rb = RB_ROOT;
    mm->rss = 0;
    spin_unlock(&mm->page_table_lock);
    mm->total_vm = 0;
    mm->locked_vm = 0;
    flush_cache_mm(mm);
    while (mpnt) {
        struct vm_area_struct * next = mpnt->vm_next;
        unsigned long start = mpnt->vm_start;
        unsigned long end = mpnt->vm_end;
        unsigned long size = end - start;

        if (mpnt->vm_ops) {
            if (mpnt->vm_ops->close)
                mpnt->vm_ops->close(mpnt);
            mm->map_count--;
            remove_shared_vm_struct(mpnt);
            zap_page_range(mm, start, size);
            if (mpnt->vm_file)
                fput(mpnt->vm_file);
            kmem_cache_free(vm_area_cachep, mpnt);
            mpnt = next;
        }
    }
    flush_tlb_mm(mm);
    /* This is just debugging */
    if (mm->map_count)
        BUG();
    clear_page_tables(mm, FIRST_USER_PGD_NR, USER_PTRS_PER_PGD);
}
```

release_segments() will release memory segments associated with the process on its Local Descriptor Table (LDT) if the architecture supports segments and the process was using them. Some applications, notably WINE use this
D.2.6 Deleting all memory regions (**exit_mmap()**) 289

feature

1132 Lock the mm

1133 **mpt** becomes the first VMA on the list

1134 Clear VMA related information from the mm so it may be unlocked

1137 Unlock the mm

1138-1139 Clear the mm statistics

1141 Flush the CPU for the address range

1142-1159 Step through every VMA that was associated with the mm

1143 Record what the next VMA to clear will be so this one may be deleted

1144-1146 Record the start, end and size of the region to be deleted

1148-1151 If there is a close operation associated with this VMA, call it

1152 Reduce the map count

1153 Remove the file/device mapping from the shared mappings list

1154 Free all pages associated with this region

1155-1156 If a file/device was mapped in this region, free it

1157 Free the VMA struct

1158 Move to the next VMA

1160 Flush the TLB for this whole mm as it is about to be unmapped

1163-1164 If the map_count is positive, it means the map count was not accounted for properly so call **BUG()** to mark it

1166 Clear the page tables associated with this region with **clear_page_tables()**
(See Section D.2.6.2)
D.2.6.2 Function: `clear_page_tables()`  *(mm/memory.c)*

This is the top-level function used to unmapped all PTEs and free pages within a region. It is used when pagetables needs to be torn down such as when the process exits or a region is unmapped.

```c
void clear_page_tables(struct mm_struct *mm,
                        unsigned long first, int nr)
```

146 Get the PGD for the `mm` being unmapped
147 Lock the pagetables
148-155 Step through all PGDs in the requested range. For each PGD found, call `free_one_pgd()` (See Section D.2.6.3)
156 Unlock the pagetables
159 Check the cache of available PGD structures. If there are too many PGDs in the PGD quicklist, some of them will be reclaimed

D.2.6.3 Function: `free_one_pgd()`  *(mm/memory.c)*

This function tears down one PGD. For each PMD in this PGD, `free_one_pmd()` will be called.

```c
static inline void free_one_pgd(pgd_t * dir)
```

109 Get the PGD for the `mm` being unmapped
110 Lock the pagetables
111-115 Step through all PMDs in the requested range. For each PMD found, call `free_one_pmd()` (See Section D.2.6.4)
116 Unlock the pagetables
118 Check the cache of available PMD structures. If there are too many PMDs in the PMD quicklist, some of them will be reclaimed

D.2.6.4 Function: `free_one_pmd()`  *(mm/memory.c)*

This function tears down one PMD. For each PTE in this PMD, `free_one_pte()` will be called.
D.2.6 Deleting all memory regions (\texttt{free_one_pgd()})

117\hspace{1em} \text{pgd\_ERROR(*dir);}  
118\hspace{1em} \text{pgd\_clear(dir);}  
119\hspace{1em} \text{return;}  
120\}  
121\  \text{pmd = pmd\_offset(dir, 0);}  
122\  \text{pgd\_clear(dir);}  
123\  \text{for (j = 0; j < PTRS\_PER\_PMD ; j++) { } }  
124\  \text{prefetchw(pmd+j+(PREFETCH\_STRIDE/16));}  
125\  \text{free\_one\_pmd(pmd+j);}  
126\  }  
127\  \text{pmd\_free(pmd);}  
128\}  

114-115 If no PGD exists here, return  
116-120 If the PGD is bad, flag the error and return  
1121 Get the first PMD in the PGD  
122 Clear the PGD entry  
123-126 For each PMD in this PGD, call \texttt{free_one_pmd()} (See Section D.2.6.4)  
127 Free the PMD page to the PMD quicklist. Later, \texttt{check_pgt_cache()} will be called and if the cache has too many PMD pages in it, they will be reclaimed  

D.2.6.4 Function: \texttt{free_one_pmd()} \hspace{1em} (\textit{mm/memory.c})

93 static inline void \texttt{free_one_pmd(pmd_t * dir)}  
94 {  
95 \hspace{1em} \text{pte_t * pte;}  
96 \hspace{1em} \  
97 \hspace{1em} if (pmd\_none(*dir))  
98 \hspace{1em} \    \text{return;}  
99 \hspace{1em} \  
100 \hspace{1em} if (pmd\_bad(*dir)) { }  
101 \hspace{1em} \    \text{pmd\_ERROR(*dir);}  
102 \hspace{1em} \    \text{pmd\_clear(dir);}  
103 \hspace{1em} \    \text{return;}  
104 \hspace{1em} \  
105 \hspace{1em} \text{pte = pte\_offset(dir, 0);}  
106 \hspace{1em} \text{pgd\_clear(dir);}  
107 \hspace{1em} \text{pte\_free(pte);}  
108 \}  

97-98 If no PMD exists here, return  
99-103 If the PMD is bad, flag the error and return
D.2.6 Deleting all memory regions (*free_one_pmd()*)

104 Get the first PTE in the PMD

105 Clear the PMD from the pagetable

106 Free the PTE page to the PTE quicklist cache with *pte_free()*. Later, *check_pgt_cache()* will be called and if the cache has too many PTE pages in it, they will be reclaimed
D.3 Searching Memory Regions

Contents

D.3 Searching Memory Regions .................................................. 293
  D.3.1 Finding a Mapped Memory Region .................................... 293
    D.3.1.1 Function: find_vma() ........................................... 293
    D.3.1.2 Function: find_vma_prev() .................................... 294
    D.3.1.3 Function: find_vma_intersection() ............................ 296
  D.3.2 Finding a Free Memory Region ........................................ 296
    D.3.2.1 Function: get_unmapped_area() ................................. 296
    D.3.2.2 Function: arch_get_unmapped_area() .......................... 297

The functions in this section deal with searching the virtual address space for mapped and free regions.

D.3.1 Finding a Mapped Memory Region

D.3.1.1 Function: find_vma() *(mm/mmap.c)*

661 struct vm_area_struct * find_vma(struct mm_struct * mm,
  unsigned long addr)
662 {
663     struct vm_area_struct *vma = NULL;
664
665     if (mm) {
666         /* Check the cache first. */
667         /* (Cache hit rate is typically around 35%). */
668         vma = mm->mmap_cache;
669         if (!((vma && vma->vm_end > addr &&
                vma->vm_start <= addr)) {
670             rb_node_t * rb_node;
671             rb_node = mm->mm_rb.rb_node;
672             vma = NULL;
673
674             while (rb_node) {
675                 struct vm_area_struct * vma_tmp;
676                 vma_tmp = rb_entry(rb_node,
677                               struct vm_area_struct, vm_rb);
678                 if (vma_tmp->vm_end > addr) {
679                     vma = vma_tmp;
680                     if (vma_tmp->vm_start <= addr)
681                         break;
682                 }
683                 rb_node = rb_node->rb_left;
684             }
685         }
The two parameters are the top level mm\_struct that is to be searched and the address the caller is interested in.

Default to returning NULL for address not found.

Make sure the caller does not try and search a bogus mm

mmap\_cache has the result of the last call to find\_vma(). This has a chance of not having to search at all through the red-black tree.

If it is a valid VMA that is being examined, check to see if the address being searched is contained within it. If it is, the VMA was the mmap\_cache one so it can be returned, otherwise the tree is searched.

Start at the root of the tree.

This block is the tree walk.

The macro, as the name suggests, returns the VMA this tree node points to.

Check if the next node traversed by the left or right leaf.

If the current VMA is what is required, exit the while loop.

If the VMA is valid, set the mmap\_cache for the next call to find\_vma().

Return the VMA that contains the address or as a side effect of the tree walk, return the VMA that is closest to the requested address.

D.3.1.2 Function: find\_vma\_prev() (mm/mmap.c)

```c
struct vm_area_struct * find_vma_prev(struct mm_struct * mm,
                                       unsigned long addr,
                                       struct vm_area_struct **pprev)
{
    if (mm) {
        /* Go through the RB tree quickly. */
        struct vm_area_struct * vma;
```
D.3.1 Finding a Mapped Memory Region (find_vma_prev())

```c
rb_node_t * rb_node, * rb_last_right, * rb_prev;
rb_node = mm->mm_rb.rb_node;
rb_last_right = rb_prev = NULL;
vma = NULL;

while (rb_node) {
    struct vm_area_struct * vma_tmp;
    vma_tmp = rb_entry(rb_node, struct vm_area_struct, vm_rb);
    if (vma_tmp->vm_end > addr) {
        vma = vma_tmp;
        rb_prev = rb_last_right;
        if (vma_tmp->vm_start <= addr)
            break;
        rb_node = rb_node->rb_left;
    } else {
        rb_last_right = rb_node;
        rb_node = rb_node->rb_right;
    }
}
if (vma) {
    if (vma->vm_rb.rb_left) {
        rb_prev = vma->vm_rb.rb_left;
        while (rb_prev->rb_right)
            rb_prev = rb_prev->rb_right;
    }
    *pprev = NULL;
    if (rb_prev)
        *pprev = rb_entry(rb_prev, struct
                          vm_area_struct, vm_rb);
    if ((rb_prev ? (*pprev)->vm_next : mm->mmap) != vma)
        BUG();
    return vma;
} else {
    *pprev = NULL;
    return NULL;
}
```

696-723 This is essentially the same as the find_vma() function already described. The only difference is that the last right node accesses is remembered as this
D.3.1 Finding a Mapped Memory Region (*find_vma_prev()*)

will represent the vma previous to the requested vma.

725-729 If the returned VMA has a left node, it means that it has to be traversed. It first takes the left leaf and then follows each right leaf until the bottom of the tree is found.

731-732 Extract the VMA from the red-black tree node

733-734 A debugging check, if this is the previous node, then its next field should point to the VMA being returned. If it is not, it is a bug

D.3.1.3 Function: *find_vma_intersection()* *(include/linux/mm.h)*

```c
673 static inline struct vm_area_struct * find_vma_intersection(
    struct mm_struct * mm,
    unsigned long start_addr, unsigned long end_addr)
674 {
    struct vm_area_struct * vma = find_vma(mm,start_addr);
    676 if (vma && end_addr <= vma->vm_start)
737 vma = NULL;
738 return vma;
680 }

675 Return the VMA closest to the starting address

677 If a VMA is returned and the end address is still less than the beginning of the returned VMA, the VMA does not intersect

679 Return the VMA if it does intersect

D.3.2 Finding a Free Memory Region

D.3.2.1 Function: *get_unmapped_area()* *(mm/mmap.c)*

The call graph for this function is shown at Figure 4.5.

```c
644 unsigned long get_unmapped_area(struct file *file, 
    unsigned long addr, 
    unsigned long len, 
    unsigned long pgoff, 
    unsigned long flags)
645 {
    if (flags & MAP_FIXED) {
647       if (addr > TASK_SIZE - len)
648           return -ENOMEM;
649       if (addr & ~PAGE_MASK)
650           return -EINVAL;
```
D.3.2 Finding a Free Memory Region \(\text{(get_unmapped_area())}\)

```c
return addr;
}

if (file && file->f_op && file->f_op->get_unmapped_area)
    return file->f_op->get_unmapped_area(file, addr,
                                           len, pgoff, flags);

return arch_get_unmapped_area(file, addr, len, pgoff, flags);
```

The parameters passed are

- file: The file or device being mapped
- addr: The requested address to map to
- len: The length of the mapping
- pgoff: The offset within the file being mapped
- flags: Protection flags

The parameters passed are

- file: The file or device being mapped
- addr: The requested address to map to
- len: The length of the mapping
- pgoff: The offset within the file being mapped
- flags: Protection flags

Sanity checked. If it is required that the mapping be placed at the specified address, make sure it will not overflow the address space and that it is page aligned.

If the `struct file` provides a `get_unmapped_area()` function, use it.

Else use `arch_get_unmapped_area()` (See Section D.3.2.2) as an anonymous version of the `get_unmapped_area()` function.

D.3.2.2 Function: arch_get_unmapped_area() \((\text{mm/mmap.c})\)

Architectures have the option of specifying this function for themselves by defining `HAVE_ARCH_UNMAPPED_AREA`. If the architectures does not supply one, this version is used.

```c
#ifndef HAVE_ARCH_UNMAPPED_AREA
static inline unsigned long arch_get_unmapped_area(
    struct file *filp,
    unsigned long addr, unsigned long len,
    unsigned long pgoff, unsigned long flags)
{
    struct vm_area_struct *vma;

    if (len > TASK_SIZE)
        return -ENOMEM;

    if (addr) {
        addr = PAGE_ALIGN(addr);
```
D.3.2 Finding a Free Memory Region (arch_get_unmapped_area())

Finding a Free Memory Region:

```c
vma = find_vma(current->mm, addr);
if (TASK_SIZE - len >= addr && (!vma || addr + len <= vma->vm_start))
    return addr;

addr = PAGE_ALIGN(TASK_UNMAPPED_BASE);
for (vma = find_vma(current->mm, addr); ; vma = vma->vm_next) {
    /* At this point: (!vma || addr < vma->vm_end). */
    if (TASK_SIZE - len < addr)
        return -ENOMEM;
    if (!vma || addr + len <= vma->vm_start)
        return addr;
    addr = vma->vm_end;
}
```

The architecture does not provide its own `arch_get_unmapped_area()` function, so this one is used instead.

The parameters are the same as those for `get_unmapped_area()` (See Section D.3.2.1).

Sanity check, make sure the required map length is not too long.

If an address is provided, use it for the mapping.

Make sure the address is page aligned.

`find_vma()` (See Section D.3.1.1) will return the region closest to the requested address.

Make sure the mapping will not overlap with another region. If it does not, return it as it is safe to use. Otherwise it gets ignored.

`TASK_UNMAPPED_BASE` is the starting point for searching for a free region to use.

Starting from `TASK_UNMAPPED_BASE`, linearly search the VMAs until a large enough region between them is found to store the new mapping. This is essentially a first fit search.

If an external function is provided, it still needs to be declared here.
D.4 Locking and Unlocking Memory Regions

Contents

D.4 Locking and Unlocking Memory Regions 299
  D.4.1 Locking a Memory Region 299
    D.4.1.1 Function: sys_mlock() 299
    D.4.1.2 Function: sys_mlockall() 300
    D.4.1.3 Function: do_mlockall() 302
    D.4.1.4 Function: do_mlock() 303
  D.4.2 Unlocking the region 305
    D.4.2.1 Function: sys_munlock() 305
    D.4.2.2 Function: sys_munlockall() 306
  D.4.3 Fixing up regions after locking/unlocking 306
    D.4.3.1 Function: mlock_fixup() 306
    D.4.3.2 Function: mlock_fixup_all() 308
    D.4.3.3 Function: mlock_fixup_start() 308
    D.4.3.4 Function: mlock_fixup_end() 309
    D.4.3.5 Function: mlock_fixup_middle() 310

This section contains the functions related to locking and unlocking a region. The main complexity in them is how the regions need to be fixed up after the operation takes place.

D.4.1 Locking a Memory Region

D.4.1.1 Function: sys_mlock() (mm/mlock.c)

The call graph for this function is shown in Figure 4.10. This is the system call mlock() for locking a region of memory into physical memory. This function simply checks to make sure that process and user limits are not exceeded and that the region to lock is page aligned.

195 asmlinkage long sys_mlock(unsigned long start, size_t len)
196 {
197    unsigned long locked;
198    unsigned long lock_limit;
199    int error = -ENOMEM;
200    
201    down_write(&current->mm->mmap_sem);
202    len = PAGE_ALIGN(len + (start & ~PAGE_MASK));
203    start &= PAGE_MASK;
204    
205    locked = len >> PAGE_SHIFT;
206    locked += current->mm->locked_vm;
207    
208    lock_limit = current->rlim[RLIMIT_MEMLOCK].rlim_cur;
209    lock_limit >>= PAGE_SHIFT;

D.4.1 Locking a Memory Region (sys_mlock())

210 211 /* check against resource limits */
212 if (locked > lock_limit)
213     goto out;
214
215 /* we may lock at most half of physical memory... */
216 /* (this check is pretty bogus, but doesn’t hurt) */
217 if (locked > num_physpages/2)
218     goto out;
219
220 error = do_mlock(start, len, 1);
221 out:
222  up_write(&current->mm->mmap_sem);
223  return error;
224 }

201 Take the semaphore, we are likely to sleep during this so a spinlock can not
    be used
202 Round the length up to the page boundary
203 Round the start address down to the page boundary
205 Calculate how many pages will be locked
206 Calculate how many pages will be locked in total by this process
208-209 Calculate what the limit is to the number of locked pages
212-213 Do not allow the process to lock more than it should
217-218 Do not allow the process to map more than half of physical memory
220 Call do_mlock() (See Section D.4.1.4) which starts the “real” work by find the
    VMA closest to the area to lock before calling mlock_fixup() (See Section D.4.3.1)
222 Free the semaphore
223 Return the error or success code from do_mlock()

D.4.1.2 Function: sys_mlockall() (mm/mlock.c)

    This is the system call mlockall() which attempts to lock all pages in the calling
process in memory. If MCL_CURRENT is specified, all current pages will be locked. If
MCL_FUTURE is specified, all future mappings will be locked. The flags may be or-ed
    together. This function makes sure that the flags and process limits are ok before
calling do_mlockall().
asmlinkage long sys_mlockall(int flags) {
    unsigned long lock_limit;
    int ret = -EINVAL;
    down_write(&current->mm->mmap_sem);
    if (!flags || (flags & ~(MCL_CURRENT | MCL_FUTURE)))
        goto out;
    lock_limit = current->rlim[RLIMIT_MEMLOCK].rlim_cur;
    lock_limit >>= PAGE_SHIFT;
    ret = -ENOMEM;
    if (current->mm->total_vm > lock_limit)
        goto out;
    /* we may lock at most half of physical memory... */
    /* (this check is pretty bogus, but doesn’t hurt) */
    if (current->mm->total_vm > num_physpages/2)
        goto out;
    ret = do_mlockall(flags);
    out:
    up_write(&current->mm->mmap_sem);
    return ret;
}

By default, return -EINVAL to indicate invalid parameters

Acquire the current mm_struct semaphore

Make sure that some valid flag has been specified. If not, goto out to
unlock the semaphore and return -EINVAL

Check the process limits to see how many pages may be locked

From here on, the default error is -ENOMEM

If the size of the locking would exceed set limits, then goto out

Do not allow this process to lock more than half of physical memory. This
is a bogus check because four processes locking a quarter of physical memory
each will bypass this. It is acceptable though as only root processes are allowed
to lock memory and are unlikely to make this type of mistake

Call the core function do_mlockall() (See Section D.4.1.3)

Unlock the semaphore and return
D.4.1.3 Function: do_mlockall() (mm/mlock.c)

238 static int do_mlockall(int flags)
239 {
240     int error;
241     unsigned int def_flags;
242     struct vm_area_struct * vma;
243
244     if (!capable(CAP_IPC_LOCK))
245         return -EPERM;
246
247     def_flags = 0;
248     if (flags & MCL_FUTURE)
249         def_flags = VM_LOCKED;
250     current->mm->def_flags = def_flags;
251
252     error = 0;
253     for (vma = current->mm->mmap; vma; vma = vma->vm_next) {
254         unsigned int newflags;
255
256         newflags = vma->vm_flags | VM_LOCKED;
257         if (!(flags & MCL_CURRENT))
258             newflags &= ~VM_LOCKED;
259         error = mlock_fixup(vma, vma->vm_start, vma->vm_end, newflags);
260         if (error)
261             break;
262     }
263     return error;
264 }

244-245 The calling process must be either root or have CAP_IPC_LOCK capabilities

248-250 The MCL_FUTURE flag says that all future pages should be locked so if set, the def_flags for VMAs should be VM_LOCKED

253-262 Cycle through all VMAs

256 Set the VM_LOCKED flag in the current VMA flags

257-258 If the MCL_CURRENT flag has not been set requesting that all current pages be locked, then clear the VM_LOCKED flag. The logic is arranged like this so that the unlock code can use this same function just with no flags

259 Call mlock_fixup() (See Section D.4.3.1) which will adjust the regions to match the locking as necessary
260-261 If a non-zero value is returned at any point, stop locking. It is interesting
to note that VMAs already locked will not be unlocked.

263 Return the success or error value.

D.4.1.4 Function: do_mlock()  (mm/mlock.c)

This function is responsible for starting the work needed to either lock or
unlock a region depending on the value of the on parameter. It is broken up into
two sections. The first makes sure the region is page aligned (despite the fact the
only two callers of this function do the same thing) before finding the VMA that
is to be adjusted. The second part then sets the appropriate flags before calling
mlock_fixup() for each VMA that is affected by this locking.

148 static int do_mlock(unsigned long start, size_t len, int on)
149 {
150    unsigned long nstart, end, tmp;
151    struct vm_area_struct * vma, * next;
152    int error;
153
154    if (on && !capable(CAP_IPC_LOCK))
155       return -EPERM;
156    len = PAGE_ALIGN(len);
157    end = start + len;
158    if (end < start)
159       return -EINVAL;
160    if (end == start)
161       return 0;
162    vma = find_vma(current->mm, start);
163    if (!vma || vma->vm_start > start)
164       return -ENOMEM;

Page align the request and find the VMA

154 Only root processes can lock pages

156 Page align the length. This is redundant as the length is page aligned in the
parent functions

157-159 Calculate the end of the locking and make sure it is a valid region. Return
-EINVAL if it is not

160-161 if locking a region of size 0, just return

162 Find the VMA that will be affected by this locking

163-164 If the VMA for this address range does not exist, return -ENOMEM
for (nstart = start ; ; ) {
    unsigned int newflags;
    newflags = vma->vm_flags | VM_LOCKED;
    if (!on)
        newflags &= ~VM_LOCKED;
    if (vma->vm_end >= end) {
        error = mlock_fixup(vma, nstart, end, newflags);
        break;
    }
    tmp = vma->vm_end;
    next = vma->vm_next;
    error = mlock_fixup(vma, nstart, tmp, newflags);
    if (error)
        break;
    nstart = tmp;
    vma = next;
    if (!vma || vma->vm_start != nstart) {
        error = -ENOMEM;
        break;
    }
}

Walk through the VMAs affected by this locking and call mlock_fixup() for each of them.

166-192 Cycle through as many VMAs as necessary to lock the pages

171 Set the VM_LOCKED flag on the VMA

172-173 Unless this is an unlock in which case, remove the flag

175-177 If this VMA is the last VMA to be affected by the unlocking, call mlock_fixup() with the end address for the locking and exit

180-190 Else this is whole VMA needs to be locked. To lock it, the end of this VMA is pass as a parameter to mlock_fixup() (See Section D.4.3.1) instead of the end of the actual locking

180 tmp is the end of the mapping on this VMA

181 next is the next VMA that will be affected by the locking
D.4.2 Unlocking the region

D.4.2.1 Function: sys_munlock() (mm/mlock.c)

Page align the request before calling do_mlock() which begins the real work of fixing up the regions.

```
asmlinkage long sys_munlock(unsigned long start, size_t len)
{
    int ret;
    down_write(&current->mm->mmap_sem);
    len = PAGE_ALIGN(len + (start & ~PAGE_MASK));
    start &= PAGE_MASK;
    ret = do_mlock(start, len, 0);
    up_write(&current->mm->mmap_sem);
    return ret;
}
```

230 Acquire the semaphore protecting the mm_struct

231 Round the length of the region up to the nearest page boundary

232 Round the start of the region down to the nearest page boundary

233 Call do_mlock() (See Section D.4.1.4) with 0 as the third parameter to unlock the region

234 Release the semaphore

235 Return the success or failure code
D.4.2.2 Function: sys_munlockall()  (mm/mlock.c)

Trivial function. If the flags to mlockall() are 0 it gets translated as none of the current pages must be present and no future mappings should be locked either which means the VM_LOCKED flag will be removed on all VMAs.

```c
asmlinkage long sys_munlockall(void)
{
    int ret;

    down_write(&current->mm->mmap_sem);
    ret = do_mlockall(0);
    up_write(&current->mm->mmap_sem);
    return ret;
}
```

297 Acquire the semaphore protecting the mm_struct

298 Call do_mlockall() (See Section D.4.1.3) with 0 as flags which will remove the VM_LOCKED from all VMAs

299 Release the semaphore

300 Return the error or success code

D.4.3 Fixing up regions after locking/unlocking

D.4.3.1 Function: mlock_fixup()  (mm/mlock.c)

This function identifies four separate types of locking that must be addressed. There first is where the full VMA is to be locked where it calls mlock_fixup_all(). The second is where only the beginning portion of the VMA is affected, handled by mlock_fixup_start(). The third is the locking of a region at the end handled by mlock_fixup_end() and the last is locking a region in the middle of the VMA with mlock_fixup_middle().

```c
static int mlock_fixup(struct vm_area_struct * vma,
        unsigned long start, unsigned long end, unsigned int newflags)
{
    int pages, retval;

    if (newflags == vma->vm_flags)
        return 0;

    if (start == vma->vm_start) {
        if (end == vma->vm_end)
            retval = mlock_fixup_all(vma, newflags);
        else
```
D.4.3 Fixing up regions after locking/unlocking (mlock_fixup())

129     retval = mlock_fixup_start(vma, end, newflags);
130 } else {
131     if (end == vma->vm_end)
132         retval = mlock_fixup_end(vma, start, newflags);
133     else
134         retval = mlock_fixup_middle(vma, start,
135                                         end, newflags);
136 }
137     if (!retval) {
138         /* keep track of amount of locked VM */
139         pages = (end - start) >> PAGE_SHIFT;
140         if (newflags & VM_LOCKED) {
141             pages = -pages;
142             make_pages_present(start, end);
143         }
144         vma->vm_mm->locked_vm -= pages;
145     }
146     return retval;

122-123 If no change is to be made, just return
125 If the start of the locking is at the start of the VMA, it means that either the full region is to be locked or only a portion at the beginning
126-127 The full VMA is being locked, call mlock_fixup_all() (See Section D.4.3.2)
128-129 Part of the VMA is being locked with the start of the VMA matching the start of the locking, call mlock_fixup_start() (See Section D.4.3.3)
130 Else either a region at the end is to be locked or a region in the middle
131-132 The end of the locking matches the end of the VMA, call mlock_fixup_end() (See Section D.4.3.4)
133-134 A region in the middle of the VMA is to be locked, call mlock_fixup_middle() (See Section D.4.3.5)
136-144 The fixup functions return 0 on success. If the fixup of the regions succeed and the regions are now marked as locked, call make_pages_present() which makes some basic checks before calling get_user_pages() which faults in all the pages in the same way the page fault handler does
D.4.3.2 Function: \textit{mlock\_fixup\_all} (\textit{mm/mlock.c})

15 static inline int mlock_fixup_all(struct vm_area_struct * vma, int newflags)
16 {
17    spin_lock(&vma->vm_mm->page_table_lock);
18    vma->vm_flags = newflags;
19    spin_unlock(&vma->vm_mm->page_table_lock);
20    return 0;
21 }

17-19 Trivial, lock the VMA with the spinlock, set the new flags, release the lock and return success

D.4.3.3 Function: \textit{mlock\_fixup\_start} (\textit{mm/mlock.c})

Slightly more complicated. A new VMA is required to represent the affected region. The start of the old VMA is moved forward

23 static inline int mlock_fixup_start(struct vm_area_struct * vma, unsigned long end, int newflags)
24 {
25    struct vm_area_struct * n;
26
27    n = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
28    if (!n)
29       return -EAGAIN;
30    *n = *vma;
31    n->vm_end = end;
32    n->vm_flags = newflags;
33    n->vm_ranend = 0;
34    if (n->vm_file)
35       get_file(n->vm_file);
36    if (n->vm_ops & n->vm_ops->open)
37       n->vm_ops->open(n);
38    vma->vm_pgoff += (end - vma->vm_start) >> PAGE_SHIFT;
39    lock_vma_mappings(vma);
40    spin_lock(&vma->vm_mm->page_table_lock);
41    vma->vm_start = end;
42    __insert_vm_struct(current->mm, n);
43    spin_unlock(&vma->vm_mm->page_table_lock);
44    unlock_vma_mappings(vma);
45    return 0;
46 }

28 Allocate a VMA from the slab allocator for the affected region
D.4.3 Fixing up regions after locking/unlocking (mlock_fixup_start())

31-34 Copy in the necessary information

35-36 If the VMA has a file or device mapping, get_file() will increment the reference count

37-38 If an open() function is provided, call it

39 Update the offset within the file or device mapping for the old VMA to be the end of the locked region

40 lock_vma_mappings() will lock any files if this VMA is a shared region

41-44 Lock the parent mm_struct, update its start to be the end of the affected region, insert the new VMA into the processes linked lists (See Section D.2.2.1) and release the lock

45 Unlock the file mappings with unlock_vma_mappings()

46 Return success

D.4.3.4 Function: mlock_fixup_end() (mm/mlock.c)

Essentially the same as mlock_fixup_start() except the affected region is at the end of the VMA.

49 static inline int mlock_fixup_end(struct vm_area_struct * vma, unsigned long start, int newflags)
50 {
51   struct vm_area_struct * n;
52   n = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
53   if (!n)
54     return -EAGAIN;
55   *n = *vma;
56   n->vm_start = start;
57   n->vm_pgoff += (n->vm_start - vma->vm_start) >> PAGE_SHIFT;
58   n->vm_flags = newflags;
59   n->vm_raend = 0;
60   if (n->vm_file)
61     get_file(n->vm_file);
62   if (n->vm_ops && n->vm_ops->open)
63     n->vm_ops->open(n);
64   lock_vma_mappings(vma);
65   spin_lock(&vma->vm_mm->page_table_lock);
66   vma->vm_end = start;
67   __insert_vm_struct(current->mm, n);
68   spin_unlock(&vma->vm_mm->page_table_lock);
69   unlock_vma_mappings(vma);
70   }

D.4.3 Fixing up regions after locking/unlocking (`mlock_fixup_end()`)  

```c
310 return 0;
```

54 Alloc a VMA from the slab allocator for the affected region
57-61 Copy in the necessary information and update the offset within the file or device mapping
62-63 If the VMA has a file or device mapping, get_file() will increment the reference count
64-65 If an open() function is provided, call it
66 lock_vma_mappings() will lock any files if this VMA is a shared region
67-70 Lock the parent mm_struct, update its start to be the end of the affected region, insert the new VMA into the processes linked lists (See Section D.2.2.1) and release the lock
71 Unlock the file mappings with unlock_vma_mappings()
72 Return success

D.4.3.5 Function: `mlock_fixup_middle()` (`mm/mlock.c`)  
Similar to the previous two fixup functions except that 2 new regions are required to fix up the mapping.

```c
75 static inline int mlock_fixup_middle(struct vm_area_struct * vma,
                                           unsigned long start, unsigned long end, int newflags)
77 {
78    struct vm_area_struct * left, * right;
79    left = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
80    if (!left)
81        return -EAGAIN;
82    right = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
83    if (!right) {
84        kmem_cache_free(vm_area_cachep, left);
85        return -EAGAIN;
86    }
87    *left = *vma;
88    *right = *vma;
89    left->vm_end = start;
90    right->vm_start = end;
91    right->vm_pgoff += (right->vm_start - left->vm_start) >> PAGE_SHIFT;
92    vma->vm_flags = newflags;
```
D.4.3 Fixing up regions after locking/unlocking (mlock_fixup_middle())

```c
94   left->vm_raend = 0;
95   right->vm_raend = 0;
96   if (vma->vm_file)
97       atomic_add(2, &vma->vm_file->f_count);
98
99   if (vma->vm_ops && vma->vm_ops->open) {
100      vma->vm_ops->open(left);
101      vma->vm_ops->open(right);
102   }
103   vma->vm_raend = 0;
104   vma->vm_pgoff += (start - vma->vm_start) >> PAGE_SHIFT;
105   lock_vma_mappings(vma);
106   spin_lock(&vma->vm_mm->page_table_lock);
107   vma->vm_start = start;
108   vma->vm_end = end;
109   vma->vm_flags = newflags;
110   __insert_vm_struct(current->mm, left);
111   __insert_vm_struct(current->mm, right);
112   spin_unlock(&vma->vm_mm->page_table_lock);
113   unlock_vma_mappings(vma);
114   return 0;
115 }
```

80-87 Allocate the two new VMAs from the slab allocator

88-89 Copy in the information from the old VMA into them

90 The end of the left region is the start of the region to be affected

91 The start of the right region is the end of the affected region

92 Update the file offset

93 The old VMA is now the affected region so update its flags

94-95 Make the readahead window 0 to ensure pages not belonging to their regions
are not accidently read ahead

96-97 Increment the reference count to the file/device mapping if there is one

99-102 Call the open() function for the two new mappings

103-104 Cancel the readahead window and update the offset within the file to be
the beginning of the locked region

105 Lock the shared file/device mappings

106-112 Lock the parent mm_struct, update the VMA and insert the two new
regions into the process before releasing the lock again
Unlock the shared mappings

Return success
This section deals with the page fault handler. It begins with the architecture specific function for the x86 and then moves to the architecture independent layer. The architecture specific functions all have the same responsibilities.

### D.5.1 x86 Page Fault Handler

**D.5.1.1 Function: do_page_fault()**

The call graph for this function is shown in Figure 4.12. This function is the x86 architecture dependent function for the handling of page fault exception handlers. Each architecture registers their own but all of them have similar responsibilities.

```c
asmlinkage void do_page_fault(struct pt_regs *regs, unsigned long error_code)
{
    struct task_struct *tsk;
    struct mm_struct *mm;
    struct vm_area_struct *vma;
    unsigned long address;
    unsigned long page;
    unsigned long fixup;
    int write;
    siginfo_t info;

    /* get the address */
    __asm__("movl %cr2,%0":"=r" (address));
```
D.5.1 x86 Page Fault Handler (do_page_fault())

153 /* It's safe to allow irq's after cr2 has been saved */
154 if (regs->eflags & X86_EFLAGS_IF)
155   local_irq_enable();
156
157 tsk = current;
158

Function preamble. Get the fault address and enable interrupts

140 The parameters are

  regs  is a struct containing what all the registers at fault time
  error_code  indicates what sort of fault occurred

152 As the comment indicates, the cr2 register holds the fault address

155-156 If the fault is from within an interrupt, enable them

158 Set the current task

173 if (address >= TASK_SIZE && !(error_code & 5))
174   goto vmalloc_fault;
175
176 mm = tsk->mm;
177 info.si_code = SEGV_MAPERR;
178
183 if (in_interrupt() || !mm)
184   goto no_context;
185
Check for exceptional faults, kernel faults, fault in interrupt and fault with no memory context

173 If the fault address is over TASK_SIZE, it is within the kernel address space.
   If the error code is 5, then it means it happened while in kernel mode and is not a protection error so handle a vmalloc fault

176 Record the working mm

183 If this is an interrupt, or there is no memory context (such as with a kernel thread), there is no way to safely handle the fault so goto no_context

186 down_read(&mm->mmap_sem);
187
188 vma = find_vma(mm, address);
189 if (!vma)
If a fault in userspace, find the VMA for the faulting address and determine if it is a good area, a bad area or if the fault occurred near a region that can be expanded such as the stack.

186 Take the long lived mm semaphore

188 Find the VMA that is responsible or is closest to the faulting address

189-190 If a VMA does not exist at all, goto bad_area

191-192 If the start of the region is before the address, it means this VMA is the correct VMA for the fault so goto good_area which will check the permissions

193-194 For the region that is closest, check if it can grow down (VM_GROWSDOWN). If it does, it means the stack can probably be expanded. If not, goto bad_area

195-204 Check to make sure it isn’t an access below the stack. If the error_code is 4, it means it is running in userspace

205-206 The stack is the only region with VM_GROWSDOWN set so if we reach here, the stack is expanded with with expand_stack() (See Section D.5.2.1), if it fails, goto bad_area

211 good_area:
212 info.si_code = SEGV_ACCERR;
213 write = 0;
214 switch (error_code & 3) {
215     default: /* 3: write, present */
216 #ifndef TEST_VERIFY_AREA
if (regs->cs == KERNEL_CS)
    printk("WP fault at %08lx\n", regs->eip);
#endif
/* fall through */

    case 2:    /* write, not present */
        if (!(vma->vm_flags & VM_WRITE))
            goto bad_area;
        write++;
        break;
    case 1:    /* read, present */
        goto bad_area;
    case 0:    /* read, not present */
        if (!(vma->vm_flags & (VM_READ | VM_EXEC)))
            goto bad_area;

There is the first part of a good area is handled. The permissions need to be checked in case this is a protection fault.

By default return an error

Check the error code against bits 0 and 1 of the error code. Bit 0 at 0 means page was not present. At 1, it means a protection fault like a write to a read-only area. Bit 1 is 0 if it was a read fault and 1 if a write

If it is 3, both bits are 1 so it is a write protection fault

Bit 1 is a 1 so it is a write fault

If the region can not be written to, it is a bad write to goto bad_area.

If the region can be written to, this is a page that is marked Copy On Write (COW)

Flag that a write has occurred

This is a read and the page is present. There is no reason for the fault so must be some other type of exception like a divide by zero, goto bad_area where it is handled

A read occurred on a missing page. Make sure it is ok to read or exec this page. If not, goto bad_area. The check for exec is made because the x86 can not exec protect a page and instead uses the read protect flag. This is why both have to be checked.

D.5.1 x86 Page Fault Handler (do_page_fault())
At this point, an attempt is going to be made to handle the fault gracefully with handle_mm_fault().

Call handle_mm_fault() with the relevant information about the fault. This is the architecture independent part of the handler.

A return of 1 means it was a minor fault. Update statistics.
A return of 2 means it was a major fault. Update statistics.
A return of 0 means some IO error happened during the fault so go to the do_sigbus handler.
Any other return means memory could not be allocated for the fault so we are out of memory. In reality this does not happen as another function out_of_memory() is invoked in mm/oom_kill.c before this could happen which is a lot more graceful about who it kills.

Release the lock to the mm.
Return as the fault has been successfully handled.
/* User mode accesses just cause a SIGSEGV */
if (error_code & 4) {
    tsk->thread.cr2 = address;
    tsk->thread.error_code = error_code;
    tsk->thread.trap_no = 14;
    info.si_signo = SIGSEGV;
    info.si_errno = 0;
    /* info.si_code has been set above */
    info.si_addr = (void *)address;
    force_sig_info(SIGSEGV, &info, tsk);
    return;
}

/*
 * Pentium F0 0F C7 C8 bug workaround.
 */
if (boot_cpu_data.f00f_bug) {
    unsigned long nr;
    nr = (address - idt) >> 3;
    if (nr == 6) {
        do_invalid_op(regs, 0);
        return;
    }
}

This is the bad area handler such as using memory with no \texttt{vm_area_struct}
managing it. If the fault is not by a user process or the f00f bug, the \texttt{no-context}
label is fallen through to.

271 An error code of 4 implies userspace so it is a simple case of sending a \texttt{SIGSEGV}
to kill the process

272-274 Set thread information about what happened which can be read by a
debugger later

275 Record that a \texttt{SIGSEGV} signal was sent

276 clear errno

278 Record the address

279 Send the \texttt{SIGSEGV} signal. The process will exit and dump all the relevant
information

280 Return as the fault has been successfully handled
An bug in the first Pentiums was called the f00f bug which caused the processor to constantly page fault. It was used as a local DoS attack on a running Linux system. This bug was trapped within a few hours and a patch released. Now it results in a harmless termination of the process rather than a rebooting system.

```c
/* Are we prepared to handle this kernel fault? */
if ((fixup = search_exception_table(regs->eip)) != 0) {
    regs->eip = fixup;
    return;
}
```

Search the exception table with `search_exception_table()` to see if this exception be handled and if so, call the proper exception handler after returning. This is really important during `copy_from_user()` and `copy_to_user()` when an exception handler is especially installed to trap reads and writes to invalid regions in userspace without having to make expensive checks. It means that a small fixup block of code can be called rather than falling through to the next block which causes an oops.

```c
/*
 * Oops. The kernel tried to access some bad page. We'll have to terminate things with extreme prejudice.
 */

bust_spinlocks(1);

if (address < PAGE_SIZE)
    printk(KERN_ALERT "Unable to handle kernel NULL pointer dereference");
else
    printk(KERN_ALERT "Unable to handle kernel paging request");

printk(" at virtual address %08lx\n", address);
printk(" printing eip:\n");
printk("%08lx\n", regs->eip);
asm("movl %cr3,%0":="r" (page));
page = ((unsigned long *)__va(page))[address >> 22];
printk(KERN_ALERT "*pde = %08lx\n", page);
if (page & 1) {
    page &= PAGE_MASK;
    address &= 0x003ff000;
    page = ((unsigned long *)
```
D.5.1 x86 Page Fault Handler (do_page_fault())

```c
__va(page))[address >> PAGE_SHIFT];
printk(KERN_ALERT "*pte = %08lx\n", page);
}
die("Oops", regs, error_code);
bust_spinlocks(0);
do_exit(SIGKILL);
```

This is the no_context handler. Some bad exception occurred which is going to end up in the process been terminated in all likeliness. Otherwise the kernel faulted when it definitely should have and an OOPS report is generated.

309-329 Otherwise the kernel faulted when it really shouldn’t have and it is a kernel bug. This block generates an oops report

309 Forcibly free spinlocks which might prevent a message getting to console

311-312 If the address is < PAGE_SIZE, it means that a null pointer was used. Linux deliberately has page 0 unassigned to trap this type of fault which is a common programming error

313-314 Otherwise it is just some bad kernel error such as a driver trying to access userspace incorrectly

315-320 Print out information about the fault

321-326 Print out information about the page been faulted

327 Die and generate an oops report which can be used later to get a stack trace so a developer can see more accurately where and how the fault occurred

329 Forcibly kill the faulting process

```c
out_of_memory:
if (tsk->pid == 1) {
yield();
goto survive;
}
up_read(&mm->mmap_sem);
printk("VM: killing process %s\n", tsk->comm);
if (error_code & 4)
do_exit(SIGKILL);
goto no_context;
```

The out of memory handler. Usually ends with the faulting process getting killed unless it is init

336-339 If the process is init, just yield and goto survive which will try to handle the fault gracefully. init should never be killed
Free the mm semaphore

Print out a helpful “You are Dead” message

If from userspace, just kill the process

If in kernel space, go to the no_context handler which in this case will probably result in a kernel oops

```c
    do_sigbus:
    up_read(&mm->mmap_sem);
    tsk->thread.cr2 = address;
    tsk->thread.error_code = error_code;
    tsk->thread.trap_no = 14;
    info.si_signo = SIGBUS;
    info.si_errno = 0;
    info.si_code = BUS_ADRERR;
    info.si_addr = (void *)address;
    force_sig_info(SIGBUS, &info, tsk);

    /* Kernel mode? Handle exceptions or die */
    if (!(error_code & 4))
        goto no_context;
    return;
```

Free the mm lock

Fill in information to show a SIGBUS occurred at the faulting address so that a debugger can trap it later

Send the signal

If in kernel mode, try and handle the exception during no_context

If in userspace, just return and the process will die in due course
### D.5.1 x86 Page Fault Handler (\texttt{do_page_fault()})

This is the \texttt{vmalloc} fault handler. When pages are mapped in the \texttt{vmalloc} space, only the reference page table is updated. As each process references this area, a fault will be trapped and the process page tables will be synchronised with the reference page table here.

376 Get the offset within a PGD

381 Copy the address of the PGD for the process from the cr3 register to pgd

382 Calculate the pgd pointer from the process PGD

383 Calculate for the kernel reference PGD

385-386 If the pgd entry is invalid for the kernel page table, goto \texttt{no_context}

386 Set the page table entry in the process page table with a copy from the kernel reference page table

```c
vmalloc_fault:
{
    int offset = __pgd_offset(address);
pgd_t *pgd, *pgd_k;
pmd_t *pmd, *pmd_k;
pte_t *pte_k;

    asm("movl %\%cr3,\%0":"=r" (pgd));
    pgd = offset + (pgd_t *)__va(pgd);
    pgd_k = init_mm.pgd + offset;

    if (!pgd_present(*pgd_k))
        goto no_context;
    set_pgd(pgd, *pgd_k);

    pmd = pmd_offset(pgd, address);
    pmd_k = pmd_offset(pgd_k, address);
    if (!pmd_present(*pmd_k))
        goto no_context;
    set_pmd(pmd, *pmd_k);

    pte_k = pte_offset(pmd_k, address);
    if (!pte_present(*pte_k))
        goto no_context;
    return;
}
```

```c
This is the \texttt{vmalloc} fault handler. When pages are mapped in the \texttt{vmalloc} space, only the reference page table is updated. As each process references this area, a fault will be trapped and the process page tables will be synchronised with the reference page table here.
```

376 Get the offset within a PGD

381 Copy the address of the PGD for the process from the cr3 register to pgd

382 Calculate the pgd pointer from the process PGD

383 Calculate for the kernel reference PGD

385-386 If the pgd entry is invalid for the kernel page table, goto \texttt{no_context}

386 Set the page table entry in the process page table with a copy from the kernel reference page table

```c
vmalloc_fault:
{
    int offset = __pgd_offset(address);
pgd_t *pgd, *pgd_k;
pmd_t *pmd, *pmd_k;
pte_t *pte_k;

    asm("movl %\%cr3,\%0":"=r" (pgd));
    pgd = offset + (pgd_t *)__va(pgd);
    pgd_k = init_mm.pgd + offset;

    if (!pgd_present(*pgd_k))
        goto no_context;
    set_pgd(pgd, *pgd_k);

    pmd = pmd_offset(pgd, address);
    pmd_k = pmd_offset(pgd_k, address);
    if (!pmd_present(*pmd_k))
        goto no_context;
    set_pmd(pmd, *pmd_k);

    pte_k = pte_offset(pmd_k, address);
    if (!pte_present(*pte_k))
        goto no_context;
    return;
}
```
D.5.2 Expanding the Stack

D.5.2.1 Function: expand_stack()  
This function is called by the architecture dependant page fault handler. The VMA supplied is guaranteed to be one that can grow to cover the address.

```c
static inline int expand_stack(struct vm_area_struct *vma, unsigned long address)
{
    unsigned long grow;
    /*
     * vma->vm_start/vm_end cannot change under us because
     * the caller is required
     * to hold the mmap_sem in write mode. We need to get the
     * spinlock only before relocating the vma range ourself.
     */
    address &= PAGE_MASK;
    spin_lock(&vma->vm_mm->page_table_lock);
    grow = (vma->vm_start - address) >> PAGE_SHIFT;
    if (vma->vm_end - address > current->rlim[RLIMIT_STACK].rlim_cur ||
        ((vma->vm_mm->total_vm + grow) << PAGE_SHIFT) >
        current->rlim[RLIMIT_AS].rlim_cur) {
        spin_unlock(&vma->vm_mm->page_table_lock);
        return -ENOMEM;
    }
    vma->vm_start = address;
    vma->vm_pgoff -= grow;
    vma->vm_mm->total_vm += grow;
    if (vma->vm_flags & VM_LOCKED)
        vma->vm_mm->locked_vm += grow;
    spin_unlock(&vma->vm_mm->page_table_lock);
    return 0;
}
```
Round the address down to the nearest page boundary

Lock the page tables spinlock

Calculate how many pages the stack needs to grow by

Check to make sure that the size of the stack does not exceed the process limits

Check to make sure that the size of the address space will not exceed process limits after the stack is grown

If either of the limits are reached, return -ENOMEM which will cause the faulting process to segfault

Grow the VMA down

Update the amount of address space used by the process

If the region is locked, update the number of locked pages used by the process

Unlock the process page tables and return success

D.5.3 Architecture Independent Page Fault Handler

This is the top level pair of functions for the architecture independent page fault handler.

D.5.3.1 Function: handle_mm_fault()  (mm/memory.c)

The call graph for this function is shown in Figure 4.14. This function allocates the PMD and PTE necessary for this new PTE hat is about to be allocated. It takes the necessary locks to protect the page tables before calling handle_pte_fault() to fault in the page itself.

```c
int handle_mm_fault(struct mm_struct *mm, 
    struct vm_area_struct *vma, 
    unsigned long address, int write_access) 
{
    pgd_t *pgd;
    pmd_t *pmd;
    current->state = TASK_RUNNING;
    pgd = pgd_offset(mm, address);
    /*
     * We need the page table lock to synchronize with kswapd
     * and the SMP-safe atomic PTE updates.
```
The parameters of the function are;

- **mm** is the `mm_struct` for the faulting process
- **vma** is the `vm_area_struct` managing the region the fault occurred in
- **address** is the faulting address
- **write_access** is 1 if the fault is a write fault

Set the current state of the process

Get the pgd entry from the top level page table

Lock the `mm_struct` as the page tables will change

`pmd_alloc()` will allocate a `pmd_t` if one does not already exist

If the `pmd` has been successfully allocated then...

Allocate a PTE for this address if one does not already exist

Handle the page fault with `handle_pte_fault()` (See Section D.5.3.2) and return the status code

Failure path, unlock the `mm_struct`

Return -1 which will be interpreted as an out of memory condition which is correct as this line is only reached if a PMD or PTE could not be allocated.
D.5.3.2 Function: handle_pte_fault() (mm/memory.c)

This function decides what type of fault this is and which function should handle it. `do_no_page()` is called if this is the first time a page is to be allocated. `do_swap_page()` handles the case where the page was swapped out to disk with the exception of pages swapped out from tmpfs. `do_wp_page()` breaks COW pages. If none of them are appropriate, the PTE entry is simply updated. If it was written to, it is marked dirty and it is marked accessed to show it is a young page.

1331 static inline int handle_pte_fault(struct mm_struct *mm,
1332 struct vm_area_struct *vma, unsigned long address,
1333 int write_access, pte_t *pte)
1334 {
1335     pte_t entry;
1336
1337     entry = *pte;
1338     if (!pte_present(entry)) {
1339         /*
1340          * If it truly wasn’t present, we know that kswapd
1341          * and the PTE updates will not touch it later. So
1342          * drop the lock.
1343          */
1344         if (pte_none(entry))
1345             return do_no_page(mm, vma, address,
1346                                 write_access, pte);
1347             return do_swap_page(mm, vma, address, pte, entry,
1348                                 write_access);
1349     }
1350
1351     if (write_access) {
1352         if (!pte_write(entry))
1353             return do_wp_page(mm, vma, address, pte, entry);
1354             entry = pte_mkdirty(entry);
1355     }
1356     entry = pte_mkyoung(entry);
1357     establish_pte(vma, address, pte, entry);
1358     spin_unlock(&mm->page_table_lock);
1359     return 1;
1360 }

1331 The parameters of the function are the same as those for handle_mm_fault() except the PTE for the fault is included
1337 Record the PTE
1338 Handle the case where the PTE is not present
1344 If the PTE has never been filled, handle the allocation of the PTE with do_no_page() (See Section D.5.4.1)

1346 If the page has been swapped out to backing storage, handle it with do_swap_page() (See Section D.5.5.1)

1349-1354 Handle the case where the page is been written to

1350-1351 If the PTE is marked write-only, it is a COW page so handle it with do_wp_page() (See Section D.5.6.1)

1353 Otherwise just simply mark the page as dirty

1355 Mark the page as accessed

1356 establish_pte() copies the PTE and then updates the TLB and MMU cache. This does not copy in a new PTE but some architectures require the TLB and MMU update

1357 Unlock the mm_struct and return that a minor fault occurred

D.5.4 Demand Allocation

D.5.4.1 Function: do_no_page() (mm/memory.c)

The call graph for this function is shown in Figure 4.15. This function is called the first time a page is referenced so that it may be allocated and filled with data if necessary. If it is an anonymous page, determined by the lack of a vm_ops available to the VMA or the lack of a nopage() function, then do_anonymous_page() is called. Otherwise the supplied nopage() function is called to allocate a page and it is inserted into the page tables here. The function has the following tasks;

- Check if do_anonymous_page() should be used and if so, call it and return the page it allocates. If not, call the supplied nopage() function and ensure it allocates a page successfully.

- Break COW early if appropriate

- Add the page to the page table entries and call the appropriate architecture dependent hooks

1245 static int do_no_page(struct mm_struct * mm, struct vm_area_struct * vma, unsigned long address, int write_access, pte_t *page_table)

1246 {
1247     struct page * new_page;
1248     pte_t entry;
1249
1250     if (!vma->vm_ops || !vma->vm_ops->nopage)
D.5.4 Demand Allocation (do_no_page())

1252 return do_anonymous_page(mm, vma, page_table, write_access, address);
1253 spin_unlock(&mm->page_table_lock);
1254
1255 new_page = vma->vm_ops->nopage(vma, address & PAGE_MASK, 0);
1256 if (new_page == NULL) /* no page was available -- SIGBUS */
1257 return 0;
1258 if (new_page == NOPAGE_OOM)
1259 return -1;

1245 The parameters supplied are the same as those for handle_pTE_fault()

1251-1252 If no vm_ops is supplied or no nopage() function is supplied, then call do_anonymous_page() (See Section D.5.4.2) to allocate a page and return it
1253 Otherwise free the page table lock as the nopage() function can not be called with spinlocks held
1255 Call the supplied nopage function, in the case of filesystems, this is frequently filemap_nopage() (See Section D.6.4.1) but will be different for each device driver
1257-1258 If NULL is returned, it means some error occurred in the nopage function such as an IO error while reading from disk. In this case, 0 is returned which results in a SIGBUS being sent to the faulting process
1259-1260 If NOPAGE_OOM is returned, the physical page allocator failed to allocate a page and -1 is returned which will forcibly kill the process

1265 if (write_access && !(vma->vm_flags & VM_SHARED)) {
1266 struct page * page = alloc_page(GFP_HIGHUSER);
1267 if (!page) {
1268 page_cache_release(new_page);
1269 return -1;
1270 }
1271 copy_user_highpage(page, new_page, address);
1272 page_cache_release(new_page);
1273 lru_cache_add(page);
1274 new_page = page;
1275 }

Break COW early in this block if appropriate. COW is broken if the fault is a write fault and the region is not shared with VM_SHARED. If COW was not broken in this case, a second fault would occur immediately upon return.

1265 Check if COW should be broken early
If so, allocate a new page for the process

If the page could not be allocated, reduce the reference count to the page returned by the `nopage()` function and return -1 for out of memory

Otherwise copy the contents

Reduce the reference count to the returned page which may still be in use by another process

Add the new page to the LRU lists so it may be reclaimed by kswapd later

Lock the page tables again as the allocations have finished and the page tables are about to be updated

Check if there is still no PTE in the entry we are about to use. If two faults hit here at the same time, it is possible another processor has already completed the page fault and this one should be backed out

If there is no PTE entered, complete the fault
Increase the RSS count as the process is now using another page. A check really should be made here to make sure it isn’t the global zero page as the RSS count could be misleading.

As the page is about to be mapped to the process space, it is possible for some architectures that writes to the page in kernel space will not be visible to the process. `flush_page_to_ram()` ensures the CPU cache will be coherent.

`flush_icache_page()` is similar in principle except it ensures the icache and dcache’s are coherent.

Create a `pte_t` with the appropriate permissions.

If this is a write, then make sure the PTE has write permissions.

Place the new PTE in the process page tables.

If the PTE is already filled, the page acquired from the `nopage()` function must be released.

Decrement the reference count to the page. If it drops to 0, it will be freed.

Release the `mm_struct` lock and return 1 to signal this is a minor page fault as no major work had to be done for this fault as it was all done by the winner of the race.

Update the MMU cache for architectures that require it.

Release the `mm_struct` lock and return 2 to signal this is a major page fault.

**Function: do_anonymous_page()** *(mm/memory.c)*

This function allocates a new page for a process accessing a page for the first time. If it is a read access, a system wide page containing only zeros is mapped into the process. If it is write, a zero filled page is allocated and placed within the page tables.

```c
static int do_anonymous_page(struct mm_struct * mm,          
    struct vm_area_struct * vma,                        
    pte_t *page_table, int write_access,                 
    unsigned long addr)
{
    pte_t entry;
    /* Read-only mapping of ZERO_PAGE. */
    entry = pte_wrprotect(mk_pte(ZERO_PAGE(addr),
        vma->vm_page_prot));
```

D.5.4.2 Function: do_anonymous_page() *(mm/memory.c)*

This function allocates a new page for a process accessing a page for the first time. If it is a read access, a system wide page containing only zeros is mapped into the process. If it is write, a zero filled page is allocated and placed within the page tables.
D.5.4 Demand Allocation (do_anonymous_page())

```c
1197    /* ..except if it's a write access */
1198    if (write_access) {
1199        struct page *page;
1200    
1201        /* Allocate our own private page. */
1202        spin_unlock(&mm->page_table_lock);
1203    
1204        page = alloc_page(GFP_HIGHUSER);
1205        if (!page)
1206            goto no_mem;
1207        clear_user_highpage(page, addr);
1208    
1209        spin_lock(&mm->page_table_lock);
1210        if (!pte_none(*page_table)) {
1211            page_cache_release(page);
1212            spin_unlock(&mm->page_table_lock);
1213            return 1;
1214        }
1215        mm->rss++;
1216        flush_page_to_ram(page);
1217        entry = pte_mkdirty(mk_pte(page, vma->vm_page_prot));
1218        lru_cache_add(page);
1219        mark_page_accessed(page);
1220    }
1221    
1222    set_pte(page_table, entry);
1223    
1224    /* No need to invalidate - it was non-present before */
1225    update_mmu_cache(vma, addr, entry);
1226    spin_unlock(&mm->page_table_lock);
1227    return 1;    /* Minor fault */
1228
1229    no_mem:
1230    return -1;
1231 }
```

The parameters are the same as those passed to handle_pte_fault() (See Section D.5.3.2)

For read accesses, simply map the system wide empty_zero_page which the ZERO_PAGE() macro returns with the given permissions. The page is write protected so that a write to the page will result in a page fault.

If this is a write fault, then allocate a new page and zero fill it
D.5.5 Demand Paging

D.5.5.1 Function: do_swap_page()  (mm/memory.c)

The call graph for this function is shown in Figure 4.16. This function handles the case where a page has been swapped out. A swapped out page may exist in the swap cache if it is shared between a number of processes or recently swapped in during readahead. This function is broken up into three parts

- Search for the page in swap cache
- If it does not exist, call swapin_readahead() to read in the page
- Insert the page into the process page tables
D.5.5 Demand Paging (do_swap_page())

static int do_swap_page(struct mm_struct * mm, 
    struct vm_area_struct * vma, unsigned long address, 
    pte_t * page_table, pte_t orig_pte, int write_access) 
{
    struct page *page;
    swp_entry_t entry = pte_to_swp_entry(orig_pte);
    pte_t pte;
    int ret = 1;
    spin_unlock(&mm->page_table_lock);
    page = lookup_swap_cache(entry);

    Function preamble, check for the page in the swap cache

    1117-1119 The parameters are the same as those supplied to handle_pte_fault() 
        (See Section D.5.3.2)

    1122 Get the swap entry information from the PTE

    1126 Free the mm_struct spinlock

    1127 Lookup the page in the swap cache

    1128    if (!page) {
    1129        swapin_readahead(entry);
    1130        page = read_swap_cache_async(entry);
    1131            if (!page) {
    1132                int retval;
    1133                spin_lock(&mm->page_table_lock);
    1134                retval = pte_same(*page_table, orig_pte) ? -1 : 1;
    1135                spin_unlock(&mm->page_table_lock);
    1136                return retval;
    1137            }
    1138        }
    1139    /* Had to read the page from swap area: Major fault */
    1140    ret = 2;
    1141 }

    If the page did not exist in the swap cache, then read it from backing storage 
    with swapin_readhead() which reads in the requested pages and a number of pages 
    after it. Once it completes, read_swap_cache_async() should be able to return the 
    page.

    1128-1145 This block is executed if the page was not in the swap cache
1129 \texttt{swapin\_readahead()} (See Section D.6.6.1) reads in the requested page and a number of pages after it. The number of pages read in is determined by the \texttt{page\_cluster} variable in \texttt{mm/swap.c} which is initialised to 2 on machines with less than 16MiB of memory and 3 otherwise. 2\texttt{page\_cluster} pages are read in after the requested page unless a bad or empty page entry is encountered.

1130 \texttt{read\_swap\_cache\_async()} (See Section K.3.1.1) will look up the requested page and read it from disk if necessary.

1131-1141 If the page does not exist, there was another fault which swapped in this page and removed it from the cache while spinlocks were dropped.

1137 Lock the \texttt{mm\_struct}.

1138 Compare the two PTEs. If they do not match, -1 is returned to signal an IO error, else 1 is returned to mark a minor page fault as a disk access was not required for this particular page.

1139-1140 Free the \texttt{mm\_struct} and return the status.

1144 The disk had to be accessed to mark that this is a major page fault.

1147 \texttt{mark\_page\_accessed(page);} \\
1148 \texttt{lock\_page(page);} \\
1150 /* \\
1152 * Back out if somebody else faulted in this pte while we \\
1153 * released the page table lock. \\
1154 */ \\
1155 \texttt{spin\_lock(&mm->page\_table\_lock);} \\
1156 \texttt{if (!pte\_same(*page\_table, orig\_pte)) { \\
1157 \texttt{spin\_unlock(&mm->page\_table\_lock);} \\
1158 \texttt{unlock\_page(page);} \\
1159 \texttt{page\_cache\_release(page);} \\
1160 \texttt{return 1;} \\
1161 }} \\
1162 */ \\
1163 \texttt{The page isn’t present yet, go ahead with the fault. */} \\
1164 \texttt{swap\_free(entry);} \\
1166 \texttt{if (vm\_swap\_full())} \\
1167 \texttt{remove\_exclusive\_swap\_page(page);} \\
1168 \texttt{mm->rss++;} \\
1170 \texttt{pte = mk\_pte(page, vma->vm\_page\_prot);} \\
1171 \texttt{if (write\_access && can\_share\_swap\_page(page))}
D.5.5 Demand Paging (do_swap_page())

```c
pte = pte_mkdirty(pte_mkwritable(pte));
unlock_page(page);
flush_page_to_ram(page);
flush_icache_page(vma, page);
set_pte(page_table, pte);

/* No need to invalidate - it was non-present before */
update_mmu_cache(vma, address, pte);
spin_unlock(&mm->page_table_lock);
return ret;
```

Place the page in the process page tables

1147 mark_page_accessed() (See Section J.2.3.1) will mark the page as active so it will be moved to the top of the active LRU list

1149 Lock the page which has the side effect of waiting for the IO swapping in the page to complete

1155-1161 If someone else faulted in the page before we could, the reference to the page is dropped, the lock freed and return that this was a minor fault

1165 The function swap_free() (See Section K.2.2.1) reduces the reference to a swap entry. If it drops to 0, it is actually freed

1166-1167 Page slots in swap space are reserved for the same page once they have been swapped out to avoid having to search for a free slot each time. If the swap space is full though, the reservation is broken and the slot freed up for another page

1169 The page is now going to be used so increment the mm_structs RSS count

1170 Make a PTE for this page

1171 If the page has been written to and is not shared between more than one process, mark it dirty so that it will be kept in sync with the backing storage and swap cache for other processes

1173 Unlock the page

1175 As the page is about to be mapped to the process space, it is possible for some architectures that write to the page in kernel space will not be visible to the process. flush_page_to_ram() ensures the cache will be coherent

1176 flush_icache_page() is similar in principle except it ensures the icache and dcache’s are coherent
D.5.5 Demand Paging (do_swap_page())

1177 Set the PTE in the process page tables

1180 Update the MMU cache if the architecture requires it

1181-1182 Unlock the mm_struct and return whether it was a minor or major page fault

D.5.5.2 Function: can_share_swap_page()  (mm/swapfile.c)

This function determines if the swap cache entry for this page may be used or not. It may be used if there is no other references to it. Most of the work is performed by exclusive_swap_page() but this function first makes a few basic checks to avoid having to acquire too many locks.

259 int can_share_swap_page(struct page *page)  
260 {  
261   int retval = 0;  
262   if (!PageLocked(page))  
263     BUG();  
264   switch (page_count(page)) {  
265     case 3:  
266       if (!page->buffers)  
267         break;  
268       /* Fallthrough */  
269     case 2:  
270       if (!PageSwapCache(page))  
271         break;  
272       retval = exclusive_swap_page(page);  
273       break;  
274     case 1:  
275       if (PageReserved(page))  
276         break;  
277       retval = 1;  
278   }  
279   return retval;  
280 }

263-264 This function is called from the fault path and the page must be locked

265 Switch based on the number of references

266-268 If the count is 3, but there is no buffers associated with it, there is more than one process using the page. Buffers may be associated for just one process if the page is backed by a swap file instead of a partition

270-273 If the count is only two, but it is not a member of the swap cache, then it has no slot which may be shared so return false. Otherwise perform a full check with exclusive_swap_page() (See Section D.5.5.3)
276-277 If the page is reserved, it is the global ZERO_PAGE so it cannot be shared otherwise this page is definitely the only one

D.5.5.3 Function: exclusive_swap_page() (mm/swapfile.c)

This function checks if the process is the only user of a locked swap page.

229 static int exclusive_swap_page(struct page *page)
230 {
231     int retval = 0;
232     struct swap_info_struct * p;
233     swp_entry_t entry;
234     entry.val = page->index;
235     p = swap_info_get(entry);
236     if (p) {
237         /* Is the only swap cache user the cache itself? */
238         if (p->swap_map[SWP_OFFSET(entry)] == 1) {
239             /* Recheck the page count with the pagecache
240               * lock held.. */
241             spin_lock(&pagecache_lock);
242             if (page_count(page) - !!page->buffers == 2)
243                 retval = 1;
244             spin_unlock(&pagecache_lock);
245         }
246         swap_info_put(p);
247     }
248     return retval;
249 }

231 By default, return false

235 The swp_entry_t for the page is stored in page->index as explained in Section 2.4

236 Get the swap_info_struct with swap_info_get()(See Section K.2.3.1)

237-247 If a slot exists, check if we are the exclusive user and return true if we are

239 Check if the slot is only being used by the cache itself. If it is, the page count needs to be checked again with the pagecache_lock held

242-243 !!page->buffers will evaluate to 1 if there is buffers are present so this
244 block effectively checks if the process is the only user of the page. If it is,
245 retval is set to 1 so that true will be returned

246 Drop the reference to the slot that was taken with swap_info_get() (See Section K.2.3.1)
D.5.6 Copy On Write (COW) Pages

D.5.6.1 Function: do_wp_page() (mm/memory.c)

The call graph for this function is shown in Figure 4.17. This function handles the case where a user tries to write to a private page shared among processes, such as what happens after fork(). Basically what happens is a page is allocated, the contents copied to the new page and the shared count decremented in the old page.

948 static int do_wp_page(struct mm_struct *mm,
                          struct vm_area_struct *vma,
                          unsigned long address, pte_t *page_table, pte_t pte)
949 {
950  struct page *old_page, *new_page;
951  
952  old_page = pte_page(pte);
953  
954  if (!VALID_PAGE(old_page))
955     goto bad_wp_page;
956
948-950 The parameters are the same as those supplied to handle_pte_fault()
953-955 Get a reference to the current page in the PTE and make sure it is valid

957     if (!TryLockPage(old_page)) {
958          int reuse = can_share_swap_page(old_page);
959          unlock_page(old_page);
960          if (reuse) {
961              flush_cache_page(vma, address);
962              establish_pte(vma, address, page_table,
                                 pte_mkyoung(pte_mkdirty(pte_mkwrite(pte_mkdirty(pte)))));
963              spin_unlock(&mm->page_table_lock);
964              return 1; /* Minor fault */
965          }
966     }
957 First try to lock the page. If 0 is returned, it means the page was previously unlocked
958 If we managed to lock it, call can_share_swap_page() (See Section D.5.5.2) to see are we the exclusive user of the swap slot for this page. If we are, it means that we are the last process to break COW and we can simply use this page rather than allocating a new one
960-965 If we are the only users of the swap slot, then it means we are the only user of this page and the last process to break COW so the PTE is simply re-established and we return a minor fault
We need to copy this page so first get a reference to the old page so it doesn’t disappear before we are finished with it.

Unlock the spinlock as we are about to call alloc_page() (See Section F.2.1) which may sleep.

Allocate a page and make sure one was returned.

No prizes what this function does. If the page being broken is the global zero page, clear_user_highpage() will be used to zero out the contents of the page, otherwise copy_user_highpage() copies the actual contents.

The page table lock was released for alloc_page() (See Section F.2.1) so reacquire it.

Make sure the PTE hasn’t changed in the meantime which could have happened if another fault occurred while the spinlock is released.

The RSS is only updated if PageReserved() is true which will only happen if the page being faulted is the global ZERO_PAGE which is not accounted.
for in the RSS. If this was a normal page, the process would be using the same number of physical frames after the fault as it was before but against the zero page, it’ll be using a new frame so \texttt{rss++}

\texttt{break_cow()} is responsible for calling the architecture hooks to ensure the CPU cache and TLBs are up to date and then establish the new page into the PTE. It first calls \texttt{flush_page_to_ram()} which must be called when a \texttt{struct page} is about to be placed in userspace. Next is \texttt{flush_cache_page()} which flushes the page from the CPU cache. Lastly is \texttt{establish_pte()} which establishes the new page into the PTE.

987 Add the page to the LRU lists

992 Release the spinlock

993-994 Drop the references to the pages

995 Return a minor fault

996-999 \texttt{bad_wp_page:}

998 \hspace{1em} \texttt{spin_unlock(&mm->page_table_lock);}
999 \hspace{1em} \texttt{printk("do_wp_page: bogus page at address %08lx (page 0x%lx)\n", address,(unsigned long)old_page);}

1000 \hspace{1em} \texttt{return -1;}

1002 \hspace{1em} \texttt{page_cache_release(old_page);}
1003 \hspace{1em} \texttt{return -1;}

1004 }

997-1000 This is a false COW break which will only happen with a buggy kernel. Print out an informational message and return

1001-1003 The page allocation failed so release the reference to the old page and return -1
D.6 Page-Related Disk IO

Contents

D.6 Page-Related Disk IO 341
D.6.1 Generic File Reading 341
  D.6.1.1 Function: generic_file_read() 341
  D.6.1.2 Function: do_generic_file_read() 344
  D.6.1.3 Function: generic_file_readahead() 351
D.6.2 Generic File mmap() 355
  D.6.2.1 Function: generic_file_mmap() 355
D.6.3 Generic File Truncation 356
  D.6.3.1 Function: vmtruncate() 356
  D.6.3.2 Function: vmtruncate_list() 358
  D.6.3.3 Function: zap_page_range() 359
  D.6.3.4 Function: zap_pmd_range() 361
  D.6.3.5 Function: zap_pte_range() 362
  D.6.3.6 Function: truncate_inode_pages() 364
  D.6.3.7 Function: truncate_list_pages() 365
  D.6.3.8 Function: truncate_complete_page() 367
  D.6.3.9 Function: do_flushpage() 368
  D.6.3.10 Function: truncate_partial_page() 368
D.6.4 Reading Pages for the Page Cache 369
  D.6.4.1 Function: filemap_nopage() 369
  D.6.4.2 Function: page_cache_read() 374
D.6.5 File Readahead for nopage() 375
  D.6.5.1 Function: nopage_sequential_readahead() 375
  D.6.5.2 Function: read_cluster_nonblocking() 377
D.6.6 Swap Related Read-Ahead 378
  D.6.6.1 Function: swapin_readahead() 378
  D.6.6.2 Function: valid_swaphandles() 379

D.6.1 Generic File Reading

This is more the domain of the IO manager than the VM but because it performs the operations via the page cache, we will cover it briefly. The operation of generic_file_write() is essentially the same although it is not covered by this book. However, if you understand how the read takes place, the write function will pose no problem to you.

D.6.1.1 Function: generic_file_read() (mm/filemap.c)

This is the generic file read function used by any filesystem that reads pages through the page cache. For normal IO, it is responsible for building a read_descriptor_t for use with do_generic_file_read() and file_read_actor(). For direct IO, this function is basically a wrapper around generic_file_direct_IO().

1695 ssize_t generic_file_read(struct file * filp,
       char * buf, size_t count,
D.6.1 Generic File Reading \textit{(generic\_file\_read())} 

```c
loff_t *ppos)
1696 {
1697     ssize_t retval;
1698     if ((ssize_t) count < 0)
1699         return -EINVAL;
1700     if (filp->f_flags & O_DIRECT)
1701         goto o_direct;
1702     retval = -EFAULT;
1703     if (access_ok(VERIFY_WRITE, buf, count)) {
1704         retval = 0;
1705         if (count) {
1706             read_descriptor_t desc;
1707             desc.written = 0;
1708             desc.count = count;
1709             desc.buf = buf;
1710             desc.error = 0;
1711             do_generic_file_read(filp, ppos, &desc, 
1712                             file_read_actor);
1713             retval = desc.written;
1714             if (!retval)
1715                 retval = desc.error;
1716         }
1717     }
1718     out:
1719     return retval;
```

This block is concern with normal file IO.

1702-1703 If this is direct IO, jump to the \texttt{o\_direct} label

1706 If the access permissions to write to a userspace page are ok, then proceed

1709 If \texttt{count} is 0, there is no IO to perform

1712-1715 Populate a \texttt{read\_descriptor\_t} structure which will be used by \texttt{file\_read\_actor()}(See Section L.3.2.3)

1716 Perform the file read

1718 Extract the number of bytes written from the read descriptor struct
1282-1683 If an error occurred, extract what the error was
1724 Return either the number of bytes read or the error that occurred

1725 o_direct:
1726 {
1728   loff_t pos = *ppos, size;
1729   struct address_space *mapping =
1730     filp->f_dentry->d_inode->i_mapping;
1731   struct inode *inode = mapping->host;
1732   retval = 0;
1733   if (!count)
1734     goto out; /* skip atime */
1735   down_read(&inode->i_alloc_sem);
1736   down(&inode->i_sem);
1737   size = inode->i_size;
1738   if (pos < size) {
1739     retval = generic_file_direct_IO(READ, filp, buf,
1740                                 count, pos);
1740   if (retval > 0)
1741     *ppos = pos + retval;
1742 }  
1743   UPDATE_ATIME(filp->f_dentry->d_inode);
1744   goto out;
1745 }  
1746 }

This block is concerned with direct IO. It is largely responsible for extracting
the parameters required for generic_file_direct_IO().

1729 Get the address_space used by this struct file

1733-1734 If no IO has been requested, jump to out to avoid updating the inodes
access time

1737 Get the size of the file

1299-1700 If the current position is before the end of the file, the read is safe so
call generic_file_direct_IO()

1740-1741 If the read was successful, update the current position in the file for
the reader

1743 Update the access time

1744 Goto out which just returns retval
D.6.1.2 Function: **do_generic_file_read()** *(mm/filemap.c)*

This is the core part of the generic file read operation. It is responsible for allocating a page if it doesn’t already exist in the page cache. If it does, it must make sure the page is up-to-date and finally, it is responsible for making sure that the appropriate readahead window is set.

```c
void do_generic_file_read(struct file * filp,
    loff_t *ppos,
    read_descriptor_t * desc,
    read_actor_t actor)
```

1349 /* Get the maximum readahead window size for this block device
1350 Calculate the page index which holds the current file position pointer
1351 Calculate the offset within the page that holds the current file position pointer
1352 */
1353 if (index > filp->f_raend ||
    index + filp->f_rawin < filp->f_raend) {
    reada_ok = 0;
    filp->f_raend = 0;
    filp->f_ralen = 0;
    filp->f_ramax = 0;
    filp->f_rawin = 0;
} else {
```
reada_ok = 1;
}
/* Adjust the current value of read-ahead max.
* If the read operation stay in the first half page, force no
* readahead. Otherwise try to increase read ahead max just
* enough to do the read request.
* Then, at least MIN_READAHEAD if read ahead is ok,
* and at most MAX_READAHEAD in all cases.
*/
if (!index && offset + desc->count <= (PAGE_CACHE_SIZE >> 1)) {
    filp->f_ramax = 0;
} else {
    unsigned long needed;
    needed = ((offset + desc->count) >> PAGE_CACHE_SHIFT) + 1;
    if (filp->f_ramax < needed)
        filp->f_ramax = needed;
    if (reada_ok && filp->f_ramax < vm_min_readahead)
        filp->f_ramax = vm_min_readahead;
    if (filp->f_ramax > max_readahead)
        filp->f_ramax = max_readahead;
}

As the comment suggests, the readahead window gets reset if the current file position is outside the current readahead window. It gets reset to 0 here and adjusted by generic_file_readahead() (See Section D.6.1.3) as necessary.

As the comment states, the readahead window gets adjusted slightly if we are in the second-half of the current page:

for (; ;) {
    struct page *page, **hash;
    unsigned long end_index, nr, ret;
    end_index = inode->i_size >> PAGE_CACHE_SHIFT;
    if (index > end_index)
        break;
    nr = PAGE_CACHE_SIZE;
    if (index == end_index) {
        nr = inode->i_size & ~PAGE_CACHE_MASK;
        if (nr <= offset)
This loop goes through each of the pages necessary to satisfy the read request.

Calculate where the end of the file is in pages.

If the current index is beyond the end, then break out as we are trying to read beyond the end of the file.

Calculate \(nr\) to be the number of bytes remaining to be read in the current page. The block takes into account that this might be the last page used by the file and where the current file position is within the page.

Search for the page in the page cache.

If the page is not in the page cache, goto noCachedPage where it will be allocated.

In this block, the page was found in the page cache.

Take a reference to the page in the page cache so it does not get freed prematurely.

If the page is not up-to-date, goto pageNotUpToDate to update the page with information on the disk.

Perform file readahead with \texttt{generic\_file\_readahead()} (See Section D.6.1.3)
page_ok:
/* If users can be writing to this page using arbitrary
 * virtual addresses, take care about potential aliasing
 * before reading the page on the kernel side.
 */
if (mapping->i_mmap_shared != NULL)
    flush_dcache_page(page);

/* Mark the page accessed if we read the
 * beginning or we just did an lseek.
 */
if (!offset || !filp->f_reada)
    mark_page_accessed(page);

/* Ok, we have the page, and it's up-to-date, so
 * now we can copy it to user space...
 * The actor routine returns how many bytes were actually used..
 * NOTE! This may not be the same as how much of a user buffer
 * we filled up (we may be padding etc), so we can only update
 * "pos" here (the actor routine has to update the user buffer
 * pointers and the remaining count).
 */
ret = actor(desc, page, offset, nr);
offset += ret;
index += offset >> PAGE_CACHE_SHIFT;
offset &= ~PAGE_CACHE_MASK;
page_cache_release(page);
if (ret == nr && desc->count)
    continue;
break;

In this block, the page is present in the page cache and ready to be read by the
file read actor function.

1440-1441 As other users could be writing this page, call flush_dcache_page()
to make sure the changes are visible

1447-1448 As the page has just been accessed, call mark_page_accessed() (See Section J.2.3.1) to move it to the active_list

1460 Call the actor function. In this case, the actor function is file_read_actor() (See Section L.3.2.3) which is responsible for copying the bytes from the page to userspace
D.6.1 Generic File Reading (*do_generic_file_read()*)

1461 Update the current offset within the file
1462 Move to the next page if necessary
1463 Update the offset within the page we are currently reading. Remember that we could have just crossed into the next page in the file
1465 Release our reference to this page
1466-1468 If there is still data to be read, loop again to read the next page. Otherwise break as the read operation is complete

1470 /*
1471 * Ok, the page was not immediately readable, so let’s try to read
1472 * ahead while we’re at it..
1472 */
1473 page_not_up_to_date:
1474 generic_file_readahead(reada_ok, filp, inode, page);
1475
1476 if (Page_Uptodate(page))
1477 goto page_ok;
1478
1479 /* Get exclusive access to the page ... */
1480 lock_page(page);
1481
1482 /* Did it get unhashed before we got the lock? */
1483 if (!page->mapping) {
1484 UnlockPage(page);
1485 page_cache_release(page);
1486 continue;
1487 }
1488
1489 /* Did somebody else fill it already? */
1490 if (Page_Uptodate(page)) {
1491 UnlockPage(page);
1492 goto page_ok;
1493 }

In this block, the page being read was not up-to-date with information on the disk. *generic_file_readahead()* is called to update the current page and readahead as IO is required anyway.

1474 Call *generic_file_readahead()* (See Section D.6.1.3) to sync the current page and readahead if necessary
1476-1477 If the page is now up-to-date, goto page_ok to start copying the bytes to userspace
D.6.1 Generic File Reading (do_generic_file_read())

1480 Otherwise something happened with readahead so lock the page for exclusive access

1483-1487 If the page was somehow removed from the page cache while spinlocks were not held, then release the reference to the page and start all over again. The second time around, the page will get allocated and inserted into the page cache all over again

1490-1493 If someone updated the page while we did not have a lock on the page then unlock it again and goto page_ok to copy the bytes to userspace

1495 readpage:
1496 /* ... and start the actual read. The read will */
1497 * unlock the page. */
1498 error = mapping->a_ops->readpage(filp, page);
1499
1500 if (!error) {
1501 if (Page_Uptodate(page))
1502 goto page_ok;
1503
1504 /* Again, try some read-ahead while waiting for */
1505 * the page to finish.. */
1506 generic_file_readahead(reada_ok, filp, inode, page);
1507 wait_on_page(page);
1508 if (Page_Uptodate(page))
1509 goto page_ok;
1510 error = -EIO;
1511 }
1512
1513 /* UHHHUH! A synchronous read error occurred. Report it */
1514 desc->error = error;
1515 page_cache_release(page);
1516 break;

At this block, readahead failed to we synchronously read the page with the address_space supplied readpage() function.

1497 Call the address_space filesystem-specific readpage() function. In many cases this will ultimately call the function block_read_full_page() declared in fs/buffer.c()

1499-1501 If no error occurred and the page is now up-to-date, goto page_ok to begin copying the bytes to userspace

1504 Otherwise, schedule some readahead to occur as we are forced to wait on IO anyway
D.6.1 Generic File Reading (do_generic_file_read())

1505-1507 Wait for IO on the requested page to complete. If it finished successfully, then goto page_ok

1508 Otherwise an error occurred so set -EIO to be returned to userspace

1512-1514 An IO error occurred so record it and release the reference to the current page. This error will be picked up from the read_descriptor_t struct by generic_file_read() (See Section D.6.1.1)

1516 no_cached_page:
1517 /*
1518 * Ok, it wasn’t cached, so we need to create a new
1519 * page..
1520 *
1521 * We get here with the page cache lock held.
1522 */
1523 if (!cached_page) {
1524 spin_unlock(&pagecache_lock);
1525 cached_page = page_cache_alloc(mapping);
1526 if (!cached_page) {
1527 desc->error = -ENOMEM;
1528 break;
1529 }
1530
1531 /*
1532 * Somebody may have added the page while we
1533 * dropped the page cache lock. Check for that.
1534 */
1535 spin_lock(&pagecache_lock);
1536 page = __find_page_nolock(mapping, index, *hash);
1537 if (page)
1538 goto found_page;
1539 }
1540
1541 /*
1542 * Ok, add the new page to the hash-queues...
1543 */
1544 page = cached_page;
1545 __add_to_page_cache(page, mapping, index, hash);
1546 spin_unlock(&pagecache_lock);
1547 lru_cache_add(page);
1548 cached_page = NULL;
1549 goto readpage;
1551 }
In this block, the page does not exist in the page cache so allocate one and add it.

1523-1539 If a cache page has not already been allocated then allocate one and make sure that someone else did not insert one into the page cache while we were sleeping.

1524 Release pagecache_lock as page_cache_alloc() may sleep

1525-1529 Allocate a page and set -ENOMEM to be returned if the allocation failed

1535-1536 Acquire pagecache_lock again and search the page cache to make sure another process has not inserted it while the lock was dropped.

1537 If another process added a suitable page to the cache already, jump to found_page as the one we just allocated is no longer necessary.

1544-1545 Otherwise, add the page we just allocated to the page cache.

1547 Add the page to the LRU lists.

1548 Set cached_page to NULL as it is now in use.

1550 Goto readpage to schedule the page to be read from disk.

1552 *ppos = ((loff_t) index << PAGE_CACHE_SHIFT) + offset;
1554 filp->f_reada = 1;
1555 if (cached_page)
1556    page_cache_release(cached_page);
1557    UPDATE_ATIME(inode);
1558 }

1553 Update our position within the file.

1555-1556 If a page was allocated for addition to the page cache and then found to be unneeded, release it here.

1557 Update the access time to the file.

D.6.1.3 Function: generic_file_readahead() (mm/filemap.c)

This function performs generic file read-ahead. Readahead is one of the few areas that is very heavily commented upon in the code. It is highly recommended that you read the comments in mm/filemap.c marked with “Read-ahead context”.
D.6.1 Generic File Reading (generic_file_readahead())

```c
static void generic_file_readahead(int reada_ok,
        struct file * filp, struct inode * inode,
        struct page * page)
{
    unsigned long end_index;
    unsigned long index = page->index;
    unsigned long max_ahead, ahead;
    unsigned long raend;
    int max_readahead = get_max_readahead(inode);
    end_index = inode->i_size >> PAGE_CACHE_SHIFT;
    raend = filp->f_raend;
    max_ahead = 0;

    /* The current page is locked.
    * If the current position is inside the previous read IO request,
    * do not try to reread previously read ahead pages.
    * Otherwise decide or not to read ahead some pages synchronously.
    * If we are not going to read ahead, set the read ahead context
    * for this page only.
    */
    if (PageLocked(page)) {
        if (!(filp->f_ralen ||
                index >= raend ||
                index + filp->f_rawin < raend)) {
            raend = index;
            if (raend < end_index)
                max_ahead = filp->f_ramax;
            filp->f_rawin = 0;
            filp->f_ralen = 1;
            if (!max_ahead) {
                filp->f_raend = index + filp->f_ralen;
                filp->f_rawin += filp->f_ralen;
            }
        }
    }
}
```
This block has encountered a page that is locked so it must decide whether to temporarily disable readahead.

1245 If the current page is locked for IO, then check if the current page is within the last readahead window. If it is, there is no point trying to readahead again. If it is not, or readahead has not been performed previously, update the readahead context.

1246 The first check is if readahead has been performed previously. The second is to see if the current locked page is after where the previous readahead finished. The third check is if the current locked page is within the current readahead window.

1247 Update the end of the readahead window.

1248-1249 If the end of the readahead window is not after the end of the file, set `max_ahead` to be the maximum amount of readahead that should be used with this `struct file(flip->f_ramax)`.

1250-1255 Set readahead to only occur with the current page, effectively disabling readahead.

1258 /*
1259 * The current page is not locked.
1260 * If we were reading ahead and,
1261 * if the current max read ahead size is not zero and,
1262 * if the current position is inside the last read-ahead IO
1263 * request, it is the moment to try to read ahead asynchronously.
1264 * We will later force unplug device in order to force
1265 * asynchronous read IO.
1266 */
1267 else if (reada_ok && filp->f_ramax && raend >= 1 &&
1268 index <= raend && index + filp->f_ralen >= raend) {
1269 /*
1270 * Add ONE page to max_ahead in order to try to have about the
1271 * same IO maxsize as synchronous read-ahead
1272 * (MAX_READAHEAD + 1)*PAGE_CACHE_SIZE.
1273 * Compute the position of the last page we have tried to read
1274 * in order to begin to read ahead just at the next page.
1275 */
1276 raend -= 1;
1277 if (raend < end_index)
1278 max_ahead = filp->f_ramax + 1;
1279 if (max_ahead) {
1280 filp->f_rawin = filp->f_ralen;
D.6.1 Generic File Reading (generic_file_readahead())

1280     filp->f_ralen = 0;
1281     reada_ok      = 2;
1282 }
1283 }

This is one of the rare cases where the in-code commentary makes the code as clear as it possibly could be. Basically, it is saying that if the current page is not locked for IO, then extend the readahead window slight and remember that readahead is currently going well.

1284 /*
1285  * Try to read ahead pages.
1286  * We hope that ll_rw_blk() plug/unplug, coalescence, requests
1287  * sort and the scheduler, will work enough for us to avoid too
1288  *
1289 */
1290 ahead = 0;
1291 while (ahead < max_ahead) {
1292     ahead ++;
1293     if ((raend + ahead) >= end_index)
1294         break;
1295     if (page_cache_read(filp, raend + ahead) < 0)
1296         break;
1297 }

This block performs the actual readahead by calling page_cache_read() for each of the pages in the readahead window. Note here how ahead is incremented for each page that is readahead.

1298 /*
1299  * If we tried to read ahead some pages,
1300  * If we tried to read ahead asynchronously,
1301  * Try to force unplug of the device in order to start an
1302  * asynchronous read IO request.
1303  * Update the read-ahead context.
1304  * Store the length of the current read-ahead window.
1305  * Double the current max read ahead size.
1306  * That heuristic avoid to do some large IO for files that are
1307  * not really accessed sequentially.
1308 */
1309 if (ahead) {
1310     filp->f_ralen += ahead;
1311     filp->f_rawin += filp->f_ralen;
1312     filp->f_raend = raend + ahead + 1;
D.6.2 Generic File mmap()

D.6.2.1 Function: generic_file_mmap() (mm/filemap.c)

This is the generic mmap() function used by many struct files as their struct file_operations. It is mainly responsible for ensuring the appropriate address_space functions exist and setting what VMA operations to use.

```c
int generic_file_mmap(struct file * file,  
                      struct vm_area_struct * vma)  
```
D.6.3 Generic File Truncation

This section covers the path where a file is being truncated. The actual system call truncate() is implemented by sys_truncate() in fs/open.c. By the time the top-level function in the VM is called (vmtruncate()), the dentry information for the file has been updated and the inode’s semaphore has been acquired.

D.6.3.1 Function: vmtruncate()  (mm/memory.c)

This is the top-level VM function responsible for truncating a file. When it completes, all page table entries mapping pages that have been truncated have been unmapped and reclaimed if possible.

1042 int vmtruncate(struct inode * inode, loff_t offset)
1043 {
1044     unsigned long pgoff;
1045     struct address_space *mapping = inode->i_mapping;
1046     unsigned long limit;
1047     if (inode->i_size < offset)
1048         goto do_expand;
1049     inode->i_size = offset;
1050     spin_lock(&mapping->i_shared_lock);
1051     spin_lock(&mapping->i_mmap_shared);
The parameters passed are the inode being truncated and the new offset marking the new end of the file. The old length of the file is stored in inode→i_size

Get the address_space responsible for the inode

If the new file size is larger than the old size, then goto do_expand where the ulimits for the process will be checked before the file is grown

Here, the file is being shrunk so update inode→i_size to match

Lock the spinlock protecting the two lists of VMAs using this inode
1052-1053 If no VMAs are mapping the inode, goto out_unlock where the pages used by the file will be reclaimed by truncate_inode_pages() (See Section D.6.3.6)

1055 Calculate pgoff as the offset within the file in pages where the truncation will begin

1056-1057 Truncate pages from all private mappings with vmtruncate_list() (See Section D.6.3.2)

1058-1059 Truncate pages from all shared mappings

1062 Unlock the spinlock protecting the VMA lists

1063 Call truncate_inode_pages() (See Section D.6.3.6) to reclaim the pages if they exist in the page cache for the file

1064 Goto out_truncate to call the filesystem specific truncate() function so the blocks used on disk will be freed

1066-1071 If the file is being expanded, make sure that the process limits for maximum file size are not being exceeded and the hosting filesystem is able to support the new filesystem

1072 If the limits are fine, then update the inodes size and fall through to call the filesystem-specific truncate function which will fill the expanded filesize with zeros

1075-1079 If the filesystem provides a truncate() function, then lock the kernel, call it and unlock the kernel again. Filesystems do not acquire the proper locks to prevent races between file truncation and file expansion due to writing or faulting so the big kernel lock is needed

1080 Return success

1082-1084 If the file size would grow to being too big, send the SIGXFSZ signal to the calling process and return -EFBIG

D.6.3.2 Function: vmtruncate_list() (mm/memory.c)
This function cycles through all VMAs in an address_spaces list and calls zap_page_range() for the range of addresses which map a file that is being truncated.

1006 static void vmtruncate_list(struct vm_area_struct *mpnt,
                               unsigned long pgoff)

1007 {
1008   do {
1009       struct mm_struct *mm = mpnt->vm_mm;
1010       unsigned long start = mpnt->vm_start;
/* mapping wholly truncated? */
if (mpnt->vm_pgoff >= pgoff) {
    zap_page_range(mm, start, len);
    continue;
}

/* mapping wholly unaffected? */
len = len >> PAGE_SHIFT;
diff = pgoff - mpnt->vm_pgoff;
if (diff >= len)
    continue;

/* Ok, partially affected.. */
start += diff << PAGE_SHIFT;
len = (len - diff) << PAGE_SHIFT;
zap_page_range(mm, start, len);
}

1008-1031 Loop through all VMAs in the list
1009 Get the mm_struct that hosts this VMA
1010-1012 Calculate the start, end and length of the VMA
1016-1019 If the whole VMA is being truncated, call the function zap_page_range() (See Section D.6.3.3) with the start and length of the full VMA
1022 Calculate the length of the VMA in pages
1023-1025 Check if the VMA maps any of the region being truncated. If the VMA in unaffected, continue to the next VMA
1028-1029 Else the VMA is being partially truncated so calculate where the start and length of the region to truncate is in pages
1030 Call zap_page_range() (See Section D.6.3.3) to unmap the affected region

D.6.3.3 Function: zap_page_range() (mm/memory.c)
This function is the top-level pagetable-walk function which unmaps userpages in the specified range from a mm_struct.
void zap_page_range(struct mm_struct *mm, unsigned long address, unsigned long size) {
    mmu_gather_t *tlb;
    pgd_t *dir;
    unsigned long start = address, end = address + size;
    int freed = 0;
    dir = pgd_offset(mm, address);
    /*
     * This is a long-lived spinlock. That's fine.
     * There's no contention, because the page table
     * lock only protects against kswapd anyway, and
     * even if kswapd happened to be looking at this
     * process we _want_ it to get stuck.
     */
    if (address >= end)
        BUG();
    spin_lock(&mm->page_table_lock);
    flush_cache_range(mm, address, end);
    tlb = tlb_gather_mmu(mm);
    do {
        freed += zap_pmd_range(tlb, dir, address, end - address);
        address = (address + PGDIR_SIZE) & PGDIR_MASK;
        dir++;
    } while (address && (address < end));
    /* this will flush any remaining tlb entries */
    tlb_finish_mmu(tlb, start, end);
    /*
     * Update rss for the mm_struct (not necessarily current->mm)
     * Notice that rss is an unsigned long.
     */
    if (mm->rss > freed)
        mm->rss -= freed;
    else
        mm->rss = 0;
    spin_unlock(&mm->page_table_lock);
}

Calculate the start and end address for zapping

Calculate the PGD (dir) that contains the starting address
D.6.3 Generic File Truncation (zap_page_range())

376-377 Make sure the start address is not after the end address

378 Acquire the spinlock protecting the page tables. This is a very long-held lock and would normally be considered a bad idea but the comment above the block explains why it is ok in this case

379 Flush the CPU cache for this range

380 tlb_gather_mmu() records the MM that is being altered. Later, tlb_remove_page() will be called to unmmap the PTE which stores the PTEs in a struct free_pte_ctx until the zapping is finished. This is to avoid having to constantly flush the TLB as PTEs are freed

382-386 For each PMD affected by the zapping, call zap_pmd_range() until the end address has been reached. Note that tlb is passed as well for tlb_remove_page() to use later

389 tlb_finish_mmu() frees all the PTEs that were unmapped by tlb_remove_page() and then flushes the TLBs. Doing the flushing this way avoids a storm of TLB flushing that would be otherwise required for each PTE unmapped

395-398 Update RSS count

399 Release the pagetable lock

D.6.3.4 Function: zap_pmd_range()  (mm/memory.c)
This function is unremarkable. It steps through the PMDs that are affected by the requested range and calls zap_pte_range() for each one.

331 static inline int zap_pmd_range(mmu_gather_t *tlb, pgd_t * dir, unsigned long address, unsigned long size)
{
  pmd_t * pmd;
  unsigned long end;
  int freed;
  
  if (pgd_none(*dir))
    return 0;
  if (pgd_bad(*dir)) {
    pgd_ERROR(*dir);
    pgd_clear(dir);
    return 0;
  }
  pmd = pmd_offset(dir, address);
  end = address + size;
  if (end > ((address + PGDIR_SIZE) & PGDIR_MASK))
D.6.3 Generic File Truncation (zap_pmd_range())

347 end = ((address + PGDIR_SIZE) & PGDIR_MASK);
348 freed = 0;
349 do {
350   freed += zap_pte_range(tlb, pmd, address, end - address);
351   address = (address + PMD_SIZE) & PMD_MASK;
352   pmd++;
353 } while (address < end);
354 return freed;

337-338 If no PGD exists, return
339-343 If the PGD is bad, flag the error and return
344 Get the starting pmd
345-347 Calculate the end address of the zapping. If it is beyond the end of this
    PGD, then set end to the end of the PGD
349-353 Step through all PMDs in this PGD. For each PMD, call zap_pte_range() (See Section D.6.3.5) to unmap the PTEs
354 Return how many pages were freed

D.6.3.5 Function: zap_pte_range() (mm/memory.c)
This function calls tlb_remove_page() for each PTE in the requested pmd within
the requested address range.

294 static inline int zap_pte_range(mmu_gather_t *tlb, pmd_t * pmd,
    unsigned long address,
    unsigned long size)
295 {
296     unsigned long offset;
297     pte_t * ptep;
298     int freed = 0;
299
300     if (pmd_none(*pmd))
301         return 0;
302     if (pmd_bad(*pmd)) {
303         pmd_ERROR(*pmd);
304         pmd_clear(pmd);
305         return 0;
306     }
307     ptep = pte_offset(pmd, address);
308     offset = address & ~PMD_MASK;
309     if (offset + size > PMD_SIZE)
size = PMD_SIZE - offset;
size &= PAGE_MASK;
for (offset=0; offset < size; ptep++, offset += PAGE_SIZE) {
    pte_t pte = *ptep;
    if (pte_none(pte))
        continue;
    if (pte_present(pte)) {
        struct page *page = pte_page(pte);
        if (VALID_PAGE(page) && !PageReserved(page))
            freed ++;
        /* This will eventually call __free_pte on the pte. */
        tlb_remove_page(tlb, ptep, address + offset);
    } else {
        free_swap_and_cache(pte_to_swp_entry(pte));
        pte_clear(ptep);
    }
}
return freed;

300-301 If the PMD does not exist, return
302-306 If the PMD is bad, flag the error and return
307 Get the starting PTE offset
308 Align hte offset to a PMD boundary
309 If the size of the region to unmap is past the PMD boundary, fix the size so
    that only this PMD will be affected
311 Align size to a page boundary
312-316 Step through all PTEs in the region
314-315 If no PTE exists, continue to the next one
316-322 If the PTE is present, then call tlb_remove_page() to unmap the page. If
    the page is reclaimable, increment the freed count
322-325 If the PTE is in use but the page is paged out or in the swap
    cache, then free the swap slot and page page with free_swap_and_cache()
    (See Section K.3.2.3). It is possible that a page is reclaimed if it was in the
    swap cache that is unaccounted for here but it is not of paramount importance
328 Return the number of pages that were freed
D.6.3.6 Function: truncate_inode_pages() (mm/filemap.c)

This is the top-level function responsible for truncating all pages from the page cache that occur after $lstart$ in a mapping.

```c
327 void truncate_inode_pages(struct address_space * mapping, llof_t lstart) {
328     unsigned long start = (lstart + PAGE_CACHE_SIZE - 1) >> PAGE_CACHE_SHIFT;
330     unsigned partial = lstart & (PAGE_CACHE_SIZE - 1);
331     int unlocked;
333     spin_lock(&pagecache_lock);
334     do {
335         unlocked = truncate_list_pages(&mapping->clean_pages, start, &partial);
336         unlocked |= truncate_list_pages(&mapping->dirty_pages, start, &partial);
337         unlocked |= truncate_list_pages(&mapping->locked_pages, start, &partial);
338     } while (unlocked);
339     /* Traversed all three lists without dropping the lock */
340     spin_unlock(&pagecache_lock);
341 }
```

329 Calculate where to start the truncation as an index in pages.
330 Calculate partial as an offset within the last page if it is being partially truncated.
333 Lock the page cache.
334 This will loop until none of the calls to truncate_list_pages() return that a page was found that should have been reclaimed.
335 Use truncate_list_pages() (See Section D.6.3.7) to truncate all pages in the clean_pages list.
336 Similarly, truncate pages in the dirty_pages list.
337 Similarly, truncate pages in the locked_pages list.
340 Unlock the page cache.
D.6.3.7 Function: \texttt{truncate_list_pages()}  \textit{(mm/filemap.c)}

This function searches the requested list (head) which is part of an address space. If pages are found after start, they will be truncated.

\begin{verbatim}
static int truncate_list_pages(struct list_head *head,  
                                unsigned long start,  
                                unsigned *partial)  
{
    struct list_head *curr;  
    struct page * page;  
    int unlocked = 0;

    restart:
    curr = head->prev;
    while (curr != head) {
        unsigned long offset;

        page = list_entry(curr, struct page, list);
        offset = page->index;

        /* Is one of the pages to truncate? */
        if ((offset >= start) ||  
            (*partial && (offset + 1) == start)) {
            int failed;

            page_cache_get(page);
            failed = TryLockPage(page);

            list_del(head);
            if (!failed)
                /* Restart after this page */
                list_add_tail(head, curr);
            else  
                /* Restart on this page */
                list_add(head, curr);

            spin_unlock(&pagecache_lock);
            unlocked = 1;

            if (!failed) {
                if (*partial && (offset + 1) == start) {
                    truncate_partial_page(page, *partial);
                    *partial = 0;
                } else
                    truncate_complete_page(page);  
        }
    }
}
\end{verbatim}
UnlockPage(page);
} else
    wait_on_page(page);
page_cache_release(page);
if (current->need_resched) {
    __set_current_state(TASK_RUNNING);
    schedule();
}
spin_lock(&pagecache_lock);
goto restart;
curr = curr->prev;
return unlocked;

266-267 Record the start of the list and loop until the full list has been scanned
270-271 Get the page for this entry and what offset within the file it represents
274 If the current page is after start or is a page that is to be partially truncated,
    then truncate this page, else move to the next one
277-278 Take a reference to the page and try to lock it
280 Remove the page from the list
281-283 If we locked the page, add it back to the list where it will be skipped over
    on the next iteration of the loop
284-286 Else add it back where it will be found again immediately. Later in the
    function, wait_on_page() is called until the page is unlocked
288 Release the pagecache lock
299 Set locked to 1 to indicate a page was found that had to be truncated. This
    will force truncate_inode_pages() to call this function again to make sure
    there are no pages left behind. This looks like an oversight and was intended
    to have the functions recalled only if a locked page was found but the way it
    is implemented means that it will called whether the page was locked or not
291-299 If we locked the page, then truncate it
If the page is to be partially truncated, call `truncate_partial_page()` (See Section D.6.3.10) with the offset within the page where the truncation begins (`partial`).

Else call `truncate_complete_page()` (See Section D.6.3.8) to truncate the whole page.

Unlock the page.

If the page locking failed, call `wait_on_page()` to wait until the page can be locked.

Release the reference to the page. If there are no more mappings for the page, it will be reclaimed.

Check if the process should call `schedule()` before continuing. This is to prevent a truncating process from hogging the CPU.

Reacquire the spinlock and restart the scanning for pages to reclaim.

The current page should not be reclaimed so move to the next page.

Return 1 if a page was found in the list that had to be truncated.

**D.6.3.8 Function: `truncate_complete_page()`**

```c
239 static void truncate_complete_page(struct page *page) {
240     /* Leave it on the LRU if it gets converted into
241          * anonymous buffers */
242     if (!page->buffers || do_flushpage(page, 0))
243         lru_cache_del(page);
244
245     /*
246      * We remove the page from the page cache _after_ we have
247      * destroyed all buffer-cache references to it. Otherwise some
248      * other process might think this inode page is not in the
249      * page cache and creates a buffer-cache alias to it causing
250      * all sorts of fun problems ...
251      */
252     ClearPageDirty(page);
253     ClearPageUptodate(page);
254     remove_inode_page(page);
255     page_cache_release(page);
256 }
```

If the page has buffers, call `do_flushpage()` (See Section D.6.3.9) to flush all buffers associated with the page. The comments in the following lines describe the problem concisely.
D.6.3 Generic File Truncation (truncate_complete_page())

243 Delete the page from the LRU

252-253 Clear the dirty and up-to-date flags for the page

254 Call remove_inode_page() (See Section J.1.2.1) to delete the page from the page cache

255 Drop the reference to the page. The page will be later reclaimed when truncate_list_pages() drops its own private reference to it

D.6.3.9 Function: do_flushpage() (mm/filemap.c)

This function is responsible for flushing all buffers associated with a page.

223 static int do_flushpage(struct page *page, unsigned long offset)
224 {
225     int (*flushpage) (struct page *, unsigned long);
226     flushpage = page->mapping->a_ops->flushpage;
227     if (flushpage)
228         return (*flushpage)(page, offset);
229     return block_flushpage(page, offset);
230 }

226-228 If the page->mapping provides a flushpage() function, call it

229 Else call block_flushpage() which is the generic function for flushing buffers associated with a page

D.6.3.10 Function: truncate_partial_page() (mm/filemap.c)

This function partially truncates a page by zeroing out the higher bytes no longer in use and flushing any associated buffers.

232 static inline void truncate_partial_page(struct page *page,
233                                           unsigned partial) {
234     memclear_highpage_flush(page, partial, PAGE_CACHE_SIZE-partial);
235     if (page->buffers)
236         do_flushpage(page, partial);
237 }

234 memclear_highpage_flush() fills an address range with zeros. In this case, it will zero from partial to the end of the page

235-236 If the page has any associated buffers, flush any buffers containing data in the truncated region
D.6.4 Reading Pages for the Page Cache

D.6.4.1 Function: filemap_nopage()  (mm/filemap.c)

This is the generic nopage() function used by many VMAs. This loops around itself with a large number of goto's which can be difficult to trace but there is nothing novel here. It is principally responsible for fetching the faulting page from either the page cache or reading it from disk. If appropriate it will also perform file read-ahead.

1994 struct page * filemap_nopage(struct vm_area_struct * area,  
       unsigned long address,  
       int unused)
1995 {
1996     int error;
1997     struct file *file = area->vm_file;
1998     struct address_space *mapping =  
           file->f_dentry->d_inode->i_mapping;
1999     struct inode *inode = mapping->host;
2000     struct page *page, **hash;
2001     unsigned long size, pgoff, endoff;
2002     pgoff = ((address - area->vm_start) >> PAGE_CACHE_SHIFT) +  
           area->vm_pgoff;
2003     endoff = ((area->vm_end - area->vm_start) >> PAGE_CACHE_SHIFT) +  
            area->vm_pgoff;
2004     retry_all:
2005     */

This block acquires the struct file, address_space and inode important for this page fault. It then acquires the starting offset within the file needed for this fault and the offset that corresponds to the end of this VMA. The offset is the end of the VMA instead of the end of the page in case file read-ahead is performed.

1997-1999 Acquire the struct file, address_space and inode required for this fault

2003 Calculate pgoff which is the offset within the file corresponding to the beginning of the fault

2004 Calculate the offset within the file corresponding to the end of the VMA

2006 retry_all:
2007     /*
2008     * An external ptracer can access pages that normally aren’t
2009     * accessible..
2010     */
2011     size = (inode->i_size + PAGE_CACHE_SIZE - 1) >> PAGE_CACHE_SHIFT;
2012     if ((pgoff >= size) && (area->vm_mm == current->mm))
return NULL;

/* The "size" of the file, as far as mmap is concerned, isn't bigger than the mapping */
if (size > endoff)
    size = endoff;

/* Do we have something in the page cache already? */
hash = page_hash(mapping, pgoff);
retry_find:
page = __find_get_page(mapping, pgoff, hash);
if (!page)
goto no_cached_page;

/* Ok, found a page in the page cache, now we need to check that it's up-to-date. */
if (!Page_Uptodate(page))
goto page_not_uptodate;

Calculate the size of the file in pages

If the faulting pgoff is beyond the end of the file and this is not a tracing process, return NULL

If the VMA maps beyond the end of the file, then set the size of the file to be the end of the mapping

Search for the page in the page cache

If it does not exist, goto no_cached_page where page_cache_read() will be called to read the page from backing storage

If the page is not up-to-date, goto page_not_uptodate where the page will either be declared invalid or else the data in the page updated

success:
/*
* Try read-ahead for sequential areas.
*/
if (VM_SequentialReadHint(area))
nopage_sequential_readahead(area, pgoff, size);
/*
* Found the page and have a reference on it, need to check sharing
* and possibly copy it over to another page..
* /
mark_page_accessed(page);
flush_page_to_ram(page);
return page;

If this mapping specified the VM_SEQ_READ hint, then the pages are the
current fault will be pre-faulted with nopage_sequential_readahead()

Mark the faulted-in page as accessed so it will be moved to the active_list

As the page is about to be installed into a process page table, call
flush_page_to_ram() so that recent stores by the kernel to the page will
definitely be visible to userspace

Return the faulted-in page

no_cached_page:
/*/ 
* If the requested offset is within our file, try to read
* a whole cluster of pages at once.
* 
* Otherwise, we’re off the end of a privately mapped file,
* so we need to map a zero page.
*/
if ((pgoff < size) && !VM_RandomReadHint(area))
error = read_cluster_nonblocking(file, pgoff, size);
else
error = page_cache_read(file, pgoff);

/*/ 
* The page we want has now been added to the page cache.
* In the unlikely event that someone removed it in the
* meantime, we’ll just come back here and read it again.
*/
if (error >= 0)
goto retry_find;

/*/ 
* An error return from page_cache_read can result if the
* system is low on memory, or a problem occurs while trying
* to schedule I/O.
*/
if (error == -ENOMEM)
2077    return NOPAGE_OOM;
2078    return NULL;

2058-2059 If the end of the file has not been reached and the random-read hint
   has not been specified, call read_cluster_nonblocking() to pre-fault in just
   a few pages near the faulting page

2061 Else, the file is being accessed randomly, so just call page_cache_read()
   (See Section D.6.4.2) to read in just the faulting page

2068-2069 If no error occurred, goto retry_find at line 1958 which will check to
   make sure the page is in the page cache before returning

2076-2077 If the error was due to being out of memory, return that so the fault
   handler can act accordingly

2078 Else return NULL to indicate that a non-existant page was faulted resulting
   in a SIGBUS signal being sent to the faulting process

2080 page_not_uptodate:
2081    lock_page(page);
2082
2083    /* Did it get unhashed while we waited for it? */
2084    if (!page->mapping) {
2085        UnlockPage(page);
2086        page_cache_release(page);
2087        goto retry_all;
2088    }
2089
2090    /* Did somebody else get it up-to-date? */
2091    if (Page_Uptodate(page)) {
2092        UnlockPage(page);
2093        goto success;
2094    }
2095
2096    if (!mapping->a_ops->readpage(file, page)) {
2097        wait_on_page(page);
2098        if (Page_Uptodate(page))
2099            goto success;
2100    }

In this block, the page was found but it was not up-to-date so the reasons for the
page not being up to date are checked. If it looks ok, the appropriate readpage() function is called to resync the page.

2081 Lock the page for IO
If the page was removed from the mapping (possible because of a file truncation) and is now anonymous, then goto retry_all which will try and fault in the page again.

Check again if the Uptodate flag in case the page was updated just before we locked the page for IO.

Call the address_space→readpage() function to schedule the data to be read from disk.

Wait for the IO to complete and if it is now up-to-date, goto success to return the page. If the readpage() function failed, fall through to the error recovery path.

/* Umm, take care of errors if the page isn’t up-to-date.
 * Try to re-read it _once_. We do this synchronously,
 * because there really aren’t any performance issues here
 * and we need to check for errors.
 */
lock_page(page);

/* Somebody truncated the page on us? */
if (!page->mapping) {
    UnlockPage(page);
    page_cache_release(page);
    goto retry_all;
}

/* Somebody else successfully read it in? */
if (Page_Uptodate(page)) {
    UnlockPage(page);
    goto success;
}
ClearPageError(page);
if (!mapping->a_ops->readpage(file, page)) {
    wait_on_page(page);
    if (Page_Uptodate(page))
        goto Uptodate;
}

/* Things didn’t work out. Return zero to tell the
 * mm layer so, possibly freeing the page cache page first.
 */
In this path, the page is not up-to-date due to some IO error. A second attempt is made to read the page data and if it fails, return.

This is almost identical to the previous block. The only difference is that ClearPageError() is called to clear the error caused by the previous IO failure.

If it still failed, release the reference to the page because it is useless.

Return NULL because the fault failed.

**Function:** page_cache_read()  *(mm/filemap.c)*

This function adds the page corresponding to the offset within the file to the page cache if it does not exist there already.
return 0;
}

Acquire the address_space mapping managing the file

The page cache is a hash table and page_hash() returns the first page in the
bucket for this mapping and offset

Search the page cache with __find_page_nolock() (See Section J.1.4.3).
This basically will traverse the list starting at hash to see if the requested page
can be found

If the page is already in the page cache, return

Allocate a new page for insertion into the page cache. page_cache_alloc() will allocate a page from the buddy allocator using GFP mask information contained in mapping

Insert the page into the page cache with add_to_page_cache_unique() (See Section J.1.1.2). This function is used because a second check needs to be made to make sure the page was not inserted into the page cache while the pagecache_lock spinlock was not acquired

If the allocated page was inserted into the page cache, it needs to be populated with data so the readpage() function for the mapping is called. This schedules the IO to take place and the page will be unlocked when the IO completes

The path in add_to_page_cache_unique() (See Section J.1.1.2) takes an extra reference to the page being added to the page cache which is dropped here. The page will not be freed

If another process added the page to the page cache, it is released here by page_cache_release() as there will be no users of the page

D.6.5 File Readahead for nopage()

D.6.5.1 Function: nopage_sequential_readahead() (mm/filemap.c)
This function is only called by filemap_nopage() when the VM_SEQ_READ flag has been specified in the VMA. When half of the current readahead-window has been faulted in, the next readahead window is scheduled for IO and pages from the previous window are freed.

static void nopage_sequential_readahead(
    struct vm_area_struct * vma,
    unsigned long pgoff, unsigned long filesize)
{
    unsigned long ra_window;
D.6.5 File Readahead for `nopage()` (`nopage_sequential_readahead()`)

```c
ra_window = get_max_readahead(vma->vm_file->f_dentry->d_inode);
ra_window = CLUSTER_OFFSET(ra_window + CLUSTER_PAGES - 1);

/* vm_raend is zero if we haven’t read ahead */
if (vma->vm_raend == 0)
  vma->vm_raend = vma->vm_pgoff + ra_window;

get_max_readahead() returns the maximum sized readahead window for the block device the specified inode resides on

CLUSTER_PAGES is the number of pages that are paged-in or paged-out in bulk. The macro CLUSTER_OFFSET() will align the readahead window to a cluster boundary

1180-1181 If read-ahead has not occurred yet, set the end of the read-ahead window (vm_reend)

/*
 * If we’ve just faulted the page half-way through our window, then schedule reads for the next window, and release the pages in the previous window.
 */
if ((pgoff + (ra_window >> 1)) == vma->vm_raend) {
  unsigned long start = vma->vm_pgoff + vma->vm_raend;
  unsigned long end = start + ra_window;
  if (end > ((vma->vm_end >> PAGE_SHIFT) + vma->vm_pgoff))
    end = (vma->vm_end >> PAGE_SHIFT) + vma->vm_pgoff;
  if (start > end)
    return;

  while ((start < end) && (start < filesize)) {
    if (read_cluster_nonblocking(vma->vm_file,
      start, filesize) < 0)
      break;
    start += CLUSTER_PAGES;
  }
  run_task_queue(&tq_disk);

  /* if we’re far enough past the beginning of this area, recycle pages that are in the previous window. */
  if (vma->vm_raend >
    (vma->vm_pgoff + ra_window + ra_window)) {
    unsigned long window = ra_window << PAGE_SHIFT;
```
D.6.5 File Readahead for `nopage()` (`nopage_sequential_readahead()`) 377

1974
1975    end = vma->vm_start + (vma->vm_raend << PAGE_SHIFT);
1976    end -= window + window;
1977    filemap_sync(vma, end - window, window, MS_INVALIDATE);
1978 }
1979
1980    vma->vm_raend += ra_window;
1981 }
1982
1983 return;
1984 }

1953 If the fault has occurred half-way through the read-ahead window then schedule the next readahead window to be read in from disk and free the pages for the first half of the current window as they are presumably not required any more.

1954-1955 Calculate the start and end of the next readahead window as we are about to schedule it for IO.

1957 If the end of the readahead window is after the end of the VMA, then set end to the end of the VMA.

1959-1960 If we are at the end of the mapping, just return as there is no more readahead to perform.

1962-1967 Schedule the next readahead window to be paged in by calling `read_cluster_nonblocking()` (See Section D.6.5.2).

1968 Call `run_task_queue()` to start the IO.

1972-1978 Recycle the pages in the previous read-ahead window with `filemap_sync()` as they are no longer required.

1980 Update where the end of the readahead window is.

D.6.5.2 Function: `read_cluster_nonblocking()` *(mm/filemap.c)*

737 static int read_cluster_nonblocking(struct file * file,
738     unsigned long offset,
739     unsigned long filesize)
740 {
741     unsigned long pages = CLUSTER_PAGES;
742     offset = CLUSTER_OFFSET(offset);
743     while ((pages-- > 0) && (offset < filesize)) {
744         int error = page_cache_read(file, offset);
D.6.6.6 Swap Related Read-Ahead

D.6.6.1 Function: `swapin_readahead()` *(mm/memory.c)*

This function will fault in a number of pages after the current entry. It will stop with either `CLUSTER_PAGES` have been swapped in or an unused swap entry is found.

```c
void swapin_readahead(swp_entry_t entry) {
    int i, num;
    struct page *new_page;
    unsigned long offset;

    /* Get the number of handles we should do readahead io to.
     */
    num = valid_swaphandles(entry, &offset);
    for (i = 0; i < num; offset++, i++) {
        /* Ok, do the async read-ahead now */
        new_page = read_swap_cache_async(SWP_ENTRY(SWP_TYPE(entry),
                                              offset));

        if (!new_page)
            break;
        page_cache_release(new_page);
    }
    return;
}
```
valid_swaphandles() is what determines how many pages should be swapped in. It will stop at the first empty entry or when CLUSTER_PAGES is reached.

Swap in the pages

Attempt to swap the page into the swap cache with read_swap_cache_async() (See Section K.3.1.1)

If the page could not be paged in, break and return

Drop the reference to the page that read_swap_cache_async() takes

Return

D.6.6.2 Function: valid_swaphandles() (mm/swapfile.c)

This function determines how many pages should be readahead from swap starting from offset. It will readahead to the next unused swap slot but at most, it will return CLUSTER_PAGES.

```c
int valid_swaphandles(swp_entry_t entry, unsigned long *offset) {
    int ret = 0, i = 1 << page_cluster;
    unsigned long toff;
    struct swap_info_struct *swapdev = SWP_TYPE(entry) + swap_info;
    if (!page_cluster) /* no readahead */
        return 0;
    toff = (SWP_OFFSET(entry) >> page_cluster) << page_cluster;
    if (!toff) /* first page is swap header */
        toff++, i--;
    *offset = toff;
    swap_device_lock(swapdev);
    do {
        /* Don’t read-ahead past the end of the swap area */
        if (toff >= swapdev->max)
            break;
        /* Don’t read in free or bad pages */
        if (!swapdev->swap_map[toff])
            break;
        if (swapdev->swap_map[toff] == SWAP_MAP_BAD)
            break;
        toff++;
        ret++;
    } while (--i);
```
```
1240   i is set to CLUSTER_PAGES which is the equivalent of the bitshift shown here
1242   Get the swap_info_struct that contains this entry
1244-1245 If readahead has been disabled, return
1246   Calculate toff to be entry rounded down to the nearest CLUSTER_PAGES-sized boundary
1247-1248 If toff is 0, move it to 1 as the first page contains information about
   the swap area
1251   Lock the swap device as we are about to scan it
1252-1263 Loop at most i, which is initialised to CLUSTER_PAGES, times
1254-1255 If the end of the swap area is reached, then that is as far as can be
   readahead
1257-1258 If an unused entry is reached, just return as it is as far as we want to
   readahead
1259-1260 Likewise, return if a bad entry is discovered
1261   Move to the next slot
1262   Increment the number of pages to be readahead
1264   Unlock the swap device
1265   Return the number of pages which should be readahead
```
Appendix E

Boot Memory Allocator

Contents

E.1 Initialising the Boot Memory Allocator ............................... 382
  E.1.1 Function: init_bootmem() .................................. 382
  E.1.2 Function: init_bootmem_node() ................................. 382
  E.1.3 Function: init_bootmem_core() ................................ 383

E.2 Allocating Memory ..................................................... 385
  E.2.1 Reserving Large Regions of Memory ............................ 385
    E.2.1.1 Function: reserve_bootmem() ............................ 385
    E.2.1.2 Function: reserve_bootmem_node() ...................... 385
    E.2.1.3 Function: reserve_bootmem_core() ...................... 386
  E.2.2 Allocating Memory at Boot Time ............................... 387
    E.2.2.1 Function: alloc_bootmem() ................................ 387
    E.2.2.2 Function: __alloc_bootmem() ............................. 387
    E.2.2.3 Function: alloc_bootmem_node() .......................... 388
    E.2.2.4 Function: __alloc_bootmem_node() ...................... 389
    E.2.2.5 Function: __alloc_bootmem_core() ...................... 389

E.3 Freeing Memory ........................................................ 395
  E.3.1 Function: free_bootmem() .................................... 395
  E.3.2 Function: free_bootmem_core() ................................. 395

E.4 Retiring the Boot Memory Allocator ................................. 397
  E.4.1 Function: mem_init() ......................................... 397
  E.4.2 Function: free_pages_init() ................................ 399
  E.4.3 Function: one_highpage_init() ................................ 400
  E.4.4 Function: free_all_bootmem() ................................ 401
  E.4.5 Function: free_all_bootmem_core() ......................... 401
E.1 Initialising the Boot Memory Allocator

Contents

E.1 Initialising the Boot Memory Allocator 382
E.1.1 Function: init_bootmem() 382
E.1.2 Function: init_bootmem_node() 382
E.1.3 Function: init_bootmem_core() 383

The functions in this section are responsible for bootstrapping the boot memory allocator. It starts with the architecture specific function setup_memory() (See Section B.1.1) but all architectures cover the same basic tasks in the architecture specific function before calling the architecture independent function init_bootmem().

E.1.1 Function: init_bootmem() (mm/bootmem.c)

This is called by UMA architectures to initialise their boot memory allocator structures.

304 unsigned long __init init_bootmem (unsigned long start,
    unsigned long pages)

305 {
    max_low_pfn = pages;
    min_low_pfn = start;
    return(init_bootmem_core(&contig_page_data, start, 0, pages));
}

304 Confusingly, the pages parameter is actually the end PFN of the memory addressable by this node, not the number of pages as the name implies.

306 Set the max PFN addressable by this node in case the architecture dependent code did not.

307 Set the min PFN addressable by this node in case the architecture dependent code did not.

308 Call init_bootmem_core() (See Section E.1.3) which does the real work of initialising the bootmem_data.

E.1.2 Function: init_bootmem_node() (mm/bootmem.c)

This is called by NUMA architectures to initialise boot memory allocator data for a given node.

284 unsigned long __init init_bootmem_node (pg_data_t *pgdat,
    unsigned long freepfn,
    unsigned long startpfn,
    unsigned long endpfn)

285 {
    return(init_bootmem_core(pgdat, freepfn, startpfn, endpfn));
}
Initiating the Boot Memory Allocator (**init_bootmem_node()**)

Just call **init_bootmem_core()**(See Section E.1.3) directly.

**E.1.3 Function: init_bootmem_core()** (*mm/bootmem.c*)

Initialises the appropriate struct bootmem_data_t and inserts the node into the linked list of nodes pgdat_list.

```c
46 static unsigned long __init init_bootmem_core (pg_data_t *pgdat,
47     unsigned long mapstart, unsigned long start, unsigned long end)
48 {
49     bootmem_data_t *bdata = pgdat->bdata;
50     unsigned long mapsize = ((end - start)+7)/8;
51     pgdat->node_next = pgdat_list;
52     pgdat_list = pgdat;
53     mapsize = (mapsize + (sizeof(long) - 1UL)) & ~((sizeof(long) - 1UL);
54     bdata->node_bootmem_map = phys_to_virt(mapstart << PAGE_SHIFT);
55     bdata->node_boot_start = (start << PAGE_SHIFT);
56     return mapsize;
57 }
```

The parameters are:

- **pgdat** is the node descriptor been initialised
- **mapstart** is the beginning of the memory that will be usable
- **start** is the beginning PFN of the node
- **end** is the end PFN of the node

Each page requires one bit to represent it so the size of the map required is the number of pages in this node rounded up to the nearest multiple of 8 and then divided by 8 to give the number of bytes required.

As the node will be shortly considered initialised, insert it into the global pgdat_list

Round the mapsize up to the closest word boundary.
56 Convert the mapstart to a virtual address and store it in bdata—node_bootmem_map

57 Convert the starting PFN to a physical address and store it on node_boot_start

58 Store the end PFN of ZONE_NORMAL in node_low_pfn

64 Fill the full map with 1's marking all pages as allocated. It is up to the architecture dependent code to mark the usable pages
E.2 Allocating Memory

Contents

E.2 Allocating Memory 385
  E.2.1 Reserving Large Regions of Memory 385
    E.2.1.1 Function: reserve_bootmem() 385
    E.2.1.2 Function: reserve_bootmem_node() 385
    E.2.1.3 Function: reserve_bootmem_core() 386
  E.2.2 Allocating Memory at Boot Time 387
    E.2.2.1 Function: alloc_bootmem() 387
    E.2.2.2 Function: __alloc_bootmem() 387
    E.2.2.3 Function: alloc_bootmem_node() 388
    E.2.2.4 Function: __alloc_bootmem_node() 389
    E.2.2.5 Function: __alloc_bootmem_core() 389

E.2.1 Reserving Large Regions of Memory

E.2.1.1 Function: reserve_bootmem() *(mm/bootmem.c)*

311 void __init reserve_bootmem (unsigned long addr, unsigned long size) 
312 { 
313       reserve_bootmem_core(contig_page_data.bdata, addr, size); 
314 }

313 Just call reserve_bootmem_core()(See Section E.2.1.3). As this is for a non-
NUMA architecture, the node to allocate from is the static contig_page_data
node.

E.2.1.2 Function: reserve_bootmem_node() *(mm/bootmem.c)*

289 void __init reserve_bootmem_node (pg_data_t *pgdat, 
    unsigned long physaddr, 
    unsigned long size) 
290 { 
291       reserve_bootmem_core(pgdat->bdata, physaddr, size); 
292 }

291 Just call reserve_bootmem_core()(See Section E.2.1.3) passing it the boot-
mem data of the requested node.
E.2.1.3 Function: `reserve_bootmem_core()`  
(`mm/bootmem.c`)

```c
74 static void __init reserve_bootmem_core(bootmem_data_t *bdata,
    unsigned long addr,
    unsigned long size)
75 {
    unsigned long i;
    /*
     * round up, partially reserved pages are considered
     * fully reserved.
     */
    unsigned long sidx = (addr - bdata->node_boot_start)/PAGE_SIZE;
    unsigned long eidx = (addr + size - bdata->node_boot_start +
        PAGE_SIZE-1)/PAGE_SIZE;
    unsigned long end = (addr + size + PAGE_SIZE-1)/PAGE_SIZE;
    if (!size) BUG();
    if (sidx < 0) BUG();
    if (eidx < 0) BUG();
    if (sidx >= eidx) BUG();
    if ((addr >> PAGE_SHIFT) >= bdata->node_low_pfn)
        BUG();
    if (end > bdata->node_low_pfn)
        BUG();
    for (i = sidx; i < eidx; i++)
        if (test_and_set_bit(i, bdata->node_bootmem_map))
            printk("hm, page %08lx reserved twice.\n",
                i*PAGE_SIZE);
```

81 The `sidx` is the starting index to serve pages from. The value is obtained by
subtracting the starting address from the requested address and dividing by
the size of a page

82 A similar calculation is made for the ending index `eidx` except that the allocation
is rounded up to the nearest page. This means that requests to partially
reserve a page will result in the full page being reserved

84 `end` is the last PFN that is affected by this reservation

86 Check that a non-zero value has been given

88-89 Check the starting index is not before the start of the node
E.2.2 Allocating Memory at Boot Time

E.2.2.1 Function: alloc_bootmem()  (mm/bootmem.c)

The callgraph for these macros is shown in Figure 5.1.

38 #define alloc_bootmem(x) \ 39   __alloc_bootmem((x), SMP_CACHE_BYTES, __pa(MAX_DMA_ADDRESS)) \ 40 #define alloc_bootmem_low(x) \ 41   __alloc_bootmem((x), SMP_CACHE_BYTES, 0) \ 42 #define alloc_bootmem_pages(x) \ 43   __alloc_bootmem((x), PAGE_SIZE, __pa(MAX_DMA_ADDRESS)) \ 44 #define alloc_bootmem_low_pages(x) \ 45   __alloc_bootmem((x), PAGE_SIZE, 0)

39 alloc_bootmem() will align to the L1 hardware cache and start searching for a page after the maximum address usable for DMA

40 alloc_bootmem_low() will align to the L1 hardware cache and start searching from page 0

42 alloc_bootmem_pages() will align the allocation to a page size so that full pages will be allocated starting from the maximum address usable for DMA

44 alloc_bootmem_pages() will align the allocation to a page size so that full pages will be allocated starting from physical address 0

E.2.2.2 Function: __alloc_bootmem()  (mm/bootmem.c)

326 void * __init __alloc_bootmem (unsigned long size, unsigned long align, unsigned long goal)
327 {
328   pg_data_t *pgdat;
329   void *ptr;
for_each_pgdat(pgdat)
    if ((ptr = __alloc_bootmem_core(pgdat->bdata, size, align, goal))
        return(ptr);

    /*
     * Whoops, we cannot satisfy the allocation request.
     */
    printk(KERN_ALERT "bootmem alloc of %lu bytes failed!\n", size);
    panic("Out of memory");
    return NULL;

The parameters are;

size is the size of the requested allocation
align is the desired alignment and must be a power of 2. Currently either
SMP_CACHE_BYTES or PAGE_SIZE
goal is the starting address to begin searching from

Cycle through all available nodes and try allocating from each in turn.
In the UMA case, this will just allocate from the contig_page_data node

If the allocation fails, the system is not going to be able to boot so the
kernel panics

E.2.2.3 Function: alloc_bootmem_node()  (mm/bootmem.c)

#define alloc_bootmem_node(pgdat, x) __alloc_bootmem_node((pgdat), (x), SMP_CACHE_BYTES, __pa(MAX_DMA_ADDRESS))
#define alloc_bootmem_pages_node(pgdat, x) __alloc_bootmem_node((pgdat), (x), PAGE_SIZE, __pa(MAX_DMA_ADDRESS))
#define alloc_bootmem_low_pages_node(pgdat, x) __alloc_bootmem_node((pgdat), (x), PAGE_SIZE, 0)

alloc_bootmem_node() will allocate from the requested node and align
to the L1 hardware cache and start searching for a page beginning with
ZONE_NORMAL (i.e. at the end of ZONE_DMA which is at MAX_DMA_ADDRESS)

alloc_bootmem_pages() will allocate from the requested node and align
the allocation to a page size so that full pages will be allocated starting from
the ZONE_NORMAL
E.2.2 Allocating Memory at Boot Time \textit{(alloc\_bootmem\_node())}

57-58 \texttt{alloc\_bootmem\_pages()} will allocate from the requested node and align the allocation to a page size so that full pages will be allocated starting from physical address 0 so that \texttt{ZONE\_DMA} will be used.

\textbf{E.2.2.4 Function: \_\_alloc\_bootmem\_node()} \texttt{(mm/bootmem.c)}

344 \texttt{void * \_\_init \_\_alloc\_bootmem\_node (pg\_data\_t *pgdat,}
\hspace{1em}\texttt{unsigned long size,}
\hspace{1em}\texttt{unsigned long align,}
\hspace{1em}\texttt{unsigned long goal)}
\begin{verbatim}
345 { void *ptr;
346     ptr = \_\_alloc\_bootmem\_core(pgdat->bdata, size, align, goal);
347     if (ptr)
348         return (ptr);
349
350     /*
351     * Whoops, we cannot satisfy the allocation request.
352     */
353     printk(KERN_ALERT "bootmem alloc of %lu bytes failed!\n", size);
354     panic("Out of memory");
355     return NULL;
356 }
\end{verbatim}

344 The parameters are the same as for \_\_\_alloc\_bootmem\_node() (See Section E.2.2.4) except the node to allocate from is specified.

348 Call the core function \_\_\_alloc\_bootmem\_core() (See Section E.2.2.5) to perform the allocation.

349-350 Return a pointer if it was successful.

355-356 Otherwise print out a message and panic the kernel as the system will not boot if memory can not be allocated even now.

\textbf{E.2.2.5 Function: \_\_\_alloc\_bootmem\_core()} \texttt{(mm/bootmem.c)}

This is the core function for allocating memory from a specified node with the boot memory allocator. It is quite large and broken up into the following tasks:

- Function preamble. Make sure the parameters are sane.
- Calculate the starting address to scan from based on the \texttt{goal} parameter.
- Check to see if this allocation may be merged with the page used for the previous allocation to save memory.
• Mark the pages allocated as 1 in the bitmap and zero out the contents of the pages

144 static void * __init __alloc_bootmem_core (bootmem_data_t *bdata,
145 unsigned long size, unsigned long align, unsigned long goal)
146 {
147 unsigned long i, start = 0;
148 void *ret;
149 unsigned long offset, remaining_size;
150 unsigned long aresize, preferred, incr;
151 unsigned long eidx = bdata->node_low_pfn -
152 (bdata->node_boot_start >> PAGE_SHIFT);
153
154 if (!size) BUG();
155
156 if (align & (align-1))
157 BUG();
158
159 offset = 0;
160 if (align &&
161 (bdata->node_boot_start & (align - 1UL)) != 0)
162 offset = (align - (bdata->node_boot_start &
163 (align - 1UL)));
164 offset >>= PAGE_SHIFT;

Function preamble, make sure the parameters are sane

144 The parameters are;

bdata is the bootmem for the struct being allocated from
size is the size of the requested allocation
align is the desired alignment for the allocation. Must be a power of 2
goal is the preferred address to allocate above if possible

151 Calculate the ending bit index eidx which returns the highest page index that
may be used for the allocation

154 Call BUG() if a request size of 0 is specified

156-156 If the alignment is not a power of 2, call BUG()

159 The default offset for alignments is 0

160 If an alignment has been specified and...

161 And the requested alignment is the same alignment as the start of the node
then calculate the offset to use
The offset to use is the requested alignment masked against the lower bits of the starting address. In reality, this offset will likely be identical to align for the prevalent values of align.

If a goal has been specified and the goal is after the starting address for this node and the PFN of the goal is less than the last PFN addressable by this node then ...

The preferred offset to start from is the goal minus the beginning of the memory addressable by this node.

Else the preferred offset is 0.

Adjust the preferred address to take the offset into account so that the address will be correctly aligned.

The number of pages that will be affected by this allocation is stored in aresize.

incr is the number of pages that have to be skipped to satisfy alignment requirements if they are over one page.

restart_scan:

for (i = preferred; i < eidx; i += incr) {
    unsigned long j;
    if (test_bit(i, bdata->node_bootmem_map))
        continue;
    for (j = i + 1; j < i + aresize; ++j) {
        if (j >= eidx)
            goto fail_block;
        if (test_bit(j, bdata->node_bootmem_map))
            goto fail_block;
E.2.2 Allocating Memory at Boot Time (_alloc_bootmem_core())

Scan through memory looking for a block large enough to satisfy this request

180 If the allocation could not be satisfied starting from goal, this label is jumped to so that the map will be rescanned

181-194 Starting from preferred, scan linearly searching for a free block large enough to satisfy the request. Walk the address space in incr steps to satisfy alignments greater than one page. If the alignment is less than a page, incr will just be 1

183-184 Test the bit, if it is already 1, it is not free so move to the next page

185-190 Scan the next areysize number of pages and see if they are also free. It fails if the end of the addressable space is reached (eidx) or one of the pages is already in use

191-192 A free block is found so record the start and jump to the found block

195-198 The allocation failed so start again from the beginning

199 If that also failed, return NULL which will result in a kernel panic

found:
201 if (start >= eidx)
202 BUG();
203
209 if (align <= PAGE_SIZE
210 && bdata->last_offset && bdata->last_pos+1 == start) {
211 offset = (bdata->last_offset+align-1) & ~(align-1);
212 if (offset > PAGE_SIZE)
213 BUG();
214 remaining_size = PAGE_SIZE-offset;
215 if (size < remaining_size) {
216 areysize = 0;
217 // last_pos unchanged
218 bdata->last_offset = offset+size;
Test to see if this allocation may be merged with the previous allocation.

201-202 Check that the start of the allocation is not after the addressable memory.
This check was just made so it is redundant

209-230 Try and merge with the previous allocation if the alignment is less than a
PAGE_SIZE, the previously page has space in it (last_offset != 0) and that
the previously used page is adjacent to the page found for this allocation

231-234 Else record the pages and offset used for this allocation to be used for
merging with the next allocation

211 Update the offset to use to be aligned correctly for the requested align

212-213 If the offset now goes over the edge of a page, BUG() is called. This
condition would require a very poor choice of alignment to be used. As the
only alignment commonly used is a factor of PAGE_SIZE, it is impossible for
normal usage

214 remaining_size is the remaining free space in the previously used page

215-221 If there is enough space left in the old page then use the old page totally
and update the bootmem_data struct to reflect it

221-228 Else calculate how many pages in addition to this one will be required
and update the bootmem_data

216 The number of pages used by this allocation is now 0
E.2.2 Allocating Memory at Boot Time (\_\_alloc_bootmem\_core())

218 Update the last_offset to be the end of this allocation
219 Calculate the virtual address to return for the successful allocation
222 remaining_size is how space will be used in the last page used to satisfy the allocation
223 Calculate how many more pages are needed to satisfy the allocation
224 Record the address the allocation starts from
226 The last page used is the start page plus the number of additional pages required to satisfy this allocation areasize
227 The end of the allocation has already been calculated
229 If the offset is at the end of the page, make it 0
231 No merging took place so record the last page used to satisfy this allocation
232 Record how much of the last page was used
233 Record the starting virtual address of the allocation

238 for (i = start; i < start+areasize; i++)
239 \quad if (test_and_set_bit(i, bdata->node_bootmem_map))
240 \quad \quad BUG();
241 \quad memset(ret, 0, size);
242 \quad return ret;
243 }

Mark the pages allocated as 1 in the bitmap and zero out the contents of the pages

238-240 Cycle through all pages used for this allocation and set the bit to 1 in the bitmap. If any of them are already 1, then a double allocation took place so call BUG()
241 Zero fill the pages
242 Return the address of the allocation
E.3 Freeing Memory

Contents

E.3 Freeing Memory 395
E.3.1 Function: free_bootmem() 395
E.3.2 Function: free_bootmem_core() 395

E.3.1 Function: free_bootmem()  
(mm/bootmem.c)

Figure E.1: Call Graph: free_bootmem()

294 void __init free_bootmem_node (pg_data_t *pgdat,  
        unsigned long physaddr, unsigned long size)  
295 {  
296    return(free_bootmem_core(pgdat->bdata, physaddr, size));  
297 }  

316 void __init free_bootmem (unsigned long addr, unsigned long size)  
317 {  
318    return(free_bootmem_core(contig_page_data.bdata, addr, size));  
319 }  

296 Call the core function with the corresponding bootmem data for the requested node  
318 Call the core function with the bootmem data for contig_page_data

E.3.2 Function: free_bootmem_core()  
(mm/bootmem.c)

103 static void __init free_bootmem_core(bootmem_data_t *bdata,  
        unsigned long addr,  
        unsigned long size)  
104 {  
105    unsigned long i;  
106    unsigned long start;
E.3 Freeing Memory (*free_bootmem_core()*)

```c
unsigned long sidx;
unsigned long eidx = (addr + size -
                     bdata->node_boot_start)/PAGE_SIZE;
unsigned long end = (addr + size)/PAGE_SIZE;
if (!size) BUG();
if (end > bdata->node_low_pfn)
  BUG();

/*
 * Round up the beginning of the address.
 */
start = (addr + PAGE_SIZE-1) / PAGE_SIZE;
sidx = start - (bdata->node_boot_start/PAGE_SIZE);
for (i = sidx; i < eidx; i++) {
  if (!test_and_clear_bit(i, bdata->node_bootmem_map))
    BUG();
}
```

112 Calculate the end index affected as eidx

113 The end address is the end of the affected area rounded down to the nearest page if it is not already page aligned

115 If a size of 0 is freed, call BUG

116-117 If the end PFN is after the memory addressable by this node, call BUG

122 Round the starting address up to the nearest page if it is not already page aligned

123 Calculate the starting index to free

125-127 For all full pages that are freed by this action, clear the bit in the boot bitmap. If it is already 0, it is a double free or is memory that was never used so call BUG
E.4 Retiring the Boot Memory Allocator

Contents

E.4 Retiring the Boot Memory Allocator 397
E.4.1 Function: mem_init() 397
E.4.2 Function: free_pages_init() 399
E.4.3 Function: one_highpage_init() 400
E.4.4 Function: free_all_bootmem() 401
E.4.5 Function: free_all_bootmem_core() 401

Once the system is started, the boot memory allocator is no longer needed so these functions are responsible for removing unnecessary boot memory allocator structures and passing the remaining pages to the normal physical page allocator.

### E.4.1 Function: mem_init() (arch/i386/mm/init.c)

The call graph for this function is shown in Figure 5.2. The important part of this function for the boot memory allocator is that it calls free_pages_init()(See Section E.4.2). The function is broken up into the following tasks:

- Function preamble, set the PFN within the global mem_map for the location of high memory and zero out the system wide zero page
- Call free_pages_init()(See Section E.4.2)
- Print out an informational message on the availability of memory in the system
- Check the CPU supports PAE if the config option is enabled and test the WP bit on the CPU. This is important as without the WP bit, the function verify_write() has to be called for every write to userspace from the kernel. This only applies to old processors like the 386
- Fill in entries for the userspace portion of the PGD for swapper_pg_dir, the kernel page tables. The zero page is mapped for all entries

```c
507 void __init mem_init(void)
508 {
509   int codesize, reservedpages, datasize, initsize;
510
511   if (!mem_map)
512     BUG();
513
514   set_max_mapnr_init();
515
516   high_memory = (void *)__va(max_low_pfn * PAGE_SIZE);
517
518   /* clear the zero-page */
519   memset(empty_zero_page, 0, PAGE_SIZE);
```
This function records the PFN high memory starts in \texttt{mem_map} (\texttt{highmem_start_page}), the maximum number of pages in the system (\texttt{max_mapnr} and \texttt{num_physpages}) and finally the maximum number of pages that may be mapped by the kernel (\texttt{num_mappedpages}).

\texttt{high_memory} is the virtual address where high memory begins.

Zero out the system wide zero page.

\begin{verbatim}
reservedpages = free_pages_init();
\end{verbatim}

Call \texttt{free_pages_init()} (See Section E.4.2) which tells the boot memory allocator to retire itself as well as initialising all pages in high memory for use with the buddy allocator.

\begin{verbatim}
codesize = (unsigned long) \&_etext - (unsigned long) \&_text;
datasize = (unsigned long) \&_edata - (unsigned long) \&_etext;
initsize = (unsigned long) \&_\_init_end - (unsigned long) \&_\_init_begin;
\end{verbatim}

\begin{verbatim}
printk(KERN_INFO "Memory: %luk/%lu available (%dk kernel code, %dk reserved, %dk data, %dk init, %ldk highmem)\n",
(unsigned long) nr_free_pages() << (PAGE_SHIFT-10),
max_mapnr << (PAGE_SHIFT-10),
codesize >> 10,
reservedpages << (PAGE_SHIFT-10),
datasize >> 10,
initsize >> 10,
(unsigned long) (totalhigh_pages << (PAGE_SHIFT-10))
);
\end{verbatim}

Print out an informational message.

Calculate the size of the code segment, data segment and memory used by initialisation code and data (all functions marked \_\_init will be in this section).

Print out a nice message on how the availability of memory and the amount of memory consumed by the kernel.

\begin{verbatim}
#If CONFIG_X86_PAE
if (!cpu_has_pae)
    panic("cannot execute a PAE-enabled kernel on a PAE-less CPU!");
#endif
\end{verbatim}
E.4 Retiring the Boot Memory Allocator (\texttt{mem_init()})

if (boot_cpu_data.wp_works_ok < 0)  
test_wp_bit();

538-539 If PAE is enabled but the processor does not support it, panic

541-542 Test for the availability of the WP bit

550 \#ifndef CONFIG_SMP
551 zap_low_mappings();
552 \#endif
553
554 }  

551 Cycle through each PGD used by the userspace portion of \texttt{swapper\_pg\_dir}
and map the zero page to it

E.4.2 Function: \texttt{free\_pages\_init()} (arch/i386/mm/init.c)

This function has two important functions, to call \texttt{free\_all\_bootmem()} (See Section E.4.4)
to retire the boot memory allocator and to free all high memory pages to the buddy allocator.

481 static int \_\_init free_pages_init(void)
482 {  
483 extern int ppro_with_ram_bug(void);
484 int bad_ppro, reservedpages, pfn;
485
486 bad_ppro = ppro_with_ram_bug();
487
488 /* this will put all low memory onto the freelists */
489 totalram_pages += free_all_bootmem();
490
491 reservedpages = 0;
492 for (pfn = 0; pfn < max_low_pfn; pfn++) {  
493 /*
494 * Only count reserved RAM pages
495 */
496 if (page_is_ram(pfn) && PageReserved(mem_map+pfn))
497     reservedpages++;
498 }
499 \#ifdef CONFIG_HIGHMEM
500 for (pfn = highend_pfn-1; pfn >= highstart_pfn; pfn--)
501   one_highpage_init((struct page *) (mem_map + pfn), pfn,
502   bad_ppro);
503 totalram_pages += totalhigh_pages;
504 \#endif
486 There is a bug in the Pentium Pros that prevent certain pages in high memory being used. The function \texttt{ppro_with_ram_bug()} checks for its existence.

489 Call \texttt{free_all_bootmem()} to retire the boot memory allocator.

491-498 Cycle through all of memory and count the number of reserved pages that were left over by the boot memory allocator.

500-501 For each page in high memory, call \texttt{one_highpage_init()} (See Section E.4.3). This function clears the \texttt{PG\_reserved} bit, sets the \texttt{PG\_high} bit, sets the count to 1, calls \texttt{\_\_free\_pages()} to give the page to the buddy allocator and increments the \texttt{totalhigh\_pages} count. Pages which kill buggy Pentium Pro’s are skipped.

**E.4.3 Function:** \texttt{one_highpage_init()} \textit{ (/arch/i386/mm/init.c)}

This function initialises the information for one page in high memory and checks to make sure that the page will not trigger a bug with some Pentium Pros. It only exists if \texttt{CONFIG\_HIGHMEM} is specified at compile time.

449 \#ifdef CONFIG\_HIGHMEM
450 void \_\_init one_highpage_init(struct page *page, int pfn, 
\hspace*{1em} int bad_ppro)
451 {
452 \hspace{1em} if (!page_is_ram(pfn)) {
453 \hspace{2em} SetPageReserved(page);
454 \hspace{1em} return;
455 \hspace{1em} }
456
457 \hspace{1em} if (bad_ppro && page_kills_ppro(pfn)) {
458 \hspace{2em} SetPageReserved(page);
459 \hspace{1em} return;
460 \hspace{1em} }
461
462 \hspace{1em} ClearPageReserved(page);
463 \hspace{1em} set_bit(PG\_highmem, &page->flags);
464 \hspace{1em} atomic_set(&page->count, 1);
465 \hspace{1em} \_\_free\_page(page);
466 \hspace{1em} totalhigh\_pages++;
467 }
468 \#endif /* CONFIG\_HIGHMEM */

452-455 If a page does not exist at the PFN, then mark the struct page as reserved so it will not be used.
If the running CPU is susceptible to the Pentium Pro bug and this page is a page that would cause a crash (page_kill_ppro() performs the check), then mark the page as reserved so it will never be allocated.

From here on, the page is a high memory page that should be used so first clear the reserved bit so it will be given to the buddy allocator later.

Set the PG_highmem bit to show it is a high memory page.

Initialise the usage count of the page to 1 which will be set to 0 by the buddy allocator.

Free the page with __free_page() (See Section F.4.2) so that the buddy allocator will add the high memory page to its free lists.

Increment the total number of available high memory pages (totalhigh_pages).

For NUMA, simply call the core function with the specified pgdat.

For UMA, call the core function with the only node contig_page_data.

This is the core function which “retires” the boot memory allocator. It is divided into two major tasks:

- For all unallocated pages known to the allocator for this node;
  - Clear the PG_reserved flag in its struct page
  - Set the count to 1
  - Call __free_pages() so that the buddy allocator can build its free lists
- Free all pages used for the bitmap and free to them to the buddy allocator.
E.4 Retiring the Boot Memory Allocator (free_all_bootmem_core())

```c
static unsigned long __init free_all_bootmem_core(pg_data_t *pgdat) {
    struct page *page = pgdat->node_mem_map;
    bootmem_data_t *bdata = pgdat->bdata;
    unsigned long i, count, total = 0;
    unsigned long idx;

    if (!bdata->node_bootmem_map) BUG();

    count = 0;
    idx = bdata->node_low_pfn -
          (bdata->node_boot_start >> PAGE_SHIFT);
    for (i = 0; i < idx; i++, page++) {
        if (!test_bit(i, bdata->node_bootmem_map)) {
            count++;
            ClearPageReserved(page);
            set_page_count(page, 1);
            __free_page(page);
        }
    }
    total += count;
}
```

252 If no map is available, it means that this node has already been freed and something woeful is wrong with the architecture dependent code so call BUG()

254 A running count of the number of pages given to the buddy allocator

255 `idx` is the last index that is addressable by this node

256-263 Cycle through all pages addressable by this node

257 If the page is marked free then...

258 Increase the running count of pages given to the buddy allocator

259 Clear the PG_reserved flag

260 Set the count to 1 so that the buddy allocator will think this is the last user of the page and place it in its free lists

261 Call the buddy allocator free function so the page will be added to it’s free lists

264 `total` will come the total number of pages given over by this function

270 `page = virt_to_page(bdata->node_bootmem_map);
271 count = 0;
272 for (i = 0;
E.4 Retiring the Boot Memory Allocator (free_all_bootmem_core())

\[ i < \left( \left( bdata->node_low_pfn - (bdata->node_boot_start \gg PAGE_SHIFT) \right) / 8 + PAGE_SIZE - 1 \right) / PAGE_SIZE; \]

\[
i++, page++ \rbrace \lbrace
\begin{align*}
& \text{count}++; \\
& \text{ClearPageReserved}(page); \\
& \text{set_page_count}(page, 1); \\
& \text{__free_page}(page); \\
& \rbrace \\
& \text{total} += \text{count}; \\
& \text{bdata->node_bootmem_map} = \text{NULL}; \\
& \text{return total}; \\
\end{align*}
\]

Free the allocator bitmap and return

270 Get the struct page that is at the beginning of the bootmem map

271 Count of pages freed by the bitmap

272-277 For all pages used by the bitmap, free them to the buddy allocator the same way the previous block of code did

279 Set the bootmem map to NULL to prevent it been freed a second time by accident

281 Return the total number of pages freed by this function, or in other words, return the number of pages that were added to the buddy allocator’s free lists
Appendix F

Physical Page Allocation

Contents

F.1 Allocating Pages ............................................. 405
  F.1.1 Function: alloc_pages() ............................... 405
  F.1.2 Function: _alloc_pages() ............................. 405
  F.1.3 Function: __alloc_pages() ............................ 406
  F.1.4 Function: rmqueue() ................................. 410
  F.1.5 Function: expand() ................................. 412
  F.1.6 Function: balance_classzone() ...................... 414
F.2 Allocation Helper Functions ................................. 418
  F.2.1 Function: alloc_page() .............................. 418
  F.2.2 Function: __get_free_page() ......................... 418
  F.2.3 Function: __get_free_pages() ......................... 418
  F.2.4 Function: __get_dma_pages() ......................... 419
  F.2.5 Function: get_zeroed_page() ......................... 419
F.3 Free Pages .................................................. 420
  F.3.1 Function: __free_pages() ............................ 420
  F.3.2 Function: __free_pages_ok() ......................... 420
F.4 Free Helper Functions ....................................... 425
  F.4.1 Function: free_pages() .............................. 425
  F.4.2 Function: __free_page() ............................ 425
  F.4.3 Function: free_page() .............................. 425
F.1 Allocating Pages

Contents

F.1 Allocating Pages 405
  F.1.1 Function: alloc_pages() 405
  F.1.2 Function: _alloc_pages() 405
  F.1.3 Function: __alloc_pages() 406
  F.1.4 Function: rmqueue() 410
  F.1.5 Function: expand() 412
  F.1.6 Function: balance_classzone() 414

F.1.1 Function: alloc_pages() (include/linux/mm.h)

The call graph for this function is shown at 6.3. It is declared as follows:

```c
static inline struct page * alloc_pages(unsigned int gfp_mask,
unsigned int order)
{
    if (order >= MAX_ORDER)
        return NULL;
    return _alloc_pages(gfp_mask, order);
}
```

The gfp_mask (Get Free Pages) flags tell the allocator how it may behave. For example GFP_WAIT is not set, the allocator will not block and instead return NULL if memory is tight. The order is the power of two number of pages to allocate.

A simple debugging check optimized away at compile time

This function is described next

F.1.2 Function: _alloc_pages() (mm/page_alloc.c)

The function _alloc_pages() comes in two varieties. The first is designed to only work with UMA architectures such as the x86 and is in mm/page_alloc.c. It only refers to the static node contig_page_data. The second is in mm/numa.c and is a simple extension. It uses a node-local allocation policy which means that memory will be allocated from the bank closest to the processor. For the purposes of this book, only the mm/page_alloc.c version will be examined but developers on NUMA architectures should read _alloc_pages() and _alloc_pages_pgdat() as well in mm/numa.c

```c
#ifdef CONFIG_DISCONTIGMEM
struct page *_alloc_pages(unsigned int gfp_mask,
unsigned int order)
{
```
The ifndef is for UMA architectures like the x86. NUMA architectures used the alloc_pages() function in mm/numa.c which employs a node local policy for allocations.

node_zone_lists is an array of preferred fallback zones to allocate from. It is initialised in build_zone_lists(). (See Section B.1.6) The lower 16 bits of gfp_mask indicate what zone is preferable to allocate from. Applying the bitmask gfp_mask & GFP_ZONE_MASK will give the index in node_zone_lists we prefer to allocate from.

F.1.3 Function: __alloc_pages() (mm/page_alloc.c)

At this stage, we’ve reached what is described as the “heart of the zoned buddy allocator”, the __alloc_pages() function. It is responsible for cycling through the fallback zones and selecting one suitable for the allocation. If memory is tight, it will take some steps to address the problem. It will wake kswapd and if necessary it will do the work of kswapd manually.

```c
struct page * __alloc_pages(unsigned int gfp_mask,
                             unsigned int order,
                             zonelist_t *zonelist)
```
F.1 Allocating Pages (_-_alloc_pages())

346    page = rmqueue(z, order);
347    if (page)
348       return page;
349   }
350   }

334 Set zone to be the preferred zone to allocate from
335 The preferred zone is recorded as the classzone. If one of the pages low
336 watermarks is reached later, the classzone is marked as needing balance
337
338-339 An unnecessary sanity check. build_zonelists() would need to be
339 seriously broken for this to happen
340
341-342 This style of block appears a number of times in this function. It reads
342 as “cycle through all zones in this fallback list and see can the allocation be
343 satisfied without violating watermarks”. Note that the pages_low for each
344 fallback zone is added together. This is deliberate to reduce the probability a
345 fallback zone will be used.
346
347 z is the zone currently been examined. The zone variable is moved to the next
348 fallback zone
349
350-351 If this is the last zone in the fallback list, break
351
352 Increment the number of pages to be allocated by the watermark for easy
353 comparisons. This happens for each zone in the fallback zones. While this
354 appears first to be a bug, this behavior is actually intended to reduce the
355 probability a fallback zone is used.
356
357 Allocate the page block if it can be assigned without reaching the
358 pages_min watermark. rmqueue() (See Section F.1.4) is responsible from re-
359 moving the block of pages from the zone
360
361-362 If the pages could be allocated, return a pointer to them
362
352    classzone->need_balance = 1;
353    mb();
354    if (waitqueue_active(&kswapd_wait))
355       wake_up_interruptible(&kswapd_wait);
356
357    zone = zonelist->zones;
358    min = 1UL << order;
359    for (; ; ) {
360       unsigned long local_min;
361       zone_t *z = *(zone++);
362       if (!z)
363          break;
local_min = z->pages_min;
if (!(gfp_mask & __GFP_WAIT))
    local_min >>= 2;
min += local_min;
if (z->free_pages > min) {
    page = rmqueue(z, order);
    if (page)
        return page;
}
}

Mark the preferred zone as needing balance. This flag will be read later by kswapd

This is a memory barrier. It ensures that all CPU’s will see any changes made to variables before this line of code. This is important because kswapd could be running on a different processor to the memory allocator.

Wake up kswapd if it is asleep

Begin again with the first preferred zone and min value

Cycle through all the zones. This time, allocate the pages if they can be allocated without hitting the pages_min watermark

local_min how low a number of free pages this zone can have

If the process can not wait or reschedule (__GFP_WAIT is clear), then allow the zone to be put in further memory pressure than the watermark normally allows

/* here we’re in the low on memory slow path */
rebalance:
if (current->flags & (PF_MEMALLOC | PF_MEMDIE)) {
    zone = zonelist->zones;
    for (;;) {
        zone_t *z = *(zone++);
        if (!z)
            break;
        page = rmqueue(z, order);
        if (page)
            return page;
    }
}
F.1 Allocating Pages (\_\_alloc\_pages())

```c
    return NULL;
}
```

This label is returned to after an attempt is made to synchronously free pages. From this line on, the low on memory path has been reached. It is likely the process will sleep.

These two flags are only set by the OOM killer. As the process is trying to kill itself cleanly, allocate the pages if at all possible as it is known they will be freed very soon.

```c
    /* Atomic allocations - we can’t balance anything */
    if (!(gfp_mask & __GFP_WAIT))
        return NULL;
    
    page = balance_classzone(classzone, gfp_mask, order, &freed);
    if (page)
        return page;

    zone = zonelist->zones;
    min = 1UL << order;
    for (;;) {
        zone_t *z = *(zone);
        if (!z)
            break;

        min += z->pages_min;
        if (z->free_pages > min) {
            page = rmqueue(z, order);
            if (page)
                return page;
        }
    }
```

If the calling process can not sleep, return NULL as the only way to allocate the pages from here involves sleeping.
balance_classzone() (See Section F.1.6) performs the work of kswapd in a synchronous fashion. The principal difference is that instead of freeing the memory into a global pool, it is kept for the process using the current → local_pages linked list.

If a page block of the right order has been freed, return it. Just because this is NULL does not mean an allocation will fail as it could be a higher order of pages that was released.

This is identical to the block above. Allocate the page blocks if it can be done without hitting the pages_min watermark.

Satisfying a large allocation like $2^4$ number of pages is difficult. If it has not been satisfied by now, it is better to simply return NULL.

Yield the processor to give kswapd a chance to work.

Attempt to balance the zones again and allocate.

Function: rmqueue() (mm/page_alloc.c)
This function is called from __alloc_pages(). It is responsible for finding a block of memory large enough to be used for the allocation. If a block of memory of the requested size is not available, it will look for a larger order that may be split into two buddies. The actual splitting is performed by the expand() (See Section F.1.5) function.

```c
198 static FASTCALL(struct page *rmqueue(zone_t *zone, unsigned int order));
199 static struct page * rmqueue(zone_t *zone, unsigned int order)
200 {
201     free_area_t * area = zone->free_area + order;
202     unsigned int curr_order = order;
203     struct list_head *head, *curr;
204     unsigned long flags;
205     struct page *page;
206     spin_lock_irqsave(&zone->lock, flags);
207     do {
208         head = &area->free_list;
209         curr = head->next;
210         if (curr != head) {
211             unsigned int index;
212             unsigned int order;
The parameters are the zone to allocate from and what order of pages are required.

Because the `free_area` is an array of linked lists, the order may be used as an index within the array.

Acquire the zone lock.

This while block is responsible for finding what order of pages we will need to allocate from. If there isn’t a free block at the order we are interested in, check the higher blocks until a suitable one is found.

`head` is the list of free page blocks for this order.

`curr` is the first block of pages.

If there is a free page block at this order, then allocate it...
F.1 Allocating Pages (rmqueue())

Page is set to be a pointer to the first page in the free block.

Sanity check that checks to make sure the page this page belongs to this zone and is within the zone_mem_map. It is unclear how this could possibly happen without severe bugs in the allocator itself that would place blocks in the wrong zones.

As the block is going to be allocated, remove it from the free list.

index treats the zone_mem_map as an array of pages so that index will be the offset within the array.

Toggle the bit that represents this pair of buddies. MARK_USED() is a macro which calculates which bit to toggle.

Update the statistics for this zone. 1UL<<order is the number of pages been allocated.

expand() (See Section F.1.5) is the function responsible for splitting page blocks of higher orders.

No other updates to the zone need to take place so release the lock.

Show that the page is in use.

Page block has been successfully allocated so return it.

If a page block was not free of the correct order, move to a higher order of page blocks and see what can be found there.

No other updates to the zone need to take place so release the lock.

No page blocks of the requested or higher order are available so return failure.

F.1.5 Function: expand() (mm/page_alloc.c)

This function splits page blocks of higher orders until a page block of the needed order is available.

177 static inline struct page * expand (zone_t *zone,
    struct page *page,
    unsigned long index,
    int low,
    int high,
    free_area_t * area)

    unsigned long size = 1 << high;
while (high > low) {
  if (BAD_RANGE(zone, page))
    BUG();
  area--;
  high--;
  size >>= 1;
  list_add(&(page)->list, &(area)->free_list);
  MARK_USED(index, high, area);
  index += size;
  page += size;
}

if (BAD_RANGE(zone, page))
  BUG();
return page;

177 The parameters are

zone is where the allocation is coming from
page is the first page of the block been split
index is the index of page within mem_map
low is the order of pages needed for the allocation
high is the order of pages that is been split for the allocation
area is the free_area_t representing the high order block of pages

180 size is the number of pages in the block that is to be split

182-192 Keep splitting until a block of the needed page order is found

183-184 Sanity check that checks to make sure the page this page belongs to this
 zone and is within the zone_mem_map

185 area is now the next free_area_t representing the lower order of page blocks

186 high is the next order of page blocks to be split

187 The size of the block been split is now half as big

188 Of the pair of buddies, the one lower in the mem_map is added to the free list
 for the lower order

189 Toggle the bit representing the pair of buddies

190 index now the index of the second buddy of the newly created pair

191 page now points to the second buddy of the newly created paid
Sanity check

The blocks have been successfully split so return the page

F.1.6 Function: balance_classzone()  (mm/page_alloc.c)

This function is part of the direct-reclaim path. Allocators which can sleep will call this function to start performing the work of kswapd in a synchronous fashion. As the process is performing the work itself, the pages it frees of the desired order are reserved in a linked list in current→local_pages and the number of page blocks in the list is stored in current→nr_local_pages. Note that page blocks is not the same as number of pages. A page block could be of any order.

```
253 static struct page * balance_classzone(zone_t * classzone,
                                          unsigned int gfp_mask,
                                          unsigned int order,
                                          int * freed)
254 {
    struct page * page = NULL;
    int __freed = 0;
258     if (!(gfp_mask & __GFP_WAIT))
259         goto out;
260     if (in_interrupt())
261         BUG();
263     current->allocation_order = order;
264     current->flags |= PF_MEMALLOC | PF_FREE_PAGES;
266     __freed = try_to_free_pages_zone(classzone, gfp_mask);
267     current->flags &= ~(PF_MEMALLOC | PF_FREE_PAGES);
258-259 If the caller is not allowed to sleep, then goto out to exit the function. For this to occur, the function would have to be called directly or __alloc_pages() would need to be deliberately broken
260-261 This function may not be used by interrupts. Again, deliberate damage would have to be introduced for this condition to occur
263 Record the desired size of the allocation in current→allocation_order. This is actually unused although it could have been used to only add pages of the desired order to the local_pages list. As it is, the order of pages in the list is stored in page→index
264 Set the flags which will the free functions to add the pages to the local_list
Free pages directly from the desired zone with `try_to_free_pages_zone()` (See Section J.5.3). This is where the direct-reclaim path intersects with `kswapd`.

Clear the flags again so that the free functions do not continue to add pages to the `local_pages` list.

```c
if (current->nr_local_pages) {
    struct list_head * entry, * local_pages;
    struct page * tmp;
    int nr_pages;

    local_pages = &current->local_pages;

    if (likely(__freed)) {
        /* pick from the last inserted so we're lifo */
        entry = local_pages->next;
        do {
            tmp = list_entry(entry, struct page, list);
            if (tmp->index == order &&
                memclass(page_zone(tmp), classzone)) {
                list_del(entry);
                current->nr_local_pages--;
                set_page_count(tmp, 1);
                page = tmp;

                if (page->buffers)
                    BUG();
                if (page->mapping)
                    BUG();
                if (!VALID_PAGE(page))
                    BUG();
                if (PageLocked(page))
                    BUG();
                if (PageLRU(page))
                    BUG();
                if (PageActive(page))
                    BUG();
                if (PageDirty(page))
                    BUG();

                break;
            }
        } while ((entry = entry->next) != local_pages);
    }
}
```
Presuming that pages exist in the local_pages list, this function will cycle through the list looking for a page block belonging to the desired zone and order.

270 Only enter this block if pages are stored in the local list
275 Start at the beginning of the list
277 If pages were freed with try_to_free_pages_zone() then...
279 The last one inserted is chosen first as it is likely to be cache hot and it is desirable to use pages that have been recently referenced
280-305 Cycle through the pages in the list until we find one of the desired order and zone
281 Get the page from this list entry
282 The order of the page block is stored in page→index so check if the order matches the desired order and that it belongs to the right zone. It is unlikely that pages from another zone are on this list but it could occur if swap_out() is called to free pages directly from process page tables
283 This is a page of the right order and zone so remove it from the list
284 Decrement the number of page blocks in the list
285 Set the page count to 1 as it is about to be freed
286 Set page as it will be returned. tmp is needed for the next block for freeing the remaining pages in the local list
288-301 Perform the same checks that are performed in __free_pages_ok() to ensure it is safe to free this page
305 Move to the next page in the list if the current one was not of the desired order and zone

```
   nr_pages = current->nr_local_pages;
   /* free in reverse order so that the global * order will be lifo */
   while ((entry = local_pages->prev) != local_pages) {
       list_del(entry);
       tmp = list_entry(entry, struct page, list);
       __free_pages_ok(tmp, tmp->index);
       if (!nr_pages--)
           BUG();
   }
   current->nr_local_pages = 0;
```
F.1 Allocating Pages (balance_classzone())

```c
417
319   out:
320   *freed = __freed;
321   return page;
322 }
```

This block frees the remaining pages in the list.

308 Get the number of page blocks that are to be freed

310 Loop until the local_pages list is empty

311 Remove this page block from the list

312 Get the struct page for the entry

313 Free the page with __free_pages_ok() (See Section F.3.2)

314-315 If the count of page blocks reaches zero and there is still pages in the list, it means that the accounting is seriously broken somewhere or that someone added pages to the local_pages list manually so call BUG()

317 Set the number of page blocks to 0 as they have all been freed

320 Update the freed parameter to tell the caller how many pages were freed in total

321 Return the page block of the requested order and zone. If the freeing failed, this will be returning NULL
F.2 Allocation Helper Functions

Contents

F.2 Allocation Helper Functions 418
  F.2.1 Function: alloc_page() 418
  F.2.2 Function: __get_free_page() 418
  F.2.3 Function: __get_free_pages() 418
  F.2.4 Function: __get_dma_pages() 419
  F.2.5 Function: get_zeroed_page() 419

This section will cover miscellaneous helper functions and macros the Buddy Allocator uses to allocate pages. Very few of them do ‘real’ work and are available just for the convenience of the programmer.

F.2.1 Function: alloc_page()  (include/linux/mm.h)

This trivial macro just calls alloc_pages() with an order of 0 to return 1 page. It is declared as follows

449 #define alloc_page(gfp_mask) alloc_pages(gfp_mask, 0)

F.2.2 Function: __get_free_page() (include/linux/mm.h)

This trivial function calls __get_free_pages() with an order of 0 to return 1 page. It is declared as follows

454 #define __get_free_page(gfp_mask) \
455 __get_free_pages((gfp_mask),0)

F.2.3 Function: __get_free_pages() (mm/page_alloc.c)

This function is for callers who do not want to worry about pages and only get back an address it can use. It is declared as follows

428 unsigned long __get_free_pages(unsigned int gfp_mask, 
428        unsigned int order)
428 {
430     struct page * page;
431
432     page = alloc_pages(gfp_mask, order);
433     if (!page)
434         return 0;
435         return (unsigned long) page_address(page);
436 }

431 alloc_pages() does the work of allocating the page block. See Section F.1.1
433-434 Make sure the page is valid
435 page_address() returns the physical address of the page
F.2.4 Function: \texttt{\_\_get\_dma\_pages()} (include/linux/mm.h)

This is of principle interest to device drivers. It will return memory from ZONE_DMA suitable for use with DMA devices. It is declared as follows

\begin{verbatim}
#define \_\_get\_dma\_pages(gfp\_mask, order) \
\_\_get\_free\_pages((gfp\_mask) | GFP\_DMA,(order))
\end{verbatim}

458 The gfp\_mask is or-ed with GFP\_DMA to tell the allocator to allocate from ZONE_DMA

F.2.5 Function: get\_zeroed\_page() (mm/page\_alloc.c)

This function will allocate one page and then zero out the contents of it. It is declared as follows

\begin{verbatim}
unsigned long get\_zeroed\_page(unsigned int gfp\_mask)
{
    struct page * page;
    page = alloc\_pages(gfp\_mask, 0);
    if (page) {
        void *address = page\_address(page);
        clear\_page(address);
        return (unsigned long) address;
    }
    return 0;
}
\end{verbatim}

438 gfp\_mask are the flags which affect allocator behaviour.
442 alloc\_pages() does the work of allocating the page block. See Section F.1.1
444 page\_address() returns the physical address of the page
445 clear\_page() will fill the contents of a page with zero
446 Return the address of the zeroed page
F.3 Free Pages

Contents

F.3.1 Function: __free_pages() 420
F.3.2 Function: __free_pages_ok() 420

F.3.1 Function: __free_pages()  (mm/page_alloc.c)

The call graph for this function is shown in Figure 6.4. Just to be confusing, the opposite to alloc_pages() is not free_pages(), it is __free_pages(). free_pages() is a helper function which takes an address as a parameter, it will be discussed in a later section.

```c
void __free_pages(struct page *page, unsigned int order)
{
    if (!PageReserved(page) && put_page_testzero(page))
        __free_pages_ok(page, order);
}
```

The parameters are the page we wish to free and what order block it is

Sanity checked. PageReserved() indicates that the page is reserved by the boot memory allocator. put_page_testzero() is just a macro wrapper around atomic_dec_and_test() decrements the usage count and makes sure it is zero

Call the function that does all the hard work

F.3.2 Function: __free_pages_ok()  (mm/page_alloc.c)

This function will do the actual freeing of the page and coalesce the buddies if possible.

```c
static void FASTCALL(__free_pages_ok (struct page *page, unsigned int order));
```

```c
static void __free_pages_ok (struct page *page, unsigned int order)
{
    unsigned long index, page_idx, mask, flags;
    free_area_t *area;
    struct page *base;
    zone_t *zone;

    if (PageLRU(page)) {
        if (unlikely(in_interrupt()))
            BUG();
        lru_cache_del(page);
    }
```
if (page->buffers)
    BUG();
if (page->mapping)
    BUG();
if (!VALID_PAGE(page))
    BUG();
if (PageLocked(page))
    BUG();
if (PageActive(page))
    BUG();
page->flags &= ~((1<<PG_referenced) | (1<<PG_dirty));

The parameters are the beginning of the page block to free and what order number of pages are to be freed.

A dirty page on the LRU will still have the LRU bit set when pinned for IO. On IO completion, it is freed so it must now be removed from the LRU list.

Sanity checks

The flags showing a page has being referenced and is dirty have to be cleared because the page is now free and not in use.

If this flag is set, the pages freed are to be kept for the process doing the freeing. This is set by balance_classzone() (See Section F.1.6) during page allocation if the caller is freeing the pages itself rather than waiting for kswapd to do the work.
The zone the page belongs to is encoded within the page flags. The `page_zone()` macro returns the zone.

The calculation of mask is discussed in companion document. It is basically related to the address calculation of the buddy base is the beginning of this `zone_mem_map`. For the buddy calculation to work, it was to be relative to an address 0 so that the addresses will be a power of two.

`page_idx` treats the `zone_mem_map` as an array of pages. This is the index page within the map.

If the index is not the proper power of two, things are severely broken and calculation of the buddy will not work.

This index is the bit index within `free_area→map`.

`area` is the area storing the free lists and map for the order block the pages are been freed from.

```c
spin_lock_irqsave(&zone->lock, flags);
zone->free_pages -= mask;
while (mask + (1 << (MAX_ORDER-1))) {
    struct page *buddy1, *buddy2;
    if (area >= zone->free_area + MAX_ORDER)
        BUG();
    if (!__test_and_change_bit(index, area->map))
        /*
         * the buddy page is still allocated.
         */
        break;
    /*
     * Move the buddy up one level.
     * This code is taking advantage of the identity:
     * -mask = 1+~mask
     */
    buddy1 = base + (page_idx ^ -mask);
    buddy2 = base + page_idx;
    if (BAD_RANGE(zone,buddy1))
        BUG();
    if (BAD_RANGE(zone,buddy2))
        BUG();
```

F.3 Free Pages (__free_pages_ok())
list_del(&buddy1->list);
mask <<= 1;
area++;
index >>= 1;
page_idx &= mask;

The zone is about to be altered so take out the lock. The lock is an interrupt
safe lock as it is possible for interrupt handlers to allocate a page in this path

Another side effect of the calculation of mask is that \(-mask\) is the number of
pages that are to be freed

The allocator will keep trying to coalesce blocks together until it either
cannot merge or reaches the highest order that can be merged. mask will be
adjusted for each order block that is merged. When the highest order that can
be merged is reached, this while loop will evaluate to 0 and exit.

If by some miracle, mask is corrupt, this check will make sure the
free_area array will not not be read beyond the end

Toggle the bit representing this pair of buddies. If the bit was previously zero,
both buddies were in use. As this buddy is been freed, one is still in use and
cannot be merged

The calculation of the two addresses is discussed in Chapter 6

Sanity check to make sure the pages are within the correct zone_mem_map
and actually belong to this zone

The buddy has been freed so remove it from any list it was part of

Prepare to examine the higher order buddy for merging

Move the mask one bit to the left for order \(2^{k+1}\)

area is a pointer within an array so area++ moves to the next index

The index in the bitmap of the higher order

The page index within the zone_mem_map for the buddy to merge

list_add((base + page_idx)->list, &area->free_list);
spin_unlock_irqrestore(&zone->lock, flags);
return;
local_freelist:
if (current->nr_local_pages)
As much merging as possible as completed and a new page block is free so add it to the free_list for this order.

Changes to the zone is complete so free the lock and return.

This is the code path taken when the pages are not freed to the main pool but instead are reserved for the process doing the freeing.

If the process already has reserved pages, it is not allowed to reserve any more so return back. This is unusual as balance_classzone() assumes that more than one page block may be returned on this list. It is likely to be an oversight but may still work if the first page block freed is the same order and zone as required by balance_classzone().

An interrupt does not have process context so it has to free in the normal fashion. It is unclear how an interrupt could end up here at all. This check is likely to be bogus and impossible to be true.

Add the page block to the list for the processes local_pages.

Record what order allocation it was for freeing later.

Increase the use count for nr_local_pages.
F.4 Free Helper Functions

Contents

F.4 Free Helper Functions 425
  F.4.1 Function: free_pages() 425
  F.4.2 Function: __free_page() 425
  F.4.3 Function: free_page() 425

These functions are very similar to the page allocation helper functions in that they do no “real” work themselves and depend on the __free_pages() function to perform the actual free.

F.4.1 Function: free_pages()  (mm/page_alloc.c)

This function takes an address instead of a page as a parameter to free. It is declared as follows

457 void free_pages(unsigned long addr, unsigned int order)
458 {
459     if (addr != 0)
460         __free_pages(virt_to_page(addr), order);
461 }

460 The function is discussed in Section F.3.1. The macro virt_to_page() returns the struct page for the addr

F.4.2 Function: __free_page()  (include/linux/mm.h)

This trivial macro just calls the function __free_pages() (See Section F.3.1) with an order 0 for 1 page. It is declared as follows

472 #define __free_page(page) __free_pages((page), 0)

F.4.3 Function: free_page()  (include/linux/mm.h)

This trivial macro just calls the function free_pages(). The essential difference between this macro and __free_page() is that this function takes a virtual address as a parameter and __free_page() takes a struct page.

472 #define free_page(addr) free_pages((addr), 0)
Appendix G

Non-Contiguous Memory Allocation

Contents

G.1 Allocating A Non-Contiguous Area ......................... 427
   G.1.1 Function: vmalloc() .................................. 427
   G.1.2 Function: __vmalloc() ................................. 427
   G.1.3 Function: get_vm_area() ............................ 428
   G.1.4 Function: vmalloc_area_pages() ....................... 430
   G.1.5 Function: __vmalloc_area_pages() ................... 431
   G.1.6 Function: alloc_area_pmd() .......................... 432
   G.1.7 Function: alloc_area_pte() ......................... 434
   G.1.8 Function: vmap() ..................................... 435

G.2 Freeing A Non-Contiguous Area ............................. 437
   G.2.1 Function: vfree() ..................................... 437
   G.2.2 Function: vmfree_area_pages() ....................... 438
   G.2.3 Function: free_area_pmd() .......................... 439
   G.2.4 Function: free_area_pte() .......................... 440
G.1 Allocating A Non-Contiguous Area

Contents

G.1 Allocating A Non-Contiguous Area 427
  G.1.1 Function: vmalloc()  427
  G.1.2 Function: __vmalloc()  427
  G.1.3 Function: get_vm_area()  428
  G.1.4 Function: vmalloc_area_pages()  430
  G.1.5 Function: __vmalloc_area_pages()  431
  G.1.6 Function: alloc_area_pmd()  432
  G.1.7 Function: alloc_area_pte()  434
  G.1.8 Function: vmap()  435

G.1.1 Function: vmalloc()  \textit{(include/linux/vmalloc.h)}

The call graph for this function is shown in Figure 7.2. The following macros only by their \texttt{GFP\_ags} (See Section 6.4). The size parameter is page aligned by \texttt{__vmalloc()} (See Section G.1.2).

37 static inline void * vmalloc (unsigned long size)  
38 {  
39  \hspace{1em} return __vmalloc(size, GFP_KERNEL | __GFP_HIGHMEM, PAGE_KERNEL);  
40  }
45
46 static inline void * vmalloc_dma (unsigned long size)  
47 {  
48  \hspace{1em} return __vmalloc(size, GFP_KERNEL|GFP_DMA, PAGE_KERNEL);  
49  }
54
55 static inline void * vmalloc_32(unsigned long size)  
56 {  
57  \hspace{1em} return __vmalloc(size, GFP_KERNEL, PAGE_KERNEL);  
58  }

37 The flags indicate that to use either \texttt{ZONE\_NORMAL} or \texttt{ZONE\_HIGHMEM} as necessary
46 The flag indicates to only allocate from \texttt{ZONE\_DMA}
55 Only physical pages from \texttt{ZONE\_NORMAL} will be allocated

G.1.2 Function: __vmalloc()  \textit{(mm/vmalloc.c)}

This function has three tasks. It page aligns the size request, asks \texttt{get_vm_area()} to find an area for the request and uses \texttt{vmalloc_area_pages()} to allocate the PTEs for the pages.

261 \hspace{1em} void * __vmalloc (unsigned long size, int gfp_mask, pgprot_t prot)  
262 {
void * addr;
struct vm_struct *area;

size = PAGE_ALIGN(size);
if (!size || (size >> PAGE_SHIFT) > num_physpages)
    return NULL;
area = get_vm_area(size, VM_ALLOC);
if (!area)
    return NULL;
addr = area->addr;
if (__vmalloc_area_pages(VMALLOC_VMADDR(addr), size, gfp_mask,
    prot, NULL)) {
    vfree(addr);
    return NULL;
}
return addr;

The parameters are the size to allocate, the GFP_ flags to use for allocation
and what protection to give the PTE

Align the size to a page size

Sanity check. Make sure the size is not 0 and that the size requested is not
larger than the number of physical pages has been requested

Find an area of virtual address space to store the allocation with get_vm_area()
(See Section G.1.3)

The addr field has been filled by get_vm_area()

Allocate the PTE entries needed for the allocation with __vmalloc_area_pages()
(See Section G.1.5). If it fails, a non-zero value -ENOMEM is returned

If the allocation fails, free any PTEs, pages and descriptions of the area

Return the address of the allocated area

G.1.3 Function: get_vm_area() (mm/vmalloc.c)
To allocate an area for the vm_struct, the slab allocator is asked to provide
the necessary memory via kmalloc(). It then searches the vm_struct list linearly
looking for a region large enough to satisfy a request, including a page pad at the
end of the area.
G.1 Allocating A Non-Contiguous Area (get_vm_area())

197 unsigned long addr, next;
198 struct vm_struct **p, *tmp, *area;
199
200 area = (struct vm_struct *) kmalloc(sizeof(*area), GFP_KERNEL);
201 if (!area)
202 return NULL;
203
204 size += PAGE_SIZE;
205 if(!size) {
206 kfree (area);
207 return NULL;
208 }
209
210 addr = VMALLOC_START;
211 write_lock(&vmlist_lock);
212 for (p = &vmlist; (tmp = *p) ; p = &tmp->next) {
213 if ((size + addr) < addr)
214 goto out;
215 if (size + addr <= (unsigned long) tmp->addr)
216 break;
217 next = tmp->size + (unsigned long) tmp->addr;
218 if (next > addr)
219 addr = next;
220 if (addr > VMALLOC_END-size)
221 goto out;
222 }
223 area->flags = flags;
224 area->addr = (void *)addr;
225 area->size = size;
226 area->next = *p;
227 *p = area;
228 write_unlock(&vmlist_lock);
229 return area;
230 out:
231 write_unlock(&vmlist_lock);
232 kfree(area);
233 return NULL;
234 }

195 The parameters is the size of the requested region which should be a multiple of the page size and the area flags, either VM_ALLOC or VM_IOREMAP

200-202 Allocate space for the vm_struct description struct

204 Pad the request so there is a page gap between areas. This is to guard against
G.1 Allocating A Non-Contiguous Area (get_vm_area())

overwrites

205-206 This is to ensure the size is not 0 after the padding due to an overflow. If something does go wrong, free the area just allocated and return NULL

210 Start the search at the beginning of the vmalloc address space

211 Lock the list

212-222 Walk through the list searching for an area large enough for the request

213-214 Check to make sure the end of the addressable range has not been reached

215-216 If the requested area would fit between the current address and the next area, the search is complete

217 Make sure the address would not go over the end of the vmalloc address space

223-225 Copy in the area information

226-227 Link the new area into the list

228-229 Unlock the list and return

231 This label is reached if the request could not be satisfied

232 Unlock the list

233-234 Free the memory used for the area descriptor and return

G.1.4 Function: vmalloc_area_pages() (mm/vmalloc.c)

This is just a wrapper around __vmalloc_area_pages(). This function exists for compatibility with older kernels. The name change was made to reflect that the new function __vmalloc_area_pages() is able to take an array of pages to use for insertion into the pagetables.

189 int vmalloc_area_pages(unsigned long address, unsigned long size,
190 int gfp_mask, pgprot_t prot)
191 {
192     return __vmalloc_area_pages(address, size, gfp_mask, prot, NULL);
193 }

192 Call __vmalloc_area_pages() with the same parameters. The pages array is passed as NULL as the pages will be allocated as necessary
This is the beginning of a standard page table walk function. This top level function will step through all PGDs within an address range. For each PGD, it will call \texttt{pmd\_alloc()} to allocate a PMD directory and call \texttt{alloc\_area\_pmd()} for the directory.

```c
155 static inline int \_\_vmalloc\_area\_pages (unsigned long address, unsigned long size, int gfp\_mask, pgprot\_t prot, struct page **pages)
156 {
157   pgd\_t * dir;
158   unsigned long end = address + size;
159   int ret;
160   dir = pgd\_offset\_k(address);
161   spin\_lock(&init\_mm.page\_table\_lock);
162   do {
163     pmd\_t *pmd;
164     pmd = pmd\_alloc(&init\_mm, dir, address);
165     ret = \-ENOMEM;
166     if (!pmd)
167       break;
168     ret = \-ENOMEM;
169     if (alloc\_area\_pmd(pmd, address, end - address, gfp\_mask, prot, pages))
170       break;
171     address = (address + PGDIR\_SIZE) & PGDIR\_MASK;
172     dir++;
173     ret = 0;
174   } while (address && (address < end));
175   spin\_unlock(&init\_mm.page\_table\_lock);
176   flush\_cache\_all();
177   return ret;
178 }
```

The parameters are:

- \texttt{address} is the starting address to allocate PMDs for
- \texttt{size} is the size of the region
**G.1 Allocating A Non-Contiguous Area (\_\_vmalloc\_area\_pages())**

\[gfp\_mask\] is the \texttt{GFP}\_\_ags for \texttt{alloc\_pages()} (See Section F.1.1)

\[prot\] is the protection to give the PTE entry

\[pages\] is an array of pages to use for insertion instead of having \texttt{alloc\_area\_pte()} allocate them one at a time. Only the \texttt{vmap()} interface passes in an array

The end address is the starting address plus the size

Get the PGD entry for the starting address

Lock the kernel reference page table

For every PGD within this address range, allocate a PMD directory and call \texttt{alloc\_area\_pmd()} (See Section G.1.6)

Allocate a PMD directory

Call \texttt{alloc\_area\_pmd()} (See Section G.1.6) which will allocate a PTE for each PTE slot in the PMD

address becomes the base address of the next PGD entry

Move \texttt{dir} to the next PGD entry

Release the lock to the kernel page table

\texttt{flush\_cache\_all()} will flush all CPU caches. This is necessary because the kernel page tables have changed

Return success

**G.1.6 Function: alloc\_area\_pmd() (mm/vmalloc.c)**

This is the second stage of the standard page table walk to allocate PTE entries for an address range. For every PMD within a given address range on a PGD, \texttt{pte\_alloc()} will creates a PTE directory and then \texttt{alloc\_area\_pte()} will be called to allocate the physical pages

```c
static inline int alloc\_area\_pmd(pmd\_t \* pmd, unsigned long address,
                                  unsigned long size, int gfp\_mask,
                                  pgprot\_t prot, struct page ***pages)
{
    unsigned long end;

    address &= ~PGDIR\_MASK;
    end = address + size;
    if (end > PGDIR\_SIZE)
        end = PGDIR\_SIZE;
    do {
```
The parameters are:

- **pmd** is the PMD that needs the allocations
- **address** is the starting address to start from
- **size** is the size of the region within the PMD to allocate for
- **gfp_mask** is the GFP_flags for alloc_pages() (See Section F.1.1)
- **prot** is the protection to give the PTE entry
- **pages** is an optional array of pages to use instead of allocating each page individually

Align the starting address to the PGD

Calculate end to be the end of the allocation or the end of the PGD, whichever occurs first

For every PMD within the given address range, allocate a PTE directory and call alloc_area_pte() (See Section G.1.7)

Allocate the PTE directory

Call alloc_area_pte() which will allocate the physical pages if an array of pages is not already supplied with pages

address becomes the base address of the next PMD entry

Move pmd to the next PMD entry

Return success
G.1.7 Function: alloc_area_pte()

This is the last stage of the page table walk. For every PTE in the given PTE directory and address range, a page will be allocated and associated with the PTE.

```c
95 static inline int alloc_area_pte (pte_t * pte, unsigned long address,
96 unsigned long size, int gfp_mask,
97 pgprot_t prot, struct page ***pages)
98 {
99     unsigned long end;
100    address &= ~PMD_MASK;
101    end = address + size;
102    if (end > PMD_SIZE)
103        end = PMD_SIZE;
104    do {
105        struct page * page;
106            if (!pages) {
107                spin_unlock(&init_mm.page_table_lock);
108                page = alloc_page(gfp_mask);
109                spin_lock(&init_mm.page_table_lock);
110            } else {
111                page = (**pages);
112                (*pages)++;
113            } else {
114                /* Add a reference to the page so we can free later */
115                if (page)
116                    atomic_inc(&page->count);
117            }
118            if (!pte_none(*pte))
119                printk(KERN_ERR "alloc_area_pte: page already exists\n");
120            if (!page)
121                return -ENOMEM;
122            set_pte(pte, mk_pte(page, prot));
123            address += PAGE_SIZE;
124            pte++;
125        } while (address < end);
126    return 0;
127 }
```

Align the address to a PMD directory.

103-104 The end address is the end of the request or the end of the directory, whichever occurs first.
105-128 Loop through every PTE in this page. If a pages array is supplied, use pages from it to populate the table, otherwise allocate each one individually

108-111 If an array of pages is not supplied, unlock the kernel reference pagetable, allocate a page with alloc_page() and reacquire the spinlock

112-120 Else, take one page from the array and increment it's usage count as it is about to be inserted into the reference page table

121-122 If the PTE is already in use, it means that the areas in the vmalloc region are overlapping somehow

123-124 Return failure if physical pages are not available

125 Set the page with the desired protection bits (prot) into the PTE

126 address becomes the address of the next PTE

127 Move to the next PTE

129 Return success

G.1.8 Function: vmap() (mm/vmalloc.c)

This function allows a caller-supplied array of pages to be inserted into the vmalloc address space. This is unused in 2.4.22 and I suspect it is an accidental backport from 2.6.x where it is used by the sound subsystem core.

281 void * vmap(struct page **pages, int count,
282 unsigned long flags, pgprot_t prot)
283 {
284     void * addr;
285     struct vm_struct *area;
286     unsigned long size = count << PAGE_SHIFT;
287
288     if (!size || size > (max_mapnr << PAGE_SHIFT))
289         return NULL;
290     area = get_vm_area(size, flags);
291     if (!area) {
292         return NULL;
293     }
294     addr = area->addr;
295     if ($_vmalloc_area_pages(VMALLOC_VMADDR(addr), size, 0,
296                              prot, &pages)) {
297         vfree(addr);
298         return NULL;
299     }
300     return addr;
301 }
The parameters are;

- **pages** is the caller-supplied array of pages to insert
- **count** is the number of pages in the array
- **flags** is the flags to use for the `vm_struct`
- **prot** is the protection bits to set the PTE with

Calculate the size in bytes of the region to create based on the size of the array.

Make sure the size of the region does not exceed limits.

Use `get_vm_area()` to find a region large enough for the mapping. If one is not found, return NULL.

Get the virtual address of the area.

Insert the array into the pagetable with `__vmalloc_area_pages()` (See Section G.1.4)

If the insertion fails, free the region and return NULL.

Return the virtual address of the newly mapped region.
G.2 Freeing A Non-Contiguous Area

Contents

G.2 Freeing A Non-Contiguous Area 437
G.2.1 Function: vfree() 437
G.2.2 Function: vmfree_area_pages() 438
G.2.3 Function: free_area_pmd() 439
G.2.4 Function: free_area_pte() 440

G.2.1 Function: vfree() (mm/vmalloc.c)

The call graph for this function is shown in Figure 7.4. This is the top level function responsible for freeing a non-contiguous area of memory. It performs basic sanity checks before finding the vm_struct for the requested addr. Once found, it calls vmfree_area_pages().

237 void vfree(void * addr)
238 {
239     struct vm_struct **p, *tmp;
240
241     if (!addr)
242         return;
243     if (((PAGE_SIZE-1) & (unsigned long) addr) {
244         printk(KERN_ERR "Trying to vfree() bad address (%p)\n", addr);
245         return;
246     }
247     write_lock(&vmlist_lock);
248     for (p = &vmlist ; (tmp = *p) ; p = &tmp->next) {
249         if (tmp->addr == addr) {
250             *p = tmp->next;
251             vmfree_area_pages(VMALLOC_VMADDR(tmp->addr),
252                 tmp->size);
253             write_unlock(&vmlist_lock);
254             kfree(tmp);
255             return;
256         }
257     }
258     write_unlock(&vmlist_lock);
259     printk(KERN_ERR "Trying to vfree() nonexistent vm area (%p)\n", addr);
260 }

The parameter is the address returned by get_vm_area() (See Section G.1.3) to either vmalloc() or ioremap()

241-243 Ignore NULL addresses
243-246 This checks the address is page aligned and is a reasonable quick guess to see if the area is valid or not.

247 Acquire a write lock to the vmlist.

248 Cycle through the vmlist looking for the correct vm_struct for addr.

249 If this it the correct address then ... 

250 Remove this area from the vmlist linked list.

251 Free all pages associated with the address range.

252 Release the vmlist lock.

253 Free the memory used for the vm_struct and return.

257-258 The vm_struct was not found. Release the lock and print a message about the failed free.

G.2.2 Function: vmfree_area_pages() (mm/vmalloc.c)

This is the first stage of the page table walk to free all pages and PTEs associated with an address range. It is responsible for stepping through the relevant PGDs and for flushing the TLB.

80 void vmfree_area_pages(unsigned long address, unsigned long size) {
81  
82  pgd_t * dir;
83  unsigned long end = address + size;
84  
85  dir = pgd_offset_k(address);
86  flush_cache_all();
87  do {
88     free_area_pmd(dir, address, end - address);
89     address = (address + PGDIR_SIZE) & PGDIR_MASK;
90     dir++;
91  } while (address && (address < end));
92  flush_tlb_all();
93 }

80 The parameters are the starting address and the size of the region.

82 The address space end is the starting address plus its size.

85 Get the first PGD for the address range.

86 Flush the cache CPU so cache hits will not occur on pages that are to be deleted. This is a null operation on many architectures including the x86.
Call free_area_pmd() (See Section G.2.3) to perform the second stage of the page table walk.

address becomes the starting address of the next PGD.

Move to the next PGD.

Flush the TLB as the page tables have now changed.

**G.2.3 Function: free_area_pmd() (mm/vmalloc.c)**

This is the second stage of the page table walk. For every PMD in this directory, call free_area_pte() to free up the pages and PTEs.

```c
56 static inline void free_area_pmd(pgd_t * dir,
    unsigned long address,
    unsigned long size)

57 {
58    pmd_t * pmd;
59    unsigned long end;
60    if (pgd_none(*dir))
61        return;
62    if (pgd_bad(*dir)) {
63        pgd_ERROR(*dir);
64        pgd_clear(dir);
65        return;
66    }
67    pmd = pmd_offset(dir, address);
68    address &= ~PGDIR_MASK;
69    end = address + size;
70    if (end > PGDIR_SIZE)
71        end = PGDIR_SIZE;
72    do {
73        free_area_pte(pmd, address, end - address);
74        address = (address + PMD_SIZE) & PMD_MASK;
75        pmd++;
76    } while (address < end);
77 }
```

The parameters are the PGD been stepped through, the starting address and the length of the region.

If there is no PGD, return. This can occur after vfree() (See Section G.2.1) is called during a failed allocation.

A PGD can be bad if the entry is not present, it is marked read-only or it is marked accessed or dirty.
G.2 Freeing A Non-Contiguous Area (free_area_pmd())

68 Get the first PMD for the address range

69 Make the address PGD aligned

70-72 end is either the end of the space to free or the end of this PGD, whichever is first

73-77 For every PMD, call free_area_pte() (See Section G.2.4) to free the PTE entries

75 address is the base address of the next PMD

76 Move to the next PMD

G.2.4 Function: free_area_pte() (mm/vmalloc.c)

This is the final stage of the page table walk. For every PTE in the given PMD within the address range, it will free the PTE and the associated page

22 static inline void free_area_pte(pmd_t * pmd, unsigned long address, unsigned long size)

23 {
24  pte_t * pte;
25  unsigned long end;
26  
27  if (pmd_none(*pmd))
28     return;
29  if (pmd_bad(*pmd)) {
30     pmd_ERROR(*pmd);
31     pmd_clear(pmd);
32     return;
33  }
34  pte = pte_offset(pmd, address);
35  address &= ~PMD_MASK;
36  end = address + size;
37  if (end > PMD_SIZE)
38     end = PMD_SIZE;
39  do {
40    pte_t page;
41    page = ptep_get_and_clear(pte);
42    address += PAGE_SIZE;
43    pte++;
44    if (pte_none(page))
45        continue;
46    if (pte_present(page)) {
47        struct page *ptpage = pte_page(page);
48        if (VALID_PAGE(ptpage) &&
G.2 Freeing A Non-Contiguous Area (**free_area_pte()**) 441

```c
(!PageReserved(ptpage))
49    __free_page(ptpage);
50    continue;
51 }
52 printk(KERN_CRIT
    "Whee.. Swapped out page in kernel page table\n");
53 } while (address < end);
54 }
```

22 The parameters are the PMD that PTEs are been freed from, the starting address and the size of the region to free

27-28 The PMD could be absent if this region is from a failed **vmalloc()**

29-33 A PMD can be bad if it’s not in main memory, it’s read only or it’s marked dirty or accessed

34 **pte** is the first PTE in the address range

35 Align the address to the PMD

36-38 The end is either the end of the requested region or the end of the PMD, whichever occurs first

38-53 Step through all PTEs, perform checks and free the PTE with its associated page

41 **ptep_get_and_clear()** will remove a PTE from a page table and return it to the caller

42 **address** will be the base address of the next PTE

43 Move to the next PTE

44 If there was no PTE, simply continue

46-51 If the page is present, perform basic checks and then free it

47 **pte_page()** uses the global **mem_map** to find the **struct page** for the PTE

48-49 Make sure the page is a valid page and it is not reserved before calling **__free_page()** to free the physical page

50 Continue to the next PTE

52 If this line is reached, a PTE within the kernel address space was somehow swapped out. Kernel memory is not swappable and so is a critical error
Appendix H

Slab Allocator

Contents

H.1 Cache Manipulation ................................................. 444
  H.1.1 Cache Creation .............................................. 444
    H.1.1.1 Function: kmem_cache_create() ....................... 444
  H.1.2 Calculating the Number of Objects on a Slab ................. 453
    H.1.2.1 Function: kmem_cache_estimate() ....................... 453
  H.1.3 Cache Shrinking ............................................. 454
    H.1.3.1 Function: kmem_cache_shrink() ......................... 455
    H.1.3.2 Function: __kmem_cache_shrink() ....................... 455
    H.1.3.3 Function: __kmem_cache_shrink_locked() ............... 456
  H.1.4 Cache Destroying ............................................ 457
    H.1.4.1 Function: kmem_cache_destroy() ....................... 458
  H.1.5 Cache Reaping .............................................. 459
    H.1.5.1 Function: kmem_cache_reap() ......................... 459

H.2 Slabs .......................................................... 464
  H.2.1 Storing the Slab Descriptor ................................. 464
    H.2.1.1 Function: kmem_cache_slabmgmt() ...................... 464
    H.2.1.2 Function: kmem_find_general_cachep() ............... 465
  H.2.2 Slab Creation .............................................. 466
    H.2.2.1 Function: kmem_cache_grow() ......................... 466
  H.2.3 Slab Destroying ............................................ 470
    H.2.3.1 Function: kmem_slab_destroy() ....................... 470

H.3 Objects ........................................................ 472
  H.3.1 Initialising Objects in a Slab ............................. 472
    H.3.1.1 Function: kmem_cache_init objs() ..................... 472
  H.3.2 Object Allocation ......................................... 474
## APPENDIX H. SLAB ALLOCATOR

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>H.3.2.1</td>
<td>Function: <code>kmem_cache_alloc()</code></td>
<td>474</td>
</tr>
<tr>
<td>H.3.2.2</td>
<td>Function: <code>__kmem_cache_alloc (UP Case)</code></td>
<td>475</td>
</tr>
<tr>
<td>H.3.2.3</td>
<td>Function: <code>__kmem_cache_alloc (SMP Case)</code></td>
<td>476</td>
</tr>
<tr>
<td>H.3.2.4</td>
<td>Function: <code>kmem_cache_alloc_head()</code></td>
<td>477</td>
</tr>
<tr>
<td>H.3.2.5</td>
<td>Function: <code>kmem_cache_alloc_one()</code></td>
<td>478</td>
</tr>
<tr>
<td>H.3.2.6</td>
<td>Function: <code>kmem_cache_alloc_one_tail()</code></td>
<td>479</td>
</tr>
<tr>
<td>H.3.2.7</td>
<td>Function: <code>kmem_cache_alloc_batch()</code></td>
<td>480</td>
</tr>
<tr>
<td>H.3.3</td>
<td>Object Freeing</td>
<td>482</td>
</tr>
<tr>
<td>H.3.3.1</td>
<td>Function: <code>kmem_cache_free()</code></td>
<td>482</td>
</tr>
<tr>
<td>H.3.3.2</td>
<td>Function: <code>__kmem_cache_free (UP Case)</code></td>
<td>482</td>
</tr>
<tr>
<td>H.3.3.3</td>
<td>Function: <code>__kmem_cache_free (SMP Case)</code></td>
<td>483</td>
</tr>
<tr>
<td>H.3.3.4</td>
<td>Function: <code>kmem_cache_free_one()</code></td>
<td>484</td>
</tr>
<tr>
<td>H.3.3.5</td>
<td>Function: <code>free_block()</code></td>
<td>485</td>
</tr>
<tr>
<td>H.3.3.6</td>
<td>Function: <code>__free_block()</code></td>
<td>486</td>
</tr>
<tr>
<td>H.4</td>
<td>Sizes Cache</td>
<td>487</td>
</tr>
<tr>
<td>H.4.1</td>
<td>Initialising the Sizes Cache</td>
<td>487</td>
</tr>
<tr>
<td>H.4.1.1</td>
<td>Function: <code>kmem_cache_sizes_init()</code></td>
<td>487</td>
</tr>
<tr>
<td>H.4.2</td>
<td><code>kmalloc()</code></td>
<td>488</td>
</tr>
<tr>
<td>H.4.2.1</td>
<td>Function: <code>kmalloc()</code></td>
<td>488</td>
</tr>
<tr>
<td>H.4.3</td>
<td><code>kfree()</code></td>
<td>489</td>
</tr>
<tr>
<td>H.4.3.1</td>
<td>Function: <code>kfree()</code></td>
<td>489</td>
</tr>
<tr>
<td>H.5</td>
<td>Per-CPU Object Cache</td>
<td>490</td>
</tr>
<tr>
<td>H.5.1</td>
<td>Enabling Per-CPU Caches</td>
<td>490</td>
</tr>
<tr>
<td>H.5.1.1</td>
<td>Function: <code>enable_all_cpucaches()</code></td>
<td>490</td>
</tr>
<tr>
<td>H.5.1.2</td>
<td>Function: <code>enable_cpucache()</code></td>
<td>491</td>
</tr>
<tr>
<td>H.5.1.3</td>
<td>Function: <code>kmem_tune_cpucache()</code></td>
<td>492</td>
</tr>
<tr>
<td>H.5.2</td>
<td>Updating Per-CPU Information</td>
<td>495</td>
</tr>
<tr>
<td>H.5.2.1</td>
<td>Function: <code>smp_call_function_all_cpus()</code></td>
<td>495</td>
</tr>
<tr>
<td>H.5.2.2</td>
<td>Function: <code>do_cpucache_update_local()</code></td>
<td>495</td>
</tr>
<tr>
<td>H.5.3</td>
<td>Draining a Per-CPU Cache</td>
<td>496</td>
</tr>
<tr>
<td>H.5.3.1</td>
<td>Function: <code>drain_cpu_caches()</code></td>
<td>496</td>
</tr>
<tr>
<td>H.6</td>
<td>Slab Allocator Initialisation</td>
<td>498</td>
</tr>
<tr>
<td>H.6.0.2</td>
<td>Function: <code>kmem_cache_init()</code></td>
<td>498</td>
</tr>
<tr>
<td>H.7</td>
<td>Interfacing with the Buddy Allocator</td>
<td>499</td>
</tr>
<tr>
<td>H.7.0.3</td>
<td>Function: <code>kmem_getpages()</code></td>
<td>499</td>
</tr>
<tr>
<td>H.7.0.4</td>
<td>Function: <code>kmem_freepages()</code></td>
<td>499</td>
</tr>
</tbody>
</table>
### H.1 Cache Manipulation

<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>H.1 Cache Manipulation</td>
</tr>
<tr>
<td>H.1.1 Cache Creation</td>
</tr>
<tr>
<td>H.1.1.1 Function: <code>kmem_cache_create()</code></td>
</tr>
<tr>
<td>H.1.2 Calculating the Number of Objects on a Slab</td>
</tr>
<tr>
<td>H.1.2.1 Function: <code>kmem_cache_estimate()</code></td>
</tr>
<tr>
<td>H.1.3 Cache Shrinking</td>
</tr>
<tr>
<td>H.1.3.1 Function: <code>kmem_cache_shrink()</code></td>
</tr>
<tr>
<td>H.1.3.2 Function: <code>__kmem_cache_shrink()</code></td>
</tr>
<tr>
<td>H.1.3.3 Function: <code>__kmem_cache_shrink_locked()</code></td>
</tr>
<tr>
<td>H.1.4 Cache Destroying</td>
</tr>
<tr>
<td>H.1.4.1 Function: <code>kmem_cache_destroy()</code></td>
</tr>
<tr>
<td>H.1.5 Cache Reaping</td>
</tr>
<tr>
<td>H.1.5.1 Function: <code>kmem_cache_reap()</code></td>
</tr>
</tbody>
</table>

### H.1.1 Cache Creation

**H.1.1.1 Function: `kmem_cache_create()` (mm/slab.c)**

The call graph for this function is shown in 8.3. This function is responsible for the creation of a new cache and will be dealt with in chunks due to its size. The chunks roughly are:

- Perform basic sanity checks for bad usage
- Perform debugging checks if `CONFIG_SLAB_DEBUG` is set
- Allocate a `kmem_cache_t` from the `cache_cache` slab cache
- Align the object size to the word size
- Calculate how many objects will fit on a slab
- Align the slab size to the hardware cache
- Calculate colour offsets
- Initialise remaining fields in cache descriptor
- Add the new cache to the cache chain
The parameters of the function are

- **name**  The human readable name of the cache
- **size**  The size of an object
- **offset**  This is used to specify a specific alignment for objects in the cache but it usually left as 0
- **flags**  Static cache flags
- **ctor**  A constructor function to call for each object during slab creation
- **dtor**  The corresponding destructor function. It is expected the destructor function leaves an object in an initialised state

These are all serious usage bugs that prevent the cache even attempting to create

If the human readable name is greater than the maximum size for a cache name (CACHE_NAMELEN)

An interrupt handler cannot create a cache as access to interrupt-safe spinlocks and semaphores are needed

The object size must be at least a word in size. The slab allocator is not suitable for objects whose size is measured in individual bytes
The largest possible slab that can be created is \(2^{\text{MAXOBJ}_-\text{ORDER}}\) number of pages which provides 32 pages.

A destructor cannot be used if no constructor is available.

The offset cannot be before the slab or beyond the boundary of the first page.

Call BUG() to exit.

```c
#if DEBUG
   if ((flags & SLAB_DEBUG_INITIAL) && !ctor) {
      printk("%sNo con, but init state check
               requested - %s\n", func_nm, name);
      flags &= ~SLAB_DEBUG_INITIAL;
   }

   if ((flags & SLAB_POISON) && ctor) {
      printk("%sPoisoning requested, but con given - %s\n",
             func_nm, name);
      flags &= ~SLAB_POISON;
   }
#elif FORCED_DEBUG
   if ((size < (PAGE_SIZE>>3)) &&
       !(flags & SLAB_MUST_HWCACHE_ALIGN))
      flags |= SLAB_RED_ZONE;
   if (!ctor)
      flags |= SLAB_POISON;
#endif
BUG_ON(flags & ~CREATE_MASK);
```

This block performs debugging checks if CONFIG_SLAB_DEBUG is set.

The flag SLAB_DEBUG_INITIAL requests that the constructor check the objects to make sure they are in an initialised state. For this, a constructor must exist. If it does not, the flag is cleared.

A slab can be poisoned with a known pattern to make sure an object wasn’t used before it was allocated but a constructor would ruin this pattern falsely reporting a bug. If a constructor exists, remove the SLAB_POISON flag if set.

Only small objects will be red zoned for debugging. Red zoning large objects would cause severe fragmentation.

If there is no constructor, set the poison bit.
The CREATE_MASK is set with all the allowable flags `kmem_cache_create()` (See Section H.1.1.1) can be called with. This prevents callers using debugging flags when they are not available and `BUG()`s it instead.

Allocate a `kmem_cache_t` from the `cache_cache` slab cache.

Allocate a cache descriptor object from the `cache_cache` with `kmem_cache_alloc()` (See Section H.3.2.1)

If out of memory goto opps which handles the oom situation

Zero fill the object to prevent surprises with uninitialised data

Align the object size to some word-sized boundary.
If the size is not aligned to the size of a word then...

Increase the object by the size of a word then mask out the lower bits, this will effectively round the object size up to the next word boundary.

Print out an informational message for debugging purposes.

If debugging is enabled then the alignments have to change slightly.

Do not bother trying to align things to the hardware cache if the slab will be red zoned. The red zoning of the object is going to offset it by moving the object one word away from the cache boundary.

The size of the object increases by two `BYTES_PER_WORD` to store the red zone mark at either end of the object.

Initialise the alignment to be to a word boundary. This will change if the caller has requested a CPU cache alignment.

If requested, align the objects to the L1 CPU cache.

If the objects are large, store the slab descriptors off-slab. This will allow better packing of objects into the slab.

If hardware cache alignment is requested, the size of the objects must be adjusted to align themselves to the hardware cache.

Try and pack objects into one cache line if they fit while still keeping the alignment. This is important to achres (e.g. Alpha or Pentium 4) with large L1 cache bytes. `align` will be adjusted to be the smallest that will give hardware cache alignment. For machines with large L1 cache lines, two or more small objects may fit into each line. For example, two objects from the size-32 cache will fit on one cache line from a Pentium 4.

Round the cache size up to the hardware cache alignment.

```c
    do {
        unsigned int break_flag = 0;
        cal_wastage:
            kmem_cache_estimate(cachep->gfporder,
                size, flags,
                &left_over,
                &cachep->num);
        if (break_flag)
            break;
        if (cachep->gfporder >= MAX_GFP_ORDER)
            break;
        if (!cachep->num)
```
H.1.1 Cache Creation (kmem_cache_create())

Calculate how many objects will fit on a slab and adjust the slab size as necessary

727-728 kmem_cache_estimate() (see Section H.1.2.1) calculates the number of objects that can fit on a slab at the current gfp order and what the amount of leftover bytes will be

729-730 The break_flag is set if the number of objects fitting on the slab exceeds the number that can be kept when offslab slab descriptors are used

731-732 The order number of pages used must not exceed MAX_GFP_ORDER (5)

733-734 If even one object didn’t fit, goto next: which will increase the gfporder used for the cache

735 If the slab descriptor is kept off-cache but the number of objects exceeds the number that can be tracked with bufctl’s off-slab then ...

737 Reduce the order number of pages used

738 Set the break_flag so the loop will exit

739 Calculate the new wastage figures
The `slab_break_gfp_order` is the order to not exceed unless 0 objects fit on the slab. This check ensures the order is not exceeded.

This is a rough check for internal fragmentation. If the wastage as a fraction of the total size of the cache is less than one eighth, it is acceptable.

If the fragmentation is too high, increase the gfp order and recalculate the number of objects that can be stored and the wastage.

If after adjustments, objects still do not fit in the cache, it cannot be created.

Free the cache descriptor and set the pointer to NULL.

Goto `opps` which simply returns the NULL pointer.

Align the slab size to the hardware cache.

It is the size of the slab struct and the number of objects * size of the bufctl

If there is enough leftover space for the slab descriptor and it was specified to place the descriptor off-slab, remove the flag and update the amount of leftover bytes there is. This will impact the cache colouring but with the large objects associated with off-slab descriptors, this is not a problem.

Calculate colour offsets.

The offset within the page the caller requested. This will make sure the offset requested is at the correct alignment for cache usage.

If somehow the offset is 0, then set it to be aligned for the CPU cache.
This is the offset to use to keep objects on different cache lines. Each slab created will be given a different colour offset.

This is the number of different offsets that can be used.

For caches with slabs of only 1 page, the CFLGS_OPTIMIZE flag is set. In reality it makes no difference as the flag is unused.

Set the cache static flags.

Zero out the gfpflags. Defunct operation as memset() after the cache descriptor was allocated would do this.

If the slab is for DMA use, set the GFP_DMA flag so the buddy allocator will use ZONE_DMA.

Initialise the spinlock for access the cache.

Copy in the object size, which now takes hardware cache alignment if necessary.

Initialise the slab lists.
H.1.1 Cache Creation (*kmem_cache_create()* )

794-795 If the descriptor is kept off-slab, allocate a slab manager and place it for use in *slab_cache*. See Section H.2.1.2

796-797 Set the pointers to the constructor and destructor functions

799 Copy in the human readable name

802-803 If per-cpu caches are enabled, create a set for this cache. See Section 8.5

806-816 Acquire the semaphore used to synchronise access to the cache chain

810-816 Check every cache on the cache chain and make sure there is no other cache with the same name. If there is, it means two caches of the same type are been created which is a serious bug

811 Get the cache from the list

814-815 Compare the names and if they match, *BUG(). It is worth noting that the new cache is not deleted, but this error is the result of sloppy programming during development and not a normal scenario

822 Link the cache into the chain.

823 Release the cache chain semaphore.

825 Return the new cache pointer
H.1.2 Calculating the Number of Objects on a Slab

H.1.2.1 Function: kmem_cache_estimate()  (mm/slab.c)

During cache creation, it is determined how many objects can be stored in a slab and how much waste-age there will be. The following function calculates how many objects may be stored, taking into account if the slab and bufctl’s must be stored on-slab.

388 static void kmem_cache_estimate (unsigned long gfporder,
389     size_t size,
390     int flags, size_t *left_over, unsigned int *num)
391 {
392     int i;
393     size_t wastage = PAGE_SIZE<<gfporder;
394     size_t extra = 0;
395     size_t base = 0;
396     if (!(flags & CFLGS_OFF_SLAB)) {
397         base = sizeof(slab_t);
398         extra = sizeof(kmem_bufctl_t);
399     }
400     i = 0;
401     while (i*size + L1_CACHE_ALIGN(base+i*extra) <= wastage)
402         i++;
403     if (i > 0)
404         i--;
405     if (i > SLAB_LIMIT)
406         i = SLAB_LIMIT;
407     *num = i;
408     wastage -= i*size;
409     wastage -= L1_CACHE_ALIGN(base+i*extra);
410     *left_over = wastage;
411 }

388 The parameters of the function are as follows

- gfporder The $2^{gfporder}$ number of pages to allocate for each slab
- size The size of each object
- flags The cache flags
- left_over The number of bytes left over in the slab. Returned to caller
- num The number of objects that will fit in a slab. Returned to caller
H.1.3 Cache Shrinking

392 wastage is decremented through the function. It starts with the maximum possible amount of wastage.

393 extra is the number of bytes needed to store kmem_bufctl_t

394 base is where usable memory in the slab starts

396 If the slab descriptor is kept on cache, the base begins at the end of the slab_t struct and the number of bytes needed to store the bufctl is the size of kmem_bufctl_t

400 i becomes the number of objects the slab can hold

401-402 This counts up the number of objects that the cache can store. i*size is the the size of the object itself. L1_CACHE_ALIGN(base+i*extra) is slightly trickier. This is calculating the amount of memory needed to store the kmem_bufctl_t needed for every object in the slab. As it is at the beginning of the slab, it is L1 cache aligned so that the first object in the slab will be aligned to hardware cache. i*extra will calculate the amount of space needed to hold a kmem_bufctl_t for this object. As wast-age starts out as the size of the slab, its use is overloaded here.

403-404 Because the previous loop counts until the slab overflows, the number of objects that can be stored is i-1.

406-407 SLAB_LIMIT is the absolute largest number of objects a slab can store. Is is defined as 0xffffffff as this the largest number kmem_bufctl_t(), which is an unsigned integer, can hold

409 num is now the number of objects a slab can hold

410 Take away the space taken up by all the objects from wastage

411 Take away the space taken up by the kmem_bufctl_t

412 Wast-age has now been calculated as the left over space in the slab

H.1.3 Cache Shrinking

The call graph for kmem_cache_shrink() is shown in Figure 8.5. Two varieties of shrink functions are provided. kmem_cache_shrink() removes all slabs from slabs_free and returns the number of pages freed as a result. __kmem_cache_shrink() frees all slabs from slabs_free and then verifies that slabs_partial and slabs_full are empty. This is important during cache destruction when it doesn’t matter how many pages are freed, just that the cache is empty.
H.1.3.1 Function: \texttt{kmem_cache_shrink()} \quad (\texttt{mm/slab.c})

This function performs basic debugging checks and then acquires the cache descriptor lock before freeing slabs. At one time, it also used to call \texttt{drain_cpu_caches()} to free up objects on the per-cpu cache. It is curious that this was removed as it is possible slabs could not be freed due to an object been allocation on a per-cpu cache but not in use.

966 int \texttt{kmem_cache_shrink(kmem_cache_t *cachep)}
967 {
968     int ret;
969
970     if (!cachep || in_interrupt() ||
971         !is_chained_kmem_cache(cachep))
972         BUG();
973
974     spin_lock_irq(&cachep->spinlock);
975     ret = \_\texttt{kmem_cache_shrink_locked(cachep)};
976     spin_unlock_irq(&cachep->spinlock);
977     return ret << cachep->gfporder;
978 }

966 The parameter is the cache been shrunk
970 Check that

- The cache pointer is not NULL
- That an interrupt is not the caller
- That the cache is on the cache chain and not a bad pointer

973 Acquire the cache descriptor lock and disable interrupts
974 Shrink the cache
975 Release the cache lock and enable interrupts
976 This returns the number of pages freed but does not take into account the objects freed by draining the CPU.

H.1.3.2 Function: \_\texttt{kmem_cache_shrink()} \quad (\texttt{mm/slab.c})

This function is identical to \texttt{kmem_cache_shrink()} except it returns if the cache is empty or not. This is important during cache destruction when it is not important how much memory was freed, just that it is safe to delete the cache and not leak memory.
H.1.3 Cache Shrinking (__kmem_cache_shrink())

945 static int __kmem_cache_shrink(kmem_cache_t *cachep)
946 {
947    int ret;
948
949    drain_cpu_caches(cachep);
950
951    spin_lock_irq(&cachep->spinlock);
952    __kmem_cache_shrink_locked(cachep);
953    ret = !list_empty(&cachep->slabs_full) ||
954        !list_empty(&cachep->slabs_partial);
955    spin_unlock_irq(&cachep->spinlock);
956    return ret;
957 }

949 Remove all objects from the per-CPU objects cache
951 Acquire the cache descriptor lock and disable interrupts
952 Free all slabs in the slabs_free list
954-954 Check the slabs_partial and slabs_full lists are empty
955 Release the cache descriptor lock and re-enable interrupts
956 Return if the cache has all its slabs free or not

H.1.3.3 Function: __kmem_cache_shrink_locked() (mm/slab.c)
This does the dirty work of freeing slabs. It will keep destroying them until the
growing flag gets set, indicating the cache is in use or until there is no more slabs
in slabs_free.

917 static int __kmem_cache_shrink_locked(kmem_cache_t *cachep)
918 {
919    slab_t *slabp;
920    int ret = 0;
921
923    while (!cachep->growing) {
924        struct list_head *p;
925
926        p = cachep->slabs_free.prev;
927        if (p == &cachep->slabs_free)
928            break;
929
930        slabp = list_entry(cachep->slabs_free.prev,
931                                    slab_t, list);
While the cache is not growing, free slabs

Get the last slab on the \texttt{slabs\_free} list

If debugging is available, make sure it is not in use. If it is not in use, it should not be on the \texttt{slabs\_free} list in the first place

Remove the slab from the list

Re-enable interrupts. This function is called with interrupts disabled and this is to free the interrupt as quickly as possible.

Delete the slab with \texttt{kmem\_slab\_destroy()} (See Section H.2.3.1)

Record the number of slabs freed

Acquire the cache descriptor lock and disable interrupts

\section*{H.1.4 Cache Destroying}

When a module is unloaded, it is responsible for destroying any cache it has created as during module loading, it is ensured there is not two caches of the same name. Core kernel code often does not destroy its caches as their existence persists for the life of the system. The steps taken to destroy a cache are

- Delete the cache from the cache chain
- Shrink the cache to delete all slabs (see Section 8.1.8)
- Free any per CPU caches (\texttt{kfree()})
- Delete the cache descriptor from the \texttt{cache\_cache} (see Section: 8.3.3)
H.1.4.1 Function: kmem_cache_destroy() (mm/slab.c)

The call graph for this function is shown in Figure 8.7.

```c
int kmem_cache_destroy (kmem_cache_t * cachep)
{
    if (!cachep || in_interrupt() || cachep->growing)
        BUG();
    /* Find the cache in the chain of caches. */
    down(&cache_chain_sem);
    /* the chain is never empty, cache_cache is never destroyed */
    if (clock_searchp == cachep)
        clock_searchp = list_entry(cachep->next.next,
            kmem_cache_t, next);
    list_del(&cachep->next);
    up(&cache_chain_sem);
    if (__kmem_cache_shrink(cachep)) {
        printk(KERN_ERR
            "kmem_cache_destroy: Can’t free all objects %p\n",
            cachep);
        down(&cache_chain_sem);
        list_add(&cachep->next,&cache_chain);
        up(&cache_chain_sem);
        return 1;
    }
    #ifdef CONFIG_SMP
    { int i;
      for (i = 0; i < NR_CPUS; i++)
        kfree(cachep->cpudata[i]);
    }
    #endif
    kmem_cache_free(&cache_cache, cachep);
    return 0;
}
```

999-1000 Sanity check. Make sure the cachep is not null, that an interrupt is not trying to do this and that the cache has not been marked as growing, indicating it is in use.

1003 Acquire the semaphore for accessing the cache chain.

1005-1007 Acquire the list entry from the cache chain.
Delete this cache from the cache chain
Release the cache chain semaphore
Shrink the cache to free all slabs with \texttt{\_\_kmem\_cache\_shrink()} (See Section H.1.3.2)
The shrink function returns true if there is still slabs in the cache. If there is, the cache cannot be destroyed so it is added back into the cache chain and the error reported
If SMP is enabled, the per-cpu data structures are deleted with \texttt{kfree()} (See Section H.4.3.1)
Delete the cache descriptor from the cache_cache with \texttt{kmem\_cache\_free()} (See Section H.3.3.1)

\section*{H.1.5 Cache Reaping}

\subsection*{H.1.5.1 Function: \texttt{kmem\_cache\_reap()} (mm/slab.c)}

The call graph for this function is shown in Figure 8.4. Because of the size of this function, it will be broken up into three separate sections. The first is simple function preamble. The second is the selection of a cache to reap and the third is the freeing of the slabs. The basic tasks were described in Section 8.1.7.

```
int kmem_cache_reap (int gfp_mask) {
  slab_t *slabp;
  kmem_cache_t *searchp;
  kmem_cache_t *best_cachep;
  unsigned int best_pages;
  unsigned int best_len;
  unsigned int scan;
  int ret = 0;

  if (gfp_mask & \_\_GFP\_WAIT)
    down(&cache_chain_sem);
  else
    if (down\_trylock(&cache_chain_sem))
      return 0;

  scan = REAP\_SCANLEN;
  best_len = 0;
  best_pages = 0;
  best_cachep = NULL;
  searchp = clock_searchp;
```
The only parameter is the GFP flag. The only check made is against the __GFP_WAIT flag. As the only caller, kswapd, can sleep, this parameter is virtually worthless.

Can the caller sleep? If yes, then acquire the semaphore.

Else, try and acquire the semaphore and if not available, return.

REAP_SCANLEN (10) is the number of caches to examine.

Set searchp to be the last cache that was examined at the last reap:

```c
    do {
        unsigned int pages;
        struct list_head* p;
        unsigned int full_free;
        
        if (searchp->flags & SLAB_NO_REAP)
            goto next;
        spin_lock_irq(&searchp->spinlock);
        if (searchp->growing)
            goto next_unlock;
        if (searchp->dflags & DFLGS_GROWN) {
            searchp->dflags &= ~DFLGS_GROWN;
            goto next_unlock;
        }
        #ifdef CONFIG_SMP
        {
            cpucache_t *cc = cc_data(searchp);
            if (cc && cc->avail) {
                __free_block(searchp, cc_entry(cc),
                             cc->avail);
                cc->avail = 0;
            }
        }
        #endif
        full_free = 0;
        p = searchp->slabs_free.next;
        while (p != &searchp->slabs_free) {
            slabp = list_entry(p, slab_t, list);
            #if DEBUG
            if (slabp->inuse)
                BUG();
            #endif
            full_free++;
        }
    }
```
This block examines `REAP_SCANLEN` number of caches to select one to free.

1767 Acquire an interrupt safe lock to the cache descriptor.

1768-1769 If the cache is growing, skip it.

1770-1773 If the cache has grown recently, skip it and clear the flag.

1775-1781 Free any per CPU objects to the global pool.

1786-1794 Count the number of slabs in the `slabs_free` list.

1801 Calculate the number of pages all the slabs hold.

1802-1803 If the objects have constructors, reduce the page count by one fifth to make it less likely to be selected for reaping.

1804-1805 If the slabs consist of more than one page, reduce the page count by one fifth. This is because high order pages are hard to acquire.

1806 If this is the best candidate found for reaping so far, check if it is perfect for reaping.
**H.1.5 Cache Reaping (kmem_cache_reap())**

1807-1809 Record the new maximums

1808 best_len is recorded so that it is easy to know how many slabs is half of the slabs in the free list

1810 If this cache is perfect for reaping then

1811 Update clock_searchp

1812 Goto perfect where half the slabs will be freed

1816 This label is reached if it was found the cache was growing after acquiring the lock

1817 Release the cache descriptor lock

1818 Move to the next entry in the cache chain

1820 Scan while REAP_SCANLEN has not been reached and we have not cycled around the whole cache chain

1822 clock_searchp = searchp;
1823
1824 if (!best_cachep)
1826 goto out;
1827
1828 spin_lock_irq(&best_cachep->spinlock);
1829 perfect:
1830 /* free only 50% of the free slabs */
1831 best_len = (best_len + 1)/2;
1832 for (scan = 0; scan < best_len; scan++) {
1833 struct list_head *p;
1834
1835 if (best_cachep->growing)
1836 break;
1837 p = best_cachep->slabs_free.prev;
1838 if (p == &best_cachep->slabs_free)
1839 break;
1840 slabp = list_entry(p, slab_t, list);
1841 #if DEBUG
1842 if (slabp->inuse)
1843 BUG();
1844 #endif
1845 list_del(&slabp->list);
1846 STATS_INC_REAPED(best_cachep);
1847 /* Safe to drop the lock. The slab is no longer
1848 * lined to the cache.*/
H.1.5 Cache Reaping \((\text{kmem\_cache\_reap})\)

1850        */
1851    spin_unlock_irq(&best_cachep->spinlock);
1852    kmem_slab_destroy(best_cachep, slabp);
1853    spin_lock_irq(&best_cachep->spinlock);
1854 }
1855    spin_unlock_irq(&best_cachep->spinlock);
1856    ret = scan * (1 << best_cachep->gfporder);
1857 out:
1858    up(&cache_chain_sem);
1859    return ret;
1860 }

This block will free half of the slabs from the selected cache

1822    Update \text{clock\_searchp} for the next cache reap

1824-1826 If a cache was not found, goto out to free the cache chain and exit

1828 Acquire the cache chain spinlock and disable interrupts. The \text{cachep} descriptor has to be held by an interrupt safe lock as some caches may be used from interrupt context. The slab allocator has no way to differentiate between interrupt safe and unsafe caches

1831 Adjust \text{best\_len} to be the number of slabs to free

1832-1854 Free \text{best\_len} number of slabs

1835-1847 If the cache is growing, exit

1837 Get a slab from the list

1838-1839 If there is no slabs left in the list, exit

1840 Get the slab pointer

1842-1843 If debugging is enabled, make sure there is no active objects in the slab

1845 Remove the slab from the \text{slabs\_free} list

1846 Update statistics if enabled

1851 Free the cache descriptor and enable interrupts

1852 Destroy the slab. See Section 8.2.8

1851 Re-acquire the cache descriptor spinlock and disable interrupts

1855 Free the cache descriptor and enable interrupts

1856 \text{ret} is the number of pages that was freed

1858-1859 Free the cache semaphore and return the number of pages freed
H.2 Slabs

Contents

H.2 Slabs 464
H.2.1 Storing the Slab Descriptor 464
  H.2.1.1 Function: kmem_cache_slabmgmt() 464
  H.2.1.2 Function: kmem_find_general_cachep() 465
H.2.2 Slab Creation 466
  H.2.2.1 Function: kmem_cache_grow() 466
H.2.3 Slab Destroying 470
  H.2.3.1 Function: kmem_slab_destroy() 470

H.2.1 Storing the Slab Descriptor

H.2.1.1 Function: kmem_cache_slabmgmt() (mm/slab.c)

This function will either allocate space to keep the slab descriptor off cache or reserve enough space at the beginning of the slab for the descriptor and the bufctl.s.

```
static inline slab_t *kmem_cache_slabmgmt (kmem_cache_t *cachep,
    void *objp,
    int colour_off,
    int local_flags)
{
    slab_t *slabp;

    if (OFF_SLAB(cachep)) {
        slabp = kmem_cache_alloc(cachep->slabp_cache,
                                local_flags);
        if (!slabp)
            return NULL;
    } else {
        slabp = objp+colour_off;
        colour_off += L1_CACHE_ALIGN(cachep->num *
            sizeof(kmem_bufctl_t) +
            sizeof(slab_t));
    }
    slabp->inuse = 0;
    slabp->colouroff = colour_off;
    slabp->s_mem = objp+colour_off;
    return slabp;
}
```
The parameters of the function are:

- `cachep`: The cache the slab is to be allocated to.
- `objp`: When the function is called, this points to the beginning of the slab.
- `colour_off`: The colour offset for this slab.
- `local_flags`: These are the flags for the cache.

If the slab descriptor is kept off cache....

Allocate memory from the sizes cache. During cache creation, `slabp_cache` is set to the appropriate size cache to allocate from.

If the allocation failed, return.

Reserve space at the beginning of the slab.

The address of the slab will be the beginning of the slab (`objp`) plus the colour offset.

`colour_off` is calculated to be the offset where the first object will be placed. The address is L1 cache aligned. `cachep->num * sizeof(kmem_bufctl_t)` is the amount of space needed to hold the bufcts for each object in the slab and `sizeof(slab_t)` is the size of the slab descriptor. This effectively has reserved the space at the beginning of the slab.

The number of objects in use on the slab is 0.

The `colouroff` is updated for placement of the new object.

The address of the first object is calculated as the address of the beginning of the slab plus the offset.

**Function: kmem_find_general_cachep()** *(mm/slab.c)*

If the slab descriptor is to be kept off-slab, this function, called during cache creation will find the appropriate sizes cache to use and will be stored within the cache descriptor in the field `slabp_cache`.

```c
kmem_cache_t * kmem_find_general_cachep (size_t size,  
    int gfpflags)
{
    cache_sizes_t *csizep = cache_sizes;
    for ( ; csizep->cs_size; csizep++) {  
        if (size > csizep->cs_size)  
            continue;
        break;
    }
```
H.2.2 Slab Creation

1633     return (gfpflags & GFP_DMA) ? csizep->cs_dmacachep : 
1634         csizep->cs_cachep;

1620  size is the size of the slab descriptor. gfpflags is always 0 as DMA memory 
is not needed for a slab descriptor

1628-1632  Starting with the smallest size, keep increasing the size until a cache 
is found with buffers large enough to store the slab descriptor

1633  Return either a normal or DMA sized cache depending on the gfpflags 
passed in. In reality, only the cs_cachep is ever passed back

H.2.2 Slab Creation

H.2.2.1 Function: kmem_cache_grow() (mm/slab.c)
The call graph for this function is shown in 8.11. The basic tasks for this function 
are;

• Perform basic sanity checks to guard against bad usage
• Calculate colour offset for objects in this slab
• Allocate memory for slab and acquire a slab descriptor
• Link the pages used for the slab to the slab and cache descriptors
• Initialise objects in the slab
• Add the slab to the cache

1105 static int kmem_cache_grow (kmem_cache_t * cachep, int flags)
1106 {
1107     slab_t  *slabp;
1108     struct page  *page;
1109     void  *objp;
1110     size_t  offset;
1111     unsigned int  i, local_flags;
1112     unsigned long  ctor_flags;
1113     unsigned long  save_flags;

Basic declarations. The parameters of the function are

cachep  The cache to allocate a new slab to

flags  The flags for a slab creation
Perform basic sanity checks to guard against bad usage. The checks are made here rather than \texttt{kmem_cache_alloc()} to protect the speed-critical path. There is no point checking the flags every time an object needs to be allocated.

1118-1119 Make sure only allowable flags are used for allocation

1120-1121 Do not grow the cache if this is set. In reality, it is never set

1129-1130 If this called within interrupt context, make sure the \texttt{ATOMI}C flag is set so we don’t sleep when \texttt{kmem_getpages()} (See Section H.7.0.3) is called

1132 This flag tells the constructor it is to init the object

1133 The local flags are just those relevant to the page allocator

1134-1139 If the \texttt{SLAB_ATOMIC} flag is set, the constructor needs to know about it in case it wants to make new allocations

1142 \texttt{spin_lock_irqsave}(&cache->spinlock, save_flags);
1143
1145 offset = cache->colour_next;
1146 cache->colour_next++;
1147 if (cache->colour_next >= cache->colour)
1148     cache->colour_next = 0;
1149 offset *= cache->colour_off;
1150 cache->dflags |= DFLGS_GROWN;
1151
1152 cache->growing++;
1153 \texttt{spin_unlock_irqrestore}(&cache->spinlock, save_flags);

Calculate colour offset for objects in this slab

1142 Acquire an interrupt safe lock for accessing the cache descriptor
H.2.2 Slab Creation (kmem_cache_grow())

1145 Get the offset for objects in this slab

1146 Move to the next colour offset

1147-1148 If colour has been reached, there is no more offsets available, so reset
colour_next to 0

1149 colour_off is the size of each offset, so offset * colour_off will give how
many bytes to offset the objects to

1150 Mark the cache that it is growing so that kmem_cache_reap() (See Section H.1.5.1)
will ignore this cache

1152 Increase the count for callers growing this cache

1153 Free the spinlock and re-enable interrupts

1165 if (!(objp = kmem_getpages(cachep, flags)))
1166     goto failed;
1167
1169 if (!(slabp = kmem_cache_slabmgmt(cachep, 
objp, offset, 
local_flags)))
1170     goto opps1;

Allocate memory for slab and acquire a slab descriptor

1165-1166 Allocate pages from the page allocator for the slab with kmem_getpages()
(See Section H.7.0.3)

1169 Acquire a slab descriptor with kmem_cache_slabmgmt() (See Section H.2.1.1)

1173  i = 1 << cachep->gfporder;
1174  page = virt_to_page(objp);
1175  do {
1176      SET_PAGE_CACHE(page, cachep);
1177      SET_PAGE_SLAB(page, slabp);
1178      PageSetSlab(page);
1179      page++;
1180  } while (--i);

Link the pages for the slab used to the slab and cache descriptors

1173 i is the number of pages used for the slab. Each page has to be linked to the
slab and cache descriptors.

1174 objp is a pointer to the beginning of the slab. The macro virt_to_page() will give the struct page for that address
1175-1180 Link each pages list field to the slab and cache descriptors

1176 `set_page_cache()` links the page to the cache descriptor using the `page->list.next` field

1178 `set_page_slab()` links the page to the slab descriptor using the `page->list.prev` field

1178 Set the PG_slab page flag. The full set of PG_flags is listed in Table 2.1

1179 Move to the next page for this slab to be linked

1182 `kmem_cache_init_objs(cache, slab, ctor_flags);`

1182 Initialise all objects (See Section H.3.1.1)

1184 `spin_lock_irqsave(&cache->spinlock, save_flags);`
1185 `cache->growing--;`
1186
1188 `list_add_tail(&slab->list, &cache->slabs_free);`
1189 `stats_inc_grown(cache);`
1190 `cache->failures = 0;`
1191
1192 `spin_unlock_irqrestore(&cache->spinlock, save_flags);`
1193 `return 1;`

Add the slab to the cache

1184 Acquire the cache descriptor spinlock in an interrupt safe fashion

1185 Decrease the growing count

1188 Add the slab to the end of the slabs_free list

1189 If STATS is set, increase the `cache->grown` field `stats_inc_grown()`

1190 Set failures to 0. This field is never used elsewhere

1192 Unlock the spinlock in an interrupt safe fashion

1193 Return success

1194 `opps1:`
1195 `kmem_free_pages(cache, objp);`
1196 `failed:`
1197 `spin_lock_irqsave(&cache->spinlock, save_flags);`
1198 `cache->growing--;`
1199 `spin_unlock_irqrestore(&cache->spinlock, save_flags);`
1300 `return 0;`
1301 `}`
H.2.3 Slab Destroying

### H.2.3.1 Function: kmem_slab_destroy()  \((mm/slab.c)\)

The call graph for this function is shown at Figure 8.13. For reability, the debugging sections have been omitted from this function but they are almost identical to the debugging section during object allocation. See Section H.3.1.1 for how the markers and poison pattern are checked.

```c
555 static void kmem_slab_destroy (kmem_cache_t *cachep, slab_t *slabp) {
556    if (cachep->dtor) {
557       int i;
558       for (i = 0; i < cachep->num; i++) {
559          void* objp = slabp->s_mem+cachep->objsize*i;
560
561          if (cachep->dtor)
562             (cachep->dtor)(objp, cachep, 0);
563
564          DEBUG: Check red zone markers
565
566          kmem_freepages(cachep, slabp->s_mem-slabp->colouroff);
567          if (OFF_SLAB(cachep))
568             kmem_cache_free(cachep->slabp_cache, slabp);
569       }
570    }
571 }
```

557-586 If a destructor is available, call it for each object in the slab

563-585 Cycle through each object in the slab
564 Calculate the address of the object to destroy

575-576 Call the destructor

588 Free the pages been used for the slab

589 If the slab descriptor is been kept off-slab, then free the memory been used for it
H.3 Objects

Contents

H.3 Objects 472
H.3.1 Initialising Objects in a Slab 472
  H.3.1.1 Function: kmem_cache_init_objs() 472
H.3.2 Object Allocation 474
  H.3.2.1 Function: kmem_cache_alloc() 474
  H.3.2.2 Function: __kmem_cache_alloc (UP Case)() 475
  H.3.2.3 Function: __kmem_cache_alloc (SMP Case)() 476
  H.3.2.4 Function: kmem_cache_alloc_head() 477
  H.3.2.5 Function: kmem_cache_alloc_one() 478
  H.3.2.6 Function: kmem_cache_alloc_one_tail() 479
  H.3.2.7 Function: kmem_cache_alloc_batch() 480
H.3.3 Object Freeing 482
  H.3.3.1 Function: kmem_cache_free() 482
  H.3.3.2 Function: __kmem_cache_free (UP Case)() 482
  H.3.3.3 Function: __kmem_cache_free (SMP Case)() 483
  H.3.3.4 Function: kmem_cache_free_one() 484
  H.3.3.5 Function: free_block() 485
  H.3.3.6 Function: __free_block() 486

This section will cover how objects are managed. At this point, most of the real hard work has been completed by either the cache or slab managers.

H.3.1 Initialising Objects in a Slab

H.3.1.1 Function: kmem_cache_init_objs() (mm/slab.c)

The vast part of this function is involved with debugging so we will start with the function without the debugging and explain that in detail before handling the debugging part. The two sections that are debugging are marked in the code excerpt below as Part 1 and Part 2.

```
static inline void kmem_cache_init_objs (kmem_cache_t * cachep, slab_t * slabp, unsigned long ctor_flags)
{
    int i;
    for (i = 0; i < cachep->num; i++) {
        void* objp = slabp->s_mem+cachep->objsize*i;

        /* Debugging Part 1 */

        if (cachep->ctor)
            cachep->ctor(objp, cachep, ctor_flags);
```

The parameters of the function are

- `cachep` The cache the objects are been initialised for
- `slabp` The slab the objects are in
- `ctor_flags` Flags the constructor needs whether this is an atomic allocation or not

Initialise cache→num number of objects

The base address for objects in the slab is `s_mem`. The address of the object to allocate is then `i * (size of a single object)`

If a constructor is available, call it

The macro `slab_bufctl()` casts `slabp` to a `slab_t` slab descriptor and adds one to it. This brings the pointer to the end of the slab descriptor and then casts it back to a `kmem_bufctl_t` effectively giving the beginning of the bufctl array.

The index of the first free object is 0 in the bufctl array

That covers the core of initialising objects. Next the first debugging part will be covered

If the cache is to be red zones then place a marker at either end of the object

Place the marker at the beginning of the object

Place the marker at the end of the object. Remember that the size of the object takes into account the size of the red markers when red zoning is enabled
Increase the objp pointer by the size of the marker for the benefit of the constructor which is called after this debugging block.

The objp pointer was increased by the size of the red marker in the previous debugging block so move it back again.

If there was no constructor, poison the object with a known pattern that can be examined later to trap uninitialised writes.

Check to make sure the red marker at the beginning of the object was preserved to trap writes before the object.

Check to make sure writes didn’t take place past the end of the object.

H.3.2 Object Allocation

H.3.2.1 Function: kmem_cache_alloc()  (mm/slab.c)

The call graph for this function is shown in Figure 8.14. This trivial function simply calls __kmem_cache_alloc().

```c
void * kmem_cache_alloc (kmem_cache_t *cachep, int flags)  
{
    return __kmem_cache_alloc(cachep, flags);
}
```
H.3.2.2 Function: \texttt{\_\_kmem\_cache\_alloc (UP Case)()}  \hfill 475

This will take the parts of the function specific to the UP case. The SMP case will be dealt with in the next section.

1338 static inline void* \texttt{\_\_kmem\_cache\_alloc (kmem\_cache\_t *cachep, int flags)}

1339 {  
1340 unsigned long save\_flags;  
1341 void* objp;  
1342  
1343 kmem\_cache\_alloc\_head(cachep, flags);  
1344 try\_again:  
1345 local\_irq\_save(save\_flags);  
1346 objp = kmem\_cache\_alloc\_one(cachep);  
1347 local\_irq\_restore(save\_flags);  
1348 return objp;  
1349 alloc\_new\_slab:  
1350 local\_irq\_restore(save\_flags);  
1351 if (kmem\_cache\_grow(cachep, flags))  
1352 goto try\_again;  
1353 return NULL;  
1354 }

1338 The parameters are the cache to allocate from and allocation specific flags

1343 This function makes sure the appropriate combination of DMA flags are in use

1345 Disable interrupts and save the flags. This function is used by interrupts so this is the only way to provide synchronisation in the UP case

1367 \texttt{kmem\_cache\_alloc\_one()} (see Section H.3.2.5) allocates an object from one of the lists and returns it. If no objects are free, this macro (note it isn’t a function) will goto alloc\_new\_slab at the end of this function

1369-1370 Restore interrupts and return

1376 At this label, no objects were free in \texttt{slabs\_partial} and \texttt{slabs\_free} is empty so a new slab is needed

1377 Allocate a new slab (see Section 8.2.2)

1379 A new slab is available so try again

1382 No slabs could be allocated so return failure
H.3.2.3 Function: `__kmem_cache_alloc (SMP Case)()` (mm/slab.c)

This is what the function looks like in the SMP case

```c
1338 static inline void * __kmem_cache_alloc (kmem_cache_t *cachep,
                                          int flags)
1339 {
1340   unsigned long save_flags;
1341   void * objp;
1342   kmem_cache_alloc_head(cachep, flags);
1344 try_again:
1345   local_irq_save(save_flags);
1346   {
1347     cpucache_t *cc = cc_data(cachep);
1349   if (cc) {
1351     if (cc->avail) {
1352       STATS_INC_ALLOCHIT(cachep);
1353       objp = cc_entry(cc)[--cc->avail];
1355     } else {
1356       STATS_INC_ALLOCMISS(cachep);
1357       objp =
1358         kmem_cache_alloc_batch(cachep,cc,flags);
1359       if (!objp)
1360         goto alloc_new_slab_nolock;
1360   } else {
1361     spin_lock(&cachep->spinlock);
1362     objp = kmem_cache_alloc_one(cachep);
1363     spin_unlock(&cachep->spinlock);
1364   }
1366   local_irq_restore(save_flags);
1370   return objp;
1371 alloc_new_slab:
1373   spin_unlock(&cachep->spinlock);
1374 alloc_new_slab_nolock:
1375   local_irq_restore(save_flags);
1377   if (kmem_cache_grow(cachep, flags))
1381     goto try_again;
1382   return NULL;
1383 }  
```

1338-1347 Same as UP case

1349 Obtain the per CPU data for this cpu
H.3.2 Object Allocation (\_\_kmem_cache_alloc (SMP Case)())

1350-1360 If a per CPU cache is available then ....
1351 If there is an object available then ....
1352 Update statistics for this cache if enabled
1353 Get an object and update the avail figure
1354 Else an object is not available so ....
1355 Update statistics for this cache if enabled
1356 Allocate batchcount number of objects, place all but one of them in the per CPU cache and return the last one to objp
1357-1358 The allocation failed, so goto alloc_new_slab_nolock to grow the cache and allocate a new slab
1360-1364 If a per CPU cache is not available, take out the cache spinlock and allocate one object in the same way the UP case does. This is the case during the initialisation for the cache_cache for example
1363 Object was successfully assigned, release cache spinlock
1366-1370 Re-enable interrupts and return the allocated object
1371-1372 If kmem_cache_alloc_one() failed to allocate an object, it will goto here with the spinlock still held so it must be released
1375-1383 Same as the UP case

H.3.2.4 Function: kmem_cache_alloc_head() (mm/slab.c)

This simple function ensures the right combination of slab and GFP flags are used for allocation from a slab. If a cache is for DMA use, this function will make sure the caller does not accidently request normal memory and vice versa

1231 static inline void kmem_cache_alloc_head(kmem_cache_t *cachep, int flags)
1232 {
1233     if (flags & SLAB_DMA) {
1234         if (!(cachep->gfpflags & GFP_DMA))
1235             BUG();
1236     } else {
1237         if (cachep->gfpflags & GFP_DMA)
1238             BUG();
1239     }
1240 }
The parameters are the cache we are allocating from and the flags requested for the allocation.

If the caller has requested memory for DMA use and ....

The cache is not using DMA memory then \texttt{BUG()}

Else if the caller has not requested DMA memory and this cache is for DMA use, \texttt{BUG()}

\textbf{H.3.2.5 Function: kmem_cache_alloc_one()} \textit{(mm/slab.c)}

This is a preprocessor macro. It may seem strange to not make this an inline function but it is a preprocessor macro for a goto optimisation in \texttt{__kmem_cache_alloc()} (see Section H.3.2.2)

\begin{verbatim}
#define kmem_cache_alloc_one(cachep) ({
  struct list_head * slabs_partial, * entry;
  slab_t *slabp;

  slabs_partial = &(cachep)->slabs_partial;
  entry = slabs_partial->next;
  if (unlikely(entry == slabs_partial)) {
    struct list_head * slabs_free;
    slabs_free = &(cachep)->slabs_free;
    entry = slabs_free->next;
    if (unlikely(entry == slabs_free))
      goto alloc_new_slab;

    list_del(entry);
    list_add(entry, slabs_partial);
  }

  slabp = list_entry(entry, slab_t, list);
  kmem_cache_alloc_one_tail(cachep, slabp);
})
\end{verbatim}

\texttt{1288-1289} Get the first slab from the \texttt{slabs_partial} list

\texttt{1290-1298} If a slab is not available from this list, execute this block

\texttt{1291-1293} Get the first slab from the \texttt{slabs_free} list

\texttt{1294-1295} If there is no slabs on \texttt{slabs_free}, then \texttt{goto alloc_new_slab()}. This goto label is in \texttt{__kmem_cache_alloc()} and it is will grow the cache by one slab

\texttt{1296-1297} Else remove the slab from the free list and place it on the \texttt{slabs_partial} list because an object is about to be removed from it
Object Allocation (*kmem_cache_alloc_one()*))

1300 Obtain the slab from the list
1301 Allocate one object from the slab

H.3.2.6 Function: *kmem_cache_alloc_one_tail()* (*mm/slab.c*)

This function is responsible for the allocation of one object from a slab. Much of it is debugging code.

1242 static inline void * kmem_cache_alloc_one_tail (  
  kmem_cache_t *cachep,  
  slab_t *slabp)  
1244 {  
  void *objp;  
1246 
1247 STATS_INC_ALLOCED(cachep);  
1248 STATS_INC_ACTIVE(cachep);  
1249 STATS_SET_HIGH(cachep);  
1250 
1252 slabp->inuse++;  
1253 objp = slabp->s_mem + slabp->free*cachep->objsize;  
1254 slabp->free=slab_bufctl(slabp)[slabp->free];  
1255 
1256 if (unlikely(slabp->free == BUFCTL_END)) {  
    list_del(&slabp->list);  
1258    list_add(&slabp->list, &cachep->slabs_full);  
1259  }  
1260 #if DEBUG  
1261   if (cachep->flags & SLAB_POISON)  
1263     if (kmem_check_poison_obj(cachep, objp))  
1265       BUG();  
1269   }  
1264   if (cachep->flags & SLAB_RED_ZONE) {  
1266     if (xchg((unsigned long *)objp, RED_MAGIC2) !=  
1267       RED_MAGIC1)  
1268       BUG();  
1269     if (xchg((unsigned long *)(objp+cachep->objsize -  
1270         BYTES_PER_WORD), RED_MAGIC2) != RED_MAGIC1)  
1271       BUG();  
1272     objp += BYTES_PER_WORD;  
1273 }  
1274 #endif  
1275 
1276 return objp;  
1276 }

1230 The parameters are the cache and slab been allocated from
1247-1249 If stats are enabled, this will set three statistics. **ALLOCED** is the total number of objects that have been allocated. **ACTIVE** is the number of active objects in the cache. **HIGH** is the maximum number of objects that were active as a single time.

1252 **inuse** is the number of objects active on this slab.

1253 Get a pointer to a free object. **s_mem** is a pointer to the first object on the slab. **free** is an index of a free object in the slab. **index * object size** gives an offset within the slab.

1254 This updates the free pointer to be an index of the next free object.

1256-1259 If the slab is full, remove it from the **slabs_partial** list and place it on the **slabs_full**.

1260-1274 Debugging code

1275 Without debugging, the object is returned to the caller.

1261-1263 If the object was poisoned with a known pattern, check it to guard against uninitialised access.

1266-1267 If red zoning was enabled, check the marker at the beginning of the object and confirm it is safe. Change the red marker to check for writes before the object later.

1269-1271 Check the marker at the end of the object and change it to check for writes after the object later.

1272 Update the object pointer to point to after the red marker.

1275 Return the object.

**H.3.2.7 Function:** **kmem_cache_alloc_batch()** *(mm/slab.c)*

This function allocate a batch of objects to a CPU cache of objects. It is only used in the SMP case. In many ways it is very similar **kmem_cache_alloc_one()** *(See Section H.3.2.5).*

1305 void* kmem_cache_alloc_batch(kmem_cache_t* cachep, cpucache_t* cc, int flags) {
1306 int batchcount = cachep->batchcount;
1308 spin_lock(&cachep->spinlock);
1310 while (batchcount--) {
1311 struct list_head * slabs_partial, * entry;
1312 slab_t *slabp;
1313 /* Get slab alloc is to come from. */
H.3.2 Object Allocation (\texttt{kmem_cache_alloc_batch()})

```
1314 slabs_partial = &(cachep)->slabs_partial;
1315 entry = slabs_partial->next;
1316 if (unlikely(entry == slabs_partial)) {
    struct list_head * slabs_free;
    slabs_free = &(cachep)->slabs_free;
    entry = slabs_free->next;
    if (unlikely(entry == slabs_free))
        break;
    list_del(entry);
    list_add(entry, slabs_partial);
}
1325 slabp = list_entry(entry, slab_t, list);
1326 cc_entry(cc)[cc->avail++] =
1327     kmem_cache_alloc_one_tail(cachep, slabp);
1328 }
1330 spin_unlock(&cachep->spinlock);
1331 if (cc->avail)
1332     return cc_entry(cc)[--cc->avail];
1334 return NULL;
```

1305 The parameters are the cache to allocate from, the per CPU cache to fill and allocation flags

1307 \texttt{batchcount} is the number of objects to allocate

1309 Obtain the spinlock for access to the cache descriptor

1310-1329 Loop \texttt{batchcount} times

1311-1324 This is example the same as \texttt{kmem_cache_alloc_one()}(See Section H.3.2.5). It selects a slab from either \texttt{slabs_partial} or \texttt{slabs_free} to allocate from. If none are available, break out of the loop

1326-1327 Call \texttt{kmem_cache_alloc_one_tail()} (See Section H.3.2.6) and place it in the per CPU cache

1330 Release the cache descriptor lock

1332-1333 Take one of the objects allocated in this batch and return it

1334 If no object was allocated, return. \texttt{__kmem_cache_alloc()} (See Section H.3.2.2) will grow the cache by one slab and try again
H.3.3 Object Freeing

H.3.3.1 Function: \texttt{kmem\_cache\_free()} \textit{(mm/slab.c)}

The call graph for this function is shown in Figure 8.15.

\begin{verbatim}
void kmem_cache_free (kmem_cache_t *cachep, void *objp)
{
    unsigned long flags;
    #if DEBUG
    CHECK_PAGE(virt_to_page(objp));
    if (cachep != GET_PAGE_CACHE(virt_to_page(objp)))
        BUG();
    #endif
    local_irq_save(flags);
    __kmem_cache_free(cachep, objp);
    local_irq_restore(flags);
}
\end{verbatim}

The parameter is the cache the object is being freed from and the object itself.

1579-1583 If debugging is enabled, the page will first be checked with \texttt{CHECK\_PAGE()} to make sure it is a slab page. Secondly the page list will be examined to make sure it belongs to this cache (See Figure 8.8)

1585 Interrupts are disabled to protect the path

1586 \texttt{__kmem\_cache\_free()} (See Section H.3.3.2) will free the object to the per-CPU cache for the SMP case and to the global pool in the normal case

1587 Re-enable interrupts

H.3.3.2 Function: \texttt{__kmem\_cache\_free\ (UP Case)()} \textit{(mm/slab.c)}

This covers what the function looks like in the UP case. Clearly, it simply releases the object to the slab.

\begin{verbatim}
static inline void __kmem_cache_free (kmem_cache_t *cachep,
    void* objp)
{
    kmem_cache_free_one(cachep, objp);
}
\end{verbatim}
H.3.3.3 Function: __kmem_cache_free (SMP Case)() (mm/slab.c)

This case is slightly more interesting. In this case, the object is released to the per-cpu cache if it is available.

1493 static inline void __kmem_cache_free (kmem_cache_t *cachep, void* objp)  
1494 {  
1496   cpucache_t *cc = cc_data(cachep);  
1497   CHECK_PAGE(virt_to_page(objp));  
1498   if (cc) {  
1499     int batchcount;  
1501     if (cc->avail < cc->limit) {  
1502       STATS_INC_FREEHIT(cachep);  
1503       cc_entry(cc)[cc->avail++] = objp;  
1504       return;  
1505     }  
1506     STATS_INC_FREEMISS(cachep);  
1507     batchcount = cachep->batchcount;  
1508     cc->avail -= batchcount;  
1509     free_block(cachep,  
1510       &cc_entry(cc)[cc->avail],batchcount);  
1511     cc_entry(cc)[cc->avail++] = objp;  
1512     return;  
1513   } else {  
1514     free_block(cachep, &objp, 1);  
1515   }  
1519 }

1496 Get the data for this per CPU cache (See Section 8.5.1)
1498 Make sure the page is a slab page
1499-1513 If a per-CPU cache is available, try to use it. This is not always available. During cache destruction for instance, the per CPU caches are already gone
1501-1505 If the number of available in the per CPU cache is below limit, then add the object to the free list and return
1506 Update statistics if enabled
1507 The pool has overflowed so batchcount number of objects is going to be freed to the global pool
1508 Update the number of available (avail) objects
1509-1510 Free a block of objects to the global cache
Free the requested object and place it on the per CPU pool.

If the per-CPU cache is not available, then free this object to the global pool.

**Function: kmem_cache_free_one()** *(mm/slab.c)*

```c
static inline void kmem_cache_free_one(kmem_cache_t *cachep,
                                        void *objp)
{
    slab_t* slabp;

    CHECK_PAGE(virt_to_page(objp));
    slabp = GET_PAGE_SLAB(virt_to_page(objp));

    if (cachep->flags & SLAB_DEBUG_INITIAL)
        cachep->ctor(objp, cachep,
                      SLAB_CTOR_CONSTRUCTOR|SLAB_CTOR_VERIFY);

    if (cachep->flags & SLAB_RED_ZONE) {
        objp -= BYTES_PER_WORD;
        if (xchg((unsigned long *)objp, RED_MAGIC1) != RED_MAGIC2)
            BUG();
        if (xchg((unsigned long *)(objp+cachep->objsize-
                          BYTES_PER_WORD), RED_MAGIC1) != RED_MAGIC2)
            BUG();
    }
    if (cachep->flags & SLAB_POISON)
        kmem_poison_obj(cachep, objp);
    if (kmem_extra_free_checks(cachep, slabp, objp))
        return;
    {
        unsigned int objnr = (objp-slabp->s_mem)/cachep->objsize;
        slab_bufctl(slabp)[objnr] = slabp->free;
        slabp->free = objnr;
    }
    STATS_DEC_ACTIVE(cachep);

    {
        int inuse = slabp->inuse;
        if (unlikely(!--slabp->inuse)) {
            ...
        }
    }
```

### H.3.3 Object Freeing (kmem_cache_free_one())

```c
/* Was partial or full, now empty. */
list_del(&slabp->list);
list_add(&slabp->list, &cachep->slabs_free);
} else if (unlikely(inuse == cachep->num)) {
    /* Was full. */
    list_del(&slabp->list);
    list_add(&slabp->list, &cachep->slabs_partial);
}
```

1418 Make sure the page is a slab page

1425 Get the slab descriptor for the page

1427-1449 Debugging material. Discussed at end of section

1451 Calculate the index for the object been freed

1454 As this object is now free, update the bufctl to reflect that

1456 If statistics are enabled, disable the number of active objects in the slab

1461-1464 If inuse reaches 0, the slab is free and is moved to the slabs_free list

1465-1468 If the number in use equals the number of objects in a slab, it is full so move it to the slabs_full list

1471 End of function

1428-1433 If SLAB_DEBUG_INITIAL is set, the constructor is called to verify the object is in an initialised state

1435-1444 Verify the red marks at either end of the object are still there. This will check for writes beyond the boundaries of the object and for double frees

1445-1446 Poison the freed object with a known pattern

1447-1448 This function will confirm the object is a part of this slab and cache. It will then check the free list (bufctl1) to make sure this is not a double free

### H.3.3.5 Function: free_block() (mm/slab.c)

This function is only used in the SMP case when the per CPU cache gets too full. It is used to free a batch of objects in bulk

```c
static void free_block (kmem_cache_t* cachep, void** objpp, int len)
```
H.3.3 Object Freeing (*free_block()*)

```c
__free_block(cachep, objpp, len);
spin_unlock(&cachep->spinlock);
}
```

1481 The parameters are:

- **cachep** The cache that objects are been freed from
- **objpp** Pointer to the first object to free
- **len** The number of objects to free

1483 Acquire a lock to the cache descriptor

1486 *__free_block*() (See Section H.3.3.6) performs the actual task of freeing up each of the pages

1487 Release the lock

H.3.3.6 Function: *__free_block*() (*mm/slab.c*)

This function is responsible for freeing each of the objects in the per-CPU array `objpp`.

```c
static inline void __free_block (kmem_cache_t* cachep,
                               void** objpp, int len)
{
    for ( ; len > 0; len--, objpp++)
        kmem_cache_free_one(cachep, *objpp);
}
```

1474 The parameters are the `cachep` the objects belong to, the list of objects(`objpp`) and the number of objects to free (`len`)

1477 Loop `len` number of times

1478 Free an object from the array
H.4 Sizes Cache

Contents

H.4 Sizes Cache 487
H.4.1 Initialising the Sizes Cache 487
  H.4.1.1 Function: kmem_cache_sizes_init() 487
H.4.2 kmalloc() 488
  H.4.2.1 Function: kmalloc() 488
H.4.3 kfree() 489
  H.4.3.1 Function: kfree() 489

H.4.1 Initialising the Sizes Cache

H.4.1.1 Function: kmem_cache_sizes_init()  (mm/slab.c)

This function is responsible for creating pairs of caches for small memory buffers suitable for either normal or DMA memory.

436 void __init kmem_cache_sizes_init(void)
437 {
438     cache_sizes_t *sizes = cache_sizes;
439     char name[20];
440
444     if (num_physpages > (32 << 20) >> PAGE_SHIFT)
445         slab_break_gfp_order = BREAK_GFP_ORDER_HI;
446     do {
452         snprintf(name, sizeof(name), "size-%Zd",
                      sizes->cs_size);
453         if (!OFF_SLAB(sizes->cs_cachep)) {
454             kmem_cache_create(name, sizes->cs_size,
455                 0, SLAB_HWCACHE_ALIGN, NULL, NULL));
456                 BUG();
457         }
458         break;
459     }
460     if (!OFF_SLAB(sizes->cs_cachep)) {
461         offfslab_limit = sizes->cs_size-sizeof(slab_t);
462         offfslab_limit /= 2;
463     }
464     snprintf(name, sizeof(name), "size-%Zd(DMA)",
                      sizes->cs_size);
465     sizes->cs_dmacachep = kmem_cache_create(name,
466             sizes->cs_size, 0,
467                 SLAB_CACHE_DMA|SLAB_HWCACHE_ALIGN,
468                 NULL, NULL);
469     if (!sizes->cs_dmacachep)
470         BUG();
Get a pointer to the cache_sizes array

The human readable name of the cache. Should be sized CACHE_NAMELEN which is defined to be 20 bytes long

slab_break_gfp_order determines how many pages a slab may use unless 0 objects fit into the slab. It is statically initialised to BREAK_GFP_ORDER_LO (1). This check sees if more than 32MiB of memory is available and if it is, allow BREAK_GFP_ORDER_HI number of pages to be used because internal fragmentation is more acceptable when more memory is available.

Create two caches for each size of memory allocation needed

Store the human readable cache name in name

Create the cache, aligned to the L1 cache

Calculate the off-slab buffer limit which determines the number of objects that can be stored in a cache when the slab descriptor is kept off-cache.

The human readable name for the cache for DMA use

Create the cache, aligned to the L1 cache and suitable for DMA user

if the cache failed to allocate, it is a bug. If memory is unavailable this early, the machine will not boot

Move to the next element in the cache_sizes array

The array is terminated with a 0 as the last element

H.4.2 kmalloc()

H.4.2.1 Function: kmalloc() (mm/slab.c)

This call graph for this function is shown in Figure 8.16.

```c
void * kmalloc (size_t size, int flags)
{
    cache_sizes_t *cszep = cache_sizes;

    for (; cszep->cs_size; cszep++) {
        if (size > cszep->cs_size)
            continue;
        return __kmem_cache_alloc(flags & GFP_DMA ?
                                  cszep->cs_dmacachep :
                                  cszep->cs_cacheep);
    }
    return NULL;
}
```
H.4.3 kfree()

\[\text{csizep} \rightarrow \text{cs_cachep, flags);}\]

1563 \}
1564 \}
1565 \}
1566 \}

1557 cache_sizes is the array of caches for each size (See Section 8.4)

1559-1564 Starting with the smallest cache, examine the size of each cache until

one large enough to satisfy the request is found

1562 If the allocation is for use with DMA, allocate an object from cs_dmacachep

else use the cs_cachep

1565 If a sizes cache of sufficient size was not available or an object could not be

allocated, return failure

H.4.3 kfree()

H.4.3.1 Function: kfree() (mm/slab.c)

The call graph for this function is shown in Figure 8.17. It is worth noting that

the work this function does is almost identical to the function kmem_cache_free() with

debugging enabled (See Section H.3.3.1).

1597 void kfree (const void *objp)
1598 {\n1599 \hspace{1em} kmem_cache_t *c;
1600 \hspace{1em} unsigned long flags;
1601 1602 \hspace{1em} if (!objp)
1603 \hspace{1em} \hspace{1em} return;
1604 \hspace{1em} local_irq_save(flags);
1605 \hspace{1em} CHECK_PAGE(virt_to_page(objp));
1606 \hspace{1em} c = GET_PAGE_CACHE(virt_to_page(objp));
1607 \hspace{1em} _kmem_cache_free(c, (void*)objp);
1608 \hspace{1em} local_irq_restore(flags);
1609 1610 \}
1612 \}
1613 \}
1614 \}

1602 Return if the pointer is NULL. This is possible if a caller used kmalloc() and had a
catch-all failure routine which called kfree() immediately

1604 Disable interrupts

1605 Make sure the page this object is in is a slab page

1606 Get the cache this pointer belongs to (See Section 8.2)

1607 Free the memory object

1608 Re-enable interrupts
H.5 Per-CPU Object Cache

Contents

H.5 Per-CPU Object Cache 490
H.5.1 Enabling Per-CPU Caches 490
  H.5.1.1 Function: enable_all_cpucaches() 490
  H.5.1.2 Function: enable_cpucache() 490
  H.5.1.3 Function: kmem_tune_cpucache() 491
H.5.2 Updating Per-CPU Information 495
  H.5.2.1 Function: smp_call_function_all_cpus() 495
  H.5.2.2 Function: do_ccupdate_local() 495
H.5.3 Draining a Per-CPU Cache 496
  H.5.3.1 Function: drain_cpu_caches() 496

The structure of the Per-CPU object cache and how objects are added or removed from them is covered in detail in Sections 8.5.1 and 8.5.2.

H.5.1 Enabling Per-CPU Caches

H.5.1.1 Function: enable_all_cpucaches() (mm/slab.c)

This function locks the cache chain and enables the cpucache for every cache. This is important after the cache_cache and sizes_cache have been enabled.

1714 static void enable_all_cpucaches (void)

Figure H.1: Call Graph: enable_all_cpucaches()
H.5.1 Enabling Per-CPU Caches (enable_all_cpucaches())

```c
1715 {
1716     struct list_head* p;
1717     down(&cache_chain_sem);
1718     p = &cache_cache.next;
1719     do {
1720         kmem_cache_t* cachep = list_entry(p, kmem_cache_t, next);
1721         enable_cpucache(cachep);
1722         p = cachep->next.next;
1723     } while (p != &cache_cache.next);
1724     up(&cache_chain_sem);
1725 }
```

1718 Obtain the semaphore to the cache chain

1719 Get the first cache on the chain

1721-1726 Cycle through the whole chain

1722 Get a cache from the chain. This code will skip the first cache on the chain but cache_cache doesn’t need a cpucache as it is so rarely used

1724 Enable the cpucache

1725 Move to the next cache on the chain

1726 Release the cache chain semaphore

H.5.1.2 Function: enable_cpucache() (mm/slab.c)

This function calculates what the size of a cpucache should be based on the size of the objects the cache contains before calling kmem_tune_cpucache() which does the actual allocation.

```c
1693 static void enable_cpucache (kmem_cache_t *cachep)
1694 {
1695     int err;
1696     int limit;
1697     if (cachep->objsize > PAGE_SIZE)
1698         return;
1699     if (cachep->objsize > 1024)
1700         limit = 60;
1701     else if (cachep->objsize > 256)
1702         limit = 124;
```
enabling per-cpu caches

```c
else
    limit = 252;
err = kmem_tune_cpucache(cachep, limit, limit/2);
if (err)
    printk(KERN_ERR
        "enable_cpucache failed for %s, error %d.\n",
        cachep->name, -err);
```

1699-1700 If an object is larger than a page, do not create a per-CPU cache as they are too expensive.

1701-1702 If an object is larger than 1KiB, keep the cpu cache below 3MiB in size. The limit is set to 124 objects to take the size of the cpucache descriptors into account.

1703-1704 For smaller objects, just make sure the cache doesn’t go above 3MiB in size.

1708 Allocate the memory for the cpucache.

1710-1711 Print out an error message if the allocation failed.

H.5.1.3 Function: kmem_tune_cpucache() (mm/slab.c)

This function is responsible for allocating memory for the cpucaches. For each CPU on the system, kmalloc gives a block of memory large enough for one cpu cache and fills a ccupdate_struct_t struct. The function smp_call_function_all_cpus() then calls do_ccupdate_local() which swaps the new information with the old information in the cache descriptor.

```c
static int kmem_tune_cpucache (kmem_cache_t* cachep,
    int limit, int batchcount)
{
    ccupdate_struct_t new;
    int i;
    /*
     * These are admin-provided, so we are more graceful.
     */
    if (limit < 0)
        return -EINVAL;
    if (batchcount < 0)
        return -EINVAL;
    if (batchcount > limit)
        return -EINVAL;
```
H.5.1 Enabling Per-CPU Caches (kmem_tune_cpucache())

if (limit != 0 & & !batchcount)
    return -EINVAL;

memset(&new.new,0,sizeof(new.new));

if (limit) {
    for (i = 0; i < smp_num_cpus; i++) {
        cpucache_t* ccnew;
        ccnew = kmalloc(sizeof(void*)*limit+sizeof(cpucache_t), GFP_KERNEL);
        if (!ccnew)
            goto oom;
        ccnew->limit = limit;
        ccnew->avail = 0;
        new.new[cpu_logical_map(i)] = ccnew;
    }
    new.cachep = cachep;
    spin_lock_irq(&cachep->spinlock);
    cachep->batchcount = batchcount;
    spin_unlock_irq(&cachep->spinlock);
    smp_call_function_all_cpus(do_ccupdate_local, (void *)&new);
    for (i = 0; i < smp_num_cpus; i++) {
        cpucache_t* ccold = new.new[cpu_logical_map(i)];
        if (!ccold)
            continue;
        local_irq_disable();
        free_block(cachep, cc_entry(ccold), ccold->avail);
        local_irq_enable();
        kfree(ccold);
    }
    return 0;
}

oom:
    for (i--; i >= 0; i--)
        kfree(new.new[cpu_logical_map(i)]);
    return -ENOMEM;

The parameters of the function are:

- cachep: The cache this cpucache is been allocated for
- limit: The total number of objects that can exist in the cpucache
batchcount  The number of objects to allocate in one batch when the cpucache is empty

1647  The number of objects in the cache cannot be negative
1649  A negative number of objects cannot be allocated in batch
1651  A batch of objects greater than the limit cannot be allocated
1653  A batchcount must be provided if the limit is positive
1656  Zero fill the update struct
1657  If a limit is provided, allocate memory for the cpucache
1658-1668 For every CPU, allocate a cpucache
1661  The amount of memory needed is limit number of pointers and the size of the cpucache descriptor
1663  If out of memory, clean up and exit
1665-1666 Fill in the fields for the cpucache descriptor
1667  Fill in the information for ccupdate_update_t struct
1670  Tell the ccupdate_update_t struct what cache is been updated
1671-1673 Acquire an interrupt safe lock to the cache descriptor and set its batchcount
1675  Get each CPU to update its cpucache information for itself. This swaps the old cpucaches in the cache descriptor with the new ones in new using do_ccupdate_local() (See Section H.5.2.2)
1677-1685 After smp_call_function_all_cpus() (See Section H.5.2.1), the old cpucaches are in new. This block of code cycles through them all, frees any objects in them and deletes the old cpucache
1686  Return success
1688  In the event there is no memory, delete all cpucaches that have been allocated up until this point and return failure
H.5.2 Updating Per-CPU Information

H.5.2.1 Function: \texttt{smp\_call\_function\_all\_cpus()} \textit{(mm/slab.c)}

This calls the function \texttt{func()} for all CPU's. In the context of the slab allocator, the function is \texttt{do\_ccupdate\_local()} and the argument is \texttt{ccupdate\_struct\_t}.

\begin{verbatim}
859 static void smp_call_function_all_cpus(void (*func) (void *arg),
   void *arg)
860 {
861   local_irq_disable();
862   func(arg);
863   local_irq_enable();
864   if (smp_call_function(func, arg, 1, 1))
865     BUG();
866 }
861-863 Disable interrupts locally and call the function for this CPU
865 For all other CPU's, call the function. \texttt{smp\_call\_function()} is an architecture specific function and will not be discussed further here.
\end{verbatim}

H.5.2.2 Function: \texttt{do\_ccupdate\_local()} \textit{(mm/slab.c)}

This function swaps the cpucache information in the cache descriptor with the information in \texttt{info} for this CPU.

\begin{verbatim}
874 static void do_ccupdate_local(void *info)
875 {
876   ccupdate_struct_t *new = (ccupdate_struct_t *)info;
877   cpucache_t *old = cc_data(new->cachep);
878   cc_data(new->cachep) = new->new[smp_processor_id()];
879   new->new[smp_processor_id()] = old;
881 }
876 \texttt{info} is a pointer to the \texttt{ccupdate\_struct\_t} which is then passed to \texttt{smp\_call\_function\_all\_cpus()} (See Section H.5.2.1)
877 Part of the \texttt{ccupdate\_struct\_t} is a pointer to the cache this cpucache belongs to. \texttt{cc\_data()} returns the \texttt{cpucache\_t} for this processor.
879 Place the new cpucache in cache descriptor. \texttt{cc\_data()} returns the pointer to the cpucache for this CPU.
880 Replace the pointer in \texttt{new} with the old cpucache so it can be deleted later by the caller of \texttt{smp\_call\_function\_call\_cpus()}, \texttt{kmem\_tune\_cpucache()} for example.
\end{verbatim}
**H.5.3 Draining a Per-CPU Cache**

This function is called to drain all objects in a per-cpu cache. It is called when a cache needs to be shrunk for the freeing up of slabs. A slab would not be freeable if an object was in the per-cpu cache even though it is not in use.

**H.5.3.1 Function:** drain_cpu_caches() (*mm/slab.c*)

```c
static void drain_cpu_caches(kmem_cache_t *cachep) {
    ccupdate_struct_t new;
    int i;

    memset(&new.new,0,sizeof(new.new));
    new.cachep = cachep;

    down(&cache_chain_sem);
    smp_call_function_all_cpus(do_ccupdate_local, (void *)&new);

    for (i = 0; i < smp_num_cpus; i++) {
        cpucache_t* ccold = new.new[cpu_logical_map(i)];
        if (!ccold || (ccold->avail == 0))
            continue;
        local_irq_disable();
        free_block(cachep, cc_entry(ccold), ccold->avail);
        local_irq_enable();
        ccold->avail = 0;
    }
    smp_call_function_all_cpus(do_ccupdate_local, (void *)&new);
    up(&cache_chain_sem);
}
```

- **Blank the update structure as it is going to be clearing all data**
- **Set new.cachep to cachep so that smp_call_function_all_cpus() knows what cache it is affecting**
- **Acquire the cache descriptor semaphore**
- **do_ccupdate_local()** (See Section H.5.2.2) swaps the cpucache_t information in the cache descriptor with the ones in new so they can be altered here
- **For each CPU in the system ....**
- **Get the cpucache descriptor for this CPU**
- **If the structure does not exist for some reason or there is no objects available in it, move to the next CPU**
901 Disable interrupts on this processor. It is possible an allocation from an interrupt handler elsewhere would try to access the per CPU cache.

902 Free the block of objects with `free_block()` (See Section H.3.3.5).

903 Re-enable interrupts.

904 Show that no objects are available.

906 The information for each CPU has been updated so call `do_ccupdate_local()` (See Section H.5.2.2) for each CPU to put the information back into the cache descriptor.

907 Release the semaphore for the cache chain.
H.6 Slab Allocator Initialisation

Contents

H.6 Slab Allocator Initialisation 498
   H.6.0.2 Function: kmem_cache_init() 498

H.6.0.2 Function: kmem_cache_init() \((\text{mm/slab.c})\)

This function will

- Initialise the cache chain linked list
- Initialise a mutex for accessing the cache chain
- Calculate the cache_cache colour

416 void __init kmem_cache_init(void)
417 {
418     size_t left_over;
419
420     init_MUTEX(&cache_chain_sem);
421     INIT_LIST_HEAD(&cache_chain);
422
423     kmem_cache_estimate(0, cache_cache.objsize, 0,
424                         &left_over, &cache_cache.num);
425     if (!cache_cache.num)
426         BUG();
427
428     cache_cache.colour = left_over/cache_cache.colour_off;
429     cache_cache.colour_next = 0;
430 }

420 Initialise the semaphore for access the cache chain

421 Initialise the cache chain linked list

423 \text{kmem_cache_estimate()} (See Section H.1.2.1) calculates the number of objects and amount of bytes wasted

425 If even one \text{kmem_cache_t} cannot be stored in a page, there is something seriously wrong

428 \text{colour} is the number of different cache lines that can be used while still keeping L1 cache alignment

429 \text{colour_next} indicates which line to use next. Start at 0
H.7 Interfacing with the Buddy Allocator

Contents

H.7 Interfacing with the Buddy Allocator 499
  H.7.0.3 Function: kmem_getpages() 499
  H.7.0.4 Function: kmem_freepages() 499

H.7.0.3 Function: kmem_getpages()  (mm/slab.c)
This allocates pages for the slab allocator

486 static inline void * kmem_getpages (kmem_cache_t *cachep,
                                          unsigned long flags)

487 {
488     void  *addr;
489     flags |= cachep->gfpflags;
490     addr = (void*) __get_free_pages(flags, cachep->gfporder);
491     return addr;
492 }

495 Whatever flags were requested for the allocation, append the cache flags to it.
The only flag it may append is ZONE_DMA if the cache requires DMA memory

496 Allocate from the buddy allocator with __get_free_pages() (See Section F.2.3)

503 Return the pages or NULL if it failed

H.7.0.4 Function: kmem_freepages()  (mm/slab.c)
This frees pages for the slab allocator. Before it calls the buddy allocator API,
it will remove the PG_slab bit from the page flags.

507 static inline void kmem_freepages (kmem_cache_t *cachep, void *addr)
508 {
509     unsigned long i = (1<<cachep->gfporder);
510     struct page *page = virt_to_page(addr);
511
517     while (i--) {
518         PageClearSlab(page);
519         page++;
520     }
521     free_pages((unsigned long)addr, cachep->gfporder);
522 }

509 Retrieve the order used for the original allocation

510 Get the struct page for the address

517-520 Clear the PG_slab bit on each page

521 Free the pages to the buddy allocator with free_pages() (See Section F.4.1)
Appendix I
High Memory Management

Contents

I.1 Mapping High Memory Pages ................................. 502
   I.1.0.5 Function: kmap() .................................. 502
   I.1.0.6 Function: kmap_nonblock() ......................... 502
   I.1.1 Function: __kmap() .................................. 502
   I.1.2 Function: kmap_high() ............................... 503
   I.1.3 Function: map_new_virtual() ......................... 503
   I.1.4 Function: flush_all_zero_pkmaps() ................ 506
I.2 Mapping High Memory Pages Atomically .................. 508
   I.2.1 Function: kmap_atomic() .............................. 508
I.3 Unmapping Pages ............................................. 510
   I.3.1 Function: kunmap() .................................. 510
   I.3.2 Function: kunmap_high() ............................. 510
I.4 Unmapping High Memory Pages Atomically .............. 512
   I.4.1 Function: kunmap_atomic() ............................ 512
I.5 Bounce Buffers .............................................. 513
   I.5.1 Creating Bounce Buffers .............................. 513
      I.5.1.1 Function: create_bounce() ..................... 513
      I.5.1.2 Function: alloc_bounce_bh() ................... 515
      I.5.1.3 Function: alloc_bounce_page() .................. 516
   I.5.2 Copying via Bounce Buffers ........................... 517
      I.5.2.1 Function: bounce_end_io_write() ......... 517
      I.5.2.2 Function: bounce_end_io_read() ............. 518
      I.5.2.3 Function: copy_from_high_bh() ............... 518
      I.5.2.4 Function: copy_to_high_bh_irq() ............ 519
APPENDIX I. HIGH MEMORY MANAGEMENT

I.5.2.5 Function: bounce_end_io() ....................... 519

I.6 Emergency Pools .......................................... 521

I.6.1 Function: init_emergency_pool() ...................... 521
I.1 Mapping High Memory Pages

Contents

I.1 Mapping High Memory Pages 502
I.1.0.5 Function: kmap() 502
I.1.0.6 Function: kmap_nonblock() 502
I.1.1 Function: __kmap() 502
I.1.2 Function: kmap_high() 503
I.1.3 Function: map_new_virtual() 503
I.1.4 Function: flush_all_zero_pkmaps() 506

I.1.0.5 Function: kmap() (include/asm-i386/highmem.c)

This API is used by callers willing to block.

62 #define kmap(page) __kmap(page, 0)

62 The core function __kmap() is called with the second parameter indicating that
the caller is willing to block

I.1.0.6 Function: kmap_nonblock() (include/asm-i386/highmem.c)

63 #define kmap_nonblock(page) __kmap(page, 1)

62 The core function __kmap() is called with the second parameter indicating that
the caller is not willing to block

I.1.1 Function: __kmap() (include/asm-i386/highmem.h)

The call graph for this function is shown in Figure 9.1.

65 static inline void *kmap(struct page *page, int nonblocking)
66 {
67 if (in_interrupt())
68 out_of_line_bug();
69 if (page < highmem_start_page)
70 return page_address(page);
71 return kmap_high(page);
72 }

67-68 This function may not be used from interrupt as it may sleep. Instead of
BUG(), out_of_line_bug() calls do_exit() and returns an error code. BUG() is not used because BUG() kills the process with extreme prejudice which would
result in the fabled “Aiee, killing interrupt handler!” kernel panic

69-70 If the page is already in low memory, return a direct mapping

71 Call kmap_high()(See Section I.1.2) for the beginning of the architecture independent work
I.1.2 Function: \texttt{kmap\_high()} \hspace{1em} (\textit{mm/highmem.c})

```c
void *kmap_high(struct page *page, int nonblocking) {
    unsigned long vaddr;
    spin_lock(&kmap_lock);
    vaddr = (unsigned long) page->virtual;
    if (!vaddr) {
        vaddr = map_new_virtual(page, nonblocking);
        if (!vaddr)
            goto out;
    }
    pkmap_count[PKMAP_NR(vaddr)]++;
    if (pkmap_count[PKMAP_NR(vaddr)] < 2)
        BUG();
out:
    spin_unlock(&kmap_lock);
    return (void*) vaddr;
}
```

142 The \texttt{kmap\_lock} protects the virtual field of a page and the \texttt{pkmap\_count} array.
143 Get the virtual address of the page.
144-148 If it is not already mapped, call \texttt{map\_new\_virtual()} which will map the page and return the virtual address. If it fails, goto out to free the spinlock and return NULL.
149 Increase the reference count for this page mapping.
150-151 If the count is currently less than 2, it is a serious bug. In reality, severe breakage would have to be introduced to cause this to happen.
153 Free the \texttt{kmap\_lock}.

I.1.3 Function: \texttt{map\_new\_virtual()} \hspace{1em} (\textit{mm/highmem.c})

This function is divided into three principle parts. The scanning for a free slot, waiting on a queue if none is available and mapping the page.

```c
static inline unsigned long map_new_virtual(struct page *page) {
    unsigned long vaddr;
    int count;
    start:
```
count = LAST_PKMAP;
/* Find an empty entry */
for (;;) {
    last_pkmap_nr = (last_pkmap_nr + 1) & LAST_PKMAP_MASK;
    if (!last_pkmap_nr) {
        flush_all_zero_pkmaps();
        count = LAST_PKMAP;
    }
    if (!pkmap_count[last_pkmap_nr])
        break; /* Found a usable entry */
    if (--count)
        continue;
    if (nonblocking)
        return 0;
}

Start scanning at the last possible slot

This loop keeps scanning and waiting until a slot becomes free. This allows
the possibility of an infinite loop for some processes if they were unlucky

last_pkmap_nr is the last pkmap that was scanned. To prevent searching over
the same pages, this value is recorded so the list is searched circularly. When
it reaches LAST_PKMAP, it wraps around to 0

When last_pkmap_nr wraps around, call flush_all_zero_pkmaps() (See Section I.1.4)
which will set all entries from 1 to 0 in the pkmap_count array before flushing
the TLB. Count is set back to LAST_PKMAP to restart scanning

If this element is 0, a usable slot has been found for the page

Move to the next index to scan

The next block of code is going to sleep waiting for a slot to be free. If the
caller requested that the function not block, return now

{  
    DECLARE_WAITQUEUE(wait, current);
    current->state = TASK_UNINTERRUPTIBLE;
    add_wait_queue(&pkmap_map_wait, &wait);
    spin_unlock(&kmap_lock);
    schedule();
    remove_wait_queue(&pkmap_map_wait, &wait);
    spin_lock(&kmap_lock);
    /* Somebody else might have mapped it while we
I.1 Mapping High Memory Pages (map_new_virtual())

```c
if (page->virtual)
    return (unsigned long) page->virtual;

/* Re-start */
goto start;
```

If there is no available slot after scanning all the pages once, we sleep on the pkmap_map_wait queue until we are woken up after an unmap.

106 Declare the wait queue

108 Set the task as interruptible because we are sleeping in kernel space

109 Add ourselves to the pkmap_map_wait queue

110 Free the kmap_lock spinlock

111 Call schedule() which will put us to sleep. We are woken up after a slot becomes free after an unmap

112 Remove ourselves from the wait queue

113 Re-acquire kmap_lock

116-117 If someone else mapped the page while we slept, just return the address and the reference count will be incremented by kmap_high()

120 Restart the scanning

```c
vaddr = PKMAP_ADDR(last_pkmap_nr);
set_pte(&(pkmap_page_table[last_pkmap_nr]), mk_pte(page, kmap_prot));
pkmap_count[last_pkmap_nr] = 1;
page->virtual = (void *) vaddr;
return vaddr;
```

A slot has been found, map the page

123 Get the virtual address for the slot found

124 Make the PTE entry with the page and required protection and place it in the page tables at the found slot
Initialise the value in the pkmap_count array to 1. The count is incremented in the parent function and we are sure this is the first mapping if we are in this function in the first place.

Set the virtual field for the page.

Return the virtual address.

**I.1.4 Function: flush_all_zero_pkmaps()** *(mm/highmem.c)*

This function cycles through the pkmap_count array and sets all entries from 1 to 0 before flushing the TLB.

```c
42 static void flush_all_zero_pkmaps(void) {
43     int i;
44     flush_cache_all();
45     for (i = 0; i < LAST_PKMAP; i++) {
46         struct page *page;
47         if (pkmap_count[i] != 1)
48             continue;
49         pkmap_count[i] = 0;
50         /* sanity check */
51         if (pte_none(pkmap_page_table[i]))
52             BUG();
53         page = pte_page(pkmap_page_table[i]);
54         pte_clear(&pkmap_page_table[i]);
55         page->virtual = NULL;
56     }
57     flush_tlb_all();
58 }
```

As the global page tables are about to change, the CPU caches of all processors have to be flushed.

48-76 Cycle through the entire pkmap_count array

57-58 If the element is not 1, move to the next element

59 Set from 1 to 0

62-63 Make sure the PTE is not somehow mapped
72-73 Unmap the page from the PTE and clear the PTE
75 Update the virtual field as the page is unmapped
77 Flush the TLB
I.2 Mapping High Memory Pages Atomically

Contents

I.2 Mapping High Memory Pages Atomically 508
I.2.1 Function: kmap_atomic() 508

The following is an example km_type enumeration for the x86. It lists the different uses interrupts have for atomically calling kmap. Note how KM_TYPE_NR is the last element so it doubles up as a count of the number of elements.

```c
enum km_type {
    KM_BOUNCE_READ,
    KM_SKB_SUNRPC_DATA,
    KM_SKB_DATA_SOFTIRQ,
    KM_USER0,
    KM_USER1,
    KM_BH_IRQ,
    KM_TYPE_NR
};
```

I.2.1 Function: kmap_atomic()  (include/asm-i386/highmem.h)

This is the atomic version of kmap(). Note that at no point is a spinlock held or does it sleep. A spinlock is not required as every processor has its own reserved space.

```c
static inline void *kmap_atomic(struct page *page,
enum km_type type)
{
    enum fixed_addresses idx;
    unsigned long vaddr;
    
    if (page < highmem_start_page)
        return page_address(page);
    
    idx = type + KM_TYPE_NR*smp_processor_id();
    vaddr = __fix_to_virt(FIX_KMAP_BEGIN + idx);
    #if HIGHMEM_DEBUG
    if (!pte_none(*(kmap_pte-idx)))
        out_of_line_bug();
    #endif
    set_pte(kmap_pte-idx, mk_pte(page, kmap_prot));
    __flush_tlb_one(vaddr);
    return (void*) vaddr;
}
```
The parameters are the page to map and the type of usage required. One slot per usage per processor is maintained.

If the page is in low memory, return a direct mapping.

type gives which slot to use. \texttt{KM\_TYPE\_NR * smp\_processor\_id()} gives the set of slots reserved for this processor.

Get the virtual address.

Debugging code. In reality a PTE will always exist.

Set the PTE into the reserved slot.

Flush the TLB for this slot.

Return the virtual address.
I.3 Unmapping Pages

Contents

I.3 Unmapping Pages 510
I.3.1 Function: *kunmap()* 510
I.3.2 Function: *kunmap_high()* 510

I.3.1 Function: *kunmap()* *(include/asm-i386/highmem.h)*

74 static inline void kunmap(struct page *page)
75 {
76     if (in_interrupt())
77         out_of_line_bug();
78     if (page < highmem_start_page)
79         return;
80     kunmap_high(page);
81 }

76-77 *kunmap()* cannot be called from interrupt so exit gracefully
78-79 If the page already is in low memory, there is no need to unmap
80 Call the architecture independent function *kunmap_high()*

I.3.2 Function: *kunmap_high()* *(mm/highmem.c)*

This is the architecture independent part of the *kunmap()* operation.

157 void kunmap_high(struct page *page)
158 {
159     unsigned long vaddr;
160     unsigned long nr;
161     int need_wakeup;
162     spin_lock(&kmap_lock);
163     vaddr = (unsigned long) page->virtual;
164     if (!vaddr)
165         BUG();
166     nr = PKMAP_NR(vaddr);
167     need_wakeup = 0;
168     switch (--pkmap_count[nr]) {
169         case 0:
170             BUG();
171         case 1:
172             need_wakeup = waitqueue_active(&pkmap_map_wait);
173         }
174     }
175 }
I.3 Unmapping Pages (kunmap_high())

190 spin_unlock(&kmap_lock);
191 */ do wake-up, if needed, race-free outside of the spin lock */
192 if (need_wakeup)
193     wake_up(&pkmap_map_wait);
194 }

163 Acquire kmap_lock protecting the virtual field and the pkmap_count array
164 Get the virtual page
165-166 If the virtual field is not set, it is a double unmapping or unmapping of
a non-mapped page so BUG()
167 Get the index within the pkmap_count array
173 By default, a wakeup call to processes calling kmap() is not needed
174 Check the value of the index after decrement
175-176 Falling to 0 is a bug as the TLB needs to be flushed to make 0 a valid
entry
177-188 If it has dropped to 1 (the entry is now free but needs a TLB flush), check
to see if there is anyone sleeping on the pkmap_map_wait queue. If necessary,
the queue will be woken up after the spinlock is freed
190 Free kmap_lock
193-194 If there are waiters on the queue and a slot has been freed, wake them up
I.4 Unmapping High Memory Pages Atomically

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I.4 Unmapping High Memory Pages Atomically</td>
<td>512</td>
</tr>
<tr>
<td>I.4.1 Function: kunmap_atomic()</td>
<td>512</td>
</tr>
</tbody>
</table>

I.4.1 Function: kunmap_atomic()  

This entire function is debug code. The reason is that as pages are only mapped here atomically, they will only be used in a tiny place for a short time before being unmapped. It is safe to leave the page there as it will not be referenced after unmapping and another mapping to the same slot will simply replace it.

```c
109 static inline void kunmap_atomic(void *kvaddr, enum km_type type) {
110   #if HIGHMEM_DEBUG
111     unsigned long vaddr = (unsigned long) kvaddr & PAGE_MASK;
112     enum fixed_addresses idx = type + KM_TYPE_NR*smp_processor_id();
113
114     if (vaddr < FIXADDR_START) // FIXME
115         return;
116
117     if (vaddr != __fix_to_virt(FIX_KMAP_BEGIN+idx))
118         out_of_line_bug();
119
120     /*
121      * force other mappings to Oops if they’ll try to access
122      * this pte without first remap it
123      */
124     pte_clear(kmap_pte-idx);
125     __flush_tlb_one(vaddr);
126 }
127 #endif
128 }
```

112 Get the virtual address and ensure it is aligned to a page boundary

115-116 If the address supplied is not in the fixed area, return

118-119 If the address does not correspond to the reserved slot for this type of usage and processor, declare it

125-126 Unmap the page now so that if it is referenced again, it will cause an Oops
I.5 Bounce Buffers

Contents

I.5 Bounce Buffers 513
I.5.1 Creating Bounce Buffers 513
  I.5.1.1 Function: create_bounce() 513
  I.5.1.2 Function: alloc_bounce_bh() 515
  I.5.1.3 Function: alloc_bounce_page() 516
I.5.2 Copying via Bounce Buffers 517
  I.5.2.1 Function: bounce_end_io_write() 517
  I.5.2.2 Function: bounce_end_io_read() 518
  I.5.2.3 Function: copy_from_high_bh() 518
  I.5.2.4 Function: copy_to_high_bh_irq() 519
  I.5.2.5 Function: bounce_end_io() 519

I.5.1 Creating Bounce Buffers

I.5.1.1 Function: create_bounce() (mm/highmem.c)

The call graph for this function is shown in Figure 9.3. High level function for
the creation of bounce buffers. It is broken into two major parts, the allocation of
the necessary resources, and the copying of data from the template.

405 struct buffer_head * create_bounce(int rw,
    struct buffer_head * bh_orig)
    {
406      struct page *page;
407      struct buffer_head *bh;
408      
409      if (!PageHighMem(bh_orig->b_page))
410        return bh_orig;
411      
412      bh = alloc_bounce_bh();
413      page = alloc_bounce_page();
414      
415      set_bh_page(bh, page, 0);
416
405 The parameters of the function are

    rw    is set to 1 if this is a write buffer
    bh_orig    is the template buffer head to copy from

410-411 If the template buffer head is already in low memory, simply return it

413 Allocate a buffer head from the slab allocator or from the emergency pool if
it fails
I.5.1 Creating Bounce Buffers (create_bounce())

Allocate a page from the buddy allocator or the emergency pool if it fails

Associate the allocated page with the allocated buffer_head

```c
bh->b_next = NULL;
bh->b_blocknr = bh_orig->b_blocknr;
bh->b_size = bh_orig->b_size;
bh->b_list = -1;
bh->b_dev = bh_orig->b_dev;
bh->b_count = bh_orig->b_count;
bh->b_rdev = bh_orig->b_rdev;
bh->b_state = bh_orig->b_state;
#endif
bh->b_flushtime = jiffies;
bh->b_next_free = NULL;
bh->b_prev_free = NULL;
/* bh->b_this_page */
bh->b_rqnext = NULL;
bh->b_private = (void *)bh_orig;
#endif
bh->b_private = (void *)bh_orig;
bh->b_rsector = bh_orig->b_rsector;
#endif
memset(&(bh->b_wait), -1, sizeof(bh->b_wait));
#endif
return bh;
```

Populate the newly created buffer_head

Copy in information essentially verbatim except for the b_list field as this
buffer is not directly connected to the others on the list

Debugging only information

441-444 If this is a buffer that is to be written to then the callback function to end
the IO is bounce_end_io_write() (See Section I.5.2.1) which is called when
the device has received all the information. As the data exists in high memory,
it is copied “down” with copy_from_high_bh() (See Section I.5.2.3)
437-438 If we are waiting for a device to write data into the buffer, then the callback function `bounce_end_io_read()` (See Section I.5.2.2) is used.

446-447 Copy the remaining information from the template `buffer_head`

452 Return the new bounce buffer

### I.5.1.2 Function: alloc_bounce_bh()  
*(mm/highmem.c)*

This function first tries to allocate a `buffer_head` from the slab allocator and if that fails, an emergency pool will be used.

369 struct buffer_head *alloc_bounce_bh (void) {
370    struct list_head *tmp;
371    struct buffer_head *bh;
372
374    bh = kmem_cache_alloc(bh_cachep, SLAB_NOHIGHIO);
375    if (bh)
376        return bh;
377
374    Try to allocate a new `buffer_head` from the slab allocator. Note how the request is made to *not* use IO operations that involve high IO to avoid recursion.
375-376 If the allocation was successful, return

381 If it was not, wake up `bdflush` to launder pages
383 repeat_alloc:
387    tmp = &emergency_bhs;
388    spin_lock_irq(&emergency_lock);
389    if (!list_empty(tmp)) {
390        bh = list_entry(tmp->next, struct buffer_head, b_inode_buffers);
391        list_del(tmp->next);
392        nr_emergency_bhs--;
393    }
394    spin_unlock_irq(&emergency_lock);
395    if (bh)
396        return bh;
397
398    /* we need to wait I/O completion */
399    run_task_queue(&tq_disk);
400
401    yield();
402    goto repeat_alloc;
403 }
I.5.1 Creating Bounce Buffers (\textit{alloc\textunderscore bounce\textunderscore bh()})

The allocation from the slab failed so allocate from the emergency pool.

387 Get the end of the emergency buffer head list
388 Acquire the lock protecting the pools
389-393 If the pool is not empty, take a \texttt{buffer\_head} from the list and decrement the \texttt{nr\_emergency\_bhs} counter
394 Release the lock
395-396 If the allocation was successful, return it
399 If not, we are seriously short of memory and the only way the pool will replenish is if high memory IO completes. Therefore, requests on \texttt{tq\_disk} are started so the data will be written to disk, probably freeing up pages in the process
401 Yield the processor
402 Attempt to allocate from the emergency pools again

I.5.1.3 \textbf{Function:} \texttt{alloc\textunderscore bounce\textunderscore page()} \textit{(mm/highmem.c)}
This function is essentially identical to \texttt{alloc\textunderscore bounce\textunderscore bh()}. It first tries to allocate a page from the buddy allocator and if that fails, an emergency pool will be used.

333 \textbf{struct page \*alloc\textunderscore bounce\textunderscore page (void)}
334 {
335 \hspace{1em} \textbf{struct list\_head \*tmp;}
336 \hspace{1em} \textbf{struct page \*page;}
337 \hspace{1em}
338 \hspace{1em} \texttt{page = alloc\textunderscore page(GFP\_NOHIGHIO)};
339 \hspace{1em} \textbf{if (page)}
340 \hspace{1.5em} \texttt{return page;}
344 \hspace{1em} \texttt{wakeup\textunderscore bdflush();}
345 repeat_alloc:
338-340 Allocate from the buddy allocator and return the page if successful
345 \hspace{1em} \texttt{Wake bdflush to launder pages}
347 repeat_alloc:
351 \hspace{1em} tmp = \&emergency\_pages;
352 \hspace{1em} spin\textunderscore lock\textunderscore irq(\&emergency\_lock);
353 \hspace{1em} \textbf{if (!list\textunderscore empty(tmp))} {
354 \hspace{1.5em} \texttt{page = list\textunderscore entry(tmp\texttt{->next}, struct page, list);}  
355 \hspace{1.5em} list\textunderscore del(tmp\texttt{->next});
356 \hspace{1.5em} nr\textunderscore emergency\_pages--;
I.5.2 Copying via Bounce Buffers

I.5.2.1 Function: bounce_end_io_write() (mm/highmem.c)

This function is called when a bounce buffer used for writing to a device completes IO. As the buffer is copied from high memory and to the device, there is nothing left to do except reclaim the resources.

```c
319 static void bounce_end_io_write (struct buffer_head *bh, int uptodate)
320 {
321     bounce_end_io(bh, uptodate);
322 }
```
I.5.2.2 Function: bounce_end_io_read() (mm/highmem.c)
This is called when data has been read from the device and needs to be copied to high memory. It is called from interrupt so has to be more careful.

324 static void bounce_end_io_read (struct buffer_head *bh, int uptodate)
325 {
326     struct buffer_head *bh_orig =
327         (struct buffer_head *) (bh->b_private);
328     if (uptodate)
329         copy_to_high_bh_irq(bh_orig, bh);
330     bounce_end_io(bh, uptodate);
331 }

328-329 The data is just copied to the bounce buffer to needs to be moved to high memory with copy_to_high_bh_irq() (See Section I.5.2.4)

330 Reclaim the resources

I.5.2.3 Function: copy_from_high_bh() (mm/highmem.c)
This function copies data from a high memory buffer_head to a bounce buffer.

215 static inline void copy_from_high_bh (struct buffer_head *to, struct buffer_head *from)
216 {
217     struct page *p_from;
218     char *vfrom;
219     p_from = from->b_page;
220     vfrom = kmap_atomic(p_from, KM_USER0);
221     memcpy(to->b_data, vfrom + bh_offset(from), to->b_size);
222     kunmap_atomic(vfrom, KM_USER0);
223 }

223 Map the high memory page into low memory. This path is protected by the IRQ safe lock io_request_lock so it is safe to call kmap_atomic() (See Section I.2.1)

224 Copy the data

225 Unmap the page
I.5.2.4 Function: `copy_to_high_bh_irq()` *(mm/highmem.c)*

Called from interrupt after the device has finished writing data to the bounce buffer. This function copies data to high memory.

```c
static inline void copy_to_high_bh_irq (struct buffer_head *to, struct buffer_head *from)
{
    struct page *p_to;
    char *vto;
    unsigned long flags;

    p_to = to->b_page;
    __save_flags(flags);
    __cli();
    vto = kmap_atomic(p_to, KM_BOUNCE_READ);
    memcpy(vto + bh_offset(to), from->b_data, to->b_size);
    kunmap_atomic(vto, KM_BOUNCE_READ);
    __restore_flags(flags);
}
```

236-237 Save the flags and disable interrupts

238 Map the high memory page into low memory

239 Copy the data

240 Unmap the page

241 Restore the interrupt flags

I.5.2.5 Function: `bounce_end_io()` *(mm/highmem.c)*

Reclaims the resources used by the bounce buffers. If emergency pools are depleted, the resources are added to it.

```c
static inline void bounce_end_io (struct buffer_head *bh, int uptodate)
{
    struct page *page;
    struct buffer_head *bh_orig =
        (struct buffer_head *)(bh->b_private);
    unsigned long flags;
    bh_orig->b_end_io(bh_orig, uptodate);
    page = bh->b_page;
```
254 spin_lock_irqsave(&emergency_lock, flags);
255 if (nr_emergency_pages >= POOL_SIZE)
256    __free_page(page);
257 else {
258     /*
259     * We are abusing page->list to manage
260     * the highmem emergency pool:
261     */
262     list_add(&page->list, &emergency_pages);
263     nr_emergency_pages++;
264 }
265
266 if (nr_emergency_bhs >= POOL_SIZE) {
267 #ifdef HIGHMEM_DEBUG
268     /* Don’t clobber the constructed slab cache */
269     init_waitqueue_head(&bh->b_wait);
270 #endif
271     kmem_cache_free(bh_cachep, bh);
272 } else {
273     /*
274     * Ditto in the bh case, here we abuse b_inode_buffers:
275     */
276     list_add(&bh->b_inode_buffers, &emergency_bhs);
277     nr_emergency_bhs++;
278 }
279 spin_unlock_irqrestore(&emergency_lock, flags);

250 Call the IO completion callback for the original buffer_head
252 Get the pointer to the buffer page to free
254 Acquire the lock to the emergency pool
255-256 If the page pool is full, just return the page to the buddy allocator
257-264 Otherwise add this page to the emergency pool
266-272 If the buffer_head pool is full, just return it to the slab allocator
272-278 Otherwise add this buffer_head to the pool
279 Release the lock
I.6 Emergency Pools

Contents

1.6 Emergency Pools 521
1.6.1 Function: init_emergency_pool() 521

There is only one function of relevance to the emergency pools and that is the init function. It is called during system startup and then the code is deleted as it is never needed again.

I.6.1 Function: init_emergency_pool() (mm/highmem.c)

Create a pool for emergency pages and for emergency buffer_heads

282 static __init int init_emergency_pool(void)
283 {
284     struct sysinfo i;
285     si_meminfo(&i);
286     si_swapinfo(&i);
287
288     if (!i.totalhigh)
289         return 0;
290
291     spin_lock_irq(&emergency_lock);
292     while (nr_emergency_pages < POOL_SIZE) {
293         struct page * page = alloc_page(GFP_ATOMIC);
294         if (!page) {
295             printk("couldn’t refill highmem emergency pages");
296             break;
297         }
298         list_add(&page->list, &emergency_pages);
299         nr_emergency_pages++;
300     }

288-289 If there is no high memory available, do not bother

291 Acquire the lock protecting the pools

292-300 Allocate POOL_SIZE pages from the buddy allocator and add them to a linked list. Keep a count of the number of pages in the pool with nr_emergency_pages

301     while (nr_emergency_bhs < POOL_SIZE) {
302         struct buffer_head * bh =
303             kmem_cache_alloc(bh_cachep, SLAB_ATOMIC);
304         if (!bh) {
305             printk("couldn’t refill highmem emergency bhs");
306             break;
307         }
I.6 Emergency Pools (*init_emergency_pool()*)

```c
306   }
307   list_add(&bh->b_inode_buffers, &emergency_bhs);
308   nr_emergency_bhs++;
309 }
310 spin_unlock_irq(&emergency_lock);
311 printk("allocated %d pages and %d bhs reserved for the
312       highmem bounces\n",
313       nr_emergency_pages, nr_emergency_bhs);
314   return 0;
315 }
```

301-309 Allocate `POOL_SIZE` buffer_heads from the slab allocator and add them to a linked list linked by `b_inode_buffers`. Keep track of how many heads are in the pool with `nr_emergency_bhs`

310 Release the lock protecting the pools

314 Return success
Appendix J

Page Frame Reclamation

Contents

J.1  Page Cache Operations ................................. 525
    J.1.1  Adding Pages to the Page Cache .......................... 525
        J.1.1.1  Function: add_to_page_cache() ....................... 525
        J.1.1.2  Function: add_to_page_cache_unique() ............... 526
        J.1.1.3  Function: __add_to_page_cache() ................... 527
        J.1.1.4  Function: add_page_to_inode_queue() ............... 528
        J.1.1.5  Function: add_page_to_hash_queue() ............... 528
    J.1.2  Deleting Pages from the Page Cache ..................... 529
        J.1.2.1  Function: remove_inode_page() .................... 529
        J.1.2.2  Function: __remove_inode_page() .................. 529
        J.1.2.3  Function: remove_page_from_inode_queue() ....... 530
        J.1.2.4  Function: remove_page_from_hash_queue() ....... 530
    J.1.3  Acquiring/Releasing Page Cache Pages ................... 531
        J.1.3.1  Function: page_cache_get() ....................... 531
        J.1.3.2  Function: page_cache_release() .................... 531
    J.1.4  Searching the Page Cache ............................. 531
        J.1.4.1  Function: find_get_page() ...................... 531
        J.1.4.2  Function: __find_get_page() .................... 531
        J.1.4.3  Function: __find_page_nolock() .................. 532
        J.1.4.4  Function: find_lock_page() ....................... 533
        J.1.4.5  Function: __find_lock_page() .................... 533
        J.1.4.6  Function: __find_lock_page_helper() ............. 533
J.2  LRU List Operations ................................. 535
    J.2.1  Adding Pages to the LRU Lists ....................... 535
APPENDIX J. PAGE FRAME RECLAMATION

J.2.1 Deleting Pages from the LRU Lists
J.2.1.1 Function: lru_cache_add() ........................................... 535
J.2.1.2 Function: add_page_to_active_list() ............................. 535
J.2.1.3 Function: add_page_to_inactive_list() ......................... 536
J.2.1.4 Function: del_page_from_active_list() ......................... 536
J.2.1.5 Function: del_page_from_inactive_list() ....................... 537
J.2.2 Deleting Pages from the LRU Lists ................................... 536
J.2.2.1 Function: lru_cache_del() .......................................... 536
J.2.2.2 Function: __lru_cache_del() ........................................ 537
J.2.2.3 Function: del_page_from_active_list() ......................... 537
J.2.2.4 Function: del_page_from_inactive_list() ....................... 537
J.2.3 Activating Pages .............................................................. 538
J.2.3.1 Function: mark_page_accessed() .................................... 538
J.2.3.2 Function: activate_lock() ............................................ 538
J.2.3.3 Function: activate_page_nolock() ................................. 538
J.3 Refilling inactive_list .......................................................... 540
J.3.1 Function: refill_inactive() ............................................. 540
J.4 Reclaiming Pages from the LRU Lists .................................... 542
J.4.1 Function: shrink_cache() .................................................. 542
J.4.2 Function: try_to_free_pages() ......................................... 550
J.4.3 Function: try_to_free_pages_zone() ................................... 551
J.5 Shrinking all caches ............................................................ 550
J.5.1 Function: shrink_caches() ............................................. 550
J.5.2 Function: try_to_free_pages() ......................................... 551
J.5.3 Function: try_to_free_pages_zone() ................................... 552
J.6 Swapping Out Process Pages ................................................ 554
J.6.1 Function: swap_out() .................................................... 554
J.6.2 Function: swap_out_mm() ................................................ 556
J.6.3 Function: swap_out_vma() ............................................. 557
J.6.4 Function: swap_out_pgd() ............................................. 558
J.6.5 Function: swap_out_pmd() ............................................. 559
J.6.6 Function: try_to_swap_out() ........................................... 561
J.7 Page Swap Daemon ............................................................... 565
J.7.1 Initialising kswapd .......................................................... 565
J.7.1.1 Function: kswapd_init() ............................................. 565
J.7.2 Kswapd Daemon ............................................................... 565
J.7.2.1 Function: kswapd() .................................................... 565
J.7.2.2 Function: kswapd_can_sleep() ...................................... 567
J.7.2.3 Function: kswapd_can_sleep_pgd() .................................. 567
J.7.2.4 Function: kswapd_balance() ......................................... 568
J.7.2.5 Function: kswapd_balance_pgd() ................................... 568
J.1 Page Cache Operations

Contents

J.1 Page Cache Operations 525
  J.1.1 Adding Pages to the Page Cache 525
    J.1.1.1 Function: add_to_page_cache() 525
    J.1.1.2 Function: add_to_page_cache_unique() 526
    J.1.1.3 Function: __add_to_page_cache() 527
    J.1.1.4 Function: add_page_to_inode_queue() 528
    J.1.1.5 Function: add_page_to_hash_queue() 528
  J.1.2 Deleting Pages from the Page Cache 529
    J.1.2.1 Function: remove_inode_page() 529
    J.1.2.2 Function: __remove_inode_page() 529
    J.1.2.3 Function: remove_page_from_inode_queue() 530
    J.1.2.4 Function: remove_page_from_hash_queue() 530
  J.1.3 Acquiring/Releasing Page Cache Pages 531
    J.1.3.1 Function: page_cache_get() 531
    J.1.3.2 Function: page_cache_release() 531
  J.1.4 Searching the Page Cache 531
    J.1.4.1 Function: find_get_page() 531
    J.1.4.2 Function: __find_get_page() 531
    J.1.4.3 Function: __find_page_nolock() 532
    J.1.4.4 Function: find_lock_page() 533
    J.1.4.5 Function: __find_lock_page() 533
    J.1.4.6 Function: __find_lock_page_helper() 533

This section addresses how pages are added and removed from the page cache and LRU lists, both of which are heavily intertwined.

J.1.1 Adding Pages to the Page Cache

J.1.1.1 Function: add_to_page_cache() (mm/filemap.c)

Acquire the lock protecting the page cache before calling __add_to_page_cache() which will add the page to the page hash table and inode queue which allows the pages belonging to files to be found quickly.

```c
667 void add_to_page_cache(struct page * page,
                      struct address_space * mapping,
                      unsigned long offset)
668 {
669     spin_lock(&pagecache_lock);
670     __add_to_page_cache(page, mapping,
671          offset, page_hash(mapping, offset));
672     spin_unlock(&pagecache_lock);
673     lru_cache_add(page);
```
Acquire the lock protecting the page hash and inode queues

Call the function which performs the “real” work

Release the lock protecting the hash and inode queue

Add the page to the page cache. page_hash() hashes into the page hash table based on the mapping and the offset within the file. If a page is returned, there was a collision and the colliding pages are chained with the page → next_hash and page → pprev_hash fields

**Function: add_to_page_cache_unique() (mm/filemap.c)**

In many respects, this function is very similar to add_to_page_cache(). The principal difference is that this function will check the page cache with the pagecache_lock spinlock held before adding the page to the cache. It is for callers may race with another process for inserting a page in the cache such as add_to_swap_cache() (See Section K.2.1.1).

```c
int add_to_page_cache_unique(struct page *page,
    struct address_space *mapping, unsigned long offset,
    struct page **hash)
{
    int err;
    struct page *alias;

    spin_lock(&pagecache_lock);
    alias = __find_page_nolock(mapping, offset, *hash);

    err = 1;
    if (!alias) {
        __add_to_page_cache(page, mapping, offset, hash);
        err = 0;
    }

    spin_unlock(&pagecache_lock);
    if (!err)
        lru_cache_add(page);
    return err;
}
```

Acquire the pagecache_lock for examining the cache

Check if the page already exists in the cache with __find_page_nolock() (See Section J.1.4.3)

If the page does not exist in the cache, add it with __add_to_page_cache() (See Section J.1.1.3)
J.1.1 Adding Pages to the Page Cache (add_to_page_cache_unique())

691 Release the pagecache_lock

692-693 If the page did not already exist in the page cache, add it to the LRU lists with lru_cache_add() (See Section J.2.1.1)

694 Return 0 if this call entered the page into the page cache and 1 if it already existed

J.1.1.3 Function: __add_to_page_cache()  (mm/filemap.c)

Clear all page flags, lock it, take a reference and add it to the inode and hash queues.

653 static inline void __add_to_page_cache(struct page * page,
654         struct address_space *mapping, unsigned long offset,
655         struct page **hash)
656 {
   657     unsigned long flags;
658     659     flags = page->flags & ~(1 << PG_uptodate |
          1 << PG_error | 1 << PG_dirty |
          1 << PG_referenced | 1 << PG_arch_1 |
          1 << PG_checked);
660     page->flags = flags | (1 << PG_locked);
661     page_cache_get(page);
662     page->index = offset;
663     add_page_to_inode_queue(mapping, page);
664     add_page_to_hash_queue(page, hash);
665 }

659 Clear all page flags

660 Lock the page

661 Take a reference to the page in case it gets freed prematurely

662 Update the index so it is known what file offset this page represents

663 Add the page to the inode queue with add_page_to_inode_queue() (See Section J.1.1.4).
   This links the page via the page→list to the clean_pages list in the
   address_space and points the page→mapping to the same address_space

664 Add it to the page hash with add_page_to_hash_queue() (See Section J.1.1.5).
   The hash page was returned by page_hash() in the parent function. The page
   hash allows page cache pages without having to linearly search the inode queue
J.1.1.4 Function: add_page_to_inode_queue()  (mm/lemap.c)

```c
85 static inline void add_page_to_inode_queue(
    struct address_space *mapping, struct page * page)
86 {
    struct list_head *head = &mapping->clean_pages;
87    mapping->nr_pages++;
88    list_add(&page->list, head);
89    page->mapping = mapping;
90 }
```

When this function is called, the page is clean, so mapping->clean_pages is the list of interest

Increment the number of pages that belong to this mapping

Add the page to the clean list

Set the page->mapping field

J.1.1.5 Function: add_page_to_hash_queue()  (mm/lemap.c)

This adds page to the top of hash bucket headed by p. Bear in mind that p is an element of the array page_hash_table.

```c
71 static void add_page_to_hash_queue(struct page * page,
    struct page **p)
72 {
    struct page *next = *p;
73    *p = page;
74    page->next_hash = next;
75    page->pprev_hash = p;
76    if (next)
77        next->pprev_hash = &page->next_hash;
78    if (page->buffers)
79        PAGE_BUG(page);
80    atomic_inc(&page_cache_size);
81 }
```

Record the current head of the hash bucket in next

Update the head of the hash bucket to be page

Point page->next_hash to the old head of the hash bucket

Point page->pprev_hash to point to the array element in page_hash_table
This will point the pprev_hash field to the head of the hash bucket completing the insertion of the page into the linked list.

Check that the page entered has no associated buffers.

Increment page_cache_size which is the size of the page cache.

### J.1.2 Deleting Pages from the Page Cache

#### J.1.2.1 Function: remove_inode_page() *(mm/filemap.c)*

```c
void remove_inode_page(struct page *page) {
    if (!PageLocked(page))
        PAGE_BUG(page);

    spin_lock(&pagecache_lock);
    __remove_inode_page(page);
    spin_unlock(&pagecache_lock);
}
```

132-133 If the page is not locked, it is a bug.

135 Acquire the lock protecting the page cache.

136 **__remove_inode_page()** (See Section J.1.2.2) is the top-level function for when the pagecache lock is held.

137 Release the pagecache lock.

#### J.1.2.2 Function: __remove_inode_page() *(mm/filemap.c)*

This is the top-level function for removing a page from the page cache for callers with the pagecache_lock spinlock held. Callers that do not have this lock acquired should call **remove_inode_page()**.

```c
void __remove_inode_page(struct page *page) {
    remove_page_from_inode_queue(page);
    remove_page_from_hash_queue(page);
}
```

126 remove_page_from_inode_queue() (See Section J.1.2.3) remove the page from it's address_space at page->mapping.

127 remove_page_from_hash_queue() removes the page from the hash table in page_hash_table.
J.1.2.3 Function: remove_page_from_inode_queue() (mm/lemap.c)

94 static inline void remove_page_from_inode_queue(struct page * page) {
95   struct address_space * mapping = page->mapping;
96
97   if (mapping->a_ops->removepage) {
98     mapping->a_ops->removepage(page);
99   }
100   list_del(&page->list);
101   page->mapping = NULL;
102   wmb();
103   mapping->nr_pages--;
104 }

96 Get the associated address_space for this page
98-99 Call the filesystem specific removepage() function if one is available
100 Delete the page from whatever list it belongs to in the mapping such as the clean_pages list in most cases or the dirty_pages in rarer cases
101 Set the page->mapping to NULL as it is no longer backed by any address_space
103 Decrement the number of pages in the mapping

J.1.2.4 Function: remove_page_from_hash_queue() (mm/filemap.c)

107 static inline void remove_page_from_hash_queue(struct page * page) {
108   { struct page *next = page->next_hash;
109     struct page **pprev = page->pprev_hash;
110
112     if (next) {
113       next->pprev_hash = pprev;
114       *pprev = next;
115       page->pprev_hash = NULL;
116       atomic_dec(&page_cache_size);
117     }

109 Get the next page after the page being removed
110 Get the pprev page before the page being removed. When the function completes, pprev will be linked to next
112 If this is not the end of the list, update next->pprev_hash to point to pprev
114 Similarly, point pprev forward to next. page is now unlinked
116 Decrement the size of the page cache
J.1.3 Acquiring/Releasing Page Cache Pages

J.1.3.1 Function: page_cache_get()  (include/linux/pagemap.h)

31 #define page_cache_get(x) get_page(x)

31 Simple call get_page() which simply uses atomic_inc() to increment the page reference count.

J.1.3.2 Function: page_cache_release()  (include/linux/pagemap.h)

32 #define page_cache_release(x) __free_page(x)

32 Call __free_page() which decrements the page count. If the count reaches 0, the page will be freed.

J.1.4 Searching the Page Cache

J.1.4.1 Function: find_get_page()  (include/linux/pagemap.h)

Top level macro for finding a page in the page cache. It simply looks up the page hash.

75 #define find_get_page(mapping, index) \
76 __find_get_page(mapping, index, page_hash(mapping, index))

76 page_hash() locates an entry in the page_hash_table based on the address_space and offset.

J.1.4.2 Function: __find_get_page()  (mm/filemap.c)

This function is responsible for finding a struct page given an entry in page_hash_table as a starting point.

931 struct page * __find_get_page(struct address_space *mapping, 
932 unsigned long offset, struct page **hash)
933 {
934     struct page *page;
935 
936     /*
937     * We scan the hash list read-only. Addition to and removal from 
938     * the hash-list needs a held write-lock.
939     */
940     spin_lock(&pagecache_lock);
941     page = __find_page_nolock(mapping, offset, *hash);
942     if (page)
943         page_cache_get(page);
944     spin_unlock(&pagecache_lock);
945     return page;
946 }
Acquire the read-only page cache lock

Call the page cache traversal function which presumes a lock is held

If the page was found, obtain a reference to it with `page_cache_get()` (See Section J.1.3.1) so it is not freed prematurely

Release the page cache lock

Return the page or NULL if not found

**J.1.4.3 Function: ** `__find_page_nolock()` *(mm/filemap.c)*

This function traverses the hash collision list looking for the page specified by the `address_space` and `offset`.

```c
443 static inline struct page * __find_page_nolock(
    struct address_space *mapping,
    unsigned long offset,
    struct page *page)
444 {
    goto inside;
446
    for (;;) {
        page = page->next_hash;
        inside:
        if (!page)
            goto not_found;
        if (page->mapping != mapping)
            continue;
        if (page->index == offset)
            break;
    }
not_found:
    return page;
}
```

Begin by examining the first page in the list

If the page is NULL, the right one could not be found so return NULL

If the `address_space` does not match, move to the next page on the collision list

If the `offset` matches, return it, else move on

Move to the next page on the hash list

Return the found page or NULL if not
**J.1.4.4 Function: `find_lock_page()`**

This is the top level function for searching the page cache for a page and having it returned in a locked state.

```c
#define find_lock_page(mapping, index)  
   __find_lock_page(mapping, index, page_hash(mapping, index))
```

Call the core function `__find_lock_page()` after looking up what hash bucket this page is using with `page_hash()`.

**J.1.4.5 Function: `__find_lock_page()`**

This function acquires the `pagecache_lock` spinlock before calling the core function `__find_lock_page_helper()` to locate the page and lock it.

```c
struct page * __find_lock_page (struct address_space *mapping,  
   unsigned long offset, struct page **hash)  
{
    struct page *page;
    spin_lock(&pagecache_lock);
    page = __find_lock_page_helper(mapping, offset, *hash);
    spin_unlock(&pagecache_lock);
    return page;
}
```

1010 Acquire the `pagecache_lock` spinlock

1011 Call `__find_lock_page_helper()` which will search the page cache and lock the page if it is found

1012 Release the `pagecache_lock` spinlock

1013 If the page was found, return it in a locked state, otherwise return NULL

**J.1.4.6 Function: `__find_lock_page_helper()`**

This function uses `__find_page_nolock()` to locate a page within the page cache. If it is found, the page will be locked for returning to the caller.

```c
static struct page * __find_lock_page_helper(  
   struct address_space *mapping,  
   unsigned long offset, struct page *hash)
{
    struct page *page;
    /*
    * We scan the hash list read-only. Addition to and removal from
```
979  * the hash-list needs a held write-lock.
980  */
981 repeat:
982  page = __find_page_nolock(mapping, offset, hash);
983  if (page) {
984      page_cache_get(page);
985      if (TryLockPage(page)) {
986          spin_unlock(&pagecache_lock);
987          lock_page(page);
988          spin_lock(&pagecache_lock);
989  /* Has the page been re-allocated while we slept? */
990  if (page->mapping != mapping || page->index != offset) {
991      UnlockPage(page);
992      page_cache_release(page);
993      goto repeat;
994  }
995  }
996  }
997 return page;
998 }

982 Use __find_page_nolock() (See Section J.1.4.3) to locate the page in the page cache

983-984 If the page was found, take a reference to it

985 Try and lock the page with TryLockPage(). This macro is just a wrapper around test_and_set_bit() which attempts to set the PG_locked bit in the page→flags

986-988 If the lock failed, release the pagecache_lock spinlock and call lock_page() (See Section B.2.1.1) to lock the page. It is likely this function will sleep until the page lock is acquired. When the page is locked, acquire the pagecache_lock spinlock again

991 If the mapping and index no longer match, it means that this page was reclaimed while we were asleep. The page is unlocked and the reference dropped before searching the page cache again

998 Return the page in a locked state, or NULL if it was not in the page cache
J.2 LRU List Operations

Contents

J.2 LRU List Operations 535
J.2.1 Adding Pages to the LRU Lists 535
  J.2.1.1 Function: lru_cache_add() 535
  J.2.1.2 Function: add_page_to_active_list() 535
  J.2.1.3 Function: add_page_to_inactive_list() 536
J.2.2 Deleting Pages from the LRU Lists 536
  J.2.2.1 Function: lru_cache_del() 536
  J.2.2.2 Function: __lru_cache_del() 537
  J.2.2.3 Function: del_page_from_active_list() 537
  J.2.2.4 Function: del_page_from_inactive_list() 537
J.2.3 Activating Pages 538
  J.2.3.1 Function: mark_page_accessed() 538
  J.2.3.2 Function: activate_lock() 538
  J.2.3.3 Function: activate_page_nolock() 538

J.2.1 Adding Pages to the LRU Lists

J.2.1.1 Function: lru_cache_add()  (mm/swap.c)

Adds a page to the LRU inactive_list.

58 void lru_cache_add(struct page * page)  
59 {  
60     if (!PageLRU(page)) {  
61         spin_lock(&pagemap_lru_lock);  
62         if (!TestSetPageLRU(page))  
63             add_page_to_inactive_list(page);  
64         spin_unlock(&pagemap_lru_lock);  
65     }  
66 }  
67
60 If the page is not already part of the LRU lists, add it
61 Acquire the LRU lock
62-63 Test and set the LRU bit. If it was clear, call add_page_to_inactive_list()
64 Release the LRU lock

J.2.1.2 Function: add_page_to_active_list()  (include/linux/swap.h)

Adds the page to the active_list

178 #define add_page_to_active_list(page)  
179 do {  
180     spin_lock(&pagemap_lru_lock);  
181     if (!TestSetPageLRU(page))  
182         add_page_to_active_list(page);  
183     spin_unlock(&pagemap_lru_lock);  
184 }  
185
178 If the page is not already part of the LRU lists, add it
179 Acquire the LRU lock
180 Test and set the LRU bit. If it was clear, call add_page_to_active_list()
J.2.1 Adding Pages to the LRU Lists (add_page_to_active_list())

```c
#define add_page_to_active_list(page) \
    do { \
    DEBUG_LRU_PAGE(page); \ 
    SetPageActive(page); \ 
    list_add(&(page)->lru, &active_list); \ 
    nr_active_pages++; \ 
    } while (0)
```

180 The DEBUG_LRU_PAGE() macro will call BUG() if the page is already on the LRU list or is marked been active
181 Update the flags of the page to show it is active
182 Add the page to the active_list
183 Update the count of the number of pages in the active_list

J.2.1.3 Function: add_page_to_inactive_list() (include/linux/swap.h)

Adds the page to the inactive_list
```
#define add_page_to_inactive_list(page) \
    do { \
    DEBUG_LRU_PAGE(page); \ 
    list_add(&(page)->lru, &inactive_list); \ 
    nr_inactive_pages++; \ 
    } while (0)
```

188 The DEBUG_LRU_PAGE() macro will call BUG() if the page is already on the LRU list or is marked been active
189 Add the page to the inactive_list
190 Update the count of the number of inactive pages on the list

J.2.2 Deleting Pages from the LRU Lists

J.2.2.1 Function: lru_cache_del() (mm/swap.c)

Acquire the lock protecting the LRU lists before calling __lru_cache_del().
```
void lru_cache_del(struct page * page) 
{
    spin_lock(&pagemap_lru_lock);
    __lru_cache_del(page);
    spin_unlock(&pagemap_lru_lock);
}
```

92 Acquire the LRU lock

93 __lru_cache_del() does the “real” work of removing the page from the LRU lists

94 Release the LRU lock
J.2.2.2 Function: __lru_cache_del()  (mm/swap.c)

Select which function is needed to remove the page from the LRU list.

```c
75 void __lru_cache_del(struct page * page) {
76     if (TestClearPageLRU(page)) {
77         if (PageActive(page)) {
78             del_page_from_active_list(page);
79         } else {
80             del_page_from_inactive_list(page);
81         }
82     }
83 }
```

Test and clear the flag indicating the page is in the LRU

78-82 If the page is on the LRU, select the appropriate removal function

78-79 If the page is active, then call del_page_from_active_list() else delete from the inactive list with del_page_from_inactive_list()

J.2.2.3 Function: del_page_from_active_list()  (include/linux/swap.h)

Remove the page from the active_list

```c
193 #define del_page_from_active_list(page) \ 
194     do { \ 
195         list_del(&(page)->lru); \ 
196         ClearPageActive(page); \ 
197         nr_active_pages--; \ 
198     } while (0)
```

195 Delete the page from the list

196 Clear the flag indicating it is part of active_list. The flag indicating it is part of the LRU list has already been cleared by __lru_cache_del()

197 Update the count of the number of pages in the active_list

J.2.2.4 Function: del_page_from_inactive_list()  (include/linux/swap.h)

```c
200 #define del_page_from_inactive_list(page) \ 
201     do { \ 
202         list_del(&(page)->lru); \ 
203         nr_inactive_pages--; \ 
204     } while (0)
```

202 Remove the page from the LRU list

203 Update the count of the number of pages in the inactive_list
J.2.3 Activating Pages

J.2.3.1 Function: mark_page_accessed()  (mm/filemap.c)

This marks that a page has been referenced. If the page is already on the active_list or the referenced flag is clear, the referenced flag will be simply set. If it is in the inactive_list and the referenced flag has been set, activate_page() will be called to move the page to the top of the active_list.

void mark_page_accessed(struct page *page) {
    if (!PageActive(page) && PageReferenced(page)) {
        activate_page(page);
        ClearPageReferenced(page);
    } else
        SetPageReferenced(page);
}

1334-1337 If the page is on the inactive_list (!PageActive()) and has been referenced recently (PageReferenced()), activate_page() is called to move it to the active_list.

J.2.3.2 Function: activate_lock()  (mm/swap.c)

Acquire the LRU lock before calling activate_page_nolock() which moves the page from the inactive_list to the active_list.

void activate_page(struct page * page) {
    spin_lock(&pagemap_lru_lock);
    activate_page_nolock(page);
    spin_unlock(&pagemap_lru_lock);
}

47 Acquire the LRU lock

50 Call the main work function

51 Release the LRU lock

J.2.3.3 Function: activate_page_nolock()  (mm/swap.c)

Move the page from the inactive_list to the active_list.

static inline void activate_page_nolock(struct page * page) {
    if (PageLRU(page) && !PageActive(page)) {
        del_page_from_inactive_list(page);
    }
41 Make sure the page is on the LRU and not already on the \texttt{active_list}

42-43 Delete the page from the \texttt{inactive_list} and add to the \texttt{active_list}
J.3 Refilling *inactive_list*

This section covers how pages are moved from the active lists to the inactive lists.

### J.3.1 Function: refill_inactive() *(mm/vmscan.c)*

Move `nr_pages` from the `active_list` to the `inactive_list`. The parameter `nr_pages` is calculated by `shrink_caches()` and is a number which tries to keep the active list two thirds the size of the page cache.

```c
533 static void refill_inactive(int nr_pages) {
534      struct list_head * entry;
535    spin_lock(&pagemap_lru_lock);
536    entry = active_list.prev;
537    while (nr_pages && entry != &active_list) {
538        struct page * page;
539        page = list_entry(entry, struct page, lru);
540        entry = entry->prev;
541        if (PageTestandClearReferenced(page)) {
542            list_del(&page->lru);
543            list_add(&page->lru, &active_list);
544            continue;
545        }
546        nr_pages--;
547    }
548    spin_unlock(&pagemap_lru_lock);
549 }
```

537 Acquire the lock protecting the LRU list

538 Take the last entry in the `active_list`

539-555 Move `nr_pages` or until the `active_list` is empty

542 Get the `struct page` for this entry
544-548 Test and clear the referenced flag. If it has been referenced, then it is moved back to the top of the active_list.

550-553 Move one page from the active_list to the inactive_list.

554 Mark it referenced so that if it is referenced again soon, it will be promoted back to the active_list without requiring a second reference.

556 Release the lock protecting the LRU list.
J.4 Reclaiming Pages from the LRU Lists

Contents

J.4 Reclaiming Pages from the LRU Lists 542
J.4.1 Function: shrink_cache() 542

This section covers how a page is reclaimed once it has been selected for pageout.

J.4.1 Function: shrink_cache() (mm/vmscan.c)

338 static int shrink_cache(int nr_pages, zone_t * classzone,
339     unsigned int gfp_mask, int priority)
340 {
341     struct list_head * entry;
342     int max_scan = nr_inactive_pages / priority;
343     int max_mapped = min((nr_pages << (10 - priority)),
344             max_scan / 10);
345     spin_lock(&pagemap_lru_lock);
346     while (--max_scan >= 0 &&
347             (entry = inactive_list.prev) != &inactive_list) {
348         The parameters are as follows;

         nr_pages The number of pages to swap out
         classzone The zone we are interested in swapping pages out for. Pages not
         belonging to this zone are skipped
         gfp_mask The gfp mask determining what actions may be taken such as if
         filesystem operations may be performed
         priority The priority of the function, starts at DEF_PRIORITY (6) and de-
         creases to the highest priority of 1

341 The maximum number of pages to scan is the number of pages in the
342 active_list divided by the priority. At lowest priority, 1/6th of the list
343 may scanned. At highest priority, the full list may be scanned

342 The maximum amount of process mapped pages allowed is either one tenth
343 of the max_scan value or nr_pages * 2^{10\text{--}priority}. If this number of pages are
344 found, whole processes will be swapped out

344 Lock the LRU list

345 Keep scanning until max_scan pages have been scanned or the inactive_list
346 is empty
struct page * page;

if (unlikely(current->need_resched)) {
    spin_unlock(&pagemap_lru_lock);
    __set_current_state(TASK_RUNNING);
    schedule();
    spin_lock(&pagemap_lru_lock);
    continue;
}

Reschedule if the quanta has been used up

Free the LRU lock as we are about to sleep

Show we are still running

Call schedule() so another process can be context switched in

Re-acquire the LRU lock

Reiterate through the loop and take an entry inactive_list again. As we slept, another process could have changed what entries are on the list which is why another entry has to be taken with the spinlock held

page = list_entry(entry, struct page, lru);

BUG_ON(!PageLRU(page));
BUG_ON(PageActive(page));

list_del(entry);
list_add(entry, &inactive_list);

/*
 * Zero page counts can happen because we unlink the pages
 * _after_ decrementing the usage count..
 */
if (unlikely(!page_count(page))
    continue;
if (!memclass(page_zone(page), classzone))
    continue;

/* Racy check to avoid trylocking when not worthwhile */
if (!page->buffers && (page_count(page) != 1 || !page->mapping))
    goto page_mapped;
J.4 Reclaiming Pages from the LRU Lists (shrink_cache() 

Get the struct page for this entry in the LRU

358-359 It is a bug if the page either belongs to the active_list or is currently marked as active

361-362 Move the page to the top of the inactive_list so that if the page is not freed, we can just continue knowing that it will be simply examined later

368-369 If the page count has already reached 0, skip over it. In __free_pages(), the page count is dropped with put_page_testzero() before __free_pages_ok() is called to free it. This leaves a window where a page with a zero count is left on the LRU before it is freed. There is a special case to trap this at the beginning of __free_pages_ok()

371-372 Skip over this page if it belongs to a zone we are not currently interested in

375-376 If the page is mapped by a process, then goto page_mapped where the max_mapped is decremented and next page examined. If max_mapped reaches 0, process pages will be swapped out

382 if (unlikely(TryLockPage(page))) {
383     if (PageLaunder(page) && (gfp_mask & __GFP_FS)) {
384         page_cache_get(page);
385         spin_unlock(&pagemap_lru_lock);
386         wait_on_page(page);
387         page_cache_release(page);
388         spin_lock(&pagemap_lru_lock);
389     }
390     continue;
391 }

Page is locked and the launder bit is set. In this case, it is the second time this page has been found dirty. The first time it was scheduled for IO and placed back on the list. This time we wait until the IO is complete and then try to free the page.

382-383 If we could not lock the page, the PG_launder bit is set and the GFP flags allow the caller to perform FS operations, then...

384 Take a reference to the page so it does not disappear while we sleep
385 Free the LRU lock
386 Wait until the IO is complete
387 Release the reference to the page. If it reaches 0, the page will be freed
388 Re-acquire the LRU lock
if (PageDirty(page) &&
is_page_cache_freeable(page) &&
page->mapping) {
    /*
     * It is not critical here to write it only if
     * the page is unmapped because any direct writer
     * like O_DIRECT would set the PG_dirty bitflag
     * on the physical page after having successfully
     * pinned it and after the I/O to the page is finished,
     * so the direct writes to the page cannot get lost.
     */
    int (*writepage)(struct page *);

    writepage = page->mapping->a_ops->writepage;
    if ((gfp_mask & __GFP_FS) && writepage) {
        ClearPageDirty(page);
        SetPageLaunder(page);
        page_cache_get(page);
        spin_unlock(&pagemap_lru_lock);

        writepage(page);
        page_cache_release(page);

        spin_lock(&pagemap_lru_lock);
        continue;
    }
}

This handles the case where a page is dirty, is not mapped by any process, has
no buffers and is backed by a file or device mapping. The page is cleaned and will
be reclaimed by the previous block of code when the IO is complete.

PageDirty() checks the PG_dirty bit, is_page_cache_freeable() will return true if it is not mapped by any process and has no buffers

Get a pointer to the necessary writepage() function for this mapping or
device

This block of code can only be executed if a writepage() function is
available and the GFP flags allow file operations

Clear the dirty bit and mark that the page is being laundered

Take a reference to the page so it will not be freed unexpectedly
Unlock the LRU list

Call the filesystem-specific writepage() function which is taken from the address_space_operations belonging to page→mapping

Release the reference to the page

Re-acquire the LRU list lock and move to the next page

if (page->buffers) {
    spin_unlock(&pagemap_lru_lock);
    /* avoid to free a locked page */
    page_cache_get(page);
    if (try_to_release_page(page, gfp_mask)) {
        if (!page->mapping) {
            spin_lock(&pagemap_lru_lock);
            UnlockPage(page);
            __lru_cache_del(page);
            /* effectively free the page here */
            page_cache_release(page);
            if (--nr_pages)
                continue;
            break;
        } else {
            page_cache_release(page);
            spin_lock(&pagemap_lru_lock);
        }
    } else {
        /* failed to drop the buffers so stop here */
        UnlockPage(page);
        page_cache_release(page);
        spin_lock(&pagemap_lru_lock);
        continue;
    }
}

Page has buffers associated with it that must be freed.

Release the LRU lock as we may sleep

Take a reference to the page
430 Call `try_to_release_page()` which will attempt to release the buffers associated with the page. Returns 1 if it succeeds.

431-447 This is a case where an anonymous page that was in the swap cache has now had its buffers cleared and removed. As it was on the swap cache, it was placed on the LRU by `add_to_swap_cache()` so remove it now from the LRU and drop the reference to the page. In `swap_writepage()`, it calls `remove_exclusive_swap_page()` which will delete the page from the swap cache when there are no more processes mapping the page. This block will free the page after the buffers have been written out if it was backed by a swap file.

438-440 Take the LRU list lock, unlock the page, delete it from the page cache and free it.

445-446 Update `nr_pages` to show a page has been freed and move to the next page.

447 If `nr_pages` drops to 0, then exit the loop as the work is completed.

449-456 If the page does have an associated mapping then simply drop the reference to the page and re-acquire the LRU lock. More work will be performed later to remove the page from the page cache at line 499.

459-464 If the buffers could not be freed, then unlock the page, drop the reference to it, re-acquire the LRU lock and move to the next page.

```c
468       spin_lock(&pagecache_lock);
469
470      /*
471     * this is the non-racy check for busy page.
472      */
473      if (!page->mapping || !is_page_cache_freeable(page)) {
474        spin_unlock(&pagecache_lock);
475        UnlockPage(page);
476        page_mapped:
477        if (--max_mapped >= 0)
478          continue;
479      }
480      spin_unlock(&pagemap_lru_lock);
481      swap_out(priority, gfp_mask, classzone);
482      return nr_pages;
483    }
```

468 From this point on, pages in the swap cache are likely to be examined which is protected by the `pagecache_lock` which must be now held.

473-487 An anonymous page with no buffers is mapped by a process.
Reclaiming Pages from the LRU Lists (\texttt{shrink_cache()})

474-475 Release the page cache lock and the page

477-478 Decrement \texttt{max\_mapped}. If it has not reached 0, move to the next page

484-485 Too many mapped pages have been found in the page cache. The LRU lock is released and \texttt{swap\_out()} is called to begin swapping out whole processes

493-497 The page has no references but could have been dirtied by the last process to free it if the dirty bit was set in the PTE. It is left in the page cache and will get laundered later. Once it has been cleaned, it can be safely deleted

500-503 If the page does not belong to the swap cache, it is part of the inode queue so it is removed

504-508 Remove it from the swap cache as there is no more references to it

511 Delete it from the page cache
Unlock the page
Free the page
Decrement nr_page and move to the next page if it is not 0
If it reaches 0, the work of the function is complete

Function exit. Free the LRU lock and return the number of pages left to free
J.5 Shrinking all caches

Contents

J.5 Shrinking all caches 550
  J.5.1 Function: shrink_caches()  550
  J.5.2 Function: try_to_free_pages()  551
  J.5.3 Function: try_to_free_pages_zone()  552

J.5.1 Function: shrink_caches()  (mm/vmscan.c)

The call graph for this function is shown in Figure 10.4.

560 static int shrink_caches(zone_t * classzone, int priority,
                             unsigned int gfp_mask, int nr_pages)
561 {
562   int chunk_size = nr_pages;
563   unsigned long ratio;
564   nr_pages -= kmem_cache_reap(gfp_mask);
565   if (nr_pages <= 0)
566     return 0;
567   nr_pages = chunk_size;
568   /* try to keep the active list 2/3 of the size of the cache */
569   ratio = (unsigned long) nr_pages *
570          nr_active_pages / ((nr_inactive_pages + 1) * 2);
571   refill_inactive(ratio);
572   nr_pages = shrink_cache(nr_pages, classzone, gfp_mask, priority);
573   if (nr_pages <= 0)
574     return 0;
575   shrink_dcache_memory(priority, gfp_mask);
576   shrink_icache_memory(priority, gfp_mask);
577   #ifdef CONFIG_QUOTA
578     shrink_dqcache_memory(DEF_PRIORITY, gfp_mask);
579   #endif
580   return nr_pages;
581 }

560 The parameters are as follows;

  classzone is the zone that pages should be freed from
  priority determines how much work will be done to free pages
  gfp_mask determines what sort of actions may be taken
nr_pages is the number of pages remaining to be freed

565-567 Ask the slab allocator to free up some pages with kmem_cache_reap() (See Section H.1.5.1). If enough are freed, the function returns otherwise nr_pages will be freed from other caches

571-572 Move pages from the active_list to the inactive_list by calling refill_inactive() (See Section J.3.1). The number of pages moved depends on how many pages need to be freed and to have active_list about two thirds the size of the page cache

574-575 Shrink the page cache, if enough pages are freed, return

578-582 Shrink the dcache, icache and dqcache. These are small objects in themselves but the cascading effect frees up a lot of disk buffers

584 Return the number of pages remaining to be freed

J.5.2 Function: try_to_free_pages() (mm/vmscan.c)

This function cycles through all pgdat's and tries to balance the preferred allocation zone (usually ZONE_NORMAL) for each of them. This function is only called from one place, buffer.c:free_more_memory() when the buffer manager fails to create new buffers or grow existing ones. It calls try_to_free_pages() with GFP_NOIO as the gfp_mask.

This results in the first zone in pg_data_t—node_zonelists having pages freed so that buffers can grow. This array is the preferred order of zones to allocate from and usually will begin with ZONE_NORMAL which is required by the buffer manager. On NUMA architectures, some nodes may have ZONE_DMA as the preferred zone if the memory bank is dedicated to IO devices and UML also uses only this zone. As the buffer manager is restricted in the zones it uses, there is no point balancing other zones.

607 int try_to_free_pages(unsigned int gfp_mask)
608 {
609    pg_data_t *pgdat;
610    zonelist_t *zonelist;
611    unsigned long pf_free_pages;
612    int error = 0;
613    pf_free_pages = current->flags & PF_FREE_PAGES;
614    current->flags &= ~PF_FREE_PAGES;
615    for_each_pgdat(pgdat) {
616        zonelist = pgdat->node_zonelists +
617                  (gfp_mask & GFP_ZONEMASK);
618        error |= try_to_free_pages_zone(
J.5 Shrinking all caches (try_to_free_pages())

614-615 This clears the PF_FREE_PAGES flag if it is set so that pages freed by the process will be returned to the global pool rather than reserved for the process itself.

617-620 Cycle through all nodes and call try_to_free_pages() for the preferred zone in each node.

618 This function is only called with GFP_NOIO as a parameter. When ANDed with GFP_ZONE_MASK, it will always result in 0.

622-623 Restore the process flags and return the result.

J.5.3 Function: try_to_free_pages_zone() (mm/vmscan.c)

Try to free SWAP_CLUSTER_MAX pages from the requested zone. As well as being used by kswapd, this function is the entry for the buddy allocator’s direct-reclaim path.

587 int try_to_free_pages_zone(zone_t *classzone,
                    unsigned int gfp_mask)
588 {
589     int priority = DEF_PRIORITY;
590     int nr_pages = SWAP_CLUSTER_MAX;
591     gfp_mask = pf_gfp_mask(gfp_mask);
592     do {
593         nr_pages = shrink_caches(classzone, priority, gfp_mask, nr_pages);
594         if (nr_pages <= 0)
595             return 1;
596     } while (--priority);
597     /* Hmm.. Cache shrink failed - time to kill something? */
598     /* Mhwahahhaha! This is the part I really like. Giggle. */
599     out_of_memory();
600     return 0;
601 }
J.5 Shrinking all caches (try_to_free_pages_zone())

589 Start with the lowest priority. Statically defined to be 6

590 Try and free SWAP_CLUSTER_MAX pages. Statically defined to be 32

592 pf_gfp_mask() checks the PF_NOIO flag in the current process flags. If no IO can be performed, it ensures there is no incompatible flags in the GFP mask

593-597 Starting with the lowest priority and increasing with each pass, call shrink_caches() until nr_pages has been freed

595-596 If enough pages were freed, return indicating that the work is complete

603 If enough pages could not be freed even at highest priority (where at worst the full inactive_list is scanned) then check to see if we are out of memory. If we are, then a process will be selected to be killed

604 Return indicating that we failed to free enough pages
J.6 Swapping Out Process Pages

Contents

J.6 Swapping Out Process Pages 554
J.6.1 Function: swap_out() 554
J.6.2 Function: swap_out_mm() 556
J.6.3 Function: swap_out_vma() 557
J.6.4 Function: swap_out_pgd() 558
J.6.5 Function: swap_out_pmd() 559
J.6.6 Function: try_to_swap_out() 561

This section covers the path where too many process mapped pages have been found in the LRU lists. This path will start scanning whole processes and reclaiming the mapped pages.

J.6.1 Function: swap_out() (mm/vmscan.c)

The call graph for this function is shown in Figure 10.5. This function linearly searches through every processes page tables trying to swap out SWAP_CLUSTER_MAX number of pages. The process it starts with is the swap_mm and the starting address is mm→swap_address

296 static int swap_out(unsigned int priority, unsigned int gfp_mask, zone_t * classzone)
297 {
298     int counter, nr_pages = SWAP_CLUSTER_MAX;
299     struct mm_struct *mm;
300
301     counter = mmlist_nr;
302     do {
303         if (unlikely(current->need_resched)) {
304             __set_current_state(TASK_RUNNING);
305             schedule();
306         }
307     }
308     spin_lock(&mmlist_lock);
309     mm = swap_mm;
310     while (mm->swap_address == TASK_SIZE || mm == &init_mm) {
311         mm->swap_address = 0;
312         mm = list_entry(mm->mmlist.next, struct mm_struct, mmlist);
313         if (mm == swap_mm)
314             goto empty;
315     }
316     swap_mm = mm;
317
318     /* Make sure the mm doesn’t disappear
J.6 Swapping Out Process Pages (`swap_out()`) 555

```c
when we drop the lock.. */
319    atomic_inc(&mm->mm_users);
320    spin_unlock(&mmlist_lock);
321
322    nr_pages = swap_out_mm(mm, nr_pages, &counter, classzone);
323
324    mmput(mm);
325
326    if (!nr_pages)
327        return 1;
328    } while (--counter >= 0);
329
330    return 0;
331
332 empty:
333    spin_unlock(&mmlist_lock);
334    return 0;
335 }
```

301 Set the counter so the process list is only scanned once

303-306 Reschedule if the quanta has been used up to prevent CPU hogging

308 Acquire the lock protecting the mm list

309 Start with the `swap_mm`. It is interesting this is never checked to make sure it is valid. It is possible, albeit unlikely that the process with the mm has exited since the last scan and the slab holding the mm `struct` has been reclaimed during a cache shrink making the pointer totally invalid. The lack of bug reports might be because the slab rarely gets reclaimed and would be difficult to trigger in reality

310-316 Move to the next process if the `swap_address` has reached the `TASK_SIZE` or if the mm is the `init_mm`

311 Start at the beginning of the process space

312 Get the mm for this process

313-314 If it is the same, there is no running processes that can be examined

315 Record the `swap_mm` for the next pass

319 Increase the reference count so that the mm does not get freed while we are scanning

320 Release the mm lock

322 Begin scanning the mm with `swap_out_mm()` (See Section J.6.2)
J.6 Swapping Out Process Pages (\texttt{swap\_out()})

324 Drop the reference to the mm

326-327 If the required number of pages has been freed, return success

328 If we failed on this pass, increase the priority so more processes will be scanned

330 Return failure

J.6.2 Function: \texttt{swap\_out\_mm()} \textit{(mm/vmscan.c)}
Walk through each VMA and call \texttt{swap\_out\_mm()} for each one.

256 static inline int swap_out_mm(struct mm_struct * mm, int count, int * mmcounter, zone_t * classzone)
257 {
258     unsigned long address;
259     struct vm_area_struct* vma;
260
265     spin_lock(&mm->page_table_lock);
266     address = mm->swap_address;
267     if (address == TASK_SIZE || swap_mm != mm) {
268         /* We raced: don’t count this mm but try again */
269         ++*mmcounter;
270         goto out_unlock;
271     }
272     vma = find_vma(mm, address);
273     if (vma) {
274         if (address < vma->vm_start)
275             address = vma->vm_start;
276
277         for (;;) {
278             count = swap_out_vma(mm, vma, address,
279                 count, classzone);
280             vma = vma->vm_next;
281             if (!vma)
282                 break;
283             if (!count)
284                 goto out_unlock;
285             address = vma->vm_start;
286         }
287         /* Indicate that we reached the end of address space */
288         mm->swap_address = TASK_SIZE;
289     } out_unlock:
290     spin_unlock(&mm->page_table_lock);
291     return count;
292 }
Acquire the page table lock for this mm

Start with the address contained in swap_address

If the address is TASK_SIZE, it means that a thread raced and scanned this process already. Increase mmcounter so that swap_out_mm() knows to go to another process

Find the VMA for this address

Presuming a VMA was found then ....

Start at the beginning of the VMA

Scan through this and each subsequent VMA calling swap_out_vma() (See Section J.6.3) for each one. If the requisite number of pages (count) is freed, then finish scanning and return

Once the last VMA has been scanned, set swap_address to TASK_SIZE so that this process will be skipped over by swap_out_mm() next time

J.6.3 Function: swap_out_vma()  (mm/vmscan.c)  
Walk through this VMA and for each PGD in it, call swap_out_pgd().

```c
static inline int swap_out_vma(struct mm_struct * mm,  
    struct vm_area_struct * vma,  
    unsigned long address, int count,  
    zone_t * classzone)
{
    pgd_t *pgdir;
    unsigned long end;

    /* Don't swap out areas which are reserved */
    if (vma->vm_flags & VM_RESERVED)
        return count;

    pgdir = pgd_offset(mm, address);
    end = vma->vm_end;
    BUG_ON(address >= end);
    do {
        count = swap_out_pgd(mm, vma, pgdir,  
            address, end, count, classzone);
        if (!count)
            break;

        address = (address + PGDIR_SIZE) & PGDIR_MASK;
        pgdir++;
```
246  } while (address && (address < end));
247  return count;
248 }

233-234 Skip over this VMA if the VM_RESERVED flag is set. This is used by some
device drivers such as the SCSI generic driver.

236 Get the starting PGD for the address.

238 Mark where the end is and BUG() it if the starting address is somehow past
the end.

240 Cycle through PGDs until the end address is reached.

241 Call swap_out_pgd() (See Section J.6.4) keeping count of how many more
pages need to be freed.

242-243 If enough pages have been freed, break and return.

244-245 Move to the next PGD and move the address to the next PGD aligned
address.

247 Return the remaining number of pages to be freed.

J.6.4 Function: swap_out_pgd() (mm/vmscan.c)
Step through all PMD’s in the supplied PGD and call swap_out_pmd().

197 static inline int swap_out_pgd(struct mm_struct * mm,
        struct vm_area_struct * vma, pgd_t *dir,
        unsigned long address, unsigned long end,
        int count, zone_t * classzone)
198 {
199   pmd_t * pmd;
200   unsigned long pgd_end;
201   if (pgd_none(*dir))
202     return count;
203   if (pgd_bad(*dir)) {
204     pgd_ERROR(*dir);
205     pgd_clear(dir);
206     return count;
207   }
208 
209   pmd = pmd_offset(dir, address);
210   pgd_end = (address + PGDIR_SIZE) & PGDIR_MASK;
211   if (pgd_end && (end > pgd_end))
J.6 Swappig Out Process Pages (swap_out_pgd())

214 end = pgd_end;
215
216 do {
217 count = swap_out_pmd(mm, vma, pmd,
218 address, end, count, classzone);
219 if (!count)
220 break;
221 address = (address + PMD_SIZE) & PMD_MASK;
222 pmd++;
223 } while (address && (address < end));
224 return count;

202-203 If there is no PGD, return
204-208 If the PGD is bad, flag it as such and return
210 Get the starting PMD
212-214 Calculate the end to be the end of this PGD or the end of the VMA been scanned, whichever is closer
216-222 For each PMD in this PGD, call swap_out_pmd() (See Section J.6.5). If enough pages get freed, break and return
223 Return the number of pages remaining to be freed

J.6.5 Function: swap_out_pmd() (mm/vmscan.c)
For each PTE in this PMD, call try_to_swap_out(). On completion, mm→swap_address is updated to show where we finished to prevent the same page been examined soon after this scan.

158 static inline int swap_out_pmd(struct mm_struct * mm,
159 struct vm_area_struct * vma, pmd_t *dir,
160 unsigned long address, unsigned long end,
161 int count, zone_t * classzone)
162 {
163 pte_t * pte;
164 unsigned long pmd_end;
165 if (pmd_none(*dir))
166 return count;
167 if (pmd_bad(*dir)) {
168 pmd_ERROR(*dir);
169 pmd_clear(dir);
170 return count;
171 }


169  
170  pte = pte_offset(dir, address);
171  
172  pmd_end = (address + PMD_SIZE) & PMD_MASK;
173  
174  if (end > pmd_end)
175      end = pmd_end;
176  
177  do {
178      if (pte_present(*pte)) {
179          struct page *page = pte_page(*pte);
180          
181          if (VALID_PAGE(page) && !PageReserved(page)) {
182              count -= try_to_swap_out(mm, vma,
183                  address, pte,
184                  page, classzone);
185              if (!count) {
186                  address += PAGE_SIZE;
187                  break;
188              }
189          }
190      }
191  } while (address && (address < end));
192  
193  mm->swap_address = address;
194  
195  return count;

163-164 Return if there is no PMD
165-169 If the PMD is bad, flag it as such and return
171 Get the starting PTE
173-175 Calculate the end to be the end of the PMD or the end of the VMA, whichever is closer
177-191 Cycle through each PTE
178 Make sure the PTE is marked present
179 Get the struct page for this PTE
181 If it is a valid page and it is not reserved then ...
182 Call try_to_swap_out()
183-186 If enough pages have been swapped out, move the address to the next page and break to return.

189-190 Move to the next page and PTE.

192 Update the swap_address to show where we last finished off.

193 Return the number of pages remaining to be freed.

J.6.6 Function: try_to_swap_out() (mm/vmscan.c)

This function tries to swap out a page from a process. It is quite a large function so will be dealt with in parts. Broadly speaking they are:

- Function preamble, ensure this is a page that should be swapped out
- Remove the page and PTE from the page tables
- Handle the case where the page is already in the swap cache
- Handle the case where the page is dirty or has associated buffers
- Handle the case where the page has been added to the swap cache

```c
47 static inline int try_to_swap_out(struct mm_struct * mm,
                      struct vm_area_struct* vma,
                      unsigned long address,
                      pte_t * page_table,
                      struct page * page,
                      zone_t * classzone)
48 {  
    pte_t pte;
    swp_entry_t entry;
    
    /* Don't look at this pte if it's been accessed recently. */
    if ((vma->vm_flags & VM_LOCKED) ||
            ptep_test_and_clear_young(page_table)) {
        mark_page_accessed(page);
        return 0;
    }

    /* Don't bother unmapping pages that are active */
    if (PageActive(page))
        return 0;

    /* Don't bother replenishing zones not under pressure.. */
    if (!memclass(page_zone(page), classzone))
        return 0;
```
if (TryLockPage(page))
    return 0;

53-56 If the page is locked (for tasks like IO) or the PTE shows the page has been accessed recently then clear the referenced bit and call mark_page_accessed() (See Section J.2.3.1) to make the struct page reflect the age. Return 0 to show it was not swapped out.

59-60 If the page is on the active_list, do not swap it out.

63-64 If the page belongs to a zone we are not interested in, do not swap it out.

66-67 If the page is already locked for IO, skip it.

74 Call the architecture hook to flush this page from all CPUs

75 Get the PTE from the page tables and clear it

76 Call the architecture hook to flush the TLB

78-79 If the PTE was marked dirty, mark the struct page dirty so it will be laundered correctly.

86 if (PageSwapCache(page)) {
    entry.val = page->index;
    swap_duplicate(entry);
    set_swap_pte:
    set_pte(page_table, swp_entry_to_pte(entry));
    drop_pte:
    mm->rss--;
    UnlockPage(page);
    { int freeable =
        page_count(page) - !!page->buffers <= 2;
        page_cache_release(page);
        return freeable;
    }
}
Handle the case where the page is already in the swap cache

86 Enter this block only if the page is already in the swap cache. Note that it can also be entered by calling goto to the set_swap_pte and drop_pte labels

87-88 Fill in the index value for the swap entry. swap_duplicate() verifies the swap identifier is valid and increases the counter in the swap_map if it is

90 Fill the PTE with information needed to get the page from swap

92 Update RSS to show there is one less page being mapped by the process

93 Unlock the page

95 The page is free-able if the count is currently 2 or less and has no buffers. If the count is higher, it is either being mapped by other processes or is a file-backed page and the "user" is the page cache

96 Decrement the reference count and free the page if it reaches 0. Note that if this is a file-backed page, it will not reach 0 even if there are no processes mapping it. The page will be later reclaimed from the page cache by shrink_cache() (See Section J.4.1)

97 Return if the page was freed or not

115 if (page->mapping)
116 goto drop_pte;
117 if (!PageDirty(page))
118 goto drop_pte;
124 if (page->buffers)
125 goto preserve;

115-116 If the page has an associated mapping, simply drop it from the page tables. When no processes are mapping it, it will be reclaimed from the page cache by shrink_cache()

117-118 If the page is clean, it is safe to simply drop it

124-125 If it has associated buffers due to a truncate followed by a page fault, then re-attach the page and PTE to the page tables as it cannot be handled yet

126
127 /*
128 * This is a dirty, swappable page. First of all,
129 * get a suitable swap entry for it, and make sure
130 * we have the swap cache set up to associate the
131 * page with that swap entry.
J.6 Swapping Out Process Pages (try_to_swap_out())

132        */
133        for (;;) {
134            entry = get_swap_page();
135            if (!entry.val)
136                break;
137        /* Add it to the swap cache and mark it dirty
138         * (adding to the page cache will clear the dirty
139         * and uptodate bits, so we need to do it again)
140         */
141            if (add_to_swap_cache(page, entry) == 0) {
142                SetPageUptodate(page);
143                set_page_dirty(page);
144                goto set_swap_pte;
145            }
146        /* Raced with "speculative" read_swap_cache_async */
147            swap_free(entry);
148        }
149
150        /* No swap space left */
151     preserve:
152            set_pte(page_table, pte);
153            UnlockPage(page);
154            return 0;
155    }

134 Allocate a swap entry for this page
135-136 If one could not be allocated, break out where the PTE and page will be
re-attached to the process page tables
141 Add the page to the swap cache
142 Mark the page as up to date in memory
143 Mark the page dirty so that it will be written out to swap soon
144 Goto set_swap_pte which will update the PTE with information needed to
get the page from swap later
147 If the add to swap cache failed, it means that the page was placed in the swap
cache already by a readahead so drop the work done here
152 Reattach the PTE to the page tables
153 Unlock the page
154 Return that no page was freed
J.7 Page Swap Daemon

Contents

J.7 Page Swap Daemon

J.7.1 Initialising kswapd

J.7.1.1 Function: kswapd_init() (mm/vmscan.c)
Start the kswapd kernel thread

static int __init kswapd_init(void)
{
    printk("Starting kswapd\n");
    swap_setup();
    kernel_thread(kswapd, NULL, CLONE_FS
                 | CLONE_FILES
                 | CLONE_SIGNAL);

    return 0;
}

J.7.2 kswapd Daemon

J.7.2.1 Function: kswapd() (mm/vmscan.c)
The main function of the kswapd kernel thread.

int kswapd(void *unused)
{
    struct task_struct *tsk = current;
    DECLARE_WAITQUEUE(wait, tsk);

This section details the main loops used by the kswapd daemon which is woken-up when memory is low. The main functions covered are the ones that determine if kswapd can sleep and how it determines which nodes need balancing.
daemonize();
strcpy(tsk->comm, "kswapd");
sigfillset(&tsk->blocked);

tsk->flags |= PF_MEMALLOC;

for (;;) {
    __set_current_state(TASK_INTERRUPTIBLE);
    add_wait_queue(&kswapd_wait, &wait);
    mb();
    if (kswapd_can_sleep())
        schedule();
    __set_current_state(TASK_RUNNING);
    remove_wait_queue(&kswapd_wait, &wait);
    kswapd_balance();
    run_task_queue(&tq_disk);
}

Call daemonize() which will make this a kernel thread, remove the mm context, close all files and re-parent the process.

Set the name of the process

Ignore all signals

By setting this flag, the physical page allocator will always try to satisfy requests for pages. As this process will always be trying to free pages, it is worth satisfying requests.

Endlessly loop

This adds kswapd to the wait queue in preparation to sleep

The Memory Block function (mb()) ensures that all reads and writes that occurred before this line will be visible to all CPU’s

kswapd_can_sleep() (See Section J.7.2.2) cycles through all nodes and zones checking the need_balance field. If any of them are set to 1, kswapd can not sleep

By calling schedule(), kswapd will now sleep until woken again by the physical page allocator in __alloc_pages() (See Section F.1.3)
Once woken up, **kswpd** is removed from the wait queue as it is now running.

**kswpd_balance()** (See Section J.7.2.4) cycles through all zones and calls **try_to_free_pages_zone()** (See Section J.5.3) for each zone that requires balance.

Run the IO task queue to start writing data out to disk.

### J.7.2.2 Function: **kswpd_can_sleep()** *(mm/vmscan.c)*

Simple function to cycle through all pgdats to call **kswpd_can_sleep_pgdat()** on each.

```c
static int kswpd_can_sleep(void)
{
    pg_data_t * pgdat;

    for_each_pgdat(pgdat) {
        if (!kswpd_can_sleep_pgdat(pgdat))
            return 0;
    }

    return 1;
}
```

**for_each_pgdat()** does exactly as the name implies. It cycles through all available pgdat’s and in this case calls **kswpd_can_sleep_pgdat()** (See Section J.7.2.3) for each. On the x86, there will only be one pgdat.

### J.7.2.3 Function: **kswpd_can_sleep_pgdat()** *(mm/vmscan.c)*

Cycles through all zones to make sure none of them need balance. The **zone→need_balance** flag is set by **__alloc_pages()** when the number of free pages in the zone reaches the **pages_low** watermark.

```c
static int kswpd_can_sleep_pgdat(pg_data_t * pgdat)
{
    zone_t * zone;
    int i;

    for (i = pgdat->nr_zones-1; i >= 0; i--) {
        zone = pgdat->node_zones + i;
        if (!zone->need_balance)
            continue;
        return 0;
    }
```
J.7.2 kswapd Daemon (

J.7.2.4 Function: kswapd_balance() (mm/vmscan.c)

Continuously cycle through each pgdat until none require balancing

J.7.2.5 Function: kswapd_balance_pgdat() (mm/vmscan.c)

This function will check if a node requires balance by examining each of the nodes in it. If any zone requires balancing, try_to_free_pages_zone() will be called.

```c
static int kswapd_balance_pgdat(pg_data_t * pgdat)
{
    int need_more_balance = 0, i;
    zone_t * zone;
```
for (i = pgdat->nr_zones-1; i >= 0; i--) {
    zone = pgdat->node_zones + i;
    if (unlikely(current->need_resched))
        schedule();
    if (!zone->need_balance)
        continue;
    if (!try_to_free_pages_zone(zone, GFP_KSWAPD)) {
        zone->need_balance = 0;
        __set_current_state(TASK_INTERRUPTIBLE);
        schedule_timeout(HZ);
        continue;
    }
    if (check_classzone_need_balance(zone))
        need_more_balance = 1;
    else
        zone->need_balance = 0;
}
return need_more_balance;}

Cycle through each zone and call try_to_free_pages_zone() (See Section J.5.3) if it needs re-balancing

node_zones is an array and i is an index within it

Call schedule() if the quanta is expired to prevent kswapd hogging the CPU

If the zone does not require balance, move to the next one

If the function returns 0, it means the out_of_memory() function was called because a sufficient number of pages could not be freed. kswapd sleeps for 1 second to give the system a chance to reclaim the killed processes pages and perform IO. The zone is marked as balanced so kswapd will ignore this zone until the the allocator function __alloc_pages() complains again

If is was successful, check_classzone_need_balance() is called to see if the zone requires further balancing or not

Return 1 if one zone requires further balancing
Appendix K

Swap Management

Contents

K.1 Scanning for Free Entries ........................................ 572
  K.1.1 Function: get_swap_page() ................................. 572
  K.1.2 Function: scan_swap_map() ................................. 574
K.2 Swap Cache ......................................................... 577
  K.2.1 Adding Pages to the Swap Cache ......................... 577
    K.2.1.1 Function: add_to_swap_cache() ....................... 577
    K.2.1.2 Function: swap_duplicate() .......................... 578
  K.2.2 Deleting Pages from the Swap Cache .................... 580
    K.2.2.1 Function: swap_free() ................................ 580
    K.2.2.2 Function: swap_entry_free() ......................... 580
  K.2.3 Acquiring/Releasing Swap Cache Pages ................... 581
    K.2.3.1 Function: swap_info_get() ........................... 581
    K.2.3.2 Function: swap_info_put() ........................... 582
  K.2.4 Searching the Swap Cache .................................. 583
    K.2.4.1 Function: lookup_swap_cache() ....................... 583
K.3 Swap Area IO ....................................................... 584
  K.3.1 Reading Backing Storage ................................... 584
    K.3.1.1 Function: read_swap_cache_async() .................. 584
  K.3.2 Writing Backing Storage .................................... 586
    K.3.2.1 Function: swap_writepage() ......................... 586
    K.3.2.2 Function: remove_exclusive_swap_page() .......... 586
    K.3.2.3 Function: free_swap_and_cache() .................... 588
  K.3.3 Block IO ....................................................... 589
    K.3.3.1 Function: rw_swap_page() ............................ 589

570
APPENDIX K. SWAP MANAGEMENT

K.3.3.2 Function: rw_swap_page_base() ..................... 590
K.3.3.3 Function: get_swaphandle_info() .................. 592

K.4 Activating a Swap Area .................................. 594
K.4.1 Function: sys_swapon() ................................. 594
K.4.2 Function: swap_setup() ................................. 605

K.5 Deactivating a Swap Area ................................ 606
K.5.1 Function: sys_swapoff() ................................. 606
K.5.2 Function: try_to_unuse() ............................... 610
K.5.3 Function: unuse_process() .............................. 615
K.5.4 Function: unuse_vma() ................................ 616
K.5.5 Function: unuse_pgd() ................................ 616
K.5.6 Function: unuse_pmd() ................................ 618
K.5.7 Function: unuse_pte() ................................ 619
K.1 Scanning for Free Entries

Contents

K.1 Scanning for Free Entries 572
  K.1.1 Function: get_swap_page()  572
  K.1.2 Function: scan_swap_map()  574

K.1.1 Function: get_swap_page()  (mm/swapfile.c)

The call graph for this function is shown in Figure 11.2. This is the high level API function for searching the swap areas for a free swap lot and returning the resulting swp_entry_t.

99 swp_entry_t get_swap_page(void)
100 {
101    struct swap_info_struct * p;
102    unsigned long offset;
103    swp_entry_t entry;
104    int type, wrapped = 0;
105
106    entry.val = 0; /* Out of memory */
107    swap_list_lock();
108    type = swap_list.next;
109    if (type < 0)
110        goto out;
111    if (nr_swap_pages <= 0)
112        goto out;
113
114    while (1) {
115        p = &swap_info[type];
116        if (((p->flags & SWP_WRITEOK) == SWP_WRITEOK) {
117            swap_device_lock(p);
118            offset = scan_swap_map(p);
119            swap_device_unlock(p);
120            if (offset) {
121                entry = SWP_ENTRY(type,offset);
122                type = swap_info[type].next;
123                if (type < 0 ||
124                    p->prio != swap_info[type].prio) {
125                    swap_list.next = swap_list.head;
126                } else {
127                    swap_list.next = type;
128                }
129                goto out;
130            }
131        }
132    }
Lock the list of swap areas

Get the next swap area that is to be used for allocating from. This list will be ordered depending on the priority of the swap areas

If there are no swap areas, return NULL

If the accounting says there are no available swap slots, return NULL

Cycle through all swap areas

Get the current swap info struct from the swap_info array

If this swap area is available for writing to and is active...

Lock the swap area

Call scan_swap_map() (See Section K.1.2) which searches the requested swap map for a free slot

Unlock the swap device

If a slot was free...

Encode an identifier for the entry with SWP_ENTRY()

Record the next swap area to use

If the next area is the end of the list or the priority of the next swap area does not match the current one, move back to the head

Otherwise move to the next area

Goto out
K.1 Scanning for Free Entries \((\text{get\_swap\_page()}\)\)

132 Move to the next swap area

133-138 Check for wraparound. Set \texttt{wrapped} to 1 if we get to the end of the list of swap areas

139-140 If there was no available swap areas, goto out

142 The exit to this function

143 Unlock the swap area list

144 Return the entry if one was found and NULL otherwise

K.1.2 Function: \texttt{scan\_swap\_map()} \((\text{mm/swapfile.c})\)

This function tries to allocate \texttt{SWAPFILE\_CLUSTER} number of pages sequentially in swap. When it has allocated that many, it searches for another block of free slots of size \texttt{SWAPFILE\_CLUSTER}. If it fails to find one, it resorts to allocating the first free slot. This clustering attempts to make sure that slots are allocated and freed in \texttt{SWAPFILE\_CLUSTER} sized chunks.

36 static inline int scan_swap_map(struct swap_info_struct *si) {
37  
38     unsigned long offset;
39     if (si->cluster_nr) {
40         while (si->cluster_next <= si->highest_bit) {
41             offset = si->cluster_next++;
42             if (si->swap_map[offset])
43                 continue;
44             si->cluster_nr--;
45             goto got_page;
46         }
47     }
48 
49 
50     Allocate \texttt{SWAPFILE\_CLUSTER} pages sequentially. \texttt{cluster\_nr} is initialised to \texttt{SWAPFILE\_CLUSTER} and decrements with each allocation

47 If \texttt{cluster\_nr} is still positive, allocate the next available sequential slot

48 While the current offset to use (\texttt{cluster\_next}) is less than the highest known free slot (\texttt{highest\_bit}) then ...

49 Record the offset and update \texttt{cluster\_next} to the next free slot

50-51 If the slot is not actually free, move to the next one

52 Slot has been found, decrement the \texttt{cluster\_nr} field

53 Goto the out path
K.1 Scanning for Free Entries (scan_swap_map())

```c
56  si->cluster_nr = SWAPFILE_CLUSTER;
57
58  /* try to find an empty (even not aligned) cluster. */
59  offset = si->lowest_bit;
60  check_next_cluster:
61  if (offset+SWAPFILE_CLUSTER-1 <= si->highest_bit)
62  {
63      int nr;
64      for (nr = offset; nr < offset+SWAPFILE_CLUSTER; nr++)
65          if (si->swap_map[nr])
66              { offset = nr+1;
67                  goto check_next_cluster;
68              }
69  } /* We found a completely empty cluster, so start
70      * using it.
71      */
72  goto got_page;
73  }
```

At this stage, SWAPFILE_CLUSTER pages have been allocated sequentially so find
the next free block of SWAPFILE_CLUSTER pages.

56 Re-initialise the count of sequential pages to allocate to SWAPFILE_CLUSTER
59 Starting searching at the lowest known free slot
61 If the offset plus the cluster size is less than the known last free slot, then
    examine all the pages to see if this is a large free block
64 Scan from offset to offset + SWAPFILE_CLUSTER
65-69 If this slot is used, then start searching again for a free slot beginning after
    this known allocated one
73 A large cluster was found so use it
75  /* No luck, so now go finegrined as usual. -Andrea */
76  for (offset = si->lowest_bit; offset <= si->highest_bit ;
77         offset++) {
78      if (si->swap_map[offset])
79          continue;
80      si->lowest_bit = offset+1;
```

This unusual for loop extract starts scanning for a free page starting from
lowest_bit

77-78 If the slot is in use, move to the next one
K.1 Scanning for Free Entries (*scan_swap_map()*)

79 Update the lowest_bit known probable free slot to the succeeding one

80 got_page:
81   if (offset == si->lowest_bit)
82     si->lowest_bit++;
83   if (offset == si->highest_bit)
84     si->highest_bit--;
85   if (si->lowest_bit > si->highest_bit) {
86     si->lowest_bit = si->max;
87     si->highest_bit = 0;
88   }
89   si->swap_map[offset] = 1;
90   nr_swap_pages--;
91   si->cluster_next = offset+1;
92   return offset;
93 }
94 si->lowest_bit = si->max;
95 si->highest_bit = 0;
96 return 0;
97 }

A slot has been found, do some housekeeping and return it

81-82 If this offset is the known lowest free slot (lowest_bit), increment it

83-84 If this offset is the highest known likely free slot, decrement it

85-88 If the low and high mark meet, the swap area is not worth searching any more because these marks represent the lowest and highest known free slots. Set the low slot to be the highest possible slot and the high mark to 0 to cut down on search time later. This will be fixed up the next time a slot is freed

89 Set the reference count for the slot

90 Update the accounting for the number of available swap pages (*nr_swap_pages*)

91 Set *cluster_next* to the adjacent slot so the next search will start here

92 Return the free slot

94-96 No free slot available, mark the area unsearchable and return 0
K.2 Swap Cache

K.2.1 Adding Pages to the Swap Cache

K.2.1.1 Function: add_to_swap_cache()  

The call graph for this function is shown in Figure 11.3. This function wraps around the normal page cache handler. It first checks if the page is already in the swap cache with swap_duplicate() and if it does not, it calls add_to_page_cache_unique() instead.

```c
70 int add_to_swap_cache(struct page *page, swp_entry_t entry)  
71 {  
72   if (page->mapping)  
73     BUG();  
74   if (!swap_duplicate(entry)) {  
75     INC_CACHE_INFO(noent_race);  
76     return -ENOENT;  
77   }  
78   if (add_to_page_cache_unique(page, &swapper_space, entry.val,  
79     page_hash(&swapper_space, entry.val)) != 0) {  
80     swap_free(entry);  
81     INC_CACHE_INFO(exist_race);  
82     return -EEXIST;  
83   }  
84   if (!PageLocked(page))  
85     BUG();  
86   if (!PageSwapCache(page))  
87     BUG();  
88   INC_CACHE_INFO(add_total);  
89   return 0;  
90 }
```
K.2.1 Adding Pages to the Swap Cache (add_to_swap_cache())

72-73 A check is made with PageSwapCache() before this function is called to make sure the page is not already in the swap cache. This check here ensures the page has no other existing mapping in case the caller was careless and did not make the check.

74-77 Use swap_duplicate() (See Section K.2.1.2) to try an increment the count for this entry. If a slot already exists in the swap_map, increment the statistic recording the number of races involving adding pages to the swap cache and return -ENOENT.

78 Try and add the page to the page cache with add_to_page_cache_unique() (See Section J.1.1.2). This function is similar to add_to_page_cache() (See Section J.1.1.1) except it searches the page cache for a duplicate entry with __find_page_nolock(). The managing address space is swapper_space. The “offset within the file” in this case is the offset within swap_map, hence entry.val and finally the page is hashed based on address_space and offset within swap_map.

80-83 If it already existed in the page cache, we raced so increment the statistic recording the number of races to insert an existing page into the swap cache and return EEXIST.

84-85 If the page is locked for IO, it is a bug.

86-87 If it is not now in the swap cache, something went seriously wrong.

88 Increment the statistic recording the total number of pages in the swap cache.

89 Return success.

K.2.1.2 Function: swap_duplicate() (mm/swapfile.c)

This function verifies a swap entry is valid and if so, increments its swap map count.

1161 int swap_duplicate(swp_entry_t entry)
1162 {
1163     struct swap_info_struct * p;
1164     unsigned long offset, type;
1165     int result = 0;
1166     type = SWP_TYPE(entry);
1167     if (type >= nr_swapfiles)
1168         goto bad_file;
1169     p = type + swap_info;
1170     offset = SWP_OFFSET(entry);
1171     swap_device_lock(p);
1174 if (offset < p->max && p->swap_map[offset]) {
1175   if (p->swap_map[offset] < SWAP_MAP_MAX - 1) {
1176     p->swap_map[offset]++;
1177     result = 1;
1178   } else if (p->swap_map[offset] <= SWAP_MAP_MAX) {
1179     if (swap_overflow++ < 5)
1180       printk(KERN_WARNING "swap_dup: swap entry overflow\n");
1181     p->swap_map[offset] = SWAP_MAP_MAX;
1182     result = 1;
1183   }
1184 }
1185 swap_device_unlock(p);
1186 out:
1187 return result;
1188
1189 bad_file:
1190 printk(KERN_ERR "swap_dup: %s\%08lx\n", Bad_file, entry.val);
1191 goto out;
1192 }

1161 The parameter is the swap entry to increase the swap_map count for

1167-1169 Get the offset within the swap_info for the swap_info_struct containing this entry. If it is greater than the number of swap areas, goto bad_file

1170-1171 Get the relevant swap_info_struct and get the offset within its swap_map

1173 Lock the swap device

1174 Make a quick sanity check to ensure the offset is within the swap_map and that the slot indicated has a positive count. A 0 count would mean the slot is not free and this is a bogus swp_entry_t

1175-1177 If the count is not SWAP_MAP_MAX, simply increment it and return 1 for success

1178-1183 Else the count would overflow so set it to SWAP_MAP_MAX and reserve the slot permanently. In reality this condition is virtually impossible

1185-1187 Unlock the swap device and return

1190-1191 If a bad device was used, print out the error message and return failure
K.2.2 Deleting Pages from the Swap Cache

K.2.2.1 Function: swap_free() \((mm/swapfile.c)\)
Decrement the corresponding swap_map entry for the swp_entry_t

\begin{verbatim}
214 void swap_free(swp_entry_t entry) {
215     struct swap_info_struct * p;
216
217     p = swap_info_get(entry);
218     if (p) {
219         swap_entry_free(p, SWP_OFFSET(entry));
220         swap_info_put(p);
221     }
222 }
\end{verbatim}

swap_info_get() (See Section K.2.3.1) fetches the correct swap_info_struct
and performs a number of debugging checks to ensure it is a valid area and a
valid swap_map entry. If all is sane, it will lock the swap device

219-222 If it is valid, the corresponding swap_map entry is decremented with
swap_entry_free() (See Section K.2.2.2) and swap_info_put() (See Section K.2.3.2)
called to free the device

K.2.2.2 Function: swap_entry_free() \((mm/swapfile.c)\)

\begin{verbatim}
192 static int swap_entry_free(struct swap_info_struct *p,
   unsigned long offset) {
193     int count = p->swap_map[offset];
194
195     if (count < SWAP_MAP_MAX) {
196         count--;
197         p->swap_map[offset] = count;
198         if (!count) {
199             if (offset < p->lowest_bit)
200                 p->lowest_bit = offset;
201             if (offset > p->highest_bit)
202                 p->highest_bit = offset;
203             nr_swap_pages++;
204         }
205     }
206 }
207     return count;
208 }
\end{verbatim}

194 Get the current count
If the count indicates the slot is not permanently reserved then...

Decrement the count and store it in the \texttt{swap\_map}

If the count reaches 0, the slot is free so update some information

If this freed slot is below \texttt{lowest\_bit}, update \texttt{lowest\_bit} which indicates the lowest known free slot

Similarly, update the \texttt{highest\_bit} if this newly freed slot is above it

Increment the count indicating the number of free swap slots

Return the current count

\subsection{K.2.3 Acquiring/Releasing Swap Cache Pages}

\subsubsection{K.2.3.1 Function: swap\_info\_get() (mm/swapfile.c)}

This function finds the \texttt{swap\_info\_struct} for the given entry, performs some basic checking and then locks the device.

\begin{verbatim}
static struct swap_info_struct * swap_info_get(swp_entry_t entry)
{
    struct swap_info_struct * p;
    unsigned long offset, type;

    if (!entry.val)
        goto out;
    type = SWP_TYPE(entry);
    if (type >= nr_swapfiles)
        goto bad_nofile;
    p = & swap_info[type];
    if (!(p->flags & SWP_USED))
        goto bad_device;
    offset = SWP_OFFSET(entry);
    if (offset >= p->max)
        goto bad_offset;
    if (!p->swap_map[offset])
        goto bad_free;
    swap_list_lock();
    if (p->prio > swap_info[swap_list.next].prio)
        swap_list.next = type;
    swap_device_lock(p);
    return p;

    bad_free:
    printk(KERN_ERR "swap_free: %s%08lx\n", Unused_offset,

\texttt{Unused\_offset},
\end{verbatim}
K.2.3 Acquiring/Releasing Swap Cache Pages (*swap_info_get()*)

entry.val);
174 bad_offset:
175 printk(KERN_ERR "swap_free: %s%08lx\n", Bad_offset,
entry.val);
176 goto out;
177 bad_device:
178 printk(KERN_ERR "swap_free: %s%08lx\n", Unused_file,
entry.val);
179 goto out;
180 bad_nofile:
181 printk(KERN_ERR "swap_free: %s%08lx\n", Bad_file,
entry.val);
182 out:
183 return NULL;
184 }

152-153 If the supplied entry is NULL, return
154 Get the offset within the *swap_info* array
155-156 Ensure it is a valid area
157 Get the address of the area
158-159 If the area is not active yet, print a bad device error and return
160 Get the offset within the *swap_map*
161-162 Make sure the offset is not after the end of the map
163-164 Make sure the slot is currently in use
165 Lock the swap area list
166-167 If this area is of higher priority than the area that would be next, ensure
   the current area is used
168-169 Lock the swap device and return the swap area descriptor

K.2.3.2 Function: *swap_info_put()* (*mm/swapfile.c*)

This function simply unlocks the area and list

186 static void swap_info_put(struct swap_info_struct * p)
187 {
188     swap_device_unlock(p);
189     swap_list_unlock();
190 }

188 Unlock the device
189 Unlock the swap area list
K.2.4 Searching the Swap Cache

K.2.4.1 Function: lookup_swap_cache()  (mm/swap_state.c)
Top level function for finding a page in the swap cache

161 struct page * lookup_swap_cache(swp_entry_t entry)
162 {
163     struct page *found;
164
165     found = find_get_page(&swapper_space, entry.val);
166     /*
167     * Unsafe to assert PageSwapCache and mapping on page found:
168     * if SMP nothing prevents swapoff from deleting this page from
169     * the swap cache at this moment. find_lock_page would prevent
170     * that, but no need to change: we _have_ got the right page.
171     */
172     INC_CACHE_INFO(find_total);
173     if (found)
174         INC_CACHE_INFO(find_success);
175     return found;
176 }

165 find_get_page() (See Section J.1.4.1) is the principle function for returning
the struct page. It uses the normal page hashing and cache functions for
quickly finding it

172 Increase the statistic recording the number of times a page was searched for
in the cache

173-174 If one was found, increment the successful find count

175 Return the struct page or NULL if it did not exist
### K.3.1 Reading Backing Storage

**Function:** read_swap_cache_async()

This function will either return the requested page from the swap cache. If it does not exist, a page will be allocated, placed in the swap cache and the data is scheduled to be read from disk with `rw_swap_page()`.

```c
struct page * read_swap_cache_async(swp_entry_t entry)
{
    struct page *found_page, *new_page = NULL;
    int err;

    do {
        found_page = find_get_page(&swapper_space, entry.val);
        if (found_page)
            break;

        /*
        * Get a new page to read into from swap.
        */
        if (!new_page) {
            new_page = alloc_page(GFP_HIGHUSER);
            if (!new_page)
                break; /* Out of memory */
        }

        /*
        * Associate the page with swap entry in the swap cache.
        * May fail (-ENOENT) if swap entry has been freed since
        * our caller observed it. May fail (-EEXIST) if there
```
K.3.1 Reading Backing Storage (*read_swap_cache_async()*).

```c
213   * is already a page associated with this entry in the
214   * swap cache: added by a racing read_swap_cache_async,
215   * or by try_to_swap_out (or shmem_writepage) re-using
216   * the just freed swap entry for an existing page.
217   */
218   err = add_to_swap_cache(new_page, entry);
219   if (!err) {
220     /*
221     * Initiate read into locked page and return.
222     */
223     rw_swap_page(READ, new_page);
224     return new_page;
225   }
226 } while (err != -ENOENT);
227
228 if (new_page)
229   page_cache_release(new_page);
230 return found_page;
231 }
```

189 Loop in case *add_to_swap_cache()* fails to add a page to the swap cache

196 First search the swap cache with *find_get_page()* (See Section J.1.4.1) to see if the page is already available. Ordinarily, *lookup_swap_cache()* (See Section K.2.4.1) would be called but it updates statistics (such as the number of cache searches) so *find_get_page()* (See Section J.1.4.1) is called directly

203-207 If the page is not in the swap cache and we have not allocated one yet, allocate one with *alloc_page()*

218 Add the newly allocated page to the swap cache with *add_to_swap_cache()* (See Section K.2.1.1)

223 Schedule the data to be read with *rw_swap_page()* (See Section K.3.3.1). The page will be returned locked and will be unlocked when IO completes

224 Return the new page

226 Loop until *add_to_swap_cache()* succeeds or another process successfully inserts the page into the swap cache

228-229 This is either the error path or another process added the page to the swap cache for us. If a new page was allocated, free it with *page_cache_release()* (See Section J.1.3.2)

230 Return either the page found in the swap cache or an error
K.3.2 Writing Backing Storage

K.3.2.1 Function: swap_writepage() (mm/swap_state.c)
This is the function registered in swap_aops for writing out pages. Its function is pretty simple. First it calls remove_exclusive_swap_page() to try and free the page. If the page was freed, then the page will be unlocked here before returning as there is no IO pending on the page. Otherwise rw_swap_page() is called to sync the page with backing storage.

```c
24 static int swap_writepage(struct page *page)
25 {
26   if (remove_exclusive_swap_page(page)) {
27     UnlockPage(page);
28     return 0;
29   }
30   rw_swap_page(WRITE, page);
31   return 0;
32 }
```

26-29 remove_exclusive_swap_page() (See Section K.3.2.2) will reclaim the page from the swap cache if possible. If the page is reclaimed, unlock it before returning.

30 Otherwise the page is still in the swap cache so synchronise it with backing storage by calling rw_swap_page() (See Section K.3.3.1)

K.3.2.2 Function: remove_exclusive_swap_page() (mm/swapfile.c)
This function will tries to work out if there is other processes sharing this page or not. If possible the page will be removed from the swap cache and freed. Once removed from the swap cache, swap_free() is decremented to indicate that the swap cache is no longer using the slot. The count will instead reflect the number of PTEs that contain a swp_entry_t for this slot.

```c
287 int remove_exclusive_swap_page(struct page *page)
288 {
289   int retval;
290   struct swap_info_struct * p;
291   swp_entry_t entry;
292
293   if (!PageLocked(page))
294     BUG();
295   if (!PageSwapCache(page))
296     return 0;
297   if (page_count(page) - !!page->buffers != 2) /* 2: us + cache */
298     return 0;
299```
K.3.2 Writing Backing Storage (\textit{remove\_exclusive\_swap\_page()})

```c
entry.val = page->index;
p = swap_info_get(entry);
if (!p)
  return 0;

/* Is the only swap cache user the cache itself? */
retval = 0;
if (p->swap_map[SWP_OFFSET(entry)] == 1) {
  /* Recheck the page count with the pagecache lock held. */
  spin_lock(&pagecache_lock);
  if (page_count(page) - !!page->buffers == 2) {
    __delete_from_swap_cache(page);
    SetPageDirty(page);
    retval = 1;
  }
  spin_unlock(&pagecache_lock);
}
swap_info_put(p);
if (retval) {
  block_flushpage(page, 0);
  swap_free(entry);
  page_cache_release(page);
}
return retval;
```

293-294 This operation should only be made with the page locked

295-296 If the page is not in the swap cache, then there is nothing to do

297-298 If there are other users of the page, then it cannot be reclaimed so return

300 The \textit{swp\_entry\_t} for the page is stored in \textit{page->index} as explained in Section 2.4

301 Get the \textit{swap\_info\_struct} with \textit{swap\_info\_get()} (See Section K.2.3.1)

307 If the only user of the swap slot is the swap cache itself (i.e. no process is mapping it), then delete this page from the swap cache to free the slot. Later the swap slot usage count will be decremented as the swap cache is no longer using it

310 If the current user is the only user of this page, then it is safe to remove from the swap cache. If another process is sharing it, it must remain here
K.3.2 Writing Backing Storage (remove_exclusive_swap_page())

311 Delete from the swap cache

313 Set retval to 1 so that the caller knows the page was freed and so that swap_free() (See Section K.2.2.1) will be called to decrement the usage count in the swap_map

317 Drop the reference to the swap slot that was taken with swap_info_get() (See Section K.2.3.1)

320 The slot is being freed to call block_flushpage() so that all IO will complete and any buffers associated with the page will be freed

321 Free the swap slot with swap_free()

322 Drop the reference to the page

K.3.2.3 Function: free_swap_and_cache() (mm/swapfile.c)
This function frees an entry from the swap cache and tries to reclaim the page. Note that this function only applies to the swap cache.

332 void free_swap_and_cache(swp_entry_t entry)
333 {
334    struct swap_info_struct * p;
335    struct page *page = NULL;
336
337    p = swap_info_get(entry);
338    if (p) {
339        if (swap_entry_free(p, SWP_OFFSET(entry)) == 1)
340            page = find_trylock_page(&swapper_space, entry.val);
341        swap_info_put(p);
342    }
343    if (page) {
344        page_cache_get(page);
345        /* Only cache user (+us), or swap space full? Free it! */
346        if (page_count(page) - !!page->buffers == 2 || vm_swap_full()) {
347            delete_from_swap_cache(page);
348            SetPageDirty(page);
349        }
350        UnlockPage(page);
351        page_cache_release(page);
352    }
353 }

337 Get the swap_info struct for the requested entry

338-342 Presuming the swap area information struct exists, call swap_entry_free() to free the swap entry. The page for the entry is then located in the swap cache using find_trylock_page(). Note that the page is returned locked
341 Drop the reference taken to the swap info struct at line 337

343-352 If the page was located then we try to reclaim it

344 Take a reference to the page so it will not be freed prematurely

346-349 The page is deleted from the swap cache if there are no processes mapping the page or if the swap area is more than 50% full (Checked by vm_swap_full())

350 Unlock the page again

351 Drop the local reference to the page taken at line 344

K.3.3 Block IO

K.3.3.1 Function: rw_swap_page() (mm/page_io.c)

This is the main function used for reading data from backing storage into a page or writing data from a page to backing storage. Which operation is performed depends on the first parameter rw. It is basically a wrapper function around the core function rw_swap_page_base(). This simply enforces that the operations are only performed on pages in the swap cache.

85 void rw_swap_page(int rw, struct page *page)
86 {
87   swp_entry_t entry;
88
89   entry.val = page->index;
90
91   if (!PageLocked(page))
92     PAGE_BUG(page);
93   if (!PageSwapCache(page))
94     PAGE_BUG(page);
95   if (!rw_swap_page_base(rw, entry, page))
96     UnlockPage(page);
97 }

85 rw indicates whether a read or write is taking place

89 Get the swp_entry_t from the index field

91-92 If the page is not locked for IO, it is a bug

93-94 If the page is not in the swap cache, it is a bug

95 Call the core function rw_swap_page_base(). If it returns failure, the page is unlocked with UnlockPage() so it can be freed
K.3.3.2 Function: **rw_swap_page_base()**  
(mm/page_io.c)

This is the core function for reading or writing data to the backing storage. Whether it is writing to a partition or a file, the block layer **brw_page()** function is used to perform the actual IO. This function sets up the necessary buffer information for the block layer to do its job. The **brw_page()** function performs asynchronous IO so it is likely it will return with the page locked which will be unlocked when the IO completes.

```c
36 static int rw_swap_page_base(int rw, swp_entry_t entry,  
                   struct page *page)
37 {
38     unsigned long offset;
39     int zones[PAGE_SIZE/512];
40     int zones_used;
41     kdev_t dev = 0;
42     int block_size;
43     struct inode *swapf = 0;
44
45     if (rw == READ) {
46         ClearPageUptodate(page);
47         kstat.pswpin++;
48     } else
49         kstat.pswpout++;
50
36

36 The parameters are:

**rw** indicates whether the operation is a read or a write

**entry** is the swap entry for locating the data in backing storage

**page** is the page that is been read or written to

39 **zones** is a parameter required by the block layer for **brw_page()**. It is expected to contain an array of block numbers that are to be written to. This is primarily of important when the backing storage is a file rather than a partition

45-47 If the page is to be read from disk, clear the Uptodate flag as the page is obviously not up to date if we are reading information from the disk. Increment the pages swapped in (**pswpin**) statistic

49 Else just update the pages swapped out (**pswpout**) statistic

```c
51     get_swaphandle_info(entry, &offset, &dev, &swapf);
52     if (dev) {
53         zones[0] = offset;
54         zones_used = 1;
55         block_size = PAGE_SIZE;
```
K.3.3 Block IO (rw_swap_page_base())

56 } else if (swapf) {
57   int i, j;
58   unsigned int block =
59     offset << (PAGE_SHIFT - swapf->i_sb->s_blocksize_bits);
60
61   block_size = swapf->i_sb->s_blocksize;
62   for (i=0, j=0; j< PAGE_SIZE ; i++, j += block_size)
63     if (!(zones[i] = bmap(swapf,block++))) {
64       printk("rw_swap_page: bad swap file\n");
65       return 0;
66     }
67     zones_used = i;
68     dev = swapf->i_dev;
69 } else {
70   return 0;
71 }
72
73 /* block_size == PAGE_SIZE/zones_used */
74 brw_page(rw, page, dev, zones, block_size);
75 return 1;
76 }

51 get_swaphandle_info() (See Section K.3.3.3) returns either the kdev_t or
struct inode that represents the swap area, whichever is appropriate

52-55 If the storage area is a partition, then there is only one block to be written
which is the size of a page. Hence, zones only has one entry which is the offset
within the partition to be written and the block_size is PAGE_SIZE

56 Else it is a swap file so each of the blocks in the file that make up the page has
to be mapped with bmap() before calling brw_page()

58-59 Calculate what the starting block is

61 The size of individual block is stored in the superblock information for the
filesystem the file resides on

62-66 Call bmap() for every block that makes up the full page. Each block is
stored in the zones array for passing to brw_page(). If any block fails to be
mapped, 0 is returned

67 Record how many blocks make up the page in zones_used

68 Record which device is being written to

74 Call brw_page() from the block layer to schedule the IO to occur. This function
returns immediately as the IO is asynchronous. When the IO is completed, a
callback function (end_buffer_io_async()) is called which unlocks the page. Any process waiting on the page will be woken up at that point.

K.3.3.3 Function: get_swaphandle_info()  (mm/swapfile.c)

This function is responsible for returning either the kdev_t or struct inode that is managing the swap area that entry belongs to.

```c
void get_swaphandle_info(swp_entry_t entry, unsigned long *offset, kdev_t *dev, struct inode **swapf)
{
    unsigned long type;
    struct swap_info_struct *p;
    type = SWP_TYPE(entry);
    if (type >= nr_swapfiles) {
        printk(KERN_ERR "rw_swap_page: %s%08lx\n", Bad_file, entry.val);
        return;
    }
    p = &swap_info[type];
    *offset = SWP_OFFSET(entry);
    if (*offset >= p->max && *offset != 0) {
        printk(KERN_ERR "rw_swap_page: %s%08lx\n", Bad_offset, entry.val);
        return;
    }
    if (p->swap_map && !p->swap_map[*offset]) {
        printk(KERN_ERR "rw_swap_page: %s%08lx\n", Unused_offset, entry.val);
        return;
    }
    if (p->swap_map && !p->swap_map[*offset]) {
        printk(KERN_ERR "rw_swap_page: %s%08lx\n", Unused_file, entry.val);
        return;
    }
    if (!(p->flags & SWP_USED)) {
        printk(KERN_ERR "rw_swap_page: %s%08lx\n", Unused_file, entry.val);
        return;
    }
    if (p->swap_device) {
        *dev = p->swap_device;
    }
    else if (p->swap_file) {
        *swapf = p->swap_file->d_inode;
    }
    else {
```
printk(KERN_ERR "rw_swap_page: no swap file or device\n");
}
return;
}

Extract which area within swap_info this entry belongs to

If the index is for an area that does not exist, then print out an information message and return. Bad_file is a static array declared near the top of mm/swapfile.c that says “Bad swap file entry”

Get the swap_info_struct from swap_info

Extract the offset within the swap area for this entry

Make sure the offset is not after the end of the file. Print out the message in Bad_offset if it is

If the offset is currently not being used, it means that entry is a stale entry so print out the error message in Unused_offset

If the swap area is currently not active, print out the error message in Unused_file

If the swap area is a device, return the kdev_t in swap_info_struct→swap_device

If it is a swap file, return the struct inode which is available via swap_info_struct→swap_file→d_inode

Else there is no swap file or device for this entry so print out the error message and return
K.4 Activating a Swap Area

Contents

K.4 Activating a Swap Area 594
K.4.1 Function: sys_swapon() 594
K.4.2 Function: swap_setup() 605

K.4.1 Function: sys_swapon()  (mm/swapfile.c)

This, quite large, function is responsible for the activating of swap space. Broadly speaking the tasks it takes are as follows;

- Find a free swap_info_struct in the swap_info array and initialise it with default values

- Call user_path_walk() which traverses the directory tree for the supplied specialfile and populates a namidata structure with the available data on the file, such as the dentry and the filesystem information for where it is stored (vfsmount)

- Populate swap_info_struct fields pertaining to the dimensions of the swap area and how to find it. If the swap area is a partition, the block size will be configured to the PAGE_SIZE before calculating the size. If it is a file, the information is obtained directly from the inode

- Ensure the area is not already activated. If not, allocate a page from memory and read the first page sized slot from the swap area. This page contains information such as the number of good slots and how to populate the swap_info_struct→swap_map with the bad entries

- Allocate memory with vmalloc() for swap_info_struct→swap_map and initialise each entry with 0 for good slots and SWAP_MAP_BAD otherwise. Ideally the header information will be a version 2 file format as version 1 was limited to swap areas of just under 128MiB for architectures with 4KiB page sizes like the x86

- After ensuring the information indicated in the header matches the actual swap area, fill in the remaining information in the swap_info_struct such as the maximum number of pages and the available good pages. Update the global statistics for nr_swap_pages and total_swap_pages

- The swap area is now fully active and initialised and so it is inserted into the swap list in the correct position based on priority of the newly activated area

855 asmlinkage long sys_swapon(const char * specialfile,
      int swap_flags)
856 {
857     struct swap_info_struct * p;
The two parameters are the path to the swap area and the flags for activation.

The activating process must have the CAP_SYS_ADMIN capability or be the superuser to activate a swap area.

Acquire the Big Kernel Lock.

Lock the list of swap areas.

Get the first swap area in the swap_info array.

for (type = 0; type < nr_swapfiles; type++, p++)
if (!(p->flags & SWP_USED))
  break;
error = -EPERM;
if (type >= MAX_SWAPFILES) {
  swap_list_unlock();
goto out;
}
if (type >= nr_swapfiles)
  nr_swapfiles = type+1;
p->flags = SWP_USED;
p->swap_file = NULL;
p->swap_vfsmnt = NULL;
p->swap_device = 0;
p->swap_map = NULL;
K.4 Activating a Swap Area (sys_swapon())

892     p->lowest_bit = 0;
893     p->highest_bit = 0;
894     p->cluster_nr = 0;
895     p->sdev_lock = SPIN_LOCK_UNLOCKED;
896     p->next = -1;
897     if (swap_flags & SWAP_FLAG_PREFER) {
898         p->prio =
899             (swap_flags & SWAP_FLAG_PRI0_MASK)>>SWAP_FLAG_PRI0_SHIFT;
900     } else {
901         p->prio = --least_priority;
902     }
903     swap_list_unlock();

Find a free swap_info_struct and initialise it with default values

877-879 Cycle through the swap_info until a struct is found that is not in use

880 By default the error returned is Permission Denied which indicates the caller did not have the proper permissions or too many swap areas are already in use

881 If no struct was free, MAX_SWAPFILE areas have already been activated so unlock the swap list and return

885-886 If the selected swap area is after the last known active area (nr_swapfiles), then update nr_swapfiles

887 Set the flag indicating the area is in use

888-896 Initialise fields to default values

897-902 If the caller has specified a priority, use it else set it to least_priority and decrement it. This way, the swap areas will be prioritised in order of activation

903 Release the swap list lock

904     error = user_path_walk(specialfile, &nd);
905     if (error)
906         goto bad_swap_2;
907
908     p->swap_file = nd.dentry;
909     p->swap_vfsmnt = nd.mnt;
910     swap_inode = nd.dentry->d_inode;
911     error = -EINVAL;
912
Traverse the VFS and get some information about the special file
user_path_walk() traverses the directory structure to obtain a nameidata structure describing the special file.

If it failed, return failure.

Fill in the swap_file field with the returned dentry.

Similarly, fill in the swap_vfsmnt.

Record the inode of the special file.

Now the default error is -EINVAL indicating that the special file was found but it was not a block device or a regular file.

```c
if (S_ISBLK(swap_inode->i_mode)) {
    kdev_t dev = swap_inode->i_rdev;
    struct block_device_operations *bdops;
    devfs_handle_t de;

    p->swap_device = dev;
    set_blocksize(dev, PAGE_SIZE);

    bd_acquire(swap_inode);
    bdev = swap_inode->i_bdev;
    de = devfs_get_handle_from_inode(swap_inode);
    bdops = devfs_get_ops(de);
    if (bdops) bdev->bd_op = bdops;

    error = blkdev_get(bdev, FMODE_READ|FMODE_WRITE, 0, BDEV_SWAP);
    devfs_put_ops(de); /* Decrement module use count
                        * now we're safe*/
    if (error)
        goto bad_swap_2;
    set_blocksize(dev, PAGE_SIZE);
    error = -ENODEV;
    if (!dev || (blk_size[MAJOR(dev)] && !blk_size[MAJOR(dev)][MINOR(dev)]))
        goto bad_swap;
    swapfilesize = 0;
    if (blk_size[MAJOR(dev)])
        swapfilesize = blk_size[MAJOR(dev)][MINOR(dev)]
                        >> (PAGE_SHIFT - 10);
    } else if (S_ISREG(swap_inode->i_mode))
        swapfilesize = swap_inode->i_size >> PAGE_SHIFT;
    else
        goto bad_swap;
```
If a partition, configure the block device before calculating the size of the area, else obtain it from the inode for the file.

913 Check if the special file is a block device

914-939 This code segment handles the case where the swap area is a partition

914 Record a pointer to the device structure for the block device

918 Store a pointer to the device structure describing the special file which will be needed for block IO operations

919 Set the block size on the device to be PAGE_SIZE as it will be page sized chunks swap is interested in

921 The bd_acquire() function increments the usage count for this block device

922 Get a pointer to the block_device structure which is a descriptor for the device file which is needed to open it

923 Get a devfs handle if it is enabled. devfs is beyond the scope of this book

924-925 Increment the usage count of this device entry

927 Open the block device in read/write mode and set the BDEV_SWAP flag which is an enumerated type but is ignored when do_open() is called

928 Decrement the use count of the devfs entry

929-930 If an error occurred on open, return failure

931 Set the block size again

932 After this point, the default error is to indicate no device could be found

933-935 Ensure the returned device is ok

937-939 Calculate the size of the swap file as the number of page sized chunks that exist in the block device as indicated by blk_size. The size of the swap area is calculated to make sure the information in the swap area is sane

941 If the swap area is a regular file, obtain the size directly from the inode and calculate how many page sized chunks exist

943 If the file is not a block device or regular file, return error

945 error = -EBUSY;
946 for (i = 0 ; i < nr_swapfiles ; i++) {
947     struct swap_info_struct *q = &swap_info[i];
948     if (i == type || !q->swap_file)
949         continue;
if (swap_inode->i_mapping == q->swap_file->d_inode->i_mapping)
  goto bad_swap;

swap_header = (void *)__get_free_page(GFP_USER);
if (!swap_header) {
  printk("Unable to start swapping: out of memory :-)\n");
  error = -ENOMEM;
  goto bad_swap;
}

lock_page(virt_to_page(swap_header));

if (!memcmp("SWAP-SPACE", swap_header->magic.magic, 10))
  swap_header_version = 1;
else if (!memcmp("SWAPSPACE2", swap_header->magic.magic, 10))
  swap_header_version = 2;
else {
  printk("Unable to find swap-space signature\n");
  error = -EINVAL;
  goto bad_swap;
}

The next check makes sure the area is not already active. If it is, the error -EBUSY will be returned

Read through the while swap_info struct and ensure the area to be activated is not already active

Allocate a page for reading the swap area information from disk

The function lock_page() locks a page and makes sure it is synced with disk if it is file backed. In this case, it'll just mark the page as locked which is required for the rw_swap_page_nolock() function

Read the first page slot in the swap area into swap_header

Check the version based on the swap area information is and set swap_header_version variable with it. If the swap area could not be identified, return -EINVAL

switch (swap_header_version) {
  case 1:
    memset(((char *) swap_header)+PAGE_SIZE-10,0,10);
K.4 Activating a Swap Area (sys_swapon())

j = 0;
p->lowest_bit = 0;
p->highest_bit = 0;
for (i = 1 ; i < 8*PAGE_SIZE ; i++) {
    if (test_bit(i,(char *) swap_header)) {
        if (!p->lowest_bit)
            p->lowest_bit = i;
        p->highest_bit = i;
        maxpages = i+1;
        j++;
    }
}

nr_good_pages = j;
p->swap_map = vmalloc(maxpages * sizeof(short));
if (!p->swap_map) {
    error = -ENOMEM;
    goto bad_swap;
}
for (i = 1 ; i < maxpages ; i++) {
    if (test_bit(i,(char *) swap_header))
        p->swap_map[i] = 0;
    else
        p->swap_map[i] = SWAP_MAP_BAD;
}
break;

Read in the information needed to populate the swap_map when the swap area is version 1.

976 Zero out the magic string identifying the version of the swap area

978-979 Initialise fields in swap_info_struct to 0

980-988 A bitmap with 8*PAGE_SIZE entries is stored in the swap area. The full page, minus 10 bits for the magic string, is used to describe the swap map limiting swap areas to just under 128MiB in size. If the bit is set to 1, there is a slot on disk available. This pass will calculate how many slots are available so a swap_map may be allocated

981 Test if the bit for this slot is set

982-983 If the lowest_bit field is not yet set, set it to this slot. In most cases, lowest_bit will be initialised to 1

984 As long as new slots are found, keep updating the highest_bit
Count the number of pages
j is the count of good pages in the area
Allocate memory for the swap_map with vmalloc()
If memory could not be allocated, return ENOMEM
For each slot, check if the slot is “good”. If yes, initialise the slot count to 0, else set it to SWAP_MAP_BAD so it will not be used
Exit the switch statement

case 2:
    if (swap_header->info.version != 1) {
        printk(KERN_WARNING
            "Unable to handle swap header version %d\n",
            swap_header->info.version);
        error = -EINVAL;
        goto bad_swap;
    }
    p->lowest_bit = 1;
    maxpages = SWP_OFFSET(SWP_ENTRY(0,~0UL)) - 1;
    if (maxpages > swap_header->info.last_page)
        maxpages = swap_header->info.last_page;
    p->highest_bit = maxpages - 1;
    error = -EINVAL;
    if (swap_header->info.nr_badpages > MAX_SWAP_BADPAGES)
        goto bad_swap;
    if (!(p->swap_map = vmalloc(maxpages * sizeof(short)))) {
        error = -ENOMEM;
        goto bad_swap;
    }
    error = 0;
    memset(p->swap_map, 0, maxpages * sizeof(short));
    for (i=0; i<swap_header->info.nr_badpages; i++) {
        int page = swap_header->info.badpages[i];
        if (page <= 0 ||
            page >= swap_header->info.last_page)
            error = -EINVAL;
        else
            p->swap_map[page] = SWAP_MAP_BAD;
    }
K.4 Activating a Swap Area (sys_swapon())

```
1039      nr_good_pages = swap_header->info.last_page -
1040          swap_header->info.nr_badpages -
1041          1 /* header page */;
1042      if (error)
1043          goto bad_swap;
1044  }

Read the header information when the file format is version 2

1006-1012 Make absolutely sure we can handle this swap file format and return
         -EINVAL if we cannot. Remember that with this version, the swap_header
         struct is placed nicely on disk

1014 Initialise lowest_bit to the known lowest available slot

1015-1017 Calculate the maxpages initially as the maximum possible size of a
         swap_map and then set it to the size indicated by the information on disk.
         This ensures the swap_map array is not accidently overloaded

1018 Initialise highest_bit

1020-1022 Make sure the number of bad pages that exist does not exceed
         MAX_SWAP_BADPAGES

1025-1028 Allocate memory for the swap_map with vmalloc()

1031 Initialise the full swap_map to 0 indicating all slots are available

1032-1038 Using the information loaded from disk, set each slot that is unusable
         to SWAP_MAP_BAD

1039-1041 Calculate the number of available good pages

1042-1043 Return if an error occurred

1045      if (swapfilesize && maxpages > swapfilesize) {
1046          printk(KERN_WARNING
1047              "Swap area shorter than signature indicates\n");
1048          error = -EINVAL;
1049          goto bad_swap;
1050      }
1052      if (!nr_good_pages) {
1053          printk(KERN_WARNING "Empty swap-file\n");
1054          error = -EINVAL;
1055          goto bad_swap;
1056      }
1057      p->swap_map[0] = SWAP_MAP_BAD;
```
K.4 Activating a Swap Area (sys_swapon())

Ensure the information loaded from disk matches the actual dimensions of the swap area. If they do not match, print a warning and return an error.

If no good pages were available, return an error.

Make sure the first page in the map containing the swap header information is not used. If it was, the header information would be overwritten the first time this area was used.

Lock the swap list and the swap device.

Fill in the remaining fields in the swap_info_struct.

Update global statistics for the number of available swap pages (nr_swap_pages) and the total number of swap pages (total_swap_pages).

Print an informational message about the swap activation.
1070-1080 Insert the new swap area into the correct slot in the swap list based on priority

1082 Unlock the swap device

1083 Unlock the swap list

1084-1085 Return success

1086 bad_swap:
1087 if (bdev)
1088 blkdev_put(bdev, BDEV_SWAP);
1089 bad_swap_2:
1090 swap_list_lock();
1091 swap_map = p->swap_map;
1092 nd.mnt = p->swap_vfsmnt;
1093 nd.dentry = p->swap_file;
1094 p->swap_device = 0;
1095 p->swap_file = NULL;
1096 p->swap_vfsmnt = NULL;
1097 p->swap_map = NULL;
1098 p->flags = 0;
1099 if (!(swap_flags & SWAP_FLAG_PREFER))
1100 ++least_priority;
1101 swap_list_unlock();
1102 if (swap_map)
1103 vfree(swap_map);
1104 path_release(&nd);
1105 out:
1106 if (swap_header)
1107 free_page((long) swap_header);
1108 unlock_kernel();
1109 return error;
1110 }

1087-1088 Drop the reference to the block device

1090-1104 This is the error path where the swap list need to be unlocked, the slot in swap_info reset to being unused and the memory allocated for swap_map freed if it was assigned

1104 Drop the reference to the special file

1106-1107 Release the page containing the swap header information as it is no longer needed

1108 Drop the Big Kernel Lock

1109 Return the error or success value
K.4.2 Function: swap_setup() (mm/swap.c)

This function is called during the initialisation of kswapd to set the size of page_cluster. This variable determines how many pages readahead from files and from backing storage when paging in data.

```c
void __init swap_setup(void)
{
    unsigned long megs = num_physpages >> (20 - PAGE_SHIFT);
    /* Use a smaller cluster for small-memory machines */
    if (megs < 16)
        page_cluster = 2;
    else
        page_cluster = 3;
    /*
    * Right now other parts of the system means that we
    * really don’t want to cluster much more
    */
}
```

102 Calculate how much memory the system has in megabytes

105 In low memory systems, set page_cluster to 2 which means that, at most, 4 pages will be paged in from disk during readahead

108 Else readahead 8 pages
K.5 Deactivating a Swap Area

Contents

K.5 Deactivating a Swap Area 606
K.5.1 Function: sys_swapoff() 606
K.5.2 Function: try_to_unuse() 610
K.5.3 Function: unuse_process() 615
K.5.4 Function: unuse_vma() 616
K.5.5 Function: unuse_pgd() 616
K.5.6 Function: unuse_pmd() 618
K.5.7 Function: unuse_pte() 619

K.5.1 Function: sys_swapoff() (mm/swapfile.c)

This function is principally concerned with updating the swap_info_struct and the swap lists. The main task of paging in all pages in the area is the responsibility of try_to_unuse(). The function tasks are broadly

- Call user_path_walk() to acquire the information about the special file to be deactivated and then take the BKL

- Remove the swap_info_struct from the swap list and update the global statistics on the number of swap pages available (nr_swap_pages) and the total number of swap entries (total_swap_pages). Once this is acquired, the BKL can be released again

- Call try_to_unuse() which will page in all pages from the swap area to be deactivated.

- If there was not enough available memory to page in all the entries, the swap area is reinserted back into the running system as it cannot be simply dropped. If it succeeded, the swap_info_struct is placed into an uninitialised state and the swap_map memory freed with vfree()

720 asmlinkage long sys_swapoff(const char * specialfile)
721 {
722    struct swap_info_struct * p = NULL;
723    unsigned short *swap_map;
724    struct nameidata nd;
725    int i, type, prev;
726    int err;
727
728    if (!capable(CAP_SYS_ADMIN))
729        return -EPERM;
730
731    err = user_path_walk(specialfile, &nd);
732    if (err)
goto out;

728-729 Only the superuser or a process with CAP_SYS_ADMIN capabilities may deactivate an area

731-732 Acquire information about the special file representing the swap area with user_path_walk(). Goto out if an error occurred

735 lock_kernel();
736 prev = -1;
737 swap_list_lock();
738 for (type = swap_list.head; type >= 0;
739     type = swap_info[type].next) {
740     p = swap_info + type;
741     if ((p->flags & SWP_WRITEOK) == SWP_WRITEOK) {
742         if (p->swap_file == nd.dentry)
743             break;
744     }
745     prev = type;
746 } err = -EINVAL;
747 if (type < 0) {
748     swap_list_unlock();
749     goto out_dput;
750 } if (prev < 0) {
751     swap_list.head = p->next;
752 } else {
753     swap_info[prev].next = p->next;
754 } if (type == swap_list.next) {
755     /* just pick something that’s safe... */
756     swap_list.next = swap_list.head;
757 } nr_swap_pages -= p->pages;
758 total_swap_pages -= p->pages;
759 p->flags = SWP_USED;

Acquire the BKL, find the swap_info_struct for the area to be deactivated and remove it from the swap list.

735 Acquire the BKL
737 Lock the swap list
K.5 Deactivating a Swap Area (sys_swapoff())

738-745 Traverse the swap list and find the swap_info_struct for the requested area. Use the dentry to identify the area.

747-750 If the struct could not be found, return.

752-760 Remove from the swap list making sure that this is not the head.

761 Update the total number of free swap slots.

762 Update the total number of existing swap slots.

763 Mark the area as active but may not be written to.

764-766 Unlock the swap list.

767 Release the BKL.

768 Page in all pages from this swap area.

767
768 if (err) {
769     /* re-insert swap space back into swap_list */
770     swap_list_unlock();
771     for (prev = -1, i = swap_list.head;
772          i >= 0;
773          prev = i, i = swap_info[i].next)
774         if (p->prio >= swap_info[i].prio)
775             break;
776     p->next = i;
777     if (prev < 0)
778         swap_list.head = swap_list.next = p - swap_info;
779     else
780         swap_info[prev].next = p - swap_info;
781     nr_swap_pages += p->pages;
782     total_swap_pages += p->pages;
783     p->flags = SWP_WRITEOK;
784     swap_list_unlock();
785     goto out_dput;
786 }

Acquire the BKL. If we failed to page in all pages, then reinsert the area into the swap list.

767 Acquire the BKL.
K.5 Deactivating a Swap Area (sys_swapoff())

770 Lock the swap list

771-778 Reinsert the area into the swap list. The position it is inserted at depends on the swap area priority

779-780 Update the global statistics

781 Mark the area as safe to write to again

782-783 Unlock the swap list and return

    if (p->swap_device)
        blkdev_put(p->swap_file->d_inode->i_bdev, BDEV_SWAP);
    path_release(&nd);

785    swap_list_lock();
786    swap_device_lock(p);
787    nd.mnt = p->swap_vfsmnt;
788    nd.dentry = p->swap_file;
789    p->swap_vfsmnt = NULL;
790    p->swap_file = NULL;
791    p->swap_device = 0;
792    p->max = 0;
793    swap_map = p->swap_map;
794    p->swap_map = NULL;
795    p->flags = 0;
796    swap_device_unlock(p);
797    swap_list_unlock();
798    vfree(swap_map);
799    err = 0;
800
801 out_dput:
802    unlock_kernel();
803    path_release(&nd);
804 out:
805    return err;
806 }

Else the swap area was successfully deactivated to close the block device and mark the swap_info_struct free

785-786 Close the block device

787 Release the path information

789-790 Acquire the swap list and swap device lock

791-799 Reset the fields in swap_info_struct to default values
K.5 Deactivating a Swap Area (sys_swapoff())

800-801 Release the swap list and swap device
801 Free the memory used for the swap_map
806 Release the BKL
807 Release the path information in the event we reached here via the error path
809 Return success or failure

K.5.2 Function: try_to_unuse() (mm/swapfile.c)
This function is heavily commented in the source code albeit it consists of speculation or is slightly inaccurate at parts. The comments are omitted here for brevity.

513 static int try_to_unuse(unsigned int type)
514 {
515     struct swap_info_struct * si = &swap_info[type];
516     struct mm_struct *start_mm;
517     unsigned short *swap_map;
518     unsigned short swcount;
519     struct page *page;
520     swp_entry_t entry;
521     int i = 0;
522     int retval = 0;
523     int reset_overflow = 0;
525     start_mm = &init_mm;
541     atomic_inc(&init_mm.mm_users);

540-541 The starting mm_struct to page in pages for is init_mm. The count is incremented even though this particular struct will not disappear to prevent having to write special cases in the remainder of the function

556     while ((i = find_next_to_unuse(si, i))) {
557         /*
558          * Get a page for the entry, using the existing swap
559          * cache page if there is one. Otherwise, get a clean
560          * page and read the swap into it.
561          */
562         swap_map = &si->swap_map[i];
563         entry = SWP_ENTRY(type, i);
564         page = read_swap_cache_async(entry);
565         if (!page) {
566             if (!*swap_map)
572             continue;
K.5 Deactivating a Swap Area (try_to_unuse())

```c
retval = -ENOMEM;
break;
}

/* Don’t hold on to start_mm if it looks like exiting. */
if (atomic_read(&start_mm->mm_users) == 1) {
    mmput(start_mm);
    start_mm = &init_mm;
    atomic_inc(&init_mm.mm_users);
}
```

This is the beginning of the major loop in this function. Starting from the beginning of the swap_map, it searches for the next entry to be freed with find_next_to_unuse() until all swap map entries have been paged in

Get the swp_entry_t and call read_swap_cache_async() (See Section K.3.1.1) to find the page in the swap cache or have a new page allocated for reading in from the disk

If we failed to get the page, it means the slot has already been freed independently by another process or thread (process could be exiting elsewhere) or we are out of memory. If independently freed, we continue to the next map, else we return -ENOMEM

Check to make sure this mm is not exiting. If it is, decrement its count and go back to init_mm

```c
/* Wait for and lock page. When do_swap_page races with try_to_unuse, do_swap_page can handle the fault much faster than try_to_unuse can locate the entry. This apparently redundant "wait_on_page" lets try_to_unuse defer to do_swap_page in such a case - in some tests, do_swap_page and try_to_unuse repeatedly compete. */
wait_on_page(page);
lock_page(page);
```

/* Remove all references to entry, without blocking. Whenever we reach init_mm, there’s no address space to search, but use it as a reminder to search shmem. */
shmem = 0;
```
K.5 Deactivating a Swap Area (try_to_unuse())

```c
604    swcount = *swap_map;
605    if (swcount > 1) {
606        flush_page_to_ram(page);
607        if (start_mm == &init_mm)
608            shmem = shmem_unuse(entry, page);
609        else
610            unuse_process(start_mm, entry, page);
611    }
```

595 Wait on the page to complete IO. Once it returns, we know for a fact the page exists in memory with the same information as that on disk

596 Lock the page

604 Get the swap map reference count

605 If the count is positive then...

606 As the page is about to be inserted into process page tables, it must be freed from the D-Cache or the process may not “see” changes made to the page by the kernel

607-608 If we are using the init_mm, call shmem_unuse() (See Section L.6.2) which will free the page from any shared memory regions that are in use

610 Else update the PTE in the current mm which references this page

```c
612    if (*swap_map > 1) {
613        int set_start_mm = (*swap_map >= swcount);
614        struct list_head *p = &start_mm->mmlist;
615        struct mm_struct *new_start_mm = start_mm;
616        struct mm_struct *mm;
617
618        spin_lock(&mmlist_lock);
619        while (*swap_map > 1 &&
620            (p = p->next) != &start_mm->mmlist) {
621            mm = list_entry(p, struct mm_struct, mmlist);
622            swcount = *swap_map;
623            if (mm == &init_mm) {
624                set_start_mm = 1;
625                spin_unlock(&mmlist_lock);
626                shmem = shmem_unuse(entry, page);
627                spin_lock(&mmlist_lock);
628            } else
629                unuse_process(mm, entry, page);
630            if (set_start_mm && *swap_map < swcount) {
```
K.5 Deactivating a Swap Area \(\text{try\_to\_unuse()}\)

631 new_start_mm = mm;
632 set_start_mm = 0;
633 }
634 }
635 atomic_inc(&new_start_mm->mm_users);
636 spin_unlock(&mmlist_lock);
637 mmput(start_mm);
638 start_mm = new_start_mm;
639 }

612-637 If an entry still exists, begin traversing through all \text{mm\_structs} finding references to this page and update the respective PTE.

618 Lock the mm list.

619-632 Keep searching until all \text{mm\_structs} have been found. Do not traverse the full list more than once.

621 Get the \text{mm\_struct} for this list entry.

623-627 Call \text{shmem\_unuse()} (See Section L.6.2) if the mm is \text{init\_mm} as that indicates that is a page from the virtual filesystem. Else call \text{unuse\_process()} (See Section K.5.3) to traverse the current process’s page tables searching for the swap entry. If found, the entry will be freed and the page reinstated in the PTE.

630-633 Record if we need to start searching \text{mm\_structs} starting from \text{init\_mm} again.

654 if (*swap_map == SWAP\_MAP\_MAX) {
655 swap_list_lock();
656 swap_device_lock(si);
657 nr_swap_pages++;
658 *swap_map = 1;
659 swap_device_unlock(si);
660 swap_list_unlock();
661 reset_overflow = 1;
662 }

654 If the swap map entry is permanently mapped, we have to hope that all processes have their PTEs updated to point to the page and in reality the swap map entry is free. In reality, it is highly unlikely a slot would be permanently reserved in the first place.

645-661 Lock the list and swap device, set the swap map entry to 1, unlock them again and record that a reset overflow occurred.
if ((*swap_map > 1) && PageDirty(page) && PageSwapCache(page)) {
    rw_swap_page(WRITE, page);
    lock_page(page);
}
if (PageSwapCache(page)) {
    if (shmem)
        swap_duplicate(entry);
    else
        delete_from_swap_cache(page);
}

In the very rare event a reference still exists to the page, write the page back to disk so at least if another process really has a reference to it, it'll copy the page back in from disk correctly.

If the page is in the swap cache and belongs to the shared memory filesystem, a new reference is taken to it with swap_duplicate() so we can try and remove it again later with shmem_unuse().

Else, for normal pages, just delete them from the swap cache.

Mark the page dirty so that the swap out code will preserve the page and if it needs to remove it again, it'll write it correctly to a new swap area.

Unlock the page.

Release our reference to it in the page cache.

Call schedule() if necessary so the deactivation of swap does not hog the entire CPU.
K.5 Deactivating a Swap Area (try_to_unuse())

717 Drop our reference to the mm

718-721 If a permanently mapped page had to be removed, then print out a
warning so that in the very unlikely event an error occurs later, there will be
a hint to what might have happend

717 Return success or failure

K.5.3 Function: unuse_process() (mm/swapfile.c)

This function begins the page table walk required to remove the requested page
and entry from the process page tables managed by mm. This is only required when
a swap area is being deactivated so, while expensive, it is a very rare operation. This
set of functions should be instantly recognisable as a standard page-table walk.

454 static void unuse_process(struct mm_struct * mm,
455 swp_entry_t entry, struct page* page)
456 {
457 struct vm_area_struct* vma;
458
459 /*
460 * Go through process’ page directory.
461 */
462 spin_lock(&mm->page_table_lock);
463 for (vma = mm->mmap; vma; vma = vma->vm_next) {
464 pgd_t * pgd = pgd_offset(mm, vma->vm_start);
465 unuse_vma(vma, pgd, entry, page);
466 }
467 spin_unlock(&mm->page_table_lock);
468 return;
469 }

462 Lock the process page tables

463 Move through every VMA managed by this mm. Remember that one page
frame could be mapped in multiple locations

462 Get the PGD managing the beginning of this VMA

465 Call unuse_vma() (See Section K.5.4) to search the VMA for the page

467-468 The full mm has been searched so unlock the process page tables and return
K.5.4 Function: `unuse_vma()` *(mm/swapfile.c)*

This function searches the requested VMA for page table entries mapping the page and using the given swap entry. It calls `unuse_pgd()` for every PGD this VMA maps.

```c
static void unuse_vma(struct vm_area_struct *vma, pgd_t *pgdir, swp_entry_t entry, struct page* page)
{
    unsigned long start = vma->vm_start, end = vma->vm_end;
    if (start >= end)
        BUG();
    do {
        unuse_pgd(vma, pgdir, start, end - start, entry, page);
        start = (start + PGDIR_SIZE) & PGDIR_MASK;
        pgdir++;
    } while (start && (start < end));
}
```

Get the virtual addresses for the `start` and `end` of the VMA.

445-446 Check that the `start` is not after the `end`. There would need to be serious braindamage in the kernel for this to occur.

447-451 Walk through the VMA in `PGDIR_SIZE`-sized strides until the end of the VMA is reached. This effectively walks through every PGD that maps portions of this VMA.

448 Call `unuse_pgd()` (See Section K.5.5) to walk through just this PGD to unmap page.

449 Move the virtual address `start` to the beginning of the next PGD.

450 Move `pgdir` to the next PGD in the VMA.

K.5.5 Function: `unuse_pgd()` *(mm/swapfile.c)*

This function searches the requested PGD for page table entries mapping the page and using the given swap entry. It calls `unuse_pmd()` for every PMD this PGD maps.

```c
static inline void unuse_pgd(struct vm_area_struct *vma, pgd_t *dir, swp_entry_t entry, struct page* page)
{
    pmd_t *pmd;
    unsigned long address, unsigned long size,
```
416   if (pgd_none(*dir))
417       return;
418   if (pgd_bad(*dir)) {
419       pgd_ERROR(*dir);
420       pgd_clear(dir);
421       return;
422   }
423   pmd = pmd_offset(dir, address);
424   offset = address & PGDIR_MASK;
425   address &= ~PGDIR_MASK;
426   end = address + size;
427   if (end > PGDIR_SIZE)
428       end = PGDIR_SIZE;
429   if (address >= end)
430       BUG();
431   do {
432       unused_pmd(vma, pmd, address, end - address, offset, entry,
433                 page);
434       address = (address + PMD_SIZE) & PMD_MASK;
435       pmd++;
436   } while (address && (address < end));

416-417 If there is no PGD here, return
418-422 If the PGD is bad, then set the appropriate error, clear the PGD and return. There are very few architectures where this condition can occur
423 Get the address of the first PMD in this PGD
424 Calculate offset as the offset within the PGD the address is for. Remember that the first time this function is called, it might be searching a partial PGD
425 Align the address to the PGD
426 Calculate the end address of the search
427-428 If the end is beyond this PGD, set the end just to the end of this PGD
429-430 If the starting address is after the end address, something is very seriously wrong
431-436 Step through the PGD in PMD_SIZE-sized strides and call unused_pmd() (See Section K.5.6) for every PMD in this PGD
K.5.6 Function: `unuse_pmd()` *(mm/swapfile.c)*

This function searches the requested PMD for page table entries mapping the page and using the given swap entry. It calls `unuse_pte()` for every PTE this PMD maps.

```c
381 static inline void unuse_pmd(struct vm_area_struct * vma, pmd_t *dir,
382     unsigned long address, unsigned long size, unsigned long offset,
383     swp_entry_t entry, struct page* page)
384 {
385     pte_t * pte;
386     unsigned long end;
387
388     if (pmd_none(*dir))
389         return;
390     if (pmd_bad(*dir)) {
391         pmd_ERROR(*dir);
392         pmd_clear(dir);
393         return;
394     }
395     pte = pte_offset(dir, address);
396     offset += address & PMD_MASK;
397     address &= ~PMD_MASK;
398     end = address + size;
399     if (end > PMD_SIZE)
400         end = PMD_SIZE;
401     do {
402         unuse_pte(vma, offset+address-vma->vm_start, pte, entry, page);
403         address += PAGE_SIZE;
404         pte++;
405     } while (address && (address < end));
406 }
```

388-389 Return if no PMD exists

390-394 Set the appropriate error and clear the PMD if it is bad. There are very few architectures where this condition can occur

395 Calculate the starting PTE for this address

396 Set offset to be the offset within the PMD we are starting at

397 Align address to the PMD

398-400 Calculate the end address. If it is beyond the end of this PMD, set it to the end of this PMD

401-405 Step through this PMD in PAGE_SIZE-sized chunks and call `unuse_pte()` (See Section K.5.7) for each PTE
K.5.7 Function: `unuse_pte()` (mm/swapfile.c)

This function checks if the PTE at `dir` matches the entry we are searching for. If it does, the swap entry is freed and a reference is taken to the page representing the PTE that will be updated to map it.

```c
static inline void unuse_pte(struct vm_area_struct * vma,
   pte_t *dir, swp_entry_t entry, struct page* page)
{
    pte_t pte = *dir;
    if (likely(pte_to_swp_entry(pte).val != entry.val))
        return;
    if (unlikely(pte_none(pte) || pte_present(pte)))
        return;
    get_page(page);
    set_pte(dir, pte_mkold(mk_pte(page, vma->vm_page_prot)));
    swap_free(entry);
    ++vma->vm_mm->rss;
}
```

370-371 If the entry does not match the PTE, return

372-373 If there is no PTE or it is already present (meaning there is no way this entry is mapped here), then return

374 Otherwise we have found the entry we are looking for so take a reference to the page as a new PTE is about to map it

375 Update the PTE to map page

376 Free the swap entry

377 Increment the RSS count for this process
Appendix L

Shared Memory Virtual Filesystem

Contents

L.1 Initialising shmfs ............................................. 622
  L.1.1 Function: init_tmpfs() .................................. 622
  L.1.2 Function: shmem_read_super() .......................... 624
  L.1.3 Function: shmem_set_size() ............................. 626

L.2 Creating Files in tmpfs ..................................... 628
  L.2.1 Function: shmem_create() ............................... 628
  L.2.2 Function: shmem_mknod() ................................. 628
  L.2.3 Function: shmem_get_inode() ............................ 629

L.3 File Operations in tmpfs .................................... 632
  L.3.1 Memory Mapping ......................................... 632
    L.3.1.1 Function: shmem_mmap() .............................. 632
  L.3.2 Reading Files ............................................. 633
    L.3.2.1 Function: shmem_file_read() ......................... 633
    L.3.2.2 Function: do_shmem_file_read() ..................... 634
    L.3.2.3 Function: file_read_actor() ......................... 637
  L.3.3 Writing .................................................. 638
    L.3.3.1 Function: shmem_file_write() ....................... 638
  L.3.4 Symbolic Linking ........................................ 641
    L.3.4.1 Function: shmem_symlink() ........................... 641
    L.3.4.2 Function: shmem_readlink_inline() .................. 644
    L.3.4.3 Function: shmem_follow_link_inline() .............. 644
    L.3.4.4 Function: shmem_readlink() ......................... 644
  L.3.5 Synchronising ............................................ 645
    L.3.5.1 Function: shmem_sync_file() ....................... 645
APPENDIX L. SHARED MEMORY VIRTUAL FILESYSTEM 621

L.4 Inode Operations in tmpfs .............................. 646
   L.4.1 Truncating ........................................... 646
      L.4.1.1 Function: shmem_truncate() .............. 646
      L.4.1.2 Function: shmem_truncate_indirect() .... 647
      L.4.1.3 Function: shmem_truncate_direct() ........ 649
      L.4.1.4 Function: shmem_free_swp() ............. 650
   L.4.2 Linking .............................................. 651
      L.4.2.1 Function: shmem_link() .................. 651
   L.4.3 Unlinking ........................................... 652
      L.4.3.1 Function: shmem_unlink() .............. 652
   L.4.4 Making Directories .................................. 652
      L.4.4.1 Function: shmem_mkdir() ............ 652
   L.4.5 Removing Directories ................................ 653
      L.4.5.1 Function: shmem_rmdir() ............ 653
      L.4.5.2 Function: shmem_empty() .......... 653
      L.4.5.3 Function: shmem_positive() .......... 654
L.5 Page Faulting within a Virtual File ....................... 655
   L.5.1 Reading Pages during Page Fault .................. 655
      L.5.1.1 Function: shmem_nopage() ........... 655
      L.5.1.2 Function: shmem_getpage() ........ 656
   L.5.2 Locating Swapped Pages .............................. 663
      L.5.2.1 Function: shmem_alloc_entry() ........ 663
      L.5.2.2 Function: shmem_swp_entry() ......... 664
L.6 Swap Space Interaction .................................... 667
   L.6.1 Function: shmem_writepage() .................. 667
   L.6.2 Function: shmem_unuse() .......................... 669
   L.6.3 Function: shmem_unuse_inode() ............ 670
   L.6.4 Function: shmem_find_swp() .................. 673
L.7 Setting up Shared Regions .................................. 674
   L.7.1 Function: shmem_zero_setup() .................. 674
   L.7.2 Function: shmem_file_setup() .................. 675
L.8 System V IPC ............................................. 678
   L.8.1 Creating a SYSV shared region .................. 678
      L.8.1.1 Function: sys_shmget() ............ 678
      L.8.1.2 Function: newseg() .................. 679
   L.8.2 Attaching a SYSV Shared Region .................. 681
      L.8.2.1 Function: sys_shmat() ............ 681
L.1 Initialising shmfs

Contents

L.1 Initialising shmfs 622
L.1.1 Function: init_tmpfs() 622
L.1.2 Function: shmem_read_super() 624
L.1.3 Function: shmem_set_size() 626

L.1.1 Function: init_tmpfs() \textit{(mm/shmem.c)}

This function is responsible for registering and mounting the tmpfs and shmemfs filesystems.

\begin{verbatim}
1451 #ifdef CONFIG_TMPFS
1453 static DECLARE_FSTYPE(shmem_fs_type, "shm",
                          shmem_read_super, FS_LITTER);
1454 static DECLARE_FSTYPE(tmpfs_fs_type, "tmpfs",
                          shmem_read_super, FS_LITTER);
1455 #else
1456 static DECLARE_FSTYPE(tmpfs_fs_type, "tmpfs",
                          shmem_read_super, FS_LITTER|FS_NOMOUNT);
1457 #endif

1560 static int __init init_tmpfs(void)
1561 {
1562    int error;
1563
1564    error = register_filesystem(&tmpfs_fs_type);
1565    if (error) {
1566      printk(KERN_ERR "Could not register tmpfs\n");
1567      goto out3;
1568    }
1569 #ifdef CONFIG_TMPFS
1570    error = register_filesystem(&shmem_fs_type);
1571    if (error) {
1572      printk(KERN_ERR "Could not register shm fs\n");
1573      goto out2;
1574    }
1575  #endif
1576  devfs_mk_dir(NULL, "shm", NULL);
1577  shm_mnt = kern_mount(&tmpfs_fs_type);
1578  if (IS_ERR(shm_mnt)) {
1579    error = PTR_ERR(shm_mnt);
1580    printk(KERN_ERR "Could not kern_mount tmpfs\n");
1581    goto out1;
1582  }
\end{verbatim}
1583 /* The internal instance should not do size checking */
1584 shmem_set_size(SHMEM_SB(shm_mnt->mnt_sb), ULONG_MAX, ULONG_MAX);
1585 return 0;
1586
1587 out1:
1588 #ifdef CONFIG_TMPFS
1589 unregister_filesystem(&shmem_fs_type);
1590 out2:
1591 #endif
1592 unregister_filesystem(&tmpfs_fs_type);
1593 out3:
1594 shm_mnt = ERR_PTR(error);
1595 return error;
1596 }
1597 }
1598 module_init(init_tmpfs)

1551 The shm filesystem is only mountable if CONFIG_TMPFS is defined at compile
time. Even if it is not specified, a tmpfs will still be setup for anonymous
shared memory resulting from a fork()

1553 DECLARE_FSTYPE(), declared in <linux/fs.h>, declares tmpfs_fs_type as
a type struct file_system_type and fills in four fields. “tmpfs” is its human
readable name. shmem_read_super() is the function which is used to read the
superblock for the filesystem (a detailed description of superblocks and how
they pertain to filesystems is beyond the scope of this book). FS_LITTER is
a flag that indicates the filesystem tree should be maintained in the dcache.
Finally, the macro sets the module owner of the filesystem to be the module
loading the filesystem

1560 __init places this function in the init section. This means that after the
kernel has finished bootstrapping, the code for the function will be removed

1564-1568 Register the filesystem tmpfs_fs_type which was declared in line 1433.
If it fails, goto out3 where the appropriate error will be returned

1569-1474 If tmpfs is specified at configure time, register the shmem filesystem. If
it fails, goto out2 where tmpfs_fs_type will be unregistered before returning
the error

1575 If /dev/ is being managed by the device filesystem (devfs), then create a new
shm directory. If the kernel does not use devfs, then the system administrator
must manually create the directory

1577 kern_mount() mounts a filesystem internally. In other words, the filesystem
is mounted and active but it is not visible to the user anywhere in the VFS.
The mount point in shm_mnt which is local to the shmem.c file and of type
L.1 Initialising shmfs (init_tmpfs())

struct vfsmount. This variable is needed for searching the filesystem and for unmounting it later.

1578-1582 Ensure the filesystem mounted correctly but if it didn’t, goto out1 where the filesystems will be unregistered before returning the error.

1585 The function shmem_set_size() (See Section L.1.3) is responsible for setting the maximum number of blocks and inodes that may be created in this filesystem.

1598 module_init() in this instance indicates that init_shmem_fs() should be called when the module is loaded. If it is compiled directly into the kernel, the function will be called on system startup.

L.1.2 Function: shmem_read_super() (mm/shmem.c)

This is the callback function provided for the filesystem which “reads” the superblock. With an ordinary filesystem, this would entail reading the information from the disk but as this is a RAM-based filesystem, it instead populates a struct super_block.

1452 static struct super_block *shmem_read_super(struct super_block *sb, void* data, int silent)

1453 {
1454  struct inode *inode;
1455  struct dentry *root;
1456  unsigned long blocks, inodes;
1457  int mode = S_IRWXUGO | S_ISVTX;
1458  uid_t uid = current->fsuid;
1459  gid_t gid = current->fsgid;
1460  struct shmem_sb_info *sbinfo = SHMEM_SB(sb);
1461  struct sysinfo si;
1462
1463  /*
1464  * Per default we only allow half of the physical ram per
1465  * tmpfs instance
1466  */
1467  si_meminfo(&si);
1468  blocks = inodes = si.totalram / 2;
1469 #ifdef CONFIG_TMPFS
1470  if (shmem_parse_options(data, &mode, &uid, &gid, &blocks, &inodes))
1471    return NULL;
1472 #endif
1473
1474  spin_lock_init(&sbinfo->stat_lock);
The parameters are;

\textbf{sb} is the super_block to populate
\textbf{data} contains the mount arguments
\textbf{silent} is unused in this function

Set the default mode, uid and gid. These may be overridden with the parameters passed as mount options

Each super_block is allowed to have a filesystem specific struct that is contained within a union called super_block\_u. The macro SHMEM\_SB() returns the struct shmem\_sb\_info contained within this union

\texttt{si\_meminfo()} populates struct sysinfo with total memory, available memory and usage statistics. The function is defined in \texttt{arch/i386/mm/init.c} and is architecture dependant

By default, only allow the filesystem to consume half of total available physical memory

If tmpfs is available, parse the mount options allowing them to override the defaults
L.1 Initialising shmfs (shmem_read_super())

1475 Acquire the lock protecting sbinfo which is the struct shmem_sb_info in the super_block

1483 Populate the sb and sbinfo fields

1484 The shmem_ops is a struct of function pointers for super block operations such as remounting the filesystem and deleting an inode

1485-1487 This block allocates a special inode which represents the root of the filesystem

1489-1490 Set the uid and gid of the root of the new filesystem

1496 Set the root inode into the super_block

1497 Return the populated superblock

L.1.3 Function: shmem_set_size() (mm/shmem.c)

This function updates the number of available blocks and inodes in the filesystem. It is set while the filesystem is being mounted or remounted.

861 static int shmem_set_size(struct shmem_sb_info *info,
862                        unsigned long max_blocks,
863                        unsigned long max_inodes)
864 {
865     int error;
866     unsigned long blocks, inodes;
867     spin_lock(&info->stat_lock);
868     blocks = info->max_blocks - info->free_blocks;
869     inodes = info->max_inodes - info->free_inodes;
870     error = -EINVAL;
871     if (max_blocks < blocks)
872         goto out;
873     if (max_inodes < inodes)
874         goto out;
875     error = 0;
876     info->max_blocks = max_blocks;
877     info->free_blocks = max_blocks - blocks;
878     info->max_inodes = max_inodes;
879     info->free_inodes = max_inodes - inodes;
880     out:
881     spin_unlock(&info->stat_lock);
882     return error;
883 }
The parameters are the info representing the filesystem superblock, the maximum number of blocks (max_blocks) and the maximum number of inodes (max_inodes)

Lock the superblock info spinlock

Calculate the number of blocks current in use by the filesystem. On initial mount, this is unimportant, but if the filesystem is being remounted, the function must make sure that the new filesystem is not too small

Calculate the number of inodes currently in use

If the remounted filesystem would have too few blocks to store the current information, goto out to return -EINVAL

Similarly, make sure there are enough available inodes or return -EINVAL

It is safe to mount the filesystem so set error to 0 indicating that this operation will be successful

Set the maximum number of blocks and number of available blocks in the filesystems superblock info struct

Set the maximum and available number of inodes

Unlock the filesystems superblock info struct

Return 0 if successful or -EINVAL if not
L.2 Creating Files in tmpfs

Contents

L.2 Creating Files in tmpfs 628
L.2.1 Function: shmem_create() 628
L.2.2 Function: shmem_mknod() 628
L.2.3 Function: shmem_get_inode() 629

L.2.1 Function: shmem_create() (mm/shmem.c)
This is the top-level function called when creating a new file.

1164 static int shmem_create(struct inode *dir,
                   struct dentry *dentry,
                   int mode)
1165 {
    return shmem_mknod(dir, dentry, mode | S_IFREG, 0);
1167 }

1164 The parameters are:

    dir    is the inode of the directory the new file is being created in
    dentry is the dentry of the new file being created
    mode   is the flags passed to the open system call

1166 Call shmem_mknod() (See Section L.2.2) adding the S_IFREG flag to the mode
flags so a regular file will be created

L.2.2 Function: shmem_mknod() (mm/shmem.c)

1139 static int shmem_mknod(struct inode *dir,
                   struct dentry *dentry,
                   int mode, int dev)
1140 {
1141     struct inode *inode = shmem_get_inode(dir->i_sb, mode, dev);
1142     int error = -ENOSPC;
1143
1144     if (inode) {
1145         dir->i_size += BOGO_DIRENT_SIZE;
1146         dir->i_ctime = dir->i_mtime = CURRENT_TIME;
1147         d_instantiate(dentry, inode);
1148         dget(dentry); /* Extra count - pin the dentry in core */
1149         error = 0;
1150     }
1151     return error;
1152 }
Call `shmem_get_inode()` (See Section L.2.3) to create a new inode

If the inode was successfully created, update the directory statistics and instantiate the new file

Update the size of the directory

Update the ctime and mtime fields

Instantiate the inode

Take a reference to the dentry so that it will be pinned and not accidentally reclaimed during pageout. Unlike normal files, there is no automatic way of recreating dentries once they are deleted

Indicate the call ended successfully

Return success or -ENOSPC on error

### L.2.3 Function: `shmem_get_inode()` *(mm/shmem.c)*

```c
struct inode *shmem_get_inode(struct super_block *sb,
    int mode,
    int dev)
{
    struct inode *inode;
    struct shmem_inode_info *info;
    struct shmem_sb_info *sbinfo = SHMEM_SB(sb);

    spin_lock(&sbinfo->stat_lock);
    if (!sbinfo->free_inodes) {
        spin_unlock(&sbinfo->stat_lock);
        return NULL;
    }

    sbinfo->free_inodes--;
    spin_unlock(&sbinfo->stat_lock);

    inode = new_inode(sb);
```

This preamble section is responsible for updating the free inode count and allocating an inode with `new_inode()`.

Acquire the `sbinfo` spinlock as it is about to be updated

Make sure there are free inodes and if not, return NULL

Update the free inode count and free the lock
new_inode() is part of the filesystem layer and declared in `<linux/fs.h>`. Exactly how it works is beyond the scope of this document but the summary is simple. It allocates an inode from the slab allocator, zeros most fields and populates inode→i_sb, inode→i_dev and inode→i_blkbits based on information in the super block.

```c
if (inode) {
    inode->i_mode = mode;
    inode->i_uid = current->fsuid;
    inode->i_gid = current->fsgid;
    inode->i_blocks = 0;
    inode->i_rdev = NODEV;
    inode->i_mapping->a_ops = &shmem_aops;
    inode->i_atime = inode->i_mtime = inode->i_ctime = CURRENT_TIME;
    info = SHMEM_I(inode);
    info->inode = inode;
    spin_lock_init(&info->lock);
    switch (mode & S_IFMT) {
        default:              
            init_special_inode(inode, mode, dev);
            break;
        case S_IFREG:
            inode->i_op = &shmem_inode_operations;
            inode->i_fop = &shmem_file_operations;
            spin_lock(&shmem_ilock);
            list_add_tail(&info->list, &shmem_inodes);
            spin_unlock(&shmem_ilock);
            break;
        case S_IFDIR:
            inode->i_nlink++;
            /* Some things misbehave if size == 0 on a directory */
            inode->i_size = 2 * BOGO_DIRENT_SIZE;
            inode->i_op = &shmem_dir_inode_operations;
            inode->i_fop = &dcache_dir_ops;
            break;
        case S_IFLNK:
            break;
    }
}
return inode;
```

824-858 Fill in the inode fields if created successfully.
825-830 Fill in the basic inode information

831 Set the address_space_operations to use shmem_aops which sets up the function shmem_writepage() (See Section L.6.1) to be used as a page write-back callback for the address_space

832-834 Fill in more basic information

835-836 Initialise the inodes semaphore and spinlock

836-856 Determine how to fill the remaining fields based on the mode flags passed in

838 In this case, a special inode is being created. Specifically, this is while the filesystem is being mounted and the root inode is being created

840-846 Create an inode for a regular file. The main point to note here is that the inode→i_op and inode→i_fop fields are set to shmem_inode_operations and shmem_file_operations respectively

847-852 Create an inode for a new directory. The i_nlink and i_size fields are updated to show the increased number of files and the size of the directory. The main point to note here is that the inode→i_op and inode→i_fop fields are set to shmem_dir_inode_operations and dcach_dir_ops respectively

854-855 If linking a file, do nothing for now as it is handled by the parent function shmem_link()

858 Return the new inode or NULL if it could not be created
L.3 File Operations in tmpfs

Contents

L.3.1 Memory Mapping 632
  L.3.1.1 Function: shmem_mmap() 632
L.3.2 Reading Files 633
  L.3.2.1 Function: shmem_file_read() 633
  L.3.2.2 Function: do_shmem_file_read() 634
  L.3.2.3 Function: file_read_actor() 637
L.3.3 Writing 638
  L.3.3.1 Function: shmem_file_write() 638
L.3.4 Symbolic Linking 641
  L.3.4.1 Function: shmem_symlink() 641
  L.3.4.2 Function: shmem_readlink_inline() 644
  L.3.4.3 Function: shmem_follow_link_inline() 644
  L.3.4.4 Function: shmem_readlink() 644
L.3.5 Synchronising 645
  L.3.5.1 Function: shmem_sync_file() 645

L.3.1 Memory Mapping

The tasks for memory mapping a virtual file are simple. The only changes that need to be made is to update the VMAs vm_operations_struct field (vma→vm_ops) to use the shmfs equivalents for faulting.

L.3.1.1 Function: shmem_mmap() (mm/shmem.c)

```c
796 static int shmem_mmap(struct file * file, struct vm_area_struct * vma) {
797     struct vm_operations_struct *ops;
798     struct inode *inode = file->f_dentry->d_inode;
799
800     ops = &shmem_vm_ops;
801     if (!S_ISREG(inode->i_mode))
802         return -EACCESS;
803     UPDATE_ATIME(inode);
804     vma->vm_ops = ops;
805     return 0;
806 }
```

801 ops is now the vm_operations_struct to be used for the virtual filesystem

802 Make sure that the inode being mapped is a regular file. If not, return -EACCESS

804 Update the atime for the inode to show it was accessed
Update vma→vm_ops so that shmem_nopage() (See Section L.5.1.1) will be used to handle page faults within the mapping.

L.3.2 Reading Files

L.3.2.1 Function: shmem_file_read() (mm/shmem.c)
This is the top-level function called for read()ing a tmpfs file.

```c
static ssize_t shmem_file_read(struct file *filp, char *buf,
    size_t count, loff_t *ppos)
```

The parameters are:
- `filp` is a pointer to the `struct file` being read
- `buf` is the buffer that should be filled
- `count` is the number of bytes that should be read
- `ppos` is the current position

- `count` cannot be negative

- `access_ok()` ensures that it is safe to write `count` number of bytes to the userspace buffer. If it can’t, -EFAULT will be returned

- Initialise a `read_descriptor_t` struct which will eventually be passed to `file_read_actor()` (See Section L.3.2.3)
Call do_shmem_file_read() to start performing the actual read

Return the number of bytes that were written to the userspace buffer

If none were written, return the error

This function retrieves the pages needed for the file read with shmem_getpage() and calls file_read_actor() to copy the data to userspace.

```c
static void do_shmem_file_read(struct file *filp, loff_t *ppos, read_descriptor_t *desc)
{
    struct inode *inode = filp->f_dentry->d_inode;
    struct address_space *mapping = inode->i_mapping;
    unsigned long index, offset;
    unsigned long end_index, nr, ret;
    index = *ppos >> PAGE_CACHE_SHIFT;
    offset = *ppos & ~PAGE_CACHE_MASK;
    for (;;) {
        struct page *page = NULL;
        unsigned long end_index, nr, ret;
        end_index = inode->i_size >> PAGE_CACHE_SHIFT;
        if (index > end_index) break;
        if (index == end_index) {
            nr = inode->i_size & ~PAGE_CACHE_MASK;
            if (nr <= offset) break;
        }
        desc->error = shmem_getpage(inode, index, &page, SGP_READ);
        if (desc->error) {
            if (desc->error == -EINVAL)
                desc->error = 0;
            break;
        }
        nr = PAGE_CACHE_SIZE;
        end_index = inode->i_size >> PAGE_CACHE_SHIFT;
        if (index == end_index) {
            nr = inode->i_size & ~PAGE_CACHE_MASK;
            if (nr <= offset) break;
        }
        desc->error = shmem_getpage(inode, index, &page, SGP_READ);
        if (desc->error) {
            if (desc->error == -EINVAL)
                desc->error = 0;
            break;
        }
        nr = PAGE_CACHE_SIZE;
        end_index = inode->i_size >> PAGE_CACHE_SHIFT;
        if (index == end_index) {
            nr = inode->i_size & ~PAGE_CACHE_MASK;
```
if (nr <= offset) {
    page_cache_release(page);
    break;
}

nr -= offset;

if (page != ZERO_PAGE(0)) {
    if (mapping->i_mmap_shared != NULL)
        flush_dcache_page(page);
    /*
    * Mark the page accessed if we read the
    * beginning or we just did an lseek.
    */
    if (!offset || !filp->f_reada)
        mark_page_accessed(page);
}

ret = file_read_actor(desc, page, offset, nr);
offset += ret;
index += offset >> PAGE_CACHE_SHIFT;
offset &= ~PAGE_CACHE_MASK;

page_cache_release(page);
if (ret != nr || !desc->count)
    break;

*ppos = (((loff_t) index << PAGE_CACHE_SHIFT) + offset);
filp->f_reada = 1;
UPDATE_ATIME(inode);
}

Retrieve the inode and mapping using the struct file

index is the page index within the file that contains the data

offset is the offset within the page that is currently being read

Loop until the requested number of bytes has been read. nr is the number of bytes that are still to be read within the current page. desc->count starts as the number of bytes to read and is decremented by file_read_actor() (See Section L.3.2.3)

end_index is the index of the last page in the file. Break when the end of the file is reached
L.3.2 Reading Files (do_shmem_file_read())

1019-1023 When the last page is reached, set nr to be the number of bytes to be read within this page. If the file pointer is after nr, break as there is no more data to be read. This could happen after the file was truncated.

1025-1030 shmem_getpage() (See Section L.5.1.2) will locate the requested page in the page cache, swap cache or page it in. If an error occurs, record it in desc→error and return.

1036 nr is the number of pages that must be read from the page so initialise it to the size of a page as this full page is being read.

1037 Initialise end_index which is index of the page at the end of the file.

1038-1044 If this is the last page in the file, update nr to be the number of bytes in the page. If nr is currently after the end of the file (could happen after truncate), then release the reference to the page (taken by shmem_getpage()) and exit the loop.

1045 Update the number of bytes to be read. Remember that offset is where the file reader is currently within the page.

1047-1061 If the page being read is not the global zero page, take care of potential aliasing problems by calling flush_dcache_page(). If the page is being read the first time or an lseek() just occurred (f_reada is zero), then mark the page accessed with mark_page_accesssed().

1073 Call file_read_actor() (See Section L.3.2.3) to copy the data to userspace. It returns the number of bytes that were copied and updates the user buffer pointers and remaining count.

1074 Update the offset within the page being read.

1075 Move the index to the next page if necessary.

1076 Ensure that offset is an offset within a page.

1078 Release the reference to the page being copied. The reference was taken by shmem_getpage().

1079-1080 If the requested bytes have been read, return.

1083 Update the file pointer.

1084 Enable file readahead.

1085 Update the access time for the inode as it has just been read from.
L.3.2.3 Function: file_read_actor() (mm/filemap.c)

This function is responsible for copying data from a page to a userspace buffer. It is ultimately called by a number of functions including generic_file_read(), generic_file_write() and shmem_file_read().

```c
int file_read_actor(read_descriptor_t * desc,
                    struct page *page,
                    unsigned long offset,
                    unsigned long size)
{
    char *kaddr;
    unsigned long left, count = desc->count;
    if (size > count)
        size = count;
    kaddr = kmap(page);
    left = __copy_to_user(desc->buf, kaddr + offset, size);
    kunmap(page);
    if (left) {
        size -= left;
        desc->error = -EFAULT;
    }
    desc->count = count - size;
    desc->written += size;
    desc->buf += size;
    return size;
}
```

The parameters are:

- **desc** is a structure containing information about the read, including the buffer and the total number of bytes that are to be read from this file
- **page** is the page containing file data that is to be copied to userspace
- **offset** is the offset within the page that is being copied
- **size** is the number of bytes to be read from page

**1672 count** is now the number of bytes that are to be read from the file

**1674-1675** Make sure to not read more bytes than are requested

**1677** Map the page into low memory with kmap(). See Section I.1.0.5

**1678** Copy the data from the kernel page to the userspace buffer
1679 Unmap the page. See Section I.3.1

1644-1647 If all the bytes were not copied, it must be because the buffer was not accessible. Update size so that desc->count will reflect how many bytes are still to be copied by the read. -EFAULT will be returned to the process performing the read.

1685-1687 Update the desc struct to show the current status of the read.

1688 Return the number of bytes that were written to the userspace buffer.

L.3.3 Writing

L.3.3.1 Function: shmem_file_write()  (mm/shmem.c)

925 shmem_file_write(struct file *file, const char *buf,
size_t count, loff_t *ppos)
926 {
927 struct inode  *inode = file->f_dentry->d_inode;
928 loff_t  pos;
929 unsigned long  written;
930 int  err;
931
932 if ((ssize_t) count < 0)
933 return -EINVAL;
934
935 if (!access_ok(VERIFY_READ, buf, count))
936 return -EFAULT;
937
938 down(&inode->i_sem);
939
940 pos = *ppos;
941 written = 0;
942
943 err = precheck_file_write(file, inode, &count, &pos);
944 if (err || !count)
945 goto out;
946
947 remove_suid(inode);
948 inode->i_ctime = inode->i_mtime = CURRENT_TIME;

Function preamble.

927 Get the inode that represents the file being written.

932-933 Return -EINVAL if the user tries to write a negative number of bytes.
935-936 Return -EFAULT if the userspace buffer is inaccessible

938 Acquire the semaphore protecting the inode

940 Record the beginning of where the write is taking place

941 Initialise the written number of bytes to 0

943 precheck_file_write() performs a number of checks to make sure the write is ok to proceed. This includes updating pos to be the end of the file if opened in append mode and checking that the process limits will not be exceeded

944-945 If the write cannot proceed, goto out

947 Clear the SUID bit if it is set

948 Update the inodes ctime and mtime

950   do {
951       struct page *page = NULL;
952       unsigned long bytes, index, offset;
953       char *kaddr;
954       int left;
955
956       offset = (pos & (PAGE_CACHE_SIZE -1)); /* Within page */
957       index = pos >> PAGE_CACHE_SHIFT;
958       bytes = PAGE_CACHE_SIZE - offset;
959       if (bytes > count)
960           bytes = count;
961
962       /*
963        * We don’t hold page lock across copy from user -
964        * what would it guard against? - so no deadlock here.
965        */
966
967       err = shmem_getpage(inode, index, &page, SGP_WRITE);
968       if (err)
969           break;
970
971       kaddr = kmap(page);
972       left = __copy_from_user(kaddr + offset, buf, bytes);
973       kunmap(page);
974
975       written += bytes;
976       count -= bytes;
977       pos += bytes;
978       buf += bytes;
if (pos > inode->i_size)
    inode->i_size = pos;
flush_dcache_page(page);
SetPageDirty(page);
SetPageReferenced(page);
page_cache_release(page);

if (left) {
    pos -= left;
    written -= left;
    err = -EFAULT;
    break;
}
}
while (count);

*ppos = pos;
if (written)
    err = written;

out:
up(&inode->i_sem);
return err;

950-993 Loop until all the requested bytes have been written
956 Set offset to be the offset within the current page being written
957 index is the page index within the file current being written
958 bytes is the number of bytes within the current page remaining to be written
959-960 If bytes indicates that more bytes should be written than was requested
(count), set bytes to count
967-969 Locate the page to be written to. The SGP_WRITE flag indicates that a
page should be allocated if one does not already exist. If the page could not
be found or allocated, break out of the loop
971-973 Map the page to be written to and copy the bytes from the userspace
buffer before unmapping the page again
975 Update the number of bytes written
976 Update the number of bytes remaining to write
977 Update the position within the file
L.3.4 Symbolic Linking

L.3.4.1 Function: shmem_symlink()  (mm/shmem.c)

This function is responsible for creating a symbolic link `symname` and deciding where to store the information. The name of the link will be stored in the inode if the name is small enough and in a page frame otherwise.

```c
1272 static int shmem_symlink(struct inode * dir, 
   struct dentry *dentry,  
   const char *symname)
```

```c
1273 {
1274   int error;
1275   int len;
1276   struct inode *inode;
1277   struct page *page = NULL;
1278   char *kaddr;
1279   struct shmem_inode_info *info;
1280   len = strlen(symname) + 1;
1281   if (len > PAGE_CACHE_SIZE) 
1282     return -ENAMETOOLONG;
1283   inode = shmem_get_inode(dir->i_sb, S_IFLNK|S_IRWXUGO, 0);
1284   if (!inode) 
1285     return -ENOSPC;
1286   info = SHMEM_I(inode);
1287   inode->i_size = len-1;
```
This block performs basic sanity checks and creating a new inode for the symbolic link.

1272 The parameter symname is the name of the link to create

1281 Calculate the length (len) of the link

1282-1283 If the name is larger than a page, return -ENAMETOOLONG

1285-1287 Allocate a new inode. Return -ENOSPC if it fails

1289 Get the private information struct

1290 The size of the inode is the length of the link

1291 if (len <= sizeof(struct shmem_inode_info)) {
1292 /* do it inline */
1293 memcpy(info, symname, len);
1294 inode->i_op = &shmem_symlink_inline_operations;
1295 } else {
1296 error = shmem_getpage(inode, 0, &page, SGP_WRITE);
1297 if (error) {
1298 iput(inode);
1299 return error;
1300 }
1301 inode->i_op = &shmem_symlink_inode_operations;
1302 spin_lock(&shmem_ilock);
1303 list_add_tail(&info->list, &shmem_inodes);
1304 spin_unlock(&shmem_ilock);
1305 kaddr = kmap(page);
1306 memcpy(kaddr, symname, len);
1307 kunmap(page);
1308 SetPageDirty(page);
1309 page_cache_release(page);
1310 }

This block is responsible for storing the link information.

1291-1295 If the length of the name is smaller than the space used for the shmem_inode_info, then copy the name into the space reserved for the private struct

1294 Set the inode->i_op to shmem_symlink_inline_operations which has functions which know the link name is in the inode

1295-1314 Allocate a page to store the the link in

1296 Acquire the private information semaphore
L.3.4 Symbolic Linking (shmem_symlink())

1297 Allocate a page with shmem_getpage_locked

1298-1302 If an error occurred, drop the reference to the inode and return the error

1301 Use shmem_symlink_inode_operations which understands that the link information is contained within a page

1302 shmem_ilock is a global spinlock which protects a global linked list of inodes which are linked via the private information structs info → list field

1303 Add the new inode to the global list

1304 Release shmem_ilock

1305 Map the page

1306 Copy in the link information

1307 Unmap the page

1308-1309 Set the page dirty and unlock it

1310 Release our reference to it

1311 Release the private information semaphore

1311     dir->i_size += BOGO_DIRENT_SIZE;
1312     dir->i_ctime = dir->i_mtime = CURRENT_TIME;
1313         d_instantiate(dentry, inode);
1314     dget(dentry);
1315     return 0;
1316 }

1311 Increment the size of the directory as a new inode has been added. BOGO_DIRENT_SIZE is just a pseudo size of inodes so that ls output looks nice

1312 Update the i_ctime and i_mtime

1313-1314 Instantiate the inode

1315 Return success
L.3.4.2 Function: `shmem_readlink_inline()` *(mm/shmem.c)*

1318 static int shmem_readlink_inline(struct dentry *dentry,  
  char *buffer, int buflen)
1319 {
  return vfs_readlink(dentry, buffer, buflen,  
    (const char *)SHMEM_I(dentry->d_inode));
1320 }

1320 The link name is contained within the inode so pass it as a parameter to the  
VFS layer with `vfs_readlink()`

L.3.4.3 Function: `shmem_follow_link_inline()` *(mm/shmem.c)*

1323 static int shmem_follow_link_inline(struct dentry *dentry,  
  struct nameidata *nd)
1324 {
  return vfs_follow_link(nd,  
    (const char *)SHMEM_I(dentry->d_inode));
1325 }

1209 The link name is contained within the inode so pass it as a parameter to the  
VFS layer with `vfs_followlink()`

L.3.4.4 Function: `shmem_readlink()` *(mm/shmem.c)*

1328 static int shmem_readlink(struct dentry *dentry,  
  char *buffer, int buflen)
1329 {
  struct page *page = NULL;
  int res = shmem_getpage(dentry->d_inode, 0, &page, SGP_READ);
  if (res)
    return res;
  res = vfs_readlink(dentry, buffer, buflen, kmap(page));
  kunmap(page);
  mark_page_accessed(page);
  page_cache_release(page);
  return res;
1330 }

1331 The link name is contained in a page associated with the symlink so call  
`shmem_getpage()`(See Section L.5.1.2) to get a pointer to it

1332-1333 If an error occurred, return NULL

1334 Map the page with `kmap()` (See Section I.1.0.5) and pass it as a pointer to  
`vfs_readlink()`. The link is at the beginning of the page
Unmap the page
Mark the page accessed
Drop our reference to the page taken by shmem_getpage()
Return the link

The link name is within a page so get the page with shmem_getpage()
Return the error if one occurred
Map the page and pass it as a pointer to vfs_follow_link()
Unmap the page
Drop our reference to the page
Return success

L.3.5 Synchronising

L.3.5.1 Function: shmem_sync_file() (mm/shmem.c)
This function simply returns 0 as the file exists only in memory and does not need to be synchronised with a file on disk.

The link name is within a page so get the page with shmem_getpage()
Return the error if one occurred
Map the page and pass it as a pointer to vfs_follow_link()
Unmap the page
Drop our reference to the page
Return success

L.3.5 Synchronising

L.3.5.1 Function: shmem_sync_file() (mm/shmem.c)
This function simply returns 0 as the file exists only in memory and does not need to be synchronised with a file on disk.
L.4 Inode Operations in tmpfs

Contents

L.4 Inode Operations in tmpfs 646
L.4.1 Truncating 646
  L.4.1.1 Function: shmem_truncate() 646
  L.4.1.2 Function: shmem_truncate_indirect() 647
  L.4.1.3 Function: shmem_truncate_direct() 649
  L.4.1.4 Function: shmem_free_swp() 650
L.4.2 Linking 651
  L.4.2.1 Function: shmem_link() 651
L.4.3 Unlinking 652
  L.4.3.1 Function: shmem_unlink() 652
L.4.4 Making Directories 652
  L.4.4.1 Function: shmem_mkdir() 652
L.4.5 Removing Directories 653
  L.4.5.1 Function: shmem_rmdir() 653
  L.4.5.2 Function: shmem_empty() 653
  L.4.5.3 Function: shmem_positive() 654

L.4.1 Truncating

L.4.1.1 Function: shmem_truncate() (mm/shmem.c)

By the time this function has been called, the inode→i_size has been set to
the new size by vmtruncate(). It is the job of this function to either create or
remove pages as necessary to set the size of the file.

351 static void shmem_truncate(struct inode *inode) 352 {
353     struct shmem_inode_info *info = SHMEM_I(inode);
354     struct shmem_sb_info *sinfo = SHMEM_SB(inode->i_sb);
355     unsigned long freed = 0;
356     unsigned long index;
357
358     inode->i_ctime = inode->i_mtime = CURRENT_TIME;
359     index = (inode->i_size + PAGE_CACHE_SIZE - 1) >> PAGE_CACHE_SHIFT;
360     if (index >= info->next_index)
361         return;
362
363     spin_lock(&info->lock);
364     while (index < info->next_index)
365         freed += shmem_truncate_indirect(info, index);
366     BUG_ON(info->swapped > info->next_index);
367     spin_unlock(&info->lock);
L.4.1 Truncating \texttt{(shmem\_truncate())}

\begin{verbatim}
spin_lock(&sbinfo->stat_lock);
sbinfo->free_blocks += freed;
inode->i_blocks -= freed*BLOCKS_PER_PAGE;
spin_unlock(&sbinfo->stat_lock);
\end{verbatim}

353 Get the private filesystem information for this inode with \texttt{SHMEM\_I()}

354 Get the superblock private information

358 Update the ctime and mtime for the inode

359 Get the index of the page that is the new end of the file. The old size is stored in \texttt{info->next\_index}

360-361 If the file is being expanded, just return as the global zero page will be used to represent the expanded region

363 Acquire the private \texttt{info} spinlock

364-365 Continually call \texttt{shmem\_truncate\_indirect()} until the file is truncated to the desired size

366 It is a bug if the \texttt{shmem\_info\_info} struct indicates that there are more pages swapped out than there are pages in the file

367 release the private \texttt{info} spinlock

369 Acquire the superblock private \texttt{info} spinlock

370 Update the number of free blocks available

371 Update the number of blocks being used by this inode

372 Release the superblock private \texttt{info} spinlock

L.4.1.2 \textbf{Function:} \texttt{shmem\_truncate\_indirect()}  \texttt{(mm/shmem.c)}

This function locates the last doubly-indirect block in the inode and calls \texttt{shmem\_truncate\_direct()} to truncate it.

\begin{verbatim}
static inline unsigned long
shmem_truncate_indirect(struct shmem_inode_info *info, unsigned long index)
{
    swp_entry_t ***base;
    unsigned long baseidx, start;
    unsigned long len = info->next_index;
    unsigned long freed;

    \end{verbatim}
316 if (len <= SHMEM_NR_DIRECT) {
317 info->next_index = index;
318 if (!info->swapped)
319 return 0;
320 freed = shmem_free_swp(info->i_direct + index,
321 info->i_direct + len);
322 info->swapped -= freed;
323 return freed;
324 }
325
326 if (len <= ENTRIES_PER_PAGE_PAGE/2 + SHMEM_NR_DIRECT) {
327 len -= SHMEM_NR_DIRECT;
328 base = (swp_entry_t ***) &info->i_indirect;
329 baseidx = SHMEM_NR_DIRECT;
330 } else {
331 len -= ENTRIES_PER_PAGE_PAGE/2 + SHMEM_NR_DIRECT;
332 BUG_ON(len > ENTRIES_PER_PAGE_PAGE*ENTRIES_PER_PAGE_PAGE/2);
333 baseidx = len - 1;
334 baseidx -= baseidx % ENTRIES_PER_PAGE_PAGE;
335 base = (swp_entry_t ***) info->i_indirect +
336 ENTRIES_PER_PAGE_PAGE/2 + baseidx/ENTRIES_PER_PAGE_PAGE;
337 len -= baseidx;
338 baseidx += ENTRIES_PER_PAGE_PAGE/2 + SHMEM_NR_DIRECT;
339 }
340
341 if (index > baseidx) {
342 info->next_index = index;
343 start = index - baseidx;
344 } else {
345 info->next_index = baseidx;
346 start = 0;
347 }
348 return *base? shmem_truncate_direct(info, base, start, len): 0;
349 }

313 len is the second last page that is currently in use by the file

316-324 If the file is small and all entries are stored in the direct block information, simply call shmem_free_swp() passing it the first swap entry in info->i_direct and the number of entries to truncate.

326-339 The pages to be truncated are in the indirect blocks somewhere. This section of code is dedicated to calculating three variables, base, baseidx and len. base is the beginning of the page that contains pointers to swap entries to be truncated. baseidx is the page index of the first entry within the indirect
block being used and \texttt{len} is the number of entries to be truncated from in this pass.

326-330 This calculates the variables for a doubly indirect block. The \texttt{base} is then set to the swap entry at the beginning of \texttt{info->i间接}. \texttt{baseidx} is \texttt{SHMEM_NR DIRECT} which is the page index at the beginning of \texttt{info->i indirect}. At this point, \texttt{len} is the number of pages in the file so the number of direct blocks is subtracted to leave the remaining number of pages.

330-339 Else this is a triply indexed block so the next level must be traversed before the \texttt{base}, \texttt{baseidx} and \texttt{len} are calculated.

341-344 If the file is going to be bigger after the truncation, update \texttt{next_index} to the new end of file and make \texttt{start} the beginning of the indirect block.

344-347 If the file is been made smaller, move the current end of the file to the beginning of this indirect block that is about to be truncated.

348 If there is a block at \texttt{base}, call \texttt{shmem_truncate_direct()} to truncate pages in it.

L.4.1.3 Function: \texttt{shmem_truncate_direct()} \textit{(mm/shmem.c)}

This function is responsible for cycling through an indirect block and calling \texttt{shmem_free_swp} for each page that contains swap vectors which are to be truncated.

264 static inline unsigned long
265 shmem_truncate_direct(struct shmem_inode_info *info, 
266        swp_entry_t ***dir, 
267        unsigned long start, unsigned long len)
268 
269     { 
270     swp_entry_t **last, **ptr; 
271     unsigned long off, freed_swp, freed = 0;
272 
273     last = *dir + (len + ENTRIES_PER_PAGE - 1) / ENTRIES_PER_PAGE;
274     off = start % ENTRIES_PER_PAGE;
275     for (ptr = *dir + start/ENTRIES_PER_PAGE; ptr < last; ptr++, off = 0) { 
276     if (!*ptr) 
277         continue;
278     if (info->swapped) {
279         freed_swp = shmem_free_swp(*ptr + off, 
280             *ptr + ENTRIES_PER_PAGE);
281         info->swapped -= freed_swp;
282         freed += freed_swp;
L.4.1 Truncating (shmem_truncate_direct())

282 } } 
283 284 if (!off) {
285 freed++; 
286 free_page((unsigned long) *ptr); 
287 *ptr = 0; 
288 }
289 }
290 
291 if (!start) {
292 freed++; 
293 free_page((unsigned long) *dir); 
294 *dir = 0; 
295 }
296 return freed;
297 }

270 last is the last page within the indirect block that is to be truncated
271 off is the offset within the page that the truncation is to if this is a partial truncation rather than a full page truncation

273-289 Beginning with the startth block in dir, truncate pages until last is reached
274-275 If no page is here, continue to the next one
277-282 If the info struct indicates that there are pages swapped out belonging to this inode, call shmem_free_swp() to free any swap slot associated with this page. If one was freed, update infoswapped and increment the count of the freed number of pages
284-288 If this is not a partial truncate, free the page
291-295 If this whole indirect block is now free, reclaim the page
296 Return the number of pages freed

L.4.1.4 Function: shmem_free_swp() (mm/shmem.c)

This frees count number of swap entries starting with the entry at dir.

240 static int shmem_free_swp(swp_entry_t *dir, swp_entry_t *edir) 
241 {
242 swp_entry_t *ptr; 
243 int freed = 0; 
244 
245 for (ptr = dir; ptr < edir; ptr++) {
L.4.2 Linking

L.4.2.1 Function: shmem_link() (mm/shmem.c)

This function creates a hard link with dentry to old_dentry.

1172 static int shmem_link(struct dentry *old_dentry,
                           struct inode *dir,
                           struct dentry *dentry)
1173 {
1174     struct inode *(inode = old_dentry->d_inode;    
1175     if (S_ISDIR(inode->i_mode))
1176         return -EPERM;
1177     dir->i_size += BOGO_DIRENT_SIZE;                  
1178     inode->i_ctime = dir->i_ctime = dir->i_mtime = CURRENT_TIME;
1179     inode->i_nlink++;
1180     atomic_inc(&inode->i_count);
1181     dget(dentry);
1182     d_instantiate(dentry, inode);
1183     return 0;
1184 }                                                  
1185 }                                                  
1186 }                                                  
1187 }                                                  
1188 }                                                  
1189 }                                                  
1190 }                                                  
1191 }                                                  
1192 }                                                  
1193 }                                                  
1194 Get the inode corresponding to old_dentry
1195
1196-1197 If it is linking to a directory, return -EPERM. Strictly speaking, root
1198 should be allowed to hard-link directories although it is not recommended
1199 because of the possibility of creating a loop within the filesystem which utilities
1200 like find get lost in. tmpfs simply does not allow the hard-linking of directories
1201
1202 Increment the size of the directory with the new link

245-251 Loop through each of the swap entries to be freed
246-250 If a swap entry exists, free it with free_swap_and_cache() and set the
247 swap entry to 0. Increment the number of pages freed
252 Return the total number of pages freed
Update the directories mtime and ctime. Update the inodes ctime.
Increment the number of links leading to inode.
Get an extra reference to the new dentry with dget().
Instantiate the new dentry.
Return success.

**L.4.3 Unlinking**

**L.4.3.1 Function: shmem_unlink() (mm/shmem.c)**

```c
1221 static int shmem_unlink(struct inode* dir,
    struct dentry *dentry)
1222 {
1223     struct inode *inode = dentry->d_inode;
1224     dir->i_size -= BOGO_DIRENT_SIZE;
1225     inode->i_ctime = dir->i_ctime = dir->i_mtime = CURRENT_TIME;
1226     inode->i_nlink--;
1227     dput(dentry);
1228     return 0;
1229 }
```

Get theinode for the dentry being unlinked.
Update the directory inodes size.
Update the various ctime and mtime variables.
Decrement the number of links to the inode.
Call dput() to decrement the reference to the dentry. This function will also call iput() to clear up the inode if it’s reference count reaches zero.

**L.4.4 Making Directories**

**L.4.4.1 Function: shmem_mkdir() (mm/shmem.c)**

```c
1154 static int shmem_mkdir(struct inode *dir,
    struct dentry *dentry,
    int mode)
1155 {
1156     int error;
1157     if ((error = shmem_mknod(dir, dentry, mode | S_IFDIR, 0)))
```
L.4.5 Removing Directories

L.4.5.1 Function: shmem_rmdir()  

Call shmem_mknod() (See Section L.2.2) to create a special file. By specifying the S_IFDIR flag, a directory will be created.

Increment the parent directory’s i_nlink field.

L.4.5 Removing Directories

L.4.5.1 Function: shmem_rmdir()  (mm/shmem.c)

Check to see if the directory is empty with shmem_empty() (See Section L.4.5.2). If it is not, return -ENOTEMPTY.

Decrement the parent directory’s i_nlink field.

Return the result of shmem_unlink() (See Section L.4.3.1) which should delete the directory.

L.4.5.2 Function: shmem_empty()  (mm/shmem.c)

This function checks to see if a directory is empty or not.

Initialize the list head for directories.

Spin lock on dcache_lock.

Retrieve the next directory in the list.

While list is not equal to dentry->d_subdirs.

Retrieve the directory entry from the list.

If shmem_positive is true.

if (shmem_positive(de)) {

}
L.4.5 Removing Directories (\texttt{shmem\_empty()})

1212 spin_unlock(&dcache_lock);
1213 return 0;
1214 }
1215 list = list->next;
1216 }
1217 spin_unlock(&dcache_lock);
1218 return 1;
1219 }

1205 The \texttt{dcache\_lock} protect many things but it mainly protects dcache lookups which is what will be required for this function so acquire it

1208 Cycle through the subdirs list, which contains all children dentries, and see can one active dentry be found. If it is, 0 will be returned indicating the directory is not empty

1209 Get the dentry for this child

1211 \texttt{shmem\_positive()} (See Section L.4.5.3) returns if the dentry has a valid inode associated with it and is currently hashed. If it’s hashed, it means that the dentry is active and the directory is not empty

1212-1213 If the directory is not empty, free the spinlock and return

1215 Move to the next child

1217-1218 The directory is empty. Free the spinlock and return

L.4.5.3 Function: \texttt{shmem\_positive()} (\texttt{mm/shmem.c})

1188 static inline int shmem_positive(struct dentry *dentry)
1189 {
1190 return dentry->d_inode && !d_unhashed(dentry);
1191 }

1190 Return true if the dentry has a valid inode and is currently hashed
L.5 Page Faulting within a Virtual File

Contents

L.5.1 Reading Pages during Page Fault 655
  L.5.1.1 Function: shmem_nopage() 655
  L.5.1.2 Function: shmem_getpage() 656
L.5.2 Locating Swapped Pages 663
  L.5.2.1 Function: shmem_alloc_entry() 663
  L.5.2.2 Function: shmem_swp_entry() 664

L.5.1 Reading Pages during Page Fault

L.5.1.1 Function: shmem_nopage() (mm/shmem.c)

This is the toplevel nopage() function that is called by do_no_page() when faulting in a page. This is called regardless of the fault being the first fault or if it is being faulted in from backing storage.

763 struct page * shmem_nopage(struct vm_area_struct *vma, unsigned long address, int unused)
764 {
765    struct inode *inode = vma->vm_file->f_dentry->d_inode;
766    struct page *page = NULL;
767    unsigned long idx;
768    int error;
769
770    idx = (address - vma->vm_start) >> PAGE_SHIFT;
771    idx += vma->vm_pgoff;
772    idx >>= PAGE_CACHE_SHIFT - PAGE_SHIFT;
773
774    error = shmem_getpage(inode, idx, &page, SGP_CACHE);
775    if (error)
776       return (error == -ENOMEM)? NOPAGE_OOM: NOPAGE_SIGBUS;
777
778    mark_page_accessed(page);
779    flush_page_to_ram(page);
780    return page;
781 }

763 The two parameters of relevance are the VMA the fault occurred in and the faulting address
765 Record the inode the fault occurred in
770-772 Calculate the idx as the offset in counts of PAGE_SIZE within the virtual file.

772 This adjustment takes into account the possibility that an entry in the page cache is a different size to a page. At the moment, there is no difference.

774-775 `shmem_getpage()` (See Section L.5.1.2) is responsible for locating the page at idx.

775-776 If an error occurred, decide whether to return an OOM error or an invalid faulting address error.

778 Mark the page accessed so it will be moved to the top of the LRU lists.

779 `flush_page_to_ram()` is responsible for avoiding d-cache aliasing problems.

780 Return the faulted-in page.

**L.5.1.2 Function: `shmem_getpage()` (mm/shmem.c)**

583 static int shmem_getpage(struct inode *inode, unsigned long idx, struct page **pagep, enum sgp_type sgp)

584 {
585     struct address_space *mapping = inode->i_mapping;
586     struct shmem_inode_info *info = SHMEM_I(inode);
587     struct shmem_sb_info *sbinfo;
588     struct page *filepage = *pagep;
589     struct page *swappage;
590     swp_entry_t *entry;
591     swp_entry_t swap;
592     int error = 0;
593
594     if (idx >= SHMEM_MAX_INDEX)
595         return -EFBIG;
596 /*
597     * Normally, filepage is NULL on entry, and either found
598     * uptodate immediately, or allocated and zeroed, or read
599     * in under swappage, which is then assigned to filepage.
600     * But `shmem_readpage` and `shmem_prepare_write` pass in a locked
601     * filepage, which may be found not uptodate by other callers
602     * too, and may need to be copied from the swappage read in.
603     */
604     repeat:
605     if (!filepage)
606         filepage = find_lock_page(mapping, idx);
if (filepage && Page_Uptodate(filepage))
    goto done;

spin_lock(&info->lock);
entry = shmem_swp_alloc(info, idx, sgp);
if (IS_ERR(entry)) {
    spin_unlock(&info->lock);
    error = PTR_ERR(entry);
    goto failed;
}
swap = *entry;

The parameters are:

inode  is the inode that the fault is occurring in
idx    is the index of the page within the file that is being faulted
pagep  if NULL will become the faulted page if successful. If a valid page is
        passed in, this function will make sure it is up to date
sgp    indicates what type of access this is which determines how a page will
        be located and returned

SHMEM_I() returns the shmem_inode_info contained with the filesystem-
specific information within the superblock information

Make sure the index is not beyond the end of the file
If no page was passed in with the pagep parameter, then try and locate
the page and lock it with find_lock_page() (See Section J.1.4.4)
If the page was found and is up to date, then goto done as this function
has nothing more to do
Lock the inode private information struct
Search for the swap entry for this idx with shmem_swp_alloc(). If one did
not previously exist, it will be allocated
If an error occurred, release the spinlock and return the error

if (swap.val) {
    /* Look it up and read it in.. */
    swappage = lookup_swap_cache(swap);
    if (!swappage) {
        spin_unlock(&info->lock);
        swapin_readahead(swap);
        swappage = read_swap_cache_async(swap);
if (!swappage) {
    spin_lock(&info->lock);
    entry = shmem_swp_alloc(info, idx, sgp);
    if (IS_ERR(entry))
        error = PTR_ERR(entry);
    else if (entry->val == swap.val)
        error = -ENOMEM;
    spin_unlock(&info->lock);
    if (error)
        goto failed;
    goto repeat;
}
wait_on_page(swappage);
page_cache_release(swappage);
goto repeat;
}

/* We have to do this with page locked to prevent races */
if (TryLockPage(swappage)) {
    spin_unlock(&info->lock);
    wait_on_page(swappage);
    page_cache_release(swappage);
    goto repeat;
}
if (!Page_Uptodate(swappage)) {
    spin_unlock(&info->lock);
    UnlockPage(swappage);
    page_cache_release(swappage);
    error = -EIO;
    goto failed;
}

In this block, a valid swap entry exists for the page. The page will be first searched for in the swap cache and if it does not exist there, it will be read in from backing storage.

619-690 This block of lines deal with the case where a valid swap entry exists
612 Search for swappage in the swap cache with lookup_swap_cache() (See Section K.2.4.1)
622-641 If the page does not exist in the swap cache, read it from backing storage with read_swap_cache_async(). Note that in line 638, wait_on_page() is called to wait until the IO completes. Once the IO completes, the reference to the page is released and the repeat label is jumped to reacquire the spinlocks and try again
Try and lock the page. If it fails, wait until it can be locked and jump to repeat to try again.

If the page is not up-to-date, the IO failed for some reason so return the error.

Delete the page from the swap cache so we can attempt to add it to the page cache.

If the caller supplied a page with the pagep parameter, then update pagep with the data in swappage.
671-680 Else try and add swappage to the page cache. Note that info→swapped is updated and the page is marked uptodate before the swap entry is freed with swap_free()

681-689 If we failed to add the page to the page cache, add it back to the swap cache with add_to_swap_cache(). The page is marked uptodate before being unlocked and goto repeat to try again

690 } else if (sgp == SGP_READ && !filepage) {
691     filepage = find_get_page(mapping, idx);
692     if (filepage &&
693           (!Page_Uptodate(filepage) || TryLockPage(filepage))) {
694         spin_unlock(&info->lock);
695         wait_on_page(filepage);
696         page_cache_release(filepage);
697         filepage = NULL;
698         goto repeat;
699     }
700     spin_unlock(&info->lock);

In this block, a valid swap entry does not exist for the idx. If the page is being read and the pagep is NULL, then locate the page in the page cache.

691 Call find_get_page() (See Section J.1.4.1) to find the page in the page cache

692-699 If the page was found but was not up to date or could not be locked, release the spinlock and wait until the page is unlocked. Then goto repeat to reacquire the spinlock and try again

700 Release the spinlock

701 } else {
702     sbinfo = SHMEM_SB(inode->i_sb);
703     spin_lock(&sbinfo->stat_lock);
704     if (sbinfo->free_blocks == 0) {
705         spin_unlock(&sbinfo->stat_lock);
706         spin_unlock(&info->lock);
707         error = -ENOSPC;
708         goto failed;
709     }
710     sbinfo->free_blocks--;
711     inode->i_blocks += BLOCKS_PER_PAGE;
712     spin_unlock(&sbinfo->stat_lock);
713     if (!filepage) {
714         spin_unlock(&info->lock);
715     }
L.5.1 Reading Pages during Page Fault (shmem_getpage())

716       filepage = page_cache_alloc(mapping);
717       if (!filepage) {
718           shmem_free_block(inode);
719           error = -ENOMEM;
720           goto failed;
721       }
722
723       spin_lock(&info->lock);
724       entry = shmem_swp_alloc(info, idx, sgp);
725       if (IS_ERR(entry))
726           error = PTR_ERR(entry);
727       if (error || entry->val ||
728           add_to_page_cache_unique(filepage,
729           mapping, idx, page_hash(mapping, idx)) != 0) {
730           spin_unlock(&info->lock);
731           page_cache_release(filepage);
732           shmem_free_block(inode);
733           filepage = NULL;
734           if (error)
735               goto failed;
736           goto repeat;
737       }
738   }
739
740       spin_unlock(&info->lock);
741       clear_highpage(filepage);
742       flush_dcache_page(filepage);
743       SetPageUptodate(filepage);
744   }

Else a page that is not in the page cache is being written to. It will need to be allocated.

702 Get the superblock info with SHMEM_SB()

703 Acquire the superblock info spinlock

704-709 If there are no free blocks left in the filesystem, release the spinlocks, set
the return error to -ENOSPC and goto failed;

710 Decrement the number of available blocks

711 Increment the block usage count for the inode

712 Release the superblock private information spinlock

714-715 If a page was not supplied via pagep, then allocate a page and swap entry
for the new page
715 Release the info spinlock as page_cache_alloc() may sleep

716 Allocate a new page

717-721 If the allocation failed, free the block with shmem_free_block() and set the return error to -ENOMEM before gotoing failed

723 Reacquire the info spinlock

724 shmem_swap_entry() locates a swap entry for the page. If one does not already exist (which is likely will not for this page), one will be allocated and returned

725-726 If no swap entry was found or allocated, set the return error

728-729 If no error occurred, add the page to the page cache

730-732 If the page was not added to the page cache (because we raced and another process inserted the page while we had the spinlock released for example), then drop the reference to the new page and free the block

734-735 If an error occurred, goto failed to report the error

736 Otherwise, goto repeat where the desired page will be searched for within the page cache again

740 Release the info spinlock

741 Zero-fill the new page

742 Flush the dcache to avoid possible CPU dcache aliasing

743 Mark the page as being up to date

745 done:
746 if (!*pagep) {
747 if (filepage) {
748 UnlockPage(filepage);
749 *pagep = filepage;
750 } else
751 *pagep = ZERO_PAGE(0);
752 }
753 return 0;
754
755 failed:
756 if (*pagep != filepage) {
757 UnlockPage(filepage);
758 page_cache_release(filepage);
759 }
760 return error;
761 }
746-752 If a page was not passed in via `pagep`, decide what to return. If a page
was allocated for writing, unlock and return `filepage`. Otherwise, the caller
is just a reader, so return the global zero-filled page

753 Return success

755 This is the failure path

756 If a page was allocated by this function and stored in `filepage`, unlock it and
drop the reference to it which will free it

760 Return the error code

L.5.2 Locating Swapped Pages

L.5.2.1 Function: `shmem_alloc_entry()` (mm/shmem.c)

This function is a top-level function that returns the swap entry corresponding
to a particular page index within a file. If the swap entry does not exist, one will
be allocated.

```c
183 static inline swp_entry_t * shmem_alloc_entry (  
    struct shmem_inode_info *info,  
    unsigned long index)
184 {
185     unsigned long page = 0;  
186     swp_entry_t * res;  
187
188     if (index >= SHMEM_MAX_INDEX)  
189         return ERR_PTR(-EFAULT);  
190
191     if (info->next_index <= index)  
192         info->next_index = index + 1;  
193
194     while ((res = shmem_swp_entry(info,index,page)) == 
195             ERR_PTR(-ENOMEM)) {
196         page = get_zeroed_page(GFP_USER);  
197         if (!page)  
198             break;  
199     }
200
188-189 SHMEM_MAX_INDEX is calculated at compile-time and it indicates the
largest possible virtual file in pages. If the var is greater than the maximum
possible sized file, return -EFAULT
next_index records the index of the page at the end of the file._inode→i_size alone is insufficient as the next_index field is needed for file truncation.

Call shmem_swapped_entry() to locate the swp_entry_t for the requested index. While searching, shmem_swapped_entry() may need a number of pages. If it does, it returns -ENOMEM which indicates that get_zeroed_page() should be called before trying again.

Return the swp_entry_t

L.5.2.2 Function: shmem_swapped_entry()  (mm/shmem.c)

This function uses information within the inode to locate the swp_entry_t for a given index. The inode itself is able to store SHMEM_NR_DIRECT swap vectors. After that indirect blocks are used.

```c
127 static swp_entry_t *shmem_swapped_entry (struct shmem_inode_info *info,
                                          unsigned long index,
                                          unsigned long page)
128 {  
129   unsigned long offset;
130   void **dir;
131
132   if (index < SHMEM_NR_DIRECT)
133     return info->i_direct+index;
134   if (!info->i_indirect) {
135     if (page) {
136       info->i_indirect = (void **) *page;
137       *page = 0;
138     }
139     return NULL;
140   }
141
142   index -= SHMEM_NR_DIRECT;
143   offset = index % ENTRIES_PER_PAGE;
144   index /= ENTRIES_PER_PAGE;
145   dir = info->i_indirect;
146
147   if (index >= ENTRIES_PER_PAGE/2) {
148     index -= ENTRIES_PER_PAGE/2;
149     dir += ENTRIES_PER_PAGE/2 + index/ENTRIES_PER_PAGE;
150     index %= ENTRIES_PER_PAGE;
151     if (!dir) {
152       if (page) {
153         *dir = (void *) *page;
154       }
155     }
156   }
```
L.5.2 Locating Swapped Pages *(shmem_swp_entry()*)

*page = 0;
}
return NULL;
}
dir = ((void **)dir);

dir += index;
if (!dir) {
    if (!page || !*page)
        return NULL;
    *dir = (void *) *page;
    *page = 0;
}
return (swp_entry_t *) *dir + offset;

132-133 If the index is below SHMEM_NR_DIRECT, then the swap vector is contained within the direct block so return it

134-140 If a page does not exist at this indirect block, install the page that was passed in with the page parameter and return NULL. This tells the called to allocate a new page and call the function again

142 Treat the indirect blocks as starting from index 0

143 ENTRIES_PER_PAGE is the number of swap vectors contained within each page in the indirect block. offset is now the index of the desired swap vector within the indirect block page when it is found

144 index is now the directory number within the indirect block list that must be found

145 Get a pointer to the first indirect block we are interested in

147-159 If the required directory (index) is greater than ENTRIES_PER_PAGE/2, then it is a triple indirect block so the next block must be traversed

148 Pointers to the next set of directory blocks is in the second half of the current block so calculate index as an offset within the second half of the current block

149 Calculate dir as a pointer to the next directory block

150 index is now a pointer within dir to a page containing the swap vectors we are interested in

151-156 If dir has not been allocated, install the page supplied with the page parameter and return NULL so the caller will allocate a new page and call the function again
158 dir is now the base of the page of swap vectors containing the one we are interested in

161 Move dir forward to the entry we want

162-167 If an entry does not exist, install the page supplied as a parameter if available. If not, return NULL so that one will be allocated and the function called again

168 Return the found swap vector
L.6 Swap Space Interaction

Contents
L.6 Swap Space Interaction 667
L.6.1 Function: shmem_writepage() 667
L.6.2 Function: shmem_unuse() 669
L.6.3 Function: shmem_unuse_inode() 670
L.6.4 Function: shmem_find_swp() 673

L.6.1 Function: shmem_writepage() (mm/shmem.c)
This function is responsible for moving a page from the page cache to the swap cache.

522 static int shmem_writepage(struct page *page)
523 {
524  struct shmem_inode_info *info;
525  swp_entry_t *entry, swap;
526  struct address_space *mapping;
527  unsigned long index;
528  struct inode *inode;
529  
530  BUG_ON(!PageLocked(page));
531  if (!PageLaunder(page))
532    return fail_writepage(page);
533  
534  mapping = page->mapping;
535  index = page->index;
536  inode = mapping->host;
537  info = SHMEM_I(inode);
538  if (info->flags & VM_LOCKED)
539    return fail_writepage(page);

This block is function preamble to make sure the operation is possible.

522 The parameter is the page to move to the swap cache
530 It is a bug if the page is already locked for IO
531-532 If the launder bit has not been set, call fail_writepage(). fail_writepage() is used by in-memory filesystems to mark the page dirty and re-activate it so that the page reclaimer does not repeatedly attempt to write the same page
534-537 Records variables that are needed as parameters later in the function
538-539 If the inode filesystem information is locked, fail
L.6 Swap Space Interaction (*shmem_writepage()*)

540  getswap:
541      swap = get_swap_page();
542      if (!swap.val)
543          return fail_writepage(page);
544
545      spin_lock(&info->lock);
546      BUG_ON(index >= info->next_index);
547      entry = shmem_swp_entry(info, index, NULL);
548      BUG_ON(!entry);
549      BUG_ON(entry->val);
550

This block is responsible for allocating a swap slot from the backing storage and a *swp_entry* within the inode.

541-543 Locate a free swap slot with *get_swap_page()* (See Section K.1.1). It fails, call *fail_writepage()*

545 Lock the inode information

547 Get a free *swp_entry* from the filesystem-specific private inode information with *shmem_swp_entry()*

551      /* Remove it from the page cache */
552      remove_inode_page(page);
553      page_cache_release(page);
554
555      /* Add it to the swap cache */
556      if (add_to_swap_cache(page, swap) != 0) {
557          /*
558           * Raced with "speculative" read_swap_cache_async.
559           * Add page back to page cache, unref swap, try again.
560           */
561          add_to_page_cache_locked(page, mapping, index);
562          spin_unlock(&info->lock);
563          swap_free(swap);
564          goto getswap;
565      }
566
567      *entry = swap;
568      info->swapped++;
569      spin_unlock(&info->lock);
570      SetPageUptodate(page);
571      set_page_dirty(page);
572      UnlockPage(page);
573      return 0;
574  }
Move from the page cache to the swap cache and update statistics.

remove_inode_page() (See Section J.1.2.1) removes the page from the inode and hash lists the page is a member of

page_cache_release() drops the local reference to the page taken for the writepage() operation

Add the page to the swap cache. After this returns, the page→mapping will now be swapper_space

The operation failed so add the page back to the page cache

Unlock the private information

free the swap slot and try again

Here, the page has successfully become part of the swap cache. Update the inode information to point to the swap slot in backing storage

Increment the counter recording the number of pages belonging to this inode that are in swap

Free the private inode information

Move the page to the address_space dirty pages list so that it will be written to backing storage

Return success

L.6.2 Function: shmem_unuse() (mm/shmem.c)

This function will search the shmem_inodes list for the inode that holds the information for the requested entry and page. It is a very expensive operation but it is only called when a swap area is being deactivated so it is not a significant problem. On return, the swap entry will be freed and the page will be moved from the swap cache to the page cache.

```c
498 int shmem_unuse(swp_entry_t entry, struct page *page)
499 { 
500   struct list_head *p;
501   struct shmem_inode_info *info;
502   spin_lock(&shmem_ilock);
503   list_for_each(p, &shmem_inodes) {
504     info = list_entry(p, struct shmem_inode_info, list);
505     if (info->swapped && shmem_unuse_inode(info, entry, page)) {
506       /* move head to start search for next from here */
507     }
508   }
```
509    list_move_tail(&shmem_inodes, &info->list);
510    found = 1;
511    break;
512 }
513 }
514 spin_unlock(&shmem_ilock);
515 return found;
516 }

503 Acquire the shmem_ilock spinlock protecting the inode list

504 Cycle through each entry in the shmem_inodes list searching for the inode holding the requested entry and page

509 Move the inode to the top of the list. In the event that we are reclaiming many pages, the next search will find the inode of interest at the top of the list

510 Indicate that the page was found

511 This page and entry have been found to break out of the loop

514 Release the shmem_ilock spinlock

515 Return if the page was found or not by shmem_unuse_inode()

L.6.3 Function: shmem_unuse_inode() (mm/shmem.c)

This function searches the inode information in info to determine if the entry and page belong to it. If they do, the entry will be cleared and the page will be removed from the swap cache and moved to the page cache instead.

436 static int shmem_unuse_inode(struct shmem_inode_info *info, swp_entry_t entry, struct page *page)
437 {
438    struct inode *inode;
439    struct address_space *mapping;
440    swp_entry_t *ptr;
441    unsigned long idx;
442    int offset;
443
444    idx = 0;
445    ptr = info->i_direct;
446    spin_lock(&info->lock);
447    offset = info->next_index;
448    if (offset > SHMEM_NR_DIRECT)
449        offset = SHMEM_NR_DIRECT;
offset = shmem_find_swp(entry, ptr, ptr + offset);
if (offset >= 0)
go to found;

for (idx = SHMEM_NR_DIRECT; idx < info->next_index;
 idx += ENTRIES_PER_PAGE) {
 ptr = shmem_swp_entry(info, idx, NULL);
 if (!ptr)
 continue;
 offset = info->next_index - idx;
 if (offset > ENTRIES_PER_PAGE)
 offset = ENTRIES_PER_PAGE;
 offset = shmem_find_swp(entry, ptr, ptr + offset);
 if (offset >= 0)
go to found;
}
spin_unlock(&info->lock);
return 0;

found:
idx += offset;
inode = info->inode;
 mapping = inode->i_mapping;
delete_from_swap_cache(page);

/* Racing against delete or truncate? *]
 * Must leave out of page cache */
 limit = (inode->i_state & I_FREEING)? 0:
 (inode->i_size + PAGE_CACHE_SIZE - 1) >> PAGE_CACHE_SHIFT;

if (idx >= limit || add_to_page_cache_unique(page,
 mapping, idx, page_hash(mapping, idx)) == 0) {
 ptr[offset].val = 0;
 info->swapped--;
} else if (add_to_swap_cache(page, entry) != 0)
 BUG();
spin_unlock(&info->lock);
SetPageUptodate(page);
/*
 * Decrement swap count even when the entry is left behind:
 * try_to_unuse will skip over mms, then reincrement count.
 */
 swap_free(entry);
return 1;
Initialise *ptr* to start at the beginning of the direct block for the inode being searched.

Lock the inode private information.

Initialise *offset* to be the last page index in the file.

If *offset* is beyond the end of the direct block, set it to the end of the direct block for the moment.

Use `shmem_find_swap()` (See Section L.6.4) to search the direct block for the entry.

If the entry was in the direct block, goto *found*, otherwise we have to search the indirect blocks.

Search each of the indirect blocks for the entry.

`shmem_swap_entry()` (See Section L.5.2.2) returns the swap vector at the current *idx* within the inode. As *idx* is incremented in `ENTRIES_PER_PAGE` sized strides, this will return the beginning of the next indirect block being searched.

If an error occurred, the indirect block does not exist, so continue, which probably will exit the loop.

Calculate how many pages are left in the end of the file to see if we only have to search a partially filled indirect block.

If *offset* is greater than the size of an indirect block, set *offset* to `ENTRIES_PER_PAGE` so this full indirect block will be searched by `shmem_find_swap()`.

Search the entire of the current indirect block for entry with `shmem_find_swap()` (See Section).

If the entry was found, goto *found*, otherwise the next indirect block will be searched. If the entry is never found, the info struct will be unlocked and 0 returned indicating that this inode did not contain the entry and page.

The entry was found, so free it with `swap_free()`.

Move *idx* to the location of the swap vector within the block.

Get the inode and mapping.

Delete the page from the swap cache.

Check if the inode is currently being deleted or truncated by examining `inode->i_state`. If it is, set *limit* to the index of the last page in the adjusted file size.
If the page is not being truncated or deleted, add it to the page cache with `add_to_page_cache_unique()`. If successful, clear the swap entry and decrement `info->swapped`.

Else add the page back to the swap cache where it will be reclaimed later.

Release the `info` spinlock.

Mark the page uptodate.

Decrement the swap count.

Return success.

**Function: `shmem_find_swp()` (mm/shmem.c)**

This function searches an indirect block between the two pointers `ptr` and `eptr` for the requested entry. Note that the two pointers must be in the same indirect block.

```c
static inline int shmem_find_swp(swp_entry_t entry,  
    swp_entry_t *dir,  
    swp_entry_t *edir)

for (ptr = dir; ptr < edir; ptr++) {
    if (ptr->val == entry.val)
        return ptr - dir;
}
return -1;
```

Loop between the `dir` and `edir` pointers.

If the current `ptr` entry matches the requested `entry` then return the offset from `dir`. As `shmem_unuse_inode()` is the only user of this function, this will result in the offset within the indirect block being returned.

Return indicating that the entry was not found.
L.7 Setting up Shared Regions

Contents

L.7 Setting up Shared Regions 674
  L.7.1 Function: shmem_zero_setup() 674
  L.7.2 Function: shmem_file_setup() 675

L.7.1 Function: shmem_zero_setup() (mm/shmem.c)

This function is called to setup a VMA that is a shared region backed by anonymous pages. The call graph which shows this function is in Figure 12.5. This occurs when mmap() creates an anonymous region with the MAP_SHARED flag.

1664 int shmem_zero_setup(struct vm_area_struct *vma)
1665 {
1666   struct file *file;
1667   loff_t size = vma->vm_end - vma->vm_start;
1668
1669   file = shmem_file_setup("dev/zero", size);
1670   if (IS_ERR(file))
1671     return PTR_ERR(file);
1672
1673   if (vma->vm_file)
1674     fput(vma->vm_file);
1675   vma->vm_file = file;
1676   vma->vm_ops = &shmem_vm_ops;
1677   return 0;
1678 }

1667 Calculate the size

1669 Call shmem_file_setup() (See Section L.7.2) to create a file called dev/zero and of the calculated size. We will see in the functions code commentary why the name does not have to be unique

1673-1674 If a file already exists for this virtual area, call fput() to drop it's reference

1675 Record the new file pointer

1675 Set the vm_ops so that shmem_nopage() (See Section L.5.1.1) will be called when a page needs to be faulted in for this VMA
L.7.2 Function: shmem_file_setup()  

This function is called to create a new file in shmfs, the internal filesystem. As the filesystem is internal, the supplied name does not have to be unique within each directory. Hence, every file that is created by an anonymous region with shmem_zero_setup() will simply be called “dev/zero” and regions created with shmget() will be called “SYSVNN” where NN is the key that is passed as the first argument to shmget().

```
1607 struct file *shmem_file_setup(char *name, loff_t size) {
1608     int error;
1609     struct file *file;
1610     struct inode *inode;
1611     struct dentry *dentry, *root;
1612     struct qstr this;
1613     int vm_enough_memory(long pages);
1614
1616     if (IS_ERR(shm_mnt))
1617         return (void *)shm_mnt;
1618     if (size > SHMEM_MAX_BYTES)
1619         return ERR_PTR(-EINVAL);
1620     if (!vm_enough_memory(VM_ACCT(size)))
1621         return ERR_PTR(-ENOMEM);
1622     this.name = name;
1623     this.len = strlen(name);
1624     this.hash = 0; /* will go */

1607 The parameters are the name of the file to create and its expected size
1614 vm_enough_memory() (See Section M.1.1) checks to make sure there is enough memory to satisfy the mapping
1616-1617 If there is an error with the mount point, return the error
1619-1620 Do not create a file greater than SHMEM_MAX_BYTES which is calculated at top of mm/shmem.c
1622-1623 Make sure there is enough memory to satisfy the mapping
1625-1627 Populate the struct qstr which is the string type used for dnodes
```

```
1628     root = shm_mnt->mnt_root;
1629     dentry = d_alloc(root, &this);
1630     if (!dentry)
```
return ERR_PTR(-ENOMEM);

error = -ENFILE;
file = get_empty_filp();
if (!file)
goto put_dentry;

error = -ENOSPC;
inode = shmem_get_inode(root->d_sb, S_IFREG | S_IRWXUGO, 0);
if (!inode)
goto close_file;

d_instantiate(dentry, inode);
inode->i_size = size;
inode->i_nlink = 0;  /* It is unlinked */
file->f_vfsmnt = mntget(shm_mnt);
file->f_dentry = dentry;
file->f_op = &shmem_file_operations;
file->f_mode = FMODE_WRITE | FMODE_READ;
return file;

close_file:
put_filp(file);
put_dentry:
dput(dentry);
return ERR_PTR(error);
}

root is assigned to be the dnode representing the root of shmfs

Allocate a new dentry with d_alloc()

Return -ENOMEM if one could not be allocated

Get an empty struct file from the file table. If one couldn’t be found,
-ENFILE will be returned indicating a file table overflow

Create a new inode which is a regular file (S_IFREG) and globally
readable, writable and executable. If it fails, return -ENOSPC indicating no
space is left in the filesysytem

d_instantiate() fills in the inode information for a dentry. It is defined in
fs/dcache.c

Fill in the remaining inode and file information

Return the newly created struct file
Error path when an inode could not be created. `put_filp()` will free up the `struct file` entry in the file table.

`dput()` will drop the reference to the dentry, destroying it.

Return the error code.
L.8 System V IPC

L.8.1 Creating a SYSV shared region

L.8.1.1 Function: sys_shmget()  (ipc/shm.c)

```c
asmlinkage long sys_shmget (key_t key, size_t size, int shmflg) {
    struct shmid_kernel *shp;
    int err, id = 0;

down(&shm_ids.sem);

    if (key == IPC_PRIVATE) {
        err = newseg(key, shmflg, size);
    } else if ((id = ipc_findkey(&shm_ids, key)) == -1) {
        if (!(shmflg & IPC_CREAT))
            err = -ENOENT;
        else
            err = newseg(key, shmflg, size);
    } else if ((shmflg & IPC_CREAT) && (shmflg & IPC_EXCL)) {
        err = -EEXIST;
    } else {
        shp = shm_lock(id);
        if (shp == NULL)
            BUG();
        if (shp->shm_segsz < size)
            err = -EINVAL;
    } else if (ipcperms(&shp->shm_perm, shmflg))
        err = -EACCES;
    else
        err = shm_buildid(id, shp->shm_perm.seq);

    shm_unlock(id);

    up(&shm_ids.sem);

    return err;
}
```

234 Acquire the semaphore protecting shared memory IDs

235-236 If IPC_PRIVATE is specified, most of the flags are ignored and the region is created with newseg(). This flag is intended to provide exclusive access to a shared region but Linux does not guarantee exclusive access

237 Else search to see if the key already exists with ipc_findkey()

238-239 If it does not and IPC_CREAT was not specified, then return -ENOENT
Else, create a new region with \texttt{newseg()}

If the region already exists and the process requested a new region that did not previously exist to be created, return \texttt{-EEXIST}

Else we are accessing an existing region, so lock it, make sure we have the required permissions, build a segment identifier with \texttt{shm_buildid()} and unlock the region again. The segment identifier will be returned back to userspace

Release the semaphore protecting IDs

Return either the error or the segment identifier

\textbf{L.8.1.2 Function: newseg()  \texttt{(ipc/shm.c)}}

This function creates a new shared segment.

\begin{verbatim}
static int newseg (key_t key, int shmflg, size_t size) {
    int error;
    struct shmid_kernel *shp;
    int numpages = (size + PAGE_SIZE -1) >> PAGE_SHIFT;
    struct file * file;
    char name[13];
    int id;

    if (size < SHMMIN || size > shm_ctlmax)
        return -EINVAL;

    if (shm_tot + numpages >= shm_ctlall)
        return -ENOSPC;

    shp = (struct shmid_kernel *) kmalloc (sizeof (*shp), GFP_USER);
    if (!shp)
        return -ENOMEM;
    sprintf (name, "SYSV%08x", key);

    this block allocates the segment descriptor.

    if (size < SHMMIN || size > shm_ctlmax)
        return -EINVAL;

    if (shm_tot + numpages >= shm_ctlall)
        return -ENOSPC;

    shp = (struct shmid_kernel *) kmalloc (sizeof (*shp), GFP_USER);
    if (!shp)
        return -ENOMEM;
    sprintf (name, "SYSV%08x", key);

    Calculate the number of pages the region will occupy

    Ensure the size of the region does not break limits

    Make sure the total number of pages required for the segment will not break limits

    Allocate the descriptor with \texttt{kmalloc()}(See Section H.4.2.1)
\end{verbatim}
L.8.1 Creating a SYSV shared region (newseg())

196 Print the name of the file to be created in shmfs. The name is SYSVNN where NN is the key identifier of the region

197 \texttt{file = shmem\_file\_setup(name, size);} \\
198 \texttt{error = PTR\_ERR(file);} \\
199 \texttt{if (IS\_ERR(file))} \\
200 \texttt{\quad goto no\_file;} \\
201 \texttt{\quad} \\
202 \texttt{\quad error = -ENOSPC;} \\
203 \texttt{\quad id = shm\_addid(shp);} \\
204 \texttt{\quad if(id == -1)} \\
205 \texttt{\quad \quad goto no\_id;} \\
206 \texttt{\quad shp->shm\_perm.key = key;} \\
207 \texttt{\quad shp->shm\_flags = (shmflg & S\_IRWXUGO);} \\
208 \texttt{\quad shp->shm\_cprid = current->pid;} \\
209 \texttt{\quad shp->shm\_lprid = 0;} \\
210 \texttt{\quad shp->shm\_atim = shp->shm\_dtim = 0;} \\
211 \texttt{\quad shp->shm\_ctim = CURRENT\_TIME;} \\
212 \texttt{\quad shp->shm\_segsz = size;} \\
213 \texttt{\quad shp->shm\_nattch = 0;} \\
214 \texttt{\quad shp->id = shm\_buildid(id,shp->shm\_perm.seq);} \\
215 \texttt{\quad shp->shm\_file = file;} \\
216 \texttt{\quad file->f\_dentry->d\_inode->i\_ino = shp->id;} \\
217 \texttt{\quad file->f\_op = &shm\_file\_operations;} \\
218 \texttt{\quad shm\_tot += numpages;} \\
219 \texttt{\quad shm\_unlock (id);} \\
220 \texttt{\quad return shp->id;} \\
221 \texttt{\quad} \\
222 no\_id: \\
223 \texttt{\quad fput(file);} \\
224 no\_file: \\
225 \texttt{kfree(shp);} \\
226 \texttt{return error;} \\
227 \}

197 Create a new file in shmfs with shmem\_file\_setup() (See Section L.7.2)

198-200 Make sure no error occurred with the file creation

202 By default, the error to return indicates that there is no shared memory identifiers available or that the size of the request is too large

206-213 Fill in fields in the segment descriptor

214 Build a segment identifier which is what is returned to the caller of shmget()

215-217 Set the file pointers and file operations structure
218 Update shm_tot to the total number of pages used by shared segments

220 Return the identifier

L.8.2 Attaching a SYSV Shared Region

L.8.2.1 Function: sys_shmat() (ipc/shm.c)

```c
asmlinkage long sys_shmat (int shmid, char *shmaddr, int shmflg, ulong *raddr)
```

568 {
    struct shmid_kernel *shp;
    unsigned long addr;
    unsigned long size;
    struct file *file;
    int err;
    unsigned long flags;
    unsigned long prot;
    unsigned long o_flags;
    int acc_mode;
    void *user_addr;

    if (shmid < 0)
        return -EINVAL;

    if ((addr = (ulong)shmaddr)) {
        if (addr & (SHMLBA-1)) {
            if (shmflg & SHM_RND)
                addr &= ~(SHMLBA-1); /* round down */
            else
                return -EINVAL;
        }
        flags = MAP_SHARED | MAP_FIXED;
    } else {
        flags = MAP_SHARED | MAP_FIXED;
    }

    if ((shmflg & SHM_RDONLY)) {
        prot = PROT_READ;
        o_flags = O_RDONLY;
        acc_mode = S_IRUGO;
    } else {
```
604 prot = PROT_READ | PROT_WRITE;
605 o_flags = O_RDWR;
606 acc_mode = S_IRUGO | S_IWUGO;
607 }

This section ensures the parameters to shmat() are valid.

581-582 Negative identifiers are not allowed so return -EINVAL is one is supplied

584-591 If the caller supplied an address, make sure it is ok

585 SHMLBA is the segment boundary address multiple. In Linux, this is always PAGE_SIZE. If the address is not page aligned, then check if the caller specified SHM_RND which allows the address to be changed. If specified, round the address down to the nearest page boundary, otherwise return -EINVAL

591 Set the flags to use with the VMA to create a shared region (MAP_SHARED) with a fixed address (MAP_FIXED)

593-596 If an address was not supplied, make sure the SHM_REMAP was specified and only use the MAP_SHARED flag with the VMA. This means that do_mmap() (See Section D.2.1.1) will find a suitable address to attach the shared region

613 shp = shm_lock(shmid);
614 if(shp == NULL)
615 return -EINVAL;
616 err = shm_checkid(shp, shmid);
617 if (err) {
618 shm_unlock(shmid);
619 return err;
620 }
621 if (ipcperms(&shp->shm_perm, acc_mode)) {
622 shm_unlock(shmid);
623 return -EACCES;
624 }
625 file = shp->shm_file;
626 size = file->f_dentry->d_inode->i_size;
627 shp->shm_nattch++;
628 shm_unlock(shmid);

This block ensures the IPC permissions are valid

613 shm_lock() locks the descriptor corresponding to shmid and returns a pointer to the descriptor

614-615 Make sure the descriptor exists

616-620 Make sure the ID matches the descriptor
L.8.2 Attaching a SYSV Shared Region (sys_shmat())

612-624 Make sure the caller has the correct permissions

625 Get a pointer to the struct file which do_mmap() requires

626 Get the size of the shared region so do_mmap() knows what size of VMA to create

627 Temporarily increment shm_nattach() which normally indicates how many VMAs are using the segment. This is to prevent the segment been freed prematurely. The real counter will be incremented by shm_open() which is the open() callback used by the vm_operations_struct used for shared regions

628 Release the descriptor

630  down_write(&current->mm->mmap_sem);
631  if (addr && !(shmflg & SHM_REMAP)) {
632     user_addr = ERR_PTR(-EINVAL);
633     if (find_vma_intersection(current->mm, addr, addr + size))
634         goto invalid;
635     /*
636      * If shm segment goes below stack, make sure there is some
637      * space left for the stack to grow (at least 4 pages).
638      */
639     if (addr < current->mm->start_stack &&
640         addr > current->mm->start_stack - size - PAGE_SIZE * 5)
641         goto invalid;
642  }
643
644  user_addr = (void*) do_mmap (file, addr, size, prot, flags, 0);

This block is where do_mmap() will be called to attach the region to the calling process.

630 Acquire the semaphore protecting the mm_struct

632-634 If an address was specified, call find_vma_intersection() (See Section D.3.1.3) to ensure no VMA overlaps the region we are trying to use

639-641 Make sure there is at least a 4 page gap between the end of the shared region and the stack

644 Call do_mmap() (See Section D.2.1.1) which will allocate the VMA and map it into the process address space

646 invalid:
647  up_write(&current->mm->mmap_sem);
648  down (&shm_ids.sem);
650 if(!(shp = shm_lock(shmid)))
651     BUG();
652 shp->shm_nattch--;
653 if(shp->shm_nattch == 0 &&
654     shp->shm_flags & SHM_DEST)
655     shm_destroy (shp);
656 else
657     shm_unlock(shmid);
658     up (&shm_ids.sem);
659
660 *raddr = (unsigned long) user_addr;
661 err = 0;
662 if (IS_ERR(user_addr))
663     err = PTR_ERR(user_addr);
664 return err;
665
666 }

647 Release the mm_struct semaphore
649 Release the region IDs semaphore
650-651 Lock the segment descriptor
652 Decrement the temporary shm_nattch counter. This will have been properly
incremented by the vm_ops->open callback
653-655 If the users reach 0 and the SHM_DEST flag has been specified, the region
is destroyed as it is no longer required
657 Otherwise, just unlock the segment
660 Set the address to return to the caller
661-663 If an error occurred, set the error to return to the caller
664 Return
Appendix M

Out of Memory Management

Contents

M.1 Determining Available Memory .................................. 686
   M.1.1 Function: vm_enough_memory() ................................. 686

M.2 Detecting and Recovering from OOM ............................ 688
   M.2.1 Function: out_of_memory() .................................... 688
   M.2.2 Function: oom_kill() ........................................... 689
   M.2.3 Function: select_bad_process() ............................... 690
   M.2.4 Function: badness() ............................................. 691
   M.2.5 Function: oom_kill_task() ..................................... 692
M.1 Determining Available Memory

Contents

M.1 Determining Available Memory 686
M.1.1 Function: vm_enough_memory() 686

M.1.1 Function: vm_enough_memory() \((mm/mmap.c)\)

53 int vm_enough_memory(long pages)
54 {
65 unsigned long free;
66
67 /* Sometimes we want to use more memory than we have. */
68 if (sysctl_overcommit_memory)
69     return 1;
70
71 /* The page cache contains buffer pages these days.. */
72 free = atomic_read(&page_cache_size);
73 free += nr_free_pages();
74 free += nr_swap_pages;
75
76 /*
77 * This double-counts: the nrpages are both in the page-cache
78 * and in the swapper space. At the same time, this compensates
79 * for the swap-space over-allocation (ie "nr_swap_pages" being
80 * too small.
81 */
82 free += swapper_space.nrpages;
83
84 /*
85 * The code below doesn’t account for free space in the inode
86 * and dentry slab cache, slab cache fragmentation, inodes and
87 * dentries which will become freeable under VM load, etc.
88 * Lets just hope all these (complex) factors balance out...
89 */
90 free += (dentry_stat.nr_unused * sizeof(struct dentry)) >> PAGE_SHIFT;
91 free += (inodes_stat.nr_unused * sizeof(struct inode)) >> PAGE_SHIFT;
92
93 return free > pages;
94 }

68-69 If the system administrator has specified via the proc interface that overcommit is allowed, return immediately saying that the memory is available

72 Start the free pages count with the size of the page cache as these pages may be easily reclaimed
M.1 Determining Available Memory (*vm_enough_memory()*)

73 Add the total number of free pages in the system

74 Add the total number of available swap slots

82 Add the number of pages managed by `swapper_space`. This double counts free slots in swaps but is balanced by the fact that some slots are reserved for pages but are not being currently used

90 Add the number of unused pages in the dentry cache

91 Add the number of unused pages in the inode cache

93 Return if there are more free pages available than the request
M.2 Detecting and Recovering from OOM

Contents

M.2 Detecting and Recovering from OOM 688
M.2.1 Function: out_of_memory() 688
M.2.2 Function: oom_kill() 689
M.2.3 Function: select_bad_process() 690
M.2.4 Function: badness() 691
M.2.5 Function: oom_kill_task() 692

M.2.1 Function: out_of_memory() (mm/oom_kill.c)

202 void out_of_memory(void)
203 {
204     static unsigned long first, last, count, lastkill;
205     unsigned long now, since;
206     if (nr_swap_pages > 0)
207         return;
208     now = jiffies;
209     since = now - last;
210     last = now;
211     if (since > 5*HZ)
212         goto reset;
213     since = now - first;
214     last = now;
215     if (since > 5*HZ)
216         goto reset;
217     since = now - first;
218     if (since < HZ)
219         return;
220     if (++count < 10)
221         return;
222     since = now - lastkill;
223     lastkill = now;
224     if (since < HZ*5)
225         return;
226     lastkill = now;
227     oom_kill();
228
229 reset:
230     first = now;
231     count = 0;
258 

210-211 If there are available swap slots, the system is no OOM

213-215 Record what time it is now in jiffies and determine how long it has been since this function was last called

222-223 If it has been more than 5 seconds since this function was last called, then reset the timer and exit the function

229-231 If it has been longer than a second since this function was last called, then exit the function. It is possible that IO is in progress which will complete soon

237-238 If the function has not been called 10 times within the last short interval, then the system is not yet OOM

245-247 If a process has been killed within the last 5 seconds, then exit the function as the dying process is likely to free memory

253 Ok, the system really is OOM so call oom_kill() (See Section M.2.2) to select a process to kill

M.2.2 Function: oom_kill() (mm/oom_kill.c)

This function first calls select_bad_process() to find a suitable process to kill. Once found, the task list is traversed and the oom_kill_task() is called for the selected process and all it’s threads.

172 static void oom_kill(void)
173 {
174     struct task_struct *p, *q;
175
176     read_lock(&tasklist_lock);
177     p = select_bad_process();
178
179     /* Found nothing?!! Either we hang forever, or we panic. */
180     if (p == NULL)
181         panic("Out of memory and no killable processes...
182
183     /* kill all processes that share the ->mm (i.e. all threads) */
184     for_each_task(q) {
185         if (q->mm == p->mm)
186             oom_kill_task(q);
187     }
188     read_unlock(&tasklist_lock);
189
190     /*
* Make kswapd go out of the way, so "p" has a good chance of
* killing itself before someone else gets the chance to ask
* for more memory.
*/
yield();
return;
}

176 Acquire the read-only semaphore to the task list

177 Call select_bad_process() (See Section M.2.3) to find a suitable process to kill

180-170 If one could not be found, panic the system because otherwise the system
will deadlock. In this case, it is better to deadlock and have a developer solve
the bug than have a mysterious hang

184-187 Cycle through the task list and call oom_kill_task() (See Section M.2.5)
for the selected process and all it's threads. Remember that threads will all
share the same mm_struct

188 Release the semaphore

195 Call yield() to allow the signals to be delivered and the processes to die. The
comments indicate that kswapd will be the sleeper but it is possible that a
process in the direct-reclaim path will be executing this function too

M.2.3 Function: select_bad_process()  (mm/oom_kill.c)

This function is responsible for cycling through the entire task list and returning
the process that scored highest with the badness() function.

121 static struct task_struct * select_bad_process(void)
122 {
123     int maxpoints = 0;
124     struct task_struct *p = NULL;
125     struct task_struct *chosen = NULL;
126
127     for_each_task(p) {
128         if (p->pid) {
129             int points = badness(p);
130             if (points > maxpoints) {
131                 chosen = p;
132                 maxpoints = points;
133             }
134         }
135     }
136     return chosen;
137 }
M.2 Detecting and Recovering from OOM (select_bad_process())

127 Cycle through all tasks in the task list
128 If the process is the system idle task, then skip over it
129 Call badness()(See Section M.2.4) to score the process
130-133 If this is the highest score so far, record it
136 Return the task_struct which scored highest with badness()

M.2.4 Function: badness()  (mm/oom_kill.c)

This calculates a score that determines how suitable the process is for killing. The scoring mechanism is explained in detail in Chapter 13.

58 static int badness(struct task_struct *p)
59 {
60  int points, cpu_time, run_time;
61  
62  if (!p->mm)
63     return 0;
64  
65  if (p->flags & PF_MEMDIE)
66     return 0;
67  
68  points = p->mm->total_vm;
69  
70  cpu_time = (p->times.tms_utime + p->times.tms_stime) >> (SHIFT_HZ + 3);
71  
72  run_time = (jiffies - p->start_time) >> (SHIFT_HZ + 10);
73  
74  points /= int_sqrt(cpu_time);
75  
76  points /= int_sqrt(int_sqrt(run_time));
77  
78  if (p->nice > 0)
79      points *= 2;
80  
81  if (cap_t(p->cap_effective) & CAP_TO_MASK(CAP_SYS_ADMIN) ||
82      p->uid == 0 || p->euid == 0)
83      points /= 4;
84  
85  if (cap_t(p->cap_effective) & CAP_TO_MASK(CAP_SYS_RAWIO))
86      points /= 4;
87  
88  #ifdef DEBUG
89      printk(KERN_DEBUG "OOMkill: task %d (%s) got %d points\n",
90      p->pid, p->comm, points);
91  #endif
return points;

62-63 If there is no mm, return 0 as this is a kernel thread

65-66 If the process has already been marked by the OOM killer as exiting, return 0 as there is no point trying to kill it multiple times

71 The total VM used by the process is the base starting point

79-80 cpu_time is calculated as the total runtime of the process in seconds. run_time is the total runtime of the process in minutes. Comments indicate that there is no basis for this other than it works well in practice

82 Divide the points by the integer square root of cpu_time

83 Divide the points by the cube root of run_time

89-90 If the process has been nice to be of lower priority, double it’s points as it is likely to be an unimportant process

96-98 On the other hand, if the process has superuser privileges or has the CAP_SYS_ADMIN capability, it is likely to be a system process so divide the points by 4

106-107 If the process has direct access to hardware then divide the process by 4. Killing these processes forcefully could potentially leave hardware in an inconsistent state. For example, forcibly killing X is never a good idea

112 Return the score

M.2.5 Function: oom_kill_task() (mm/oom_kill.c)

This function is responsible for sending the appropriate kill signals to the selected task.

void oom_kill_task(struct task_struct *p)
{
    printk(KERN_ERR "Out of Memory: Killed process %d (%s).\n",
            p->pid, p->comm);
    p->counter = 5 * HZ;
    p->flags |= PF_MEMALLOC | PF_MEMDIE;
156 /* This process has hardware access, be more careful. */
157 if (cap_t(p->cap_effective) & CAP_TO_MASK(CAP_SYS_RAWIO)) {
158    force_sig(SIGTERM, p);
159  } else {
160    force_sig(SIGKILL, p);
161  }
162 }

146 Print an informational message on the process being killed
153 This gives the dying process lots of time on the CPU so it can kill itself off quickly
154 These flags will tell the allocator to give favourably treatment to the process if it requires more pages before cleaning itself up
157-158 If the process can directly access hardware, send it the SIGTERM signal to give it a chance to exit cleanly
160 Otherwise send it the SIGKILL signal to force the process to be killed
Bibliography


[McK96] Marshall Kirk McKusick. The design and implementation of the 4.4BSD operating system. Addison-Wesley, 1996.


Code Commentary Index

activate_lock(), 538
activate_page_nolock(), 538
add_page_to_active_list(), 535
add_page_to_hash_queue(), 528
add_page_to_inactive_list(), 536
add_page_to_inode_queue(), 528
add_to_page_cache_unique(), 526
add_to_swap_cache(), 577
allocate_mm(), 239
alloc_area_pmd(), 432
alloc_area_pte(), 434
__alloc_bootmem_core(), 389
alloc_bootmem_node(), 388
__alloc_bootmem_node(), 389
__alloc_bootmem(), 387
alloc_bounce_bh(), 515
alloc_bounce_page(), 516
alloc_one_pte(), 278
__alloc_pages(), 405
alloc_pages(), 405
__alloc_pages(), 406
alloc_page(), 418
arch_get_unmapped_area(), 246, 297
badness(), 691
balance_classzone(), 414
block_flushpage(), 368
block_read_full_page(), 349
bounce_end_io_read(), 518
bounce_end_io_write(), 517
bounce_end_io(), 519
break_cow(), 340
BREAK_GFP_ORDER_HI, 488
BREAK_GFP_ORDER_LO, 488
brw_page(), 590
build_zonelists(), 214
CACHE_NAMELEN, 488
calc_vm_flags(), 247
can_share_swap_page(), 336
can_vma_merge(), 260
CHECK_PAGE(), 482
clear_page_tables(), 290
clear_user_highpage(), 339
CLUSTER_OFFSET(), 376, 378
CLUSTER_PAGES, 376, 378
contig_page_data, 405
copy_from_high_bh(), 518
copy_mm(), 236
copy_one_pte(), 279
copy_to_high_bh_irq(), 519
copy_user_highpage(), 339
create_bounce(), 513
DECLARE_FSTYPE(), 623
DEFAULT_MAX_MAP_COUNT, 246
del_page_from_active_list(), 537
del_page_from_inactive_list(), 537
do_anonymous_page(), 330
do_ccupdate_local(), 495
do_flushpage(), 368
do_generic_file_read(), 344
do_mlockall(), 302
do_mlock(), 303
do_mmap_pgoff(), 244
do_mmap(), 243
do_mremap(), 261
do_munmap(), 280
do_no_page(), 327
do_page_fault(), 313
do_shmem_file_read(), 634
do_swap_page(), 332
do_wp_page(), 338
drain_cpu_caches(), 496
enable_all_cpucaches(), 490
enable_cpucache(), 491
__end, 203
end_buffer_io_async(), 592
__end_of_fixed_addresses, 227
exclusive_swap_page(), 337
exit_mmap(), 240, 287
generic_file_mmap(), 355
expand_stack(), 315, 323
generic_file_read(), 341
expand(), 410, 412
generic_file_readahead(), 351
fail_writepage(), 667
generic_file_vm_ops(), 356
filemap_nopage(), 369
__get_dma_pages(), 419
file_read_actor(), 637
__get_free_pages(), 418
__find_get_page(), 531
__get_free_page(), 418
find_get_page(), 531
generic_file_read(), 341
__find_lock_page_helper(), 533
expand_stack(), 323
__find_lock_page(), 533
expand(), 410, 412
find_lock_page(), 533
get_max_readahead(), 376
find_max_low_pfn(), 203
get_one_pte(), 277
find_max_pfn(), 203
get_swaphandle_info(), 592
__find_page_nolock(), 532
get_unmapped_area(), 296
find_vma_intersection(), 296
get_zeroed_page(), 419
find_vma_prepare(), 253
gfp_mask, 405
find_vma_prev(), 294
handle_mm_fault(), 324
find_vma(), 293
handle_pte_fault(), 326
FIXADDR_TOP, 226
highstart_pfn, 203
fixrange_init(), 227
__init, 398
__fix_to_virt(), 226
init_bootmem_core(), 383
flush_all_zero_pkmaps(), 506
init_bootmem_node(), 382
follow_page(), 230
init_bootmem(), 401
init_emergency_pool(), 521
free_all_bootmem_core(), 401
init_mm, 236
free_all_bootmem(), 401
INIT_MM(), 236
free_area_init_core(), 208
init_tmpfs(), 622
free_area_init(), 206
__insert_vm_struct(), 252
free_area_init_node(), 206
insert_vm_struct(), 252, 253
free_area_pmd(), 439
ipc_findkey(), 678
free_area_pte(), 440
__KERNPG_TABLE, 225
free_block(), 485
kfree(), 489
__free_block(), 486
kmalloc(), 488
free_bootmem_core(), 395
kmap_atomic(), 508
free_bootmem(), 395
kmap_get_fixmap_pte(), 229
free_one_pgd(), 290
kmap_high(), 503
free_one_pmd(), 291
kmap_init(), 222, 228
free_pages_init(), 399
kmap_nonblock(), 502
__free_pages(), 420
kmap_prot, 229
__free_pages_ok(), 420
kmap_pte, 229
free_pages(), 420, 425
kmap_vstart, 229
__free_page(), 425
__kmap(), 502
free_page(), 425
__kmem_cache_alloc (SMP Case)(), 476
free_swap_and_cache(), 588
generic_file_read(), 341
flush_all_zero_pkmaps(), 506
__kmem_cache_alloc (SMP Case)(), 476
put_page_testzero(), 420
readpage(), 349
read_cluster_nonblocking(), 377
read_swap_cache_async(), 584
refill_inactive(), 540
register_bootmem_low_pages(), 204
remove_exclusive_swap_page(), 586
__remove_inode_page(), 529
remove_inode_page(), 529, 669
remove_page_from_hash_queue(), 530
remove_page_from_inode_queue(), 530
reserve_bootmem_core(), 386
reserve_bootmem_node(), 385
reserve_bootmem(), 385
rmqueue(), 410
run_task_queue(), 377
rw_swap_page_base(), 590
rw_swap_page(), 589
scan_swap_map(), 574
search_exception_table(), 319
select_bad_process(), 690
setup_memory(), 202
SET_PAGE_CACHE(), 469
SET_PAGE_SLAB(), 469
SGP_WRITE, 640
shmem_alloc_entry(), 663
shmem_create(), 628
shmem_empty(), 653
shmem_file_read(), 633
shmem_file_setup(), 675
shmem_file_write(), 638
shmem_find_swp(), 673
shmem_follow_link_inline(), 644
shmem_free_swp(), 650
shmem_getpage(), 656
shmem_get_inode(), 629
shmem_ilock, 643
SHMEM_I(), 647, 657
shmem_link(), 651
SHMEM_MAX_INDEX, 663
shmem_mkdir(), 652
shmem_mknod(), 628
shmem_mmap(), 632
shmem_nopage(), 655
SHMEM_NR_DIRECT, 664
shmem_positive(), 654
shmem_readlink_inline(), 644
shmem_readlink(), 644
shmem_read_super(), 624
shmem_rmdir(), 653
SHMEM_SB(), 625
shmem_set_size(), 626
shmem_swap_entry(), 662, 664
shmem_symlink(), 641
shmem_sync_file(), 645
shmem_truncate_direct(), 649
shmem_truncate_indirect(), 647
shmem_truncate(), 646
shmem_unlink(), 652
shmem_unuse_inode(), 670
shmem_unuse(), 669
shmem_writepage(), 667
shmem_zero_setup(), 674
SHMLBA, 682
shm_lock(), 682
SHM_RND, 682
shm_tot, 681
shrink_caches(), 550
shrink_cache(), 542
slab_break_gfp_order, 488
smp_call_function_all_cpus(), 495
start_pfn, 203
STATS_INC_GROWN(), 469
swapin_readahead(), 378
swapper_pg_dir, 222
swap_duplicate(), 578
swap_entry_free(), 580
swap_free(), 580
swap_info_get(), 581
swap_info_put(), 582
swap_out_mm(), 556
swap_out_pgd(), 558
swap_out_pmd(), 559
swap_out_vma(), 557
swap_out(), 554
swap_setup(), 605
swap_writepage(), 586
SWP_ENTRY(), 573
sync_page(), 217
sys_mlockall(), 300
sys_mlock(), 299
sys_mremap(), 261
sys_munlockall(), 306
sys_munlock(), 305
sys_shmat(), 681
sys_shmget(), 678
sys_swapoff(), 606
sys_swapon(), 594
tlb_finish_mmu(), 361
tlb_gather_mmu(), 361
tlb_remove_page(), 361, 362
totalhigh_pages, 401
truncate_complete_page(), 367
truncate_inode_pages(), 364
truncate_list_pages(), 365
truncate_partial_page(), 368
truncate(), 356
TryLockPage(), 216, 534
try_to_free_pages(), 551
try_to_free_pages_zone(), 552
try_to_swap_out(), 561
try_to_unuse(), 610
unlock_page(), 218
unmap_fixup(), 284
unuse_pgd(), 616
unuse_pmd(), 618
unuse_process(), 615
unuse_pte(), 619
unuse_vma(), 616
valid_swaphandles(), 379
vfree(), 437
vmalloc_area_pages(), 430
__vmalloc_area_pages(), 431
__vmalloc(), 427
vmalloc(), 427
vmap(), 435
__vma_link_file(), 257
__vma_link_list(), 256
__vma_link_rb(), 257
vma_link(), 255
__vma_link(), 256
vma_merge(), 258
vmfree_area_pages(), 438
vmtruncut_list(), 358
vmtruncut(), 356, 646
vm_enough_memory(), 686
vm_reend, 376
VM_SEQ_READ, 371
vm_swap_full(), 589
___wait_on_page(), 219
wait_on_page(), 219
zap_page_range(), 359
zap_pmd_range(), 361
zap_pte_range(), 362
zone_sizes_init(), 205
Index

3GiB/1GiB Split, 53, 144

896MiB limit of ZONE_NORMAL, 54
Accessing userspace, 82
access_ok(), 83
activate_page(), 160
active_list, 25, 154, 159
Address mapping to pages, 41
Address space management, 54
Address space regions, 60
Address space, 52
address_space, 24, 54, 64
add_to_page_cache_unique(), 157
add_to_page_cache(), 157
add_to_swap_cache(), 175, 176
Advanced Programmable Interrupt Controller (APIC), 54
allocate_mm(), 59
__alloc_bootmem_core(), 94
alloc_bootmem_low_pages_node(), 91, 94
alloc_bootmem_low_pages(), 90, 93
alloc_bootmem_low(), 90, 93
alloc_bootmem_node(), 91, 94
__alloc_bootmem_node(), 94
alloc_bootmem_pages_node(), 91, 94
alloc_bootmem_pages(), 90, 93
alloc_bootmem(), 90, 93
__alloc_bootmem(), 93
alloc_bounce_bh(), 149
alloc_bounce_page(), 149
alloc_pages(), 100
alloc_page(), 100
Anonymous pages, 78
Anonymous pages, backing, 182
arch_get_unmapped_area(), 68
arch_set_page_uptodate(), 25
associative mapping, 44
AS_EIO, 84
AS_ENOSPC, 84
autoconf, 2
automake, 2
Backing storage, 167
Benchmarking kernels, 12
Big Kernel Lock (BKL), 179
Binary Buddy Allocator, 98
BitKeeper, 5
bmap(), 178
Boot allocator initialisation, 92
Boot map representation, 90
Boot Memory Allocator, 89
bootmem_bootmap_pages(), 90
bootmem_data, 90
Bootstrap finalisation, 95
Bounce buffers, 144, 148
bounce_end_io_write(), 149
bounce_end_io(), 150
Browsing Code, 10
brw_page(), 178
Buddies, 98
Buddy coalescing, 98, 102
buffer_head, 149
BUG(), 95
Cache chain, 115
Cache colouring, 117
Cache creation (Slab), 125
Caches (slab allocator), 118
cache_cache, 141
cache_sizes, 137
cache_sizes_t, 137
Call graph generation, 11
cupdate_t, 140
cc_data(), 139
cc_entry(), 140
CFGS_OFF_SLAB, 123, 130
CFLGS_OPTIMIZE, 123
check_pgt_cache(), 39
ClearPageActive(), 31
ClearPageDirty(), 31
ClearPageError(), 31
ClearPageLaunder(), 31
INDEX

ClearPageReferenced(), 31
ClearPageReserved(), 31
ClearPageUptodate(), 31
clear_user_highpage(), 79
clear_user_page(), 83
clock_searchp, 126
Code Comprehension, 11
CodeViz, 10
Coloring, 116, 117
committed_space, 197
Companion CD, iii
CONFIG_SLAB_DEBUG, 117, 122
ConT est, 12
contig_page_data, 14
Copy On Write(COW), 316
Copy-On-Write (COW), 78, 82
Copying to/from userspace, 82
copy_from_high_bh(), 150
copy_from_user(), 83
copy_mm(), 59
copy_to_user(), 83
copy_user_page(), 83
CPU cache hit, 45
CPU cache line, 44
CPU Cache Management, 43
CPU cache miss, 45
cpucache, 138
cpu_vm_mask, 58
create_bounce(), 149
CREATE_MASK, 123
Creating files in tmpfs, 187
Creating VMAs, 66
ctags, 10
dbench, 12
Deactivating swap area, 180
def_flags, 58
DEF_PRIORITY, 162
Deleting a VMA, 75
Demand Allocation, 77
Demand allocation, 78
Demand Fetch, 76
Demand Paging, 78
Demand paging, 81
Device backed regions, 64
DFLGS_GROWN, 123
diff, 4, 6
difffunc, 9
diffstruct, 9
direct mapping, 44
direct-reclaim, 18, 19
do_anonymous_page(), 78
do_clear_update_local(), 141
do_mmap2(), 66
do_mmap_pgoff(), 66
do_no_page(), 77, 78
do_page_fault(), 76
do_swap_page(), 33, 78, 81
do_wp_page(), 78, 82
dummy_security_ops, 197
Eliminating fragmentation (internal), 116
empty_zero_page, 79
enable_all_cpuscaches(), 140
enable_cpucache(), 140
_end, 20
Exception handling, 75
Exception table, 319
exception_table_entry, 75
exit_mmap(), 59, 60
Expanding the stack, 77
EXPORT_SYMBOL(), 25
External fragmentation, 106
__ex_table, 75
File backed regions, 64
File/device backed pages, 80
filemap_nopage(), 64, 80
Filesyste, shared memory, 182
file_operations, 185
Finalising memory bootstrapping, 95
find_max_low_pfn(), 20
find_max_pfn(), 20
__find_page_nolock(), 158
find_vma_intersection(), 68, 69
find_vma_prepare(), 69, 70
find_vma_prev(), 67, 69
find_vma(), 67, 69
First Fit, 89
INDEX

__FIXADDR_SIZE, 54
FIXADDR_START, 54, 144
FIXADDR_TOP, 54
fixrange_init(), 40
FIX_KMAP_BEGIN, 40, 148
FIX_KMAP_END, 40, 148
flush_cache_all(), 46
flush_cache_mm(), 46
flush_cache_range(), 46
flush_dcache_page(), 47
flush_icache_page(), 47
flush_icache_range(), 47
flush_icache_user_range(), 47
flush_page_to_ram(), 47, 79
flush_tlb_all(), 43
flush_tlb_mm(), 43
flush_tlb_page(), 44
flush_tlb_pgtables(), 44
flush_tlb_range(), 43
for_each_pgdat(), 17
Fragmentation elimination (external), 110
Fragmentation, 106
Free lists, 98
free_all_bootmem_core(), 96
free_all_bootmem_node(), 91, 96
free_all_bootmem(), 90, 96
free_area_init_node(), 22, 23
free_area_init(), 23
free_area_t, 98
free_bootmem_node(), 91, 94
free_bootmem(), 90, 94
free_initmem(), 97
free_mm(), 59
free_pages_init(), 95
__free_pages(), 102
__free_page(), 102
free_page(), 102
free_pgtables(), 75
free_swap_and_cache(), 190
Frequently Asked Questions (FAQ), 3
generic_file_vm_ops, 64
Get Free Page (GFP), 103
Get Free Pages (GFP) flags, 103
__get_dma_pages(), 100
__get_free_pages(), 100
__get_free_page(), 100
get_free_page(), 100
GET_PAGE_CACHE(), 129
GET_PAGE_SLAB(), 129
get_pgd_fast(), 39
get_pgd_slow(), 39
get_swaphandle_info(), 178
get_swap_page(), 173, 176
get_unmapped_area(), 68, 69
get_user(), 83
get_vm_area(), 111
GFP flags, 103
GFP_ATOMIC, 104, 105
__GFP_DMA, 103
GFP_DMA, 103
__GFP_FS, 104
__GFP_HIGH, 104
__GFP.Highlight, 104
__GFP.Highmem, 103
GFP_HIGHUSER, 104, 105
__GFP.I0, 104
GFP_KERNEL, 104, 105
GFP_KSWAPD, 104, 105
gfp_mask, 99
GFP_NFS, 104, 105
__GFP.NOFail, 108
GFP_NOFS, 104, 105
GFP_NOHIGHIO, 104, 105
GFP_NOI0, 104, 105
__GFP.NORETRY, 108
__GFP.REPEAT, 108
GFP_USER, 104, 105
__GFP.WAIT, 104
Global zero page, 52
golden ratio, 22
GOLDEN_RATIO.PRIIME, 22
GraphViz, 10
g_cpucache_up, 140
handle_mm_fault(), 76
handle_pte_fault(), 77
High memory atomic mappings, 147
High Memory IO, 148
High memory mapping, 145
High Memory, 26, 144
highend_pfn, 92
highstart_pfn, 92
Huge TLB Filesystem (hugetlbfs), 51
Huge TLB Filesystem, 51
inactive_list, 25, 154, 159
inclusion property, 154
__init, 97
Initialising buddy allocator, 97
Initialising kmem_bufctl_t, 133
Initialising mm_struct, 59
Initialising Objects, 135
Initialising page hash table, 156
Initialising shared regions, 191
Initialising Slab Allocator, 141
Initialising swap areas, 179
Initialising the boot memory allocator, 92
Initialising virtual filesystem, 183
Initialising page tables, 39
__init_begin, 97
init_bootmem_core(), 17
init_bootmem_node(), 91
init_bootmem(), 90
init_emergency_pool(), 151
__init_end, 97
INIT_MM(), 59
init_mm(), 59
Inode queue, 157
inode_operations, 185
Inserting a memory region, 69
insert_vm_struct(), 69
Internal fragmentation, 106
Internet Relay Chat (IRC), 3
InterProcessor Interrupt (IPI), 58
IPC, 192
Kernel Address Space, 53
Kernel Benchmarking, 12
Kernel Configuration, 2
Kernel Documentation, 3
Kernel image location, 39
Kernel Patching, 4
Kernel Subtrees, 5
Kernel Traffic, 4
Kernel Trap, 4
kern_mount(), 182
kfree(), 118, 138
kmalloc(), 118, 138
kmap_atomic(), 40, 145, 147, 148
kmap_high(), 145
kmap_nonblock(), 145, 147
kmap(), 54, 145, 147
kmem_bufctl_t, 130, 131
kmem_bufctl_t types, 130
kmem_cache, 141
kmem_cache_alloc(), 118, 133
kmem_cache_create(), 118
kmem_cache_destroy(), 118
kmem_cache_free(), 118
kmem_cache_init(), 142
kmem_cache_reap(), 118
kmem_cache_shrink(), 118
kmem_cache_slabmgmt(), 130
kmem_freepages(), 142
kmem_getpages(), 142
kmem_tune_cpucache(), 140
km_type, 147
KM_TYPE_NR, 147
kswapd, 18, 101, 154
kswapd_balance(), 165
kswapd_can_sleep(), 164
kswapd_init(), 164
kswapd_wait, 165
kswapd(), 164
kunmap_atomic(), 147, 148
kunmap_high(), 146, 147
kunmap(), 145-147
LAST_PKMAP, 145
lastPkmap_nr, 146
Lazy buddy, 107
lazy TLB, 55
Least Recently Used (LRU), 154
Level 1 CPU, 43
Linear address macros, 34
Linear Address Space, 53
Linux Cross-Referencing (LXR), 10
Linux Kernel Mailing List (LKML), 4
Linux Kernel Newbies, 4
Linux Weekly News (LWN), 4
Linux-MM Website, 4
Imbench, 12
local_pages, 410
locked_vm, 58
Locking regions, 72
LockPage(), 31
lookup_swap_cache(), 176
LRU 2Q, 154
LRU list page reclaim, 160
LRU lists, 153, 159
LRU rotation, 160
lru_cache_add(), 160
lru_cache_del(), 160

Major page faults, 76
mapping_gfp_mask(), 84
map_new_virtual(), 145
MAP_POPULATE, 87
mark_page_accessed(), 160
MARK_USED(), 99
MAX_DMA_ADDRESS, 94
max_low_pfn, 92
max_mapped, 160
MAX_NR_ZONES, 25
MAX_ORDER, 98
max_pfn, 92
max_scan, 160
MAX_SWAPFILES, 168
MAX_SWAP_BADPAGES, 171
Memory Management Unit (MMU), 32
Memory pools, 151
Memory pressure, 18
Memory regions, 60
mem_init(), 95
mem_map, 15
mem_map_initialisation, 23
mem_map_t, 25
merge_segments(), 71
Minor page faults, 76
min_low_pfn, 92
mkswap, 170
mk_pte_phys(), 38
mk_pte(), 38
mlockall(), 72
mlock_fixup_all(), 73
mlock_fixup_end(), 73
mlock_fixup_middle(), 73
mlock_fixup_start(), 73
mlock_fixup(), 73
mlock(), 72
mmap_sem, 58
mmdrop(), 60
mmlist, 58
mmput(), 59
MMU, 32
mm_alloc(), 59
mm_count, 56
mm_init(), 59
mm_struct, 54
mm_users, 56
Move-To-Front heuristic, 154
move_page_tables(), 71
move_vma(), 71
munmap() 74
newseg(), 192
Node ID (NID), 17
Node structure, 15
Node-Local Allocation, 15, 100
Nodes, 14
Non-Uniform Memory Access (NUMA), 14
NRPTE, 47
nr_pages, 160
NUMA, 14

Object allocation, 115, 135
Object coloring, 116
Object freeing, 136
Object initialisation, 135
Objects, 135
one_highpage_init(), 96
OOM Detection, 195
OOM Killing, 196
OOM Management, 194
OOM Prevention, 194
oom_kill(), 196
Order allocation, 98
out_of_memory(), 195

Page allocation, 98
Page cache, 153, 155, 158
Page colour, 117
Page directory describing, 33
Page fault exception handler, 82
Page faulting (shmem), 188
Page faults, 76
Page flags, 25, 30
Page Fragmentation, 106
Page Frame Number (PFN), 16, 89
Page Global Directory (PGD), 32, 33
Page hash table, 156
page hash, 156
Page index, 24
Page lists, 24
Page Middle Directory (PMD), 32
Page reclamation, 153
Page replacement policy, 154
Page Size Extension (PSE), 40
page struct, 23
Page structure, 23
Page Table Entry (PTE), 32
Page table initialisation, 39
Page table layout, 33
Page table management, 32
Page table protection, 35
Page to zone mapping, 25
Page wait queues, 21
PageActive(), 31
PageChecked(), 31
PageClearSlab(), 31
PageDirty(), 31
PageError(), 31
PageHighMem(), 31
PageLauder(), 31
PageLocked(), 31
PageLRU(), 31
Pageout Daemon, 164
Pageout of process pages, 163
PageReferenced(), 31
PageReserved(), 31
PageSetSlab(), 31
PageSlab(), 31
PageSwapCache(), 174
pages_high, 19
pages_low, 19
pages_min, 19
pagetable_init(), 40
PageUptodate(), 31
pagevec, 166
_PAGE_ACCESSSED, 36
_PAGE_ALIGN(), 34
page_cache_alloc(), 157
page_cache_get(), 157
page_cache_init(), 156
page_cache_read(), 157, 158
page_cache_release(), 157
page_cluster, 81
_PAGE_DIRTY, 36
_page_hashfn(), 156
page_hash_bits, 156
page_hash_table, 156
PAGE_OFFSET, 53
PAGE_PER_WAITQUEUE, 21
_PAGE_PRESENT, 36
_PAGE_PROTNONE, 36
_PAGE_RW, 36
PAGE_SHIFT, 34
page_state, 29
_PAGE_USER, 36
page_waitqueue(), 22
Paging out, 164
Paging, 167
paging_init(), 40
Patch Generation, 6
Patch submission, 12
Patch usage, 4
patch, 6
PatchSet, 8
__pa(), 41
Per-CPU cache, 138
Persistent Kernel Map (PKMap), 144
per_cpu_pages, 29
<table>
<thead>
<tr>
<th>Index Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>per_cpu_pageset</td>
<td>28</td>
</tr>
<tr>
<td>PFN</td>
<td>16</td>
</tr>
<tr>
<td>PF_FREE_PAGES</td>
<td>106</td>
</tr>
<tr>
<td>PF_MEMALLOC</td>
<td>106</td>
</tr>
<tr>
<td>PF_MEMDIE</td>
<td>106</td>
</tr>
<tr>
<td>pg0</td>
<td>39</td>
</tr>
<tr>
<td>pg1</td>
<td>39</td>
</tr>
<tr>
<td>PGD</td>
<td>32</td>
</tr>
<tr>
<td>pgdat_list</td>
<td>14, 17</td>
</tr>
<tr>
<td>PGDIR_SHIFT</td>
<td>35</td>
</tr>
<tr>
<td>pgd_alloc</td>
<td>38</td>
</tr>
<tr>
<td>pgd_free</td>
<td>38</td>
</tr>
<tr>
<td>pgd_offset</td>
<td>36</td>
</tr>
<tr>
<td>pgd_quicklist</td>
<td>38</td>
</tr>
<tr>
<td>pgd_t</td>
<td>33</td>
</tr>
<tr>
<td>pgd_val</td>
<td>35</td>
</tr>
<tr>
<td>__pgd</td>
<td>35</td>
</tr>
<tr>
<td>pglist_data</td>
<td>14, 15</td>
</tr>
<tr>
<td>pgprot_t</td>
<td>35</td>
</tr>
<tr>
<td>pgprot_val</td>
<td>35</td>
</tr>
<tr>
<td>__pgprot</td>
<td>35</td>
</tr>
<tr>
<td>PG_active</td>
<td>30</td>
</tr>
<tr>
<td>PG_arch_1</td>
<td>30</td>
</tr>
<tr>
<td>PG_checked</td>
<td>30</td>
</tr>
<tr>
<td>pg_data_t</td>
<td>14, 15</td>
</tr>
<tr>
<td>PG_dirty</td>
<td>30</td>
</tr>
<tr>
<td>PG_error</td>
<td>30</td>
</tr>
<tr>
<td>PG_fs</td>
<td>30</td>
</tr>
<tr>
<td>PG_highmem</td>
<td>30</td>
</tr>
<tr>
<td>PG_launder</td>
<td>30</td>
</tr>
<tr>
<td>PG_locked</td>
<td>30</td>
</tr>
<tr>
<td>PG_lru</td>
<td>30</td>
</tr>
<tr>
<td>PG_referenced</td>
<td>30</td>
</tr>
<tr>
<td>PG_reserved</td>
<td>30</td>
</tr>
<tr>
<td>PG_skip</td>
<td>30</td>
</tr>
<tr>
<td>PG_slab</td>
<td>30</td>
</tr>
<tr>
<td>PG_unused</td>
<td>30</td>
</tr>
<tr>
<td>PG_uptodate</td>
<td>30</td>
</tr>
<tr>
<td>Physical Address Extension (PAE)</td>
<td>26</td>
</tr>
</tbody>
</table>
pte_mkyoung(), 38
pte_modify(), 37
pte_offset_map(), 50
pte_offset(), 36
pte_old(), 38
pte_page(), 38
pte_quicklist, 38
pte_rdprotect(), 37
pte_read(), 37
pte_t, 33
pte_to_swap_entry(), 171
pte_val(), 35
pte_write(), 37
pte_wrprotect(), 37
pte_young(), 38
__pte(), 35
PTRS_PER_PGD, 35
PTRS_PER_PMD, 35
PTRS_PER_PTE, 35
put_user(), 83

quicklists, 38
RAM based filesystem, 182
read_swap_cache_async(), 176
REAP_SCANLEN, 126
Refilling inactive_list, 159
refill_inactive(), 154, 159
remap_file_pages(), 87
remove_exclusive_swap_page(), 177
remove_inode_page(), 157
remove_page_from_hash_queue(), 157
remove_page_from_inode_queue(), 157
reserve_bootmem_node(), 91
reserve_bootmem(), 90
Resident Set Size (RSS), 58
Retiring boot memory, 95
Reverse Mapping (rmap), 46
Reverse Mapping (RMAP), 81
Reverse mapping objects, 48
Reverse mapping pages, 46
rss, 58
rw_swap_page_base(), 178
rw_swap_page(), 177, 178
scan_swap_map(), 173
search_exception_table(), 75
security_operations, 197
security_ops, 197
security_vm_enough_memory(), 197
set associative mapping, 44
SetPageActive(), 31
SetPageChecked(), 31
SetPageDirty(), 31
SetPageError(), 31
SetPageLaunder(), 31
SetPageReferenced(), 31
SetPageReserved(), 31
SetPageUptodate(), 25, 31
setup_arch(), 92
setup_arg_flags(), 63
setup_memory(), 92
SET_PAGE_CACHE(), 129
SET_PAGE_SLAB(), 129
set_page_zone(), 26
set_pte(), 38
set_shrinker(), 143
Shared Regions, 191
shm, 182
shmat(), 192
Shmem file creation, 187
Shmem functions, 184
shmem_commit_write(), 184
shmem_dir_inode_operations, 185
shmem_file_operations, 185
shmem_getpage(), 186
shmem_inodes, 182
shmem_inode_cache, 193
shmem_inode_info, 183
shmem_inode_operations, 185
SHMEM_I(), 183
SHMEM_MAX_INDEX, 189
shmem_prepare_write(), 184
shmem_readpage(), 184
shmem symlink inline operations, 185
shmem symlink inode operations, 185
shmem_writepage(), 184
shmget(), 192
SHM_DEST, 192
SHM_HUGETLB, 51
SHM_LOCK, 184
INDEX

SHM_UNLOCK, 184
shm_vm_ops, 192
shrink_caches(), 162
shrink_cache(), 160
size-N cache, 137
size-N(DMA) cache, 137
Slab allocator, 115
Slab cache allocation flags, 124
Slab cache chain, 115
Slab cache colouring, 124
Slab cache creation, 125
Slab cache deletion, 128
Slab cache dynamic flags, 123
Slab cache reaup, 126
Slab cache shrinking, 127
Slab cache static flags, 123
Slab cache, 115
Slab caches, per-CPU, 138
Slab creation, 131
Slab debugging, 117
Slab deletion, 135
slab descriptor, 130
Slab descriptors, 130
Slab free object tracking, 131
Slab initialisation, 141
Slab object allocation, 135, 136
Slab objects, see Objects, 135
slabinfo, 118
Slabs, 115, 129
Slabs, buddy interaction, 142
Slabs, finding free objects, 133
Slabs, number of objects, 134
slabs_free, 119
slabs_full, 119
slabs_partial, 119
SLAB_ATOMIC, 124
slab_bufctl(), 132
SLAB_CACHE_DMA, 123
SLAB_CTOR_ATOMIC, 125
SLAB_CTOR_CONSTRUCTOR, 125
SLAB_CTOR_VERIFY, 125
SLAB_DEBUG_FREE, 124
SLAB_DEBUG_INITIAL, 124
SLAB_DMA, 124
SLAB_HWCACHE_ALIGN, 123
SLAB_KERNEL, 124
SLAB_MUST_HWCACHE_ALIGN, 123
SLAB_NFS, 124
SLAB_NOFS, 124
SLAB_NOHIGHIO, 124
SLAB_NOIO, 124
SLAB_NO_REAP, 123
SLAB_POISON, 124
SLAB_RED_ZONE, 124
SLAB_USER, 124
Small allocation caches, 116, 137
Sourceforge.net, 3
SPEC, 12
Stack algorithm, 154
Stack Expansion, 77
startup_32(), 39
strlen_user(), 83
strncpy_from_user(), 83
struct kmem_cache_s, 120
Swap area deactivating, 180
Swap area initialising, 179
Swap area, 167
Swap area, describing, 168
Swap cache, 153, 173
Swap entries, 171
Swap management, 167
Swap reading, 176, 178
Swap writing, 177
SWAPFILE_CLUSTER, 173
swapin_readahead(), 76, 81
swapper_pg_dir, 39
Swapping, 167
SWAP_CLUSTER_MAX, 162
swap_duplicate(), 175, 176
swap_free(), 176
swap_header, 170
swap_info, 168, 171
swap_info_struct, 168
swap_list, 169
SWAP_MAP_BAD, 169
SWAP_MAP_MAX, 169
swap_mm, 163
swap_ops, 174
swap_out_mm(), 164
swap_out_vma(), 164
swap_out(), 153, 163
swp_entry_t, 171
swp_entry_to_pte(), 171
SWP_ENTRY(), 172
SWP_OFFSET(), 171
SWP_TYPE(), 171
SWP_USED, 168
SWP_WRITEOK, 168
System V IPC, 192
sys_mmap2(), 66
sys_mprotect(), 71
sys_mremap(), 71
sys_munlockall(), 73
sys_munlock(), 73
sys_swapoff(), 180
sys_swapon(), 179
TestClearPageLRU(), 31
TestSetPageLRU(), 31
Thread identification, 54
thundering herd, 21
TLB API, 43
TLB flushing, 55
TLB, 32, 42
tmpfs, 182
total_vm, 58
tq_disk, 165
Translation Lookaside Buffer (TLB), 32, 42
Trivial Patch Monkey, 13
try_to_free_buffers(), 104
try_to_swap_out(), 164
try_to_unuse(), 180
Understanding the Linux Kernel, 1
unified diffs, 4
UnlockPage(), 21, 31
unmap_fixup(), 75
update_mmu_cache(), 44
UseNet, 3
Userspace accessing, 82
__va(), 41
vfree(), 113
Virtual Memory Area, 60
virt_to_page(), 42
VM Regress, 12
VMA Creation, 66
VMA deletion, 75
VMA insertion, 69
VMA locking, 72
VMA merging, 71
VMA operations, 62
VMA remapping, 71
VMA searching, 67
VMA unlocking, 73
VMA, 60
vmalloc address space, 111
vmalloc areas, 110
vmalloc_32(), 111, 112
vmalloc_dma(), 111, 112
VMALLOC_END, 110
VMALLOC_OFFSET, 53
VMALLOC_RESERVE, 54
VMALLOC_START, 110
vmalloc(), 106, 110, 112
vmap(), 114
__vma_link(), 70
vma_link(), 70
vma_merge(), 69, 71
vmlist_lock, 111
vmspace, 54
VM_ACCOUNT, 86, 197
vm_acct_memory(), 197
VMALLOC, 111
vm_area_struct, 54, 60
VM_DENYWRITE, 63
vm_enough_memory(), 194
VM_EXECUTABLE, 63
VM_GROWSDOWN, 63
VM_GROWSUP, 63
VM_IO, 63
VM_IOREMAP, 111
VM_LOCKED, 63
VM_MAYEXEC, 63
VM_MAYREAD, 63
VM_MAYSHARE, 63
VM_MAYWRITE, 63
vm_operations_struct, 62
VM_RAND_READ, 63
VM_RESERVED, 63
INDEX

VM_SEQ_READ, 63
VM_SHM, 63
VM_STACK_FLAGS, 63
vm_struct, 110
vm_unacct_memory(), 197
vsyscall page, 84

Waiting on pages, 21
wait_on_page(), 21
wait_table_size(), 21
working set, 154

Zone balance, 19
Zone dimensions, 14
Zone fallbacks, 16
Zone initialisation, 22
Zone modifiers, 103
Zone pressure, 18
Zone size calculation, 20
Zone structure, 17
Zone watermarks, 18
Zones, 14
zones_sizes, 22
ZONE_DMA, 14
ZONE_HIGHMEM, 14
zone_holes, 22
ZONE_NORMAL, 14
ZONE_NORMAL at 896MiB, 54
ZONE_PADDING(), 28
zone_sizes_init(), 41
zone_start_paddr, 22
zone_struct, 14, 17
zone_t, 14, 17
zone_table, 25