University of Limerick

Department of Computer Science and Information Systems

An Investigation into the Theoretical Foundations and Implementation of the Linux Virtual Memory Manager
by
Mel Gorman

Prepared under the direction of Dr. Patrick Healy

A thesis presented to the University of Limerick
in fulfillment of the requirements for the degree of

Master of Science
April, 2003
Limerick
University of Limerick

Department of Computer Science and Information Systems

Abstract

An Investigation into the Theoretical Foundations and Implementation of the
Linux Virtual Memory Manager
by Mel Gorman

Adviser: Dr. Patrick Healy
April, 2003
Limerick

The development of Linux is unusual in that it was built with a practical rather
than a theoretical emphasis. While many of the algorithms in the Virtual Mem-
ory (VM) system were designed by theorists, the implementations have diverged
considerably. Instead of following the traditional development cycle of design to
implementation, changes are made based on “real world” behavior and intuitive de-
cisions by developers.

This has resulted in a poorly documented VM understood only by a small core
group of developers and is addressed only by overviews in a small number of books
or websites. This requires that even a casual observer invest a large amount of
time to study code and the field of Memory Management to understand the VM
implementation. The problem is compounded by the fact that code only states what
occurs in a very small instance making it difficult to see how the overall system
functions. This is analogous to using a microscope to identify a piece of furniture.

As Linux gains in popularity in the business and academic worlds, more de-
velopers are expressing an interest in the Linux Kernel and the lack of detailed
documentation is a significant barrier to entry. The objective of this thesis is to
document fully how the VM in kernel 2.4.20 is implemented including its structure,
the algorithms used, the implementations thereof and the Linux specific features.
Combined with the companion document “Code Commentary on the Linux Virtual
Memory Manager” these documents represents a detailed tour of the VM explaining
line by line how the it operates, its theoretical basis and approaches to studying and
understanding the VM.

It is envisioned that this will drastically reduce the amount of time a developer
or researcher needs to invest to understand what is happening inside the Linux VM.
To my family, especially my parents who kept me in college long after I should have got a job. To my girlfriend, who listened to more tech talk than any geek. To my first supervisor Dr. John O’Gorman, who encouraged and guided me through the bulk of the work, may he rest in peace. Finally, to the thousands of people who have contributed to the Linux kernel whose dedication has produced such a high quality system.
Contents

List of Figures v
List of Tables viii

1 Introduction 1
  1.1 Getting Started ........................................... 2
  1.2 Managing the Source ....................................... 5
  1.3 Browsing the Code .......................................... 11
  1.4 Reading the Code ........................................... 14
  1.5 Submitting Patches ......................................... 15

2 Describing Physical Memory 17
  2.1 Nodes ....................................................... 18
  2.2 Zones ....................................................... 20
  2.3 Zone Initialisation ......................................... 26
  2.4 Pages ....................................................... 28
  2.5 High Memory ............................................... 32
  2.6 What’s New In 2.6 ........................................... 33

3 Page Table Management 37
  3.1 Describing the Page Directory ............................. 38
  3.2 Describing a Page Table Entry ............................ 40
  3.3 Using Page Table Entries .................................. 42
  3.4 Translating and Setting Page Table Entries .............. 44
  3.5 Allocating and Freeing Page Tables ....................... 44
  3.6 Kernel Page Tables ......................................... 45
  3.7 Mapping addresses to a struct page ....................... 47
  3.8 Translation Lookaside Buffer (TLB) ....................... 49
  3.9 Level 1 CPU Cache Management ............................ 50
  3.10 What’s New In 2.6 .......................................... 53
CONTENTS

4 Process Address Space 61
  4.1 Linear Address Space ........................................... 62
  4.2 Managing the Address Space ..................................... 63
  4.3 Process Address Space Descriptor ............................... 64
  4.4 Memory Regions .................................................. 70
  4.5 Exception Handling ................................................ 86
  4.6 Page Faulting ...................................................... 87
  4.7 Copying To/From Userspace ...................................... 93
  4.8 What’s New in 2.6 .................................................. 95

5 Boot Memory Allocator 104
  5.1 Representing the Boot Map ....................................... 105
  5.2 Initialising the Boot Memory Allocator ......................... 106
  5.3 Allocating Memory ................................................ 107
  5.4 Freeing Memory .................................................... 109
  5.5 Retiring the Boot Memory Allocator .............................. 109
  5.6 What’s New in 2.6 .................................................. 111

6 Physical Page Allocation 115
  6.1 Managing Free Blocks ............................................. 115
  6.2 Allocating Pages .................................................. 117
  6.3 Free Pages .......................................................... 119
  6.4 Get Free Page (GFP) Flags ........................................ 121
  6.5 Avoiding Fragmentation ........................................... 123
  6.6 What’s New In 2.6 .................................................. 124

7 Non-Contiguous Memory Allocation 128
  7.1 Describing Virtual Memory Areas ................................. 128
  7.2 Allocating A Non-Contiguous Area .............................. 130
  7.3 Freeing A Non-Contiguous Area .................................. 131
  7.4 What’s New in 2.6 .................................................. 131

8 Slab Allocator 134
  8.1 Caches .............................................................. 137
  8.2 Slabs ............................................................... 147
  8.3 Objects ............................................................. 154
  8.4 Sizes Cache ........................................................ 156
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.5</td>
<td>Per-CPU Object Cache</td>
<td>158</td>
</tr>
<tr>
<td>8.6</td>
<td>Slab Allocator Initialisation</td>
<td>161</td>
</tr>
<tr>
<td>8.7</td>
<td>Interfacing with the Buddy Allocator</td>
<td>162</td>
</tr>
<tr>
<td>8.8</td>
<td>What’s New in 2.6</td>
<td>162</td>
</tr>
<tr>
<td>9</td>
<td>High Memory Management</td>
<td>168</td>
</tr>
<tr>
<td>9.1</td>
<td>Managing the PKMap Address Space</td>
<td>169</td>
</tr>
<tr>
<td>9.2</td>
<td>Mapping High Memory Pages</td>
<td>169</td>
</tr>
<tr>
<td>9.3</td>
<td>Mapping High Memory Pages Atomically</td>
<td>172</td>
</tr>
<tr>
<td>9.4</td>
<td>Bounce Buffers</td>
<td>173</td>
</tr>
<tr>
<td>9.5</td>
<td>Emergency Pools</td>
<td>175</td>
</tr>
<tr>
<td>9.6</td>
<td>What’s New in 2.6</td>
<td>176</td>
</tr>
<tr>
<td>10</td>
<td>Page Frame Reclamation</td>
<td>179</td>
</tr>
<tr>
<td>10.1</td>
<td>Page Replacement Policy</td>
<td>180</td>
</tr>
<tr>
<td>10.2</td>
<td>Page Cache</td>
<td>182</td>
</tr>
<tr>
<td>10.3</td>
<td>LRU Lists</td>
<td>185</td>
</tr>
<tr>
<td>10.4</td>
<td>Shrinking all caches</td>
<td>189</td>
</tr>
<tr>
<td>10.5</td>
<td>Swapping Out Process Pages</td>
<td>189</td>
</tr>
<tr>
<td>10.6</td>
<td>Pageout Daemon (kswapd)</td>
<td>191</td>
</tr>
<tr>
<td>10.7</td>
<td>What’s New in 2.6</td>
<td>192</td>
</tr>
<tr>
<td>11</td>
<td>Swap Management</td>
<td>196</td>
</tr>
<tr>
<td>11.1</td>
<td>Describing the Swap Area</td>
<td>197</td>
</tr>
<tr>
<td>11.2</td>
<td>Mapping Page Table Entries to Swap Entries</td>
<td>201</td>
</tr>
<tr>
<td>11.3</td>
<td>Allocating a swap slot</td>
<td>202</td>
</tr>
<tr>
<td>11.4</td>
<td>Swap Cache</td>
<td>204</td>
</tr>
<tr>
<td>11.5</td>
<td>Reading Pages from Backing Storage</td>
<td>207</td>
</tr>
<tr>
<td>11.6</td>
<td>Writing Pages to Backing Storage</td>
<td>208</td>
</tr>
<tr>
<td>11.7</td>
<td>Reading/Writing Swap Area Blocks</td>
<td>208</td>
</tr>
<tr>
<td>11.8</td>
<td>Activating a Swap Area</td>
<td>209</td>
</tr>
<tr>
<td>11.9</td>
<td>Deactivating a Swap Area</td>
<td>210</td>
</tr>
<tr>
<td>11.10</td>
<td>What’s New in 2.6</td>
<td>212</td>
</tr>
<tr>
<td>12</td>
<td>Out Of Memory Management</td>
<td>213</td>
</tr>
<tr>
<td>12.1</td>
<td>Checking Available Memory</td>
<td>213</td>
</tr>
<tr>
<td>12.2</td>
<td>Determining OOM Status</td>
<td>214</td>
</tr>
</tbody>
</table>
12.3 Selecting a Process .................................................. 215
12.4 Killing the Selected Process ................................. 216
12.5 Is That It? ............................................................... 216
12.6 What’s New in 2.6 ...................................................... 216

13 The Final Word .......................................................... 218
## List of Figures

1.1 Example Patch ................................................. 8
2.1 Relationship Between Nodes, Zones and Pages .................... 18
2.2 Zone Watermarks ................................................ 23
2.3 Call Graph: `setup_memory()` .................................. 24
2.4 Sleeping On a Locked Page ...................................... 25
2.5 Call Graph: `free_area_init()` ................................ 27
3.1 Page Table Layout .............................................. 39
3.2 Linear Address Bit Size Macros ................................. 39
3.3 Linear Address Size and Mask Macros ........................... 40
3.4 Call Graph: `paging_init()` .................................... 47
4.1 Kernel Address Space ........................................... 62
4.2 Data Structures related to the Address Space .................... 65
4.3 Memory Region Flags ............................................ 73
4.4 Call Graph: `sys mmap2()` ..................................... 78
4.5 Call Graph: `get_unmapped_area()` .............................. 80
4.6 Call Graph: `insert_vm_struct()` ................................. 80
4.7 Call Graph: `sys_mremap()` .................................... 82
4.8 Call Graph: `move_vma()` ....................................... 82
4.9 Call Graph: `move_page_tables()` ............................... 83
4.10 Call Graph: `sys_mlock()` .................................... 84
4.11 Call Graph: `do_munmap()` .................................... 85
4.12 Call Graph: `do_page_fault()` ................................ 88
4.13 `do_page_fault()` Flow Diagram ................................ 89
4.14 Call Graph: `handle_mm_fault()` ............................... 90
4.15 Call Graph: `do_no_page()` .................................... 91
LIST OF FIGURES

4.16 Call Graph: do_swap_page() ........................................ 92
4.17 Call Graph: do_wp_page() ........................................ 93

5.1 Call Graph: alloc_bootmem() ................................. 107
5.2 Call Graph: mem_init() ........................................ 110
5.3 Initialising mem_map and the Main Physical Page Allocator 111

6.1 Free page block management ...................................... 116
6.2 Allocating physical pages ........................................ 118
6.3 Call Graph: alloc_pages() ...................................... 119
6.4 Call Graph: __free_pages() .................................... 120

7.1 vmalloc Address Space ........................................... 129
7.2 Call Graph: vmalloc() ........................................... 130
7.3 Relationship between vmalloc(), alloc_page() and Page Faulting 132
7.4 Call Graph: vfree() .............................................. 133

8.1 Layout of the Slab Allocator .................................... 135
8.2 Slab page containing Objects Aligned to L1 CPU Cache 136
8.3 Call Graph: kmem_cache_create() ............................ 144
8.4 Call Graph: kmem_cache_reap() ................................ 145
8.5 Call Graph: kmem_cache_shrink() ............................. 146
8.6 Call Graph: __kmem_cache_shrink() ......................... 146
8.7 Call Graph: kmem_cache_destroy() .......................... 147
8.8 Page to Cache and Slab Relationship .......................... 148
8.9 Slab With Descriptor On-Slab .................................. 149
8.10 Slab With Descriptor Off-Slab ................................ 150
8.11 Call Graph: kmem_cache_grow() ............................ 150
8.12 Initialised kmem_bufctl_t Array ............................ 152
8.13 Call Graph: kmem_slab_destroy() ......................... 154
8.14 Call Graph: kmem_cache_alloc() .......................... 155
8.15 Call Graph: kmem_cache_free() ............................ 156
8.16 Call Graph: kmalloc() ....................................... 157
8.17 Call Graph: kfree() .......................................... 158

9.1 Call Graph: kmap() ............................................. 170
9.2 Call Graph: kunmap() ........................................... 172
9.3 Call Graph: create_bounce() ...................... 174
9.4 Call Graph: bounce_end_io_read/write() .......... 175
9.5 Acquiring Pages from Emergency Pools ........... 176

10.1 Page Cache LRU Lists ................................. 181
10.2 Call Graph: generic_file_read() .......... 185
10.3 Call Graph: add_to_page_cache() ................. 185
10.4 Call Graph: shrink_caches() ...................... 189
10.5 Call Graph: swap_out() .............................. 190
10.6 Call Graph: kswapd() ............................... 191

11.1 Storing Swap Entry Information in swp_entry_t ........ 202
11.2 Call Graph: get_swap_page() ...................... 203
11.3 Call Graph: add_to_swap_cache() ................. 204
11.4 Adding a Page to the Swap Cache .................. 205
11.5 Call Graph: read_swap_cache_async() .......... 207
11.6 Call Graph: sys_writepage() ...................... 208

12.1 Call Graph: out_of_memory() ....................... 215

13.1 Broad Overview on how VM Sub-Systems Interact ..... 219
List of Tables

1.1 Kernel size as an indicator of complexity .............................. 1
2.1 Flags Describing Page Status .................................................. 31
2.2 Macros For Testing, Setting and Clearing page→flags Status Bits . 32
3.1 Page Table Entry Protection and Status Bits ................................. 41
3.2 Translation Lookaside Buffer Flush API ...................................... 50
3.3 Translation Lookaside Buffer Flush API (cont) ............................... 51
3.4 Cache and TLB Flush Ordering .................................................. 52
3.5 CPU Cache Flush API ............................................................ 53
3.6 CPU D-Cache and I-Cache Flush API ......................................... 54
4.1 System Calls Related to Memory Regions ..................................... 66
4.2 Functions related to memory region descriptors ............................. 69
4.3 Memory Region VMA API ........................................................ 101
4.4 Reasons For Page Faulting ....................................................... 102
4.5 Accessing Process Address Space API ......................................... 103
5.1 Boot Memory Allocator API for UMA Architectures .................................. 113
5.2 Boot Memory Allocator API for NUMA Architectures ........................ 114
6.1 Physical Pages Allocation API .................................................... 117
6.2 Physical Pages Free API .......................................................... 119
6.3 Low Level GFP Flags Affecting Zone Allocation ............................. 121
6.4 Low Level GFP Flags Affecting Allocator behaviour ......................... 122
6.5 Low Level GFP Flag Combinations For High Level Use .................... 122
6.6 High Level GFP Flags Affecting Allocator Behaviour ....................... 123
6.7 Process Flags Affecting Allocator behaviour .................................. 124
7.1 Non-Contiguous Memory Allocation API ...................................... 131
LIST OF TABLES

7.2 Non-Contiguous Memory Free API ........................................ 131

8.1 Slab Allocator API for caches ........................................... 165
8.2 Internal cache static flags ............................................... 166
8.3 Cache static flags set by caller ......................................... 166
8.4 Cache static debug flags .................................................. 166
8.5 Cache Allocation Flags ................................................... 166
8.6 Cache Constructor Flags .................................................. 167

9.1 High Memory Mapping API ............................................... 171
9.2 High Memory Unmapping API ............................................ 172

10.1 Page Cache API .......................................................... 194
10.2 LRU List API .............................................................. 195

11.1 Swap Cache API ............................................................ 206
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 RTFS\textsuperscript{1} or else are referred to the kernel newbies mailing list (http://www.kerneldnewbies.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

\textsuperscript{1}Read The Flaming Source. It doesn’t really stand for Flaming but there could be children watching.
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

Kernel Configuration

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 README
or INSTALL file at the top-level of the source tree [FF02]. In fact, some automated
build tools such as 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 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 Makefile by hand but this is rarely the case now. Free software usually uses at least autoconf\footnote{http://www.gnu.org/software/autoconf/} to automate testing of the build environment and automake\footnote{http://www.gnu.org/software/automake/} to simplify the creation of Makefiles so building is often as simple as:

```
mel@joshua: project $ ./configure && make
```

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\footnote{http://www.tldp.org/HOWTO/Kernel-HOWTO/index.html} 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

**Kernel Documentation**

Open Source projects will usually have a home page, especially since free project hosting sites such as http://www.sourceforge.net 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 newsgroups, 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/Linux Weekly News (LWN). 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.kernelnewbies.org/Linux Kernel Newbies which is a vast source of information on the kernel that is invaluable to new and experienced people alike.

There 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 Linux-MM Website 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.kernelltrap.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**  Patch usage Kernel Patching

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 compare two full versions of a file side by side. A developer familiar with the code can easily see

\(^5\)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.
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 Kernel 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 BitKeeper In more recent times, some developers have started using
a source code control system called BitKeeper (http://www.bitmover.com), a pro-
prietary 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

Patch Generation

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 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

\(^6\)http://www.gnu.org
affected C function.

```plaintext
--- 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.
 *
 * -- wli
 + *
 + * There is a brief explanation of how a buddy algorithm works at
 + * http://www.memorymanagement.org/articles/alloc.html . A better idea
 + * is to read the explanation from a book like UNIX Internals by
 + * Uresh Vahalia
 + *
 + */
+/**
+ *
+ * __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
+ *
+ * This function returns the pages allocated by __alloc_pages and tries to
+ * merge buddies if possible. Do not call directly, use free_pages()
+ **/
+static void FASTCALL(___free_pages_ok (struct page *page, unsigned int order));
+static void ___free_pages_ok (struct page *page, unsigned int order) {
+
```

Figure 1.1: Example Patch

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.
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

**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/](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/](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
```

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 direc-
tory. 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.
1.3. Browsing the Code

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 pack-
ages 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 struc-
tures 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

Browsing 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
1.3.1 Analysing Code Flow

contained as inline functions in headers. To complicate matters, files of interest may be buried beneath architecture specific directories making tracking them down time consuming.

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

**Call 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:

```
me1@joshua: patchset $ download 2.4.22
me1@joshua: patchset $ createset 2.4.22
me1@joshua: patchset $ make-gengraph.sh 2.4.22
me1@joshua: patchset $ cd kernels/linux-2.4.22
me1@joshua: linux-2.4.22 $ gengraph -t -s "alloc_bootmem_low_pages \ zone_sizes_init" -f paging_init
me1@joshua: linux-2.4.22 $ gv paging_init.ps
```
1.4 Reading the Code

Code Comprehension

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.

Benchmarking kernslsKernel Benchmarking

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 \url{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

Patch submission

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.

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.

\(^7\)\url{http://www.kernel.org/pub/linux/kernel/people/rusty/trivial/}
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
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 dimensions**

- **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 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.

![Figure 2.1: Relationship Between Nodes, Zones and Pages](image)

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

**Node structure**

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* **Node-Local Allocation** 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>`:

```c
typedef struct pglist_data {
    zone_t node_zones[MAX_NR_ZONES];
    zonelist_t node_zonelists[GFP_ZONEMASK+1];
    int nr_zones;
    struct page *node_mem_map;
    unsigned long *valid_addr_bitmap;
    struct bootmem_data *bdata;
    unsigned long node_start_paddr;
    unsigned long node_start_mapnr;
    unsigned long node_size;
    int node_id;
    struct pglist_data *node_next;
} pg_data_t;
```

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`; Zone fallbacks

- **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)/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:

\[(\text{page_phys_addr >> PAGE_SHIFT})\]

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;

node_size The total number of pages in this zone;

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

node_next 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

### Zone structure

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;
2.2.1 Zone Watermarks

Zone watermarks Memory pressure Zone pressure

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 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).

`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;
2.2.2 Calculating The Size of Zones

**pages_high** Once kswapd has been woken to start freeing pages it will not consider the zone to be “balanced” Zone balance when pages_high pages are free. Once the watermark has been reached, kswapd will go back to sleep. In Solaris, this is called lotsfree and in BSD, it is called free_target. The default for pages_high is three times the value of 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.

### 2.2.2 Calculating The Size of Zones

**Zone size calculation**

The PFN is an offset, counted in pages, within the physical memory map. The first PFN usable by the system, 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 mm/bootmem.c for use with the boot memory allocator.
2.2.3 Zone Wait Queue Table

Page wait queues Waiting on pages

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.

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:

$$
\text{wait\_table\_size} = \log_2\left(\frac{\text{NoPages} \times 2}{\text{PAGE\_PER\_WAITQUEUE}} - 1\right)
$$

The field `zone_t→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 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 struct page being hashed.

It works by simply multiplying the address by GOLDEN_RATIO_PRIME and shifting the result zone_t->wait_table_shift bits right to index the result within the hash table. GOLDEN_RATIO_PRIME[Lev00] is the largest prime that is closest to the golden ratio[Knu68] of the largest integer that may be represented by the architecture.

2.3 Zone Initialisation

Zone initialisation

The zones are initialised after the kernel page tables have been fully setup by 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 free_area_init() for UMA architectures or free_area_init_node() for NUMA. The only parameter required for UMA is zones_size. The full list of parameters:

nid is the Node ID which is the logical identifier of the node whose zones are being initialised;

pgdat is the node’s pg_data_t that is being initialised. In UMA, this will simply be contig_page_data;

pmap is set later by free_area_init_core() to point to the beginning of the local lmem_map array allocated for the node. In NUMA, this is ignored as NUMA treats mem_map as a virtual array starting at PAGE_OFFSET. In UMA, this pointer is the global mem_map variable which is now mem_map gets initialised in UMA.

zones_sizes is an array containing the size of each zone in pages;

zone_start_paddr is the starting physical address for the first zone;

zone_holes is an array containing the total size of memory holes in the zones;
2.3.1. Initialising mem_map

It is the core function free_area_init_core() which is responsible for filling in each zone_t with the relevant information and the allocation of the 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

mem_map initialisation

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

### Page structure

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 its 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**  
**Page lists** 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** Page 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** Page 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

Page to zone mapping

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 pages 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:

```c
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

```c
733 zone_table[nid * MAX_NR_ZONES + j] = zone;
```

Where nid is the node ID, j is the zone index and zone is the zone_t struct. For each page, the function set_page_zone() is called as

```c
788 set_page_zone(page, nid * MAX_NR_ZONES + j);
```

The parameter, page, is the page whose zone is being set. So, clearly the index in the zone_table is stored in the page.
<table>
<thead>
<tr>
<th>Bit name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PG_active</td>
<td>This bit <strong>Page flags</strong> is set if a page is on the <strong>active_list</strong> 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 <strong>mem_init()</strong></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 writepage() 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 <strong>active_list</strong> or the <strong>inactive_list</strong>, 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
2.5 High Memory

As the addresses 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

<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>ClearPageChecked()</td>
</tr>
<tr>
<td>PG_dirty</td>
<td>SetPageDirty()</td>
<td>PageDirty()</td>
<td>ClearPageDirty()</td>
</tr>
<tr>
<td>PG_error</td>
<td>SetPageError()</td>
<td>PageError()</td>
<td>ClearPageError()</td>
</tr>
<tr>
<td>PG_highmem</td>
<td>n/a</td>
<td>PageHighMem()</td>
<td>n/a</td>
</tr>
<tr>
<td>PG_launder</td>
<td>SetPageLaunder()</td>
<td>PageLaunder()</td>
<td>ClearPageLaunder()</td>
</tr>
<tr>
<td>PG_locked</td>
<td>LockPage()</td>
<td>PageLocked()</td>
<td>UnlockPage()</td>
</tr>
<tr>
<td>PG_lru</td>
<td>TestSetPageLRU()</td>
<td>PageLRU()</td>
<td>TestClearPageLRU()</td>
</tr>
<tr>
<td>PG_referenced</td>
<td>SetPageReferenced()</td>
<td>PageReferenced()</td>
<td>ClearPageReferenced()</td>
</tr>
<tr>
<td>PG_reserved</td>
<td>SetPageReserved()</td>
<td>PageReserved()</td>
<td>ClearPageReserved()</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>PageSlab()</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>

Table 2.2: Macros For Testing, Setting and Clearing page→flags Status Bits
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`. 
2.6. What's New In 2.6

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.** PTE chains 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 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 readins. 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()`.
Chapter 3
Page Table Management

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...
3.1 Describing the Page Directory

Page directory describing

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 Page table layout.

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 tables as illustrated in Figure 3.2 Linear address macros.

The 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 SIZE macros reveal how many bytes are addressed by each entry at each level. The relationship between the SIZE and MASK macros is illustrated in Figure 3.3.

For the calculation of each of the triplets, only SHIFT is important as the other
3.1. Describing the Page Directory

Figure 3.1: Page Table Layout

Figure 3.2: Linear Address Bit Size Macros

two are calculated based on it. For example, the three macros for page level on the x86 are:

```c
5 #define PAGE_SHIFT 12
6 #define PAGE_SIZE (1UL << PAGE_SHIFT)
7 #define PAGE_MASK (~(PAGE_SIZE-1))
```

`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 $2^{PAGE\_SHIFT}$ which is the equivalent of the code above. Finally the mask is calculated as the negation of the bits which make up the `PAGE\_SIZE - 1`. If a page needs to be aligned on a page boundary, `PAGE\_ALIGN()` is used. This macro adds `PAGE\_SIZE - 1` to
Figure 3.3: Linear Address Size and Mask Macros

the address before simply ANDing it with the PAGE_MASK to zero out the page offset bits.

PMD_SHIFT is the number of bits in the linear address which are mapped by the second level part of the table. The PMD_SIZE and PMD_MASK are calculated in a similar way to the page level macros.

PGDIR_SHIFT is the number of bits which are mapped by the top, or first level, of the page table. The PGDIR_SIZE and PGDIR_MASK are calculated in the same manner as above.

The last three macros of importance are the PTRS_PER_x which determine the number of entries in each level of the page table. PTRS_PER_PGD is the number of pointers in the PGD, 1024 on an x86 without PAE. PTRS_PER_PMD is for the PMD, 1 on the x86 without PAE and PTRS_PER_PTE is for the lowest level, 1024 on the x86.

3.2 Describing a Page Table Entry

Page table protection PTE Protection Bits

As mentioned, each entry is described by the structspte_t, pmd_t and 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, 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 asm/page.h, which takes the above types and returns the relevant part of the structs. They are pte_val(), pmd_val(),
3.2. Describing a Page Table Entry

pgd_val() and pgprot_val(). To reverse the type casting, 4 more macros are provided __pte(), __pmd(), __pgd() and __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 pte_t is simply a 32 bit integer within a struct. Each 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 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.

<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

These bits are self-explanatory except for the _PAGE_PROTNONE which we will discuss further. On the x86 with Pentium III and higher, this bit is called the 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 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 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

PTE macros

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:

```plaintext
407 pgd_t *pgd;  
408 pmd_t *pmd;  
409 pte_t *ptep, pte;  
410 pgd = pgd_offset(mm, address);  
```
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_rdprotect();
- The write permissions are tested with pte_write(), set with pte_mkwite() 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 noting 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.

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.
3.4 Translating and Setting Page Table Entries

PTE instantiation

This set of functions and macros deal with the mapping of addresses and pages to PTEs and the setting of the individual entries.

The macro `mk_pte()` takes a `struct page` and protection bits and combines them together to form the `pte_t` that needs to be inserted into the page table. A similar macro `mk_pte_phys()` exists which takes a physical page address as a parameter.

The macro `pte_page()` returns the `struct page` which corresponds to the PTE entry. `pmd_page()` returns the `struct page` containing the set of PTEs.

The macro `set_pte()` takes a `pte_t` such as that returned by `mk_pte()` and places it within the processes page tables. `pte_clear()` is the reverse operation. An additional function is provided called `ptep_get_and_clear()` which clears an entry from the process page table and returns the `pte_t`. This is important when some modification needs to be made to either the PTE protection or the `struct page` itself.

3.5 Allocating and Freeing Page Tables

PTE allocation

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.

PTE freeing 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

Page table initialisation Initialising 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

#### Kernel image location

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 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.

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_HIGHMEM` 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 pmu_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 kmapings 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().

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

Address mapping to pages

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
3.7.1 Mapping Physical to Virtual Kernel Addresses

Physical to virtual address mapping

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 a PFN 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>:

```
#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)

**TLB 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 care 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 TLB API 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).

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().

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

#### Level 1 CPU CPU Cache Management

As Linux manages the CPU Cache in a very similar fashion to the TLB, this
3.9. Level 1 CPU Cache Management

Table 3.3: Translation Lookaside Buffer Flush API (cont)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void(flush_tlb_page)</code></td>
<td><code>struct vm_area_struct *vma, unsigned long addr</code> 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.</td>
</tr>
<tr>
<td><code>void(flush_tlb_pgtables)</code></td>
<td><code>struct mm_struct *mm, unsigned long start, unsigned long end</code> 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.</td>
</tr>
<tr>
<td><code>void(update_mmu_cache)</code></td>
<td><code>struct vm_area_struct *vma, unsigned long addr, pte_t pte</code> This API is only called after a page fault completes. It tells the architecture dependant code that a new translation now exists at <code>pte</code> for the virtual address <code>addr</code>. 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.</td>
</tr>
</tbody>
</table>

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.

CPU caches are organised into linesCPU cache line. 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 may 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 CPU cache hit and as few cache misses CPU cache miss 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 and 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>flush_cache_mm()</td>
<td>flush_cache_range()</td>
<td>flush_cache_page()</td>
</tr>
<tr>
<td>Change all page tables</td>
<td>Change page table range</td>
<td>Change single PTE</td>
</tr>
<tr>
<td>flush_tlb_mm()</td>
<td>flush_tlb_range()</td>
<td>flush_tlb_page()</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.
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 mmap() for example. This is to support architectures, usually microcontrollers, that have no MMU. Much of the work in this area was
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().

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

developed by the uCLinux Project (http://www.uclinux.org).

Reverse Mapping  Reverse mapping pages
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 linearaly
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 \textit{PTE chain} associated with every \texttt{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 \texttt{union pte} that is a field in \texttt{struct page}. This has union has two fields, a pointer to a \texttt{struct pte\_chain} called \texttt{chain} and a \texttt{pte\_addr\_t} called \texttt{direct}. The union is an optimisation whereby \texttt{direct} is used to save memory if there is only one PTE mapping the entry, otherwise a chain is used. The type \texttt{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 \texttt{pte\_t} for simplicity.

The \texttt{struct pte\_chain} is a little more complex. The struct itself is very simple but it is \textit{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 \texttt{struct pte\_chains} as it is this type of task the slab allocator is best at. Each \texttt{struct pte\_chain} can hold up to \texttt{NRPTE} pointers to PTE structures. Once that many PTEs have been filled, a \texttt{struct pte\_chain} is allocated and added to the chain.

The \texttt{struct pte\_chain} has two fields. The first is \texttt{unsigned long next\_and\_idx} which has two purposes. When \texttt{next\_and\_idx} is ANDeD with \texttt{NRPTE}, it returns the number of PTEs currently in this \texttt{struct pte\_chain} indicating where the next free slot is. When \texttt{next\_and\_idx} is ANDeD with the negation of \texttt{NRPTE} (i.e. \texttt{\sim NRPTE}), a pointer to the next \texttt{struct pte\_chain} in the chain is returned\(^1\). 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 \texttt{pte\_chain} with \texttt{pte\_chain\_alloc()}. This allocated chain is passed with the \texttt{struct page} and the PTE to \texttt{page\_add\_rmap()}. If the existing PTE chain associated with the page has slots available, it will be used and the \texttt{pte\_chain}\(^1\text{Told you it was compact}\)
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  Reverse mapping objects

The reverse mapping required for each page can have very expensive space re-
quirements. 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 re-
verse mapping for each page, all the VMAs which map a particular page would be
traversed and unmap 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\textsuperscript{2}. At the time of
writing, this feature has not been merged yet and was last seen in kernel 2.5.68-mm1

\textsuperscript{2}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"
but there is a strong incentive to have it 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_referred` 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_referred()` is implemented.

`page_referred` calls `page_referred_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_referred_obj_one()` is called with the VMA and the page as parameters. The function `page_referred_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`.

\(^3\)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

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 unmap 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 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 dif-
ferent. In particular, to find the PTE for a given address, the code now reads as (taken from mm/memory.c);

```c
ptep = pte_offset_map(pmd, address);
if (!ptep)
    goto out;
pte = *ptep;
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.

**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
3.10. What’s New In 2.6

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

Process address space 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.

Global 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

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 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

Kernel Address Space

8MiB (the amount of memory addressed by two PGDs) is reserved at 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 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 mem_map will be scattered throughout this region and where they are actually located is architecture dependent.
The region between `PAGE_OFFSET` and `VMALLOC_START - 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 `PAGE_OFFSET`. Between the physical memory map and the vmalloc address space, there is a gap of space `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, `VMALLOC_START` will be located at `PAGE_OFFSET + 0x02000000 + 0x00800000`.

In low memory systems, the remaining amount of the virtual address space, minus a 2 page gap, is used by `vmalloc()` for representing non-contiguous memory allocations in a contiguous virtual address space. In high-memory systems, the vmalloc area extends as far as `PKMAP_BASE` minus the two page gap and two extra regions are introduced. The first, which begins at `PKMAP_BASE`, is an area reserved 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 `FIXADDR_START` to `FIXADDR_TOP`. Fixed virtual addresses are needed for subsystems that need to know the virtual address at compile time such as the Advanced Programmable Interrupt Controller (APIC). `FIXADDR_TOP` is statically defined to be `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 `__FIXADDR_SIZE` and used to index back from `FIXADDR_TOP` to give the start of the region `FIXADDR_START`.

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

4.2 Managing the Address Space

Address space management
The address space usable by the process is managed by a high level `mm_struct` which is roughly analogous to the `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 `struct vm_area_struct` and are roughly analogous to the `vm_map_entry` struct in BSD. For clarity, a region may represent the process heap for use with `malloc()`, a memory mapped file such as a shared library or a block of anonymous memory allocated with `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 `vm_file` field will be set. By traversing `vm_file→f_dentry→d_inode→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.

### 4.3 Process Address Space Descriptor

**Process space descriptor**

The process address space is described by the `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 `task_struct`s which have pointers to the same `mm_struct`.  

**Thread identification**

A unique `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 the master page table. As a `mm_struct` is not needed for kernel threads, the `task_struct→mm` field for kernel threads is always NULL. For some tasks such as the boot idle task, the `mm_struct` is never setup but for kernel threads, a call to `daemonize()` will call `exit_mm()` to decrement the usage counter.

**TLB flushing**

As TLB flushes are extremely expensive, especially with archi-
4.3. Process Address Space Descriptor

tectures 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 `switch_mm()`, which results in a TLB flush, is avoided by “borrowing” the `mm_struct` used by the previous task and placing it in `task_struct→active_mm`. This technique has made large improvements to context switches times.

When entering lazy TLB, the function `enter_lazy_tlb()` is called to ensure that a `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 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
### System Call

<table>
<thead>
<tr>
<th>System Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>fork()</em></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><em>clone()</em></td>
<td>clone allows a new process to be created that shares parts of its context with its parent and is how threading is implemented in Linux. <em>clone()</em> without the CLONE_VM set will create a new address space which is essentially the same as <em>fork()</em></td>
</tr>
<tr>
<td><em>mmap()</em></td>
<td>mmap creates a new region within the process linear address space</td>
</tr>
<tr>
<td><em>mremap()</em></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><em>munmap()</em></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><em>shmat()</em></td>
<td>This attaches a shared memory segment to a process address space</td>
</tr>
<tr>
<td><em>shmdt()</em></td>
<td>Removes a shared memory segment from an address space</td>
</tr>
<tr>
<td><em>execve()</em></td>
<td>This loads a new executable file replacing the current address space</td>
</tr>
<tr>
<td><em>exit()</em></td>
<td>Destroys an address space and all regions</td>
</tr>
</tbody>
</table>

Table 4.1: System Calls Related to Memory Regions

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:
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;

**mmlist** All `mm_struct` 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;

category Architecture specific MMU context.

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

<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

4.3.1 Allocating a Descriptor

Process descriptor allocation

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.

4.3.2 Initialising a Descriptor

Initialising mm_struct

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) \n239 { \n240     mm_rb: RB_ROOT, \n241     pgd: swapper_pg_dir, \n242     mm_users: ATOMIC_INIT(2), \n243     mm_count: ATOMIC_INIT(1), \n244     mmap_sem: __RWSEM_INITIALIZER(name.mmap_sem), \n245     page_table_lock: SPIN_LOCK_UNLOCKED, \n```
Once it is established, new `mm_struct`s 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 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.

### 4.4 Memory Regions

**Address space regions Memory regions VMA Virtual Memory Area**

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>`:

```c
246 mmlist: LIST_HEAD_INIT(name.mmlist), \
247 }
```
4.4. Memory Regions

```c
struct vm_area_struct {
    struct mm_struct * vm_mm;
    unsigned long vm_start;
    unsigned long vm_end;

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

    pgprot_t vm_page_prot;
    unsigned long vm_flags;

    rb_node_t vm_rb;

    struct vm_area_struct *vm_next_share;
    struct vm_area_struct **vm_pprev_share;

    /* Function pointers to deal with this struct. */
    struct vm_operations_struct * vm_ops;

    /* Information about our backing store: */
    unsigned long vm_pgoff;
    struct file * vm_file;
    unsigned long vm_raend;
    void * vm_private_data;
};
```

`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;

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 functions pointers for open(), close() and nopage(). These are needed for syncing with information from the disk;

vm_pgo 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

VMA operations

There are three operations which a VMA may support called open(), close() and nopage(). 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),
```
### 4.4.1. Memory Region Operations

#### Protection Flags

<table>
<thead>
<tr>
<th>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>

#### mmap Related Flags

<table>
<thead>
<tr>
<th>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 <code>mmap()</code> translates to. Now unused</td>
</tr>
<tr>
<td>VM_EXECUTABLE</td>
<td>What MAP_EXECUTABLE for <code>mmap()</code> translates to. Now unused</td>
</tr>
<tr>
<td>VM_STACK_FLAGS</td>
<td>Flags used by <code>setup_arg_flags()</code> to setup the stack</td>
</tr>
</tbody>
</table>

#### Locking Flags

<table>
<thead>
<tr>
<th>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 <code>mlock()</code></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>

#### madvise() Flags

<table>
<thead>
<tr>
<th>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
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()`. The callback is responsible for locating the page in the page cache or allocating a page and populating it with the required data before returning it.

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 locating 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

**File backed 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;
};
```
4.4.2. File/Device backed memory regions

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 its own `address_space_operations` although they sometimes use generic functions;

**host** The host inode the file belongs to;

**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>`:
struct address_space_operations {
    int (*writepage)(struct page *);
    int (*readpage)(struct file *, struct page *);
    int (*sync_page)(struct page *);
    /* ext3 requires that a successful prepare_write() call be
     * followed by a commit_write() call - they must be balanced
     */
    int (*prepare_write)(struct file *, struct page *,
                         unsigned, unsigned);
    int (*commit_write)(struct file *, struct page *,
                         unsigned, unsigned);
    /* Unfortunately this kludge is needed for FIBMAP.
     * Don't use it */
    int (*bmap)(struct address_space *, long);
    int (*flushpage) (struct page *, unsigned long);
    int (*releasepage) (struct page *, int);
    #define KERNEL_HAS_O_DIRECT
    int (*direct_IO)(int, struct inode *, struct kiobuf *,
                     unsigned long, int);
    #define KERNEL_HAS_DIRECT_FILEIO
    int (*direct_fileIO)(int, struct file *, struct kiobuf *,
                         unsigned long, int);
    void (*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();

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();
4.4.3 Creating A Memory Region

**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_free_buffers().

**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

#### VMA Creation

Creating VMAs

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 \texttt{pgoфф} and finally
that the process does not have too many mapped regions already.

This rest of the function is large but broadly speaking it takes the following steps:

- Sanity check the parameters;
- Find a free linear address space large enough for the memory mapping. If
  a filesystem or device specific \texttt{get\_ unmapped\_ area()} function is provided, it
  will be used otherwise \texttt{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 that it is
  suitable for the new mapping;
- Allocate a \texttt{vm\_area\_struct} from the slab allocator and fill in its entries;
- Link in the new VMA;
- Call the filesystem or device specific \texttt{mmap} function;
- Update statistics and exit.

### 4.4.4 Finding a Mapped Memory Region

**VMA searching**

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
\texttt{find\_vma(\cdot)}. The function \texttt{find\_vma()} and other API functions affecting memory
regions are listed in Table 4.3.
4.4.5 Finding a Free Memory Region

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 desired VMA which is required as the list is a singly linked list. find_vma_prev() is rarely used but notably, it is used when two VMAs are being compared to determine if they may be merged. It is also used when removing a memory region so that the singly linked list may be updated.

The last function of note for searching VMAs is find_vma_intersection() which is used to find a VMA which overlaps a given address range. The most notable use of this is during a call to do_brk() when a region is growing up. It is important to ensure that the growing region will not overlap an old region.

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 has 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

VMA insertion

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 first calls `find_vma_prepare()` to find the appropriate VMAs the new region is to be inserted between and the correct nodes within the red-black tree. It then calls `__vma_link()` to do the work of linking in the new VMA.

The function `insert_vm_struct()` is rarely used as it does not increase the
4.4.7 Merging contiguous regions

VMA merging

Merging VMAs

Linux used to have a function called \texttt{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 \texttt{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 \texttt{merge_segments()}. The equivalent function which exists now is called \texttt{vma_merge()} and it is only used in two places. The first is user is \texttt{sys_mmap()} which calls it if an anonymous region is being mapped, as anonymous regions are frequently mergable. The second time is during \texttt{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 \texttt{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
4.4.8 Remapping and moving a memory region

VMA remapping

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.

Figure 4.7: Call Graph: `sys_mremap()`

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()`.

Figure 4.8: Call Graph: `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.
4.4.9 Locking a Memory Region

Literally one PTE at a time. Next \texttt{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, \texttt{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 call the device specific \texttt{do\_nopage()} function.

4.4.9 Locking a Memory Region

VMA locking

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
4.4.10 Unlocking the region

VMA unlocking

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()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.

Figure 4.11: Call Graph: do_munmap()

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 unmapped, remove_shared_vm_struct() is still called to remove the shared file mapping. Again, if this is a partial unmapping, it will be recreated during fixup.
4.4.13 Deleting all memory regions

zap_page_range() is called to remove all the pages associated with the region about to be unmapped before unmap_fixup() is called to handle partial un mappings.

Lastly free_pgt ables() is called to try and free up all the page table entries associated with the unmapped region. It is important to note that the page table entry freeing is not exhaustive. It will only unmap full PGD directories and their entries so for example, if only half a PGD was used for the mapping, no page table entries will be freed. This is because a finer grained freeing of page table entries would be too expensive to free up data structures that are both small and likely to be used again.

4.4.13 Deleting all memory regions

VMA deletion Deleting a VMA

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

Exception handling

A very important part of VM is how kernel address space exceptions that are not bugs are caught. 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 PAGE_OFFSET. The second is where the kernel uses copy_from_user() or copy_to_user() to read or write data from userspace.

At compile time, the linker creates an exception table in the __ex_table section of the kernel code segment which starts at __start__ex_table and ends at __stop__ex_table. Each entry is of type exception_table_entry which is a pair consisting of an execution point and a fix up routine. When an exception occurs that
the page fault handler cannot manage, it calls `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 module's 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.

## 4.6 Page Faulting

### Page faults

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 prefetching policy. Linux would likely benefit from a prefetching 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 Minor page faults, or soft page fault. Linux maintains statistics on the number of these types of page faults with the `task_struct→majflt` 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.
4.6.1 Handling a Page Fault

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 \texttt{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.

\texttt{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 \texttt{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 \texttt{handle_mm_fault()}, whose 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 \texttt{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...
4.6.1. Handling a Page Fault

not present or if it has been allocated with which is checked by \texttt{pte\_present()} and \texttt{pte\_none()}. If no PTE has been allocated (\texttt{pte\_none()} returned true), \texttt{do\_no\_page()} is called which handles Demand Allocation. Otherwise it is a page that has been swapped out to disk and \texttt{do\_swap\_page()} performs Demand Paging. There is a rare exception where swapped out pages belonging to a virtual file are handled by \texttt{do\_no\_page()}. This particular case is covered in Section ??.

The second option is if the page is being written to. If the PTE is write protected, then \texttt{do\_wp\_page()} is called as the page is a Copy-On-Write (COW) page. A COW page is one which is shared between multiple processes(usually a parent and child) until a write occurs after which a private copy is made for the writing process. A COW page is recognised because the VMA for the region is marked writable even though the individual PTE is not. If it is not a COW page, the page is simply
4.6.2 Demand Allocation

Figure 4.14: Call Graph: handle_mm_fault()

marked dirty as it has been written to.

The last option is if the page has been read and is present but a fault still occurred. This can occur with some architectures that do not have a three level page table. In this case, the PTE is simply established and marked young.

4.6.2 Demand Allocation

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  Anonymous pages

If vm_area_struct→vm_op 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 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
4.6.2. Demand Allocation

page is zeroed out in the function mem_init().

Figure 4.15: Call Graph: do_no_page()

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.

Handling file/device backed pages  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 ??). 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.
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

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 ???. 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 can not 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 can not 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

**Copy-On-Write (COW)**

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

**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 **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
byte strides otherwise the distinction between the two copy functions is not import-
tant.

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 *vsyscall 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 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 updated and contains a handle to the backing backing_dev_info to control readahead;

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 Molnar\(^1\) which allows the kernel to optionally have its own full 4GiB address space. The patches are available from [http://redhat.com/mingo/4g-patches/](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 range, almost the entire of low memory is consumed by `mem_map`. By giving the kernel its 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 pagetables 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.

\(^1\)See [http://lwn.net/Articles/39283/](http://lwn.net/Articles/39283/) for the first announcement of the patch.
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 whereas 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 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 require 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 processors 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 `VMFAULT_MINOR`, `VMFAULT_MAJOR`, `VMFAULT_SIGBUS` and `VMFAULT_OOM`.
4.8. What's New in 2.6

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct vm_area_struct *(find_vma)</td>
<td>Finds the VMA that covers a given address. If the region does not exist, it returns the VMA closest to the requested address.</td>
</tr>
<tr>
<td>struct vm_area_struct *(find_vma_prev)</td>
<td>Same as find_vma() except it also gives the VMA pointing to the returned VMA. It is not often used, with sys_mprotect() being the notable exception, as it is usually find_vma_prepare() that is required.</td>
</tr>
<tr>
<td>struct vm_area_struct *(find_vma_prepare)</td>
<td>Same as find_vma() except that it will also the preceeding VMA in the linked list as well as the red-black tree nodes needed to perform an insertion into the tree.</td>
</tr>
<tr>
<td>struct vm_area_struct *(find_vma_intersection)</td>
<td>Returns the VMA which intersects a given address range. Useful when checking if a linear address region is in use by any VMA.</td>
</tr>
<tr>
<td>int(vma_merge)</td>
<td>Attempts to expand the supplied VMA to cover a new address range. If the VMA can not be expanded forwards, the next VMA is checked to see if it may be expanded backwards to cover the address range instead. Regions may be merged if there is no file/device mapping and the permissions match.</td>
</tr>
<tr>
<td>unsigned long(get_unmapped_area)</td>
<td>Returns the address of a free region of memory large enough to cover the requested size of memory. Used principally when a new VMA is to be created.</td>
</tr>
<tr>
<td>void(insert_vm_struct)</td>
<td>Inserts a new VMA into a linear address space.</td>
</tr>
</tbody>
</table>

Table 4.3: Memory Region VMA API
### 4.8. What’s New in 2.6

<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 <strong>Stack Expansion</strong> Expanding the stack 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 <strong>SIGSEGV</strong> signal</td>
</tr>
<tr>
<td>Region is invalid or process has no permissions to access</td>
<td>Error</td>
<td>Send a <strong>SIGSEGV</strong> 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 <code>vmalloc</code> area of the address space, the current process page tables are updated against the master page table held by <code>init_mm</code>. 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
4.8. What’s New in 2.6

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>unsigned long(copy_from_user)</code></td>
<td>void *to, const void *from, unsigned long n  Copies n bytes from the user address(from) to the kernel address space(to)</td>
</tr>
<tr>
<td><code>unsigned long(copy_to_user)</code></td>
<td>void *to, const void *from, unsigned long n  Copies n bytes from the kernel address(from) to the user address space(to)</td>
</tr>
<tr>
<td><code>void(copy_user_page)</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>void(clear_user_page)</code></td>
<td>void *page, unsigned long address  Similar to <code>copy_user_page()</code> except it is for zeroing a page</td>
</tr>
<tr>
<td><code>void(get_user)</code></td>
<td>void *to, void *from   Copies an integer value from userspace (from) to kernel space (to)</td>
</tr>
<tr>
<td><code>void(put_user)</code></td>
<td>void *from, void *to   Copies an integer value from kernel space (from) to userspace (to)</td>
</tr>
<tr>
<td><code>long(strncpy_from_user)</code></td>
<td>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)</td>
</tr>
<tr>
<td><code>long(strlen_user)</code></td>
<td>const char *s, long n   Returns the length, upper bound by n, of the userspace string including the terminating NULL</td>
</tr>
<tr>
<td><code>int(access_ok)</code></td>
<td>int type, unsigned long addr, unsigned long size  Returns non-zero if the userspace block of memory is valid and zero otherwise</td>
</tr>
</tbody>
</table>

Table 4.5: Accessing Process Address Space API
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 [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 [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 [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 initialised 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

Boot map representation

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
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;
```

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 the page of the end of the last allocation.
  If 0, the page used is full;
- **last_pos** This is the thePFN 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

Initialising the boot memory allocator Boot allocator initialisation

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:

$$\text{mapsize} = \frac{(\text{end_pfn} - \text{start_pfn}) + 7}{8}$$
The bitmap is stored at the physical address pointed to by 
\texttt{bootmem_data_t→node_boot_start} and the virtual address to the map is placed in 
\texttt{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 \texttt{register_bootmem_low_pages()} 
reads through the e820 map and calls \texttt{free_bootmem()} for each usable page to set 
the bit to 0 before using \texttt{reserve_bootmem()} to reserve the pages needed by the 
actual bitmap.

5.3 Allocating Memory

The \texttt{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 func-
tions provided for easy allocations on UMA architectures called \texttt{alloc_bootmem()}, \texttt{alloc_bootmem_low()}, \texttt{alloc_bootmem_pages()} and \texttt{alloc_bootmem_low_pages()} 
which are fully described in Table 5.1. All of these macros call \texttt{__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()](image)

Similar functions exist for NUMA which take the node as an additional 
parameter, as listed in Table 5.2. They are called \texttt{alloc_bootmem_node()}, \texttt{alloc_bootmem_pages_node()} and \texttt{alloc_bootmem_low_pages_node()}. All of 
these macros call \texttt{__alloc_bootmem_node()} with different parameters.

The parameters to either \texttt{__alloc_bootmem()} and \texttt{__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 where as 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 function 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, BUG() is called to show a double-free occured. 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

Finalising memory bootstrapping Bootstrap finalisation Retiring boot memory

Late in the bootstrapping process, the function 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 mem_init() that is responsible for destroying the boot memory allocator and its associated structures.

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 free_pages_init().

This function first tells the boot memory allocator to retire itself by calling free_all_bootmem() for UMA architectures or free_all_bootmem_node() for NUMA. Both call the core function 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 `PG_reserved` flag in its struct page;
- Set the count to 1;
- Call `__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 `free_all_bootmem()` returns, it first counts the number of reserved pages for accounting purposes. The remainder of the `free_pages_init()` function is responsible for the high memory pages. However, at this point, it should be clear how the global `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.

Once `free_all_bootmem()` returns, all the pages in `ZONE_NORMAL` have been given to the buddy allocator. To initialise the high memory pages, `free_pages_init()` calls `one_highpage_init()` for every page between `highstart_pfn` and `highend_pfn`. `one_highpage_init()` simply clears the `PG_reserved` flag, sets the `PG_highmem`
flag, sets the count to 1 and calls \texttt{\_free\_pages()} to release it to the buddy allocator in the same manner \texttt{free\_all\_bootmem\_core()} did.

**Initialising buddy allocator** 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 \texttt{\_init} such as the following:

321 unsigned long \_init free\_all\_bootmem (void)

All of these functions are placed together in the \texttt{.init} section by the linker. On the x86, the function \texttt{free\_initmem()} walks through all pages from \texttt{\_init\_begin} to \texttt{\_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.
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
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)
pdg_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
Chapter 6

Physical Page Allocation

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 Knollton [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] Buddy coalescing 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 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

Free lists

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 Order allocation. An array of free_area_t structs are
6.1. Managing Free Blocks

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 $2^{\text{MAX}_\text{ORDER} - 1}$ number of pages, where the \text{MAX}_\text{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 \texttt{page->list}.

Each zone has a \texttt{free_area_t} struct array called \texttt{free_area[MAX_ORDER]}. It is declared in \texttt{<linux/mm.h>} as follows:

22 typedef struct free_area_struct {
23 struct list_head free_list;
24 unsigned long *map;
25 } free_area_t;

The fields in this struct are simply:

\texttt{free_list} A linked list of free page blocks;

\texttt{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 \texttt{MARK_USED()} in \texttt{page_alloc.c} is used which is declared as follows:

164 #define MARK_USED(index, order, area) \
165 __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 $1+\text{order}$ 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 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.

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

Table 6.1: Physical Pages Allocation API

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.

![Figure 6.2: Allocating physical pages](Image)

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: **Node-Local Allocation** 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 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.
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>__free_pages()</td>
<td>Free an order number of pages from the given page.</td>
</tr>
<tr>
<td>__free_page()</td>
<td>Free a single page.</td>
</tr>
<tr>
<td>free_page()</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 \texttt{__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 \textbf{Buddy coalescing} 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 \texttt{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 [Knu68]. As the allocations are always in blocks of size $2^k$, the address of the block, or at least its offset within \texttt{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 from 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

**GFP flags Get Free Pages (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: *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>__GFP_DMA</td>
<td>Allocate from <em>ZONE_DMA</em> if possible</td>
</tr>
<tr>
<td>__GFP_HIGHMEM</td>
<td>Allocate from <em>ZONE_HIGHMEM</em> if possible</td>
</tr>
<tr>
<td>GFP_DMA</td>
<td>Alias for __GFP_DMA</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.

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
6.4.1 Process Flags

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.
6.5 Avoiding Fragmentation

Fragmentation Page Fragmentation

One important problem that must be addressed with any allocator is the problem of internal Internal fragmentation and external External fragmentation 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
### 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 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

---

<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 Out Of Memory (OOM) killer which is discussed in Chapter 12. 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 PF_MEMALLOC 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

---

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.
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 Lazy buddy 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 method 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 \texttt{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 function have been introduced for the freeing of pages called `free_hot_page()` and `free_cold_page()`. Predictably, the determine 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 flag's 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, fragmentation elimination (external) 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

vmalloc 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.

![vmalloc address space](image)

Figure 7.1: vmalloc Address Space

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 `ioremaping`), this function will be called directly to map the requested area.
7.2 Allocating A Non-Contiguous Area

The functions \texttt{vmalloc()}, \texttt{vmalloc_dma()} and \texttt{vmalloc_32()} are provided to allocate a memory area that is contiguous in virtual address space. They all take a single parameter \texttt{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 \texttt{get_vm_area()} is to find a region large enough to store the request. It searches through a linear linked list of \texttt{vm_struct}s and returns a new struct describing the allocated region.

The second step is to allocate the necessary PGD entries with \texttt{vmalloc_area_pages()}, PMD entries with \texttt{alloc_area_pmd()} and PTE entries with \texttt{alloc_area_pte()} before finally allocating the page with \texttt{alloc_page()}.

The page table updated by \texttt{vmalloc()} is not the current process but the reference page table stored at \texttt{init_mm}\rightarrow\texttt{pgd}. This means that a process accessing the \texttt{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
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.

7.4 Whats New in 2.6

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()`
Figure 7.3: Relationship between vmalloc(), alloc_page() and Page Faulting

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().
Figure 7.4: Call Graph: vfree()
Chapter 8

Slab Allocator

Slab allocator Object allocation

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 fragmen-
Eliminating fragmentation (internal) Small allocation caches 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
8.1. Caches

Linux does not attempt to color page allocations based on their physical address [Kes91]. Page colour, 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 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 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.

8.1 Caches

Caches (slab allocator)

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;

slabinfo - version: 1.1 (SMP)
kmem_cache 80 80 248 5 5 1 : 252 126
urb_priv 0 0 64 0 0 1 : 252 126
tcp_bind_bucket 15 226 32 2 2 1 : 252 126
inode_cache 5714 5992 512 856 856 1 : 124 62
dentry_cache 5160 5160 128 172 172 1 : 252 126
mm_struct 240 240 160 10 10 1 : 252 126
vm_area_struct 3911 4480 96 112 112 1 : 252 126
size-64(DMA) 0 0 64 0 0 1 : 252 126
size-64 432 1357 64 23 23 1 : 252 126
size-32(DMA) 17 113 32 1 1 1 : 252 126
size-32 850 2712 32 24 24 1 : 252 126

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.

```c
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;
};
```
8.1.1. Cache Descriptor

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 $2^{\text{gfporder}}$ pages as these are the allocation sizes the buddy allocator provides;
8.1.1. Cache Descriptor

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 caches 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.

227 char name[CACHE_NAMELEN];
228 struct list_head next;

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.

229 #ifdef CONFIG_SMP
231 cpucache_t *cpudata[NR_CPUS];
232 #endif

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

allocahit This is the total number of times an allocation has used the per-cpu cache;

allocmiss To complement allocahit, 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

Slab 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.

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.

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

Slab 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 the cache for reaping. When the function does find a cache with this flag set, it skips the cache and removes the flag.

8.1.4 Cache Allocation Flags

Slab 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.

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

**Slab 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 `BYTES_PER_WORD` unless `SLAB_HWCACHE_ALIGN` is set in which 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

- **colour**  This is the number of different offsets that can be used;
- **colour_off**  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

**Slab cache creation**

Cache creation (Slab)

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
8.1.7 Cache Reaping

- 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 object size to the hardware cache;
- 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.

Figure 8.3: Call Graph: `kmem_cache_create()`

8.1.7 Cache Reaping

Slab cache reap

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
8.1.7. Cache Reaping

under no memory pressure but this is not a problem for architectures like the x86 which has only one bank of memory.

Figure 8.4: Call Graph: kmem_cache_reap()

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;
- Count the number of free slabs in slabs_free and calculate how many pages that would free in the variable pages;
- If the cache has constructors or large slabs, adjust pages to make it less likely for the cache to be selected;
- If the number of pages that would be freed exceeds REAP_PERFECT, free half of the slabs in slabs_free;
- Otherwise scan the rest of the caches and select the one that would free the most pages for freeing half of its slabs in slabs_free.
8.1.8 Cache Shrinking

Slab 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 \texttt{slabs\_free} unless the growing flag gets set.

Linux is nothing, if not subtle.

---

Figure 8.5: Call Graph: \texttt{kmem\_cache\_shrink()}

Two varieties of shrink functions are provided with confusingly similar names. \texttt{kmem\_cache\_shrink()} removes all slabs from \texttt{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.

---

Figure 8.6: Call Graph: \texttt{\_\_kmem\_cache\_shrink()}

The second function \texttt{\_\_kmem\_cache\_shrink()} frees all slabs from \texttt{slabs\_free} and then verifies that \texttt{slabs\_partial} and \texttt{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.
8.1.9 Cache Destroying

Slab cache deletion

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`.

Figure 8.7: Call Graph: `kmem_cache_destroy()`

8.2 Slabs

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.

Figure 8.8: Page to Cache and Slab Relationship

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 are placed are determined by the size of the object during cache creation. It is important to note that in 8.8, the struct slab_t could be stored at the beginning
of the page frame although the figure implies the `struct slab_` is separate from the page frame.

### 8.2.1 Storing the Slab Descriptor

**Slab descriptors**

If the objects are larger than a threshold (512 bytes on x86), `CFG_S_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.

![Figure 8.9: Slab With Descriptor On-Slab](image)

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 `kmem_bufctl_t` types. 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 \texttt{slab\_full}, \texttt{slab\_partial} and \texttt{slabs\_free}. New slabs are allocated to a cache by calling the function \texttt{kmem\_cache\_grow()}. This is frequently called “cache growing” and occurs when no objects are left in the \texttt{slabs\_partial} list and there are no slabs in \texttt{slabs\_free}. The tasks it fulfills are

\begin{itemize}
  \item Perform basic sanity checks to guard against bad usage;
  \item Calculate colour offset for objects in this slab;
  \item Allocate memory for slab and acquire a slab descriptor;
\end{itemize}
8.2.3 Tracking Free Objects

- 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

Slab free object tracking

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], 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 struct page and 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.

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 slab_bufctl() is provided.

#define slab_bufctl(slabp) \ 
((kmem_bufctl_t *)(((slab_t*)slabp)+1))

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

**Initialising `kmem_bufctl_t`**

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.

![Initialised `kmem_bufctl_t` Array](image)

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

**Slabs, finding free objects**

When allocating an object, `kmem_cache_alloc()` performs the "real" work of updating 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

```c
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.

The index of the next free object is stored at `kmem_bufctl_t[slabp->free]`. There is no pointer directly to the array hence the helper macro `slab_bufctl()`
8.2.6. Updating kmem_bufctl_t

is used. Note that the 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 kmem_bufctl_t array are not pointed to by any other element.

8.2.6 Updating kmem_bufctl_t

The kmem_bufctl_t list is only updated when an object is freed in the function kmem_cache_free_one(). The array is updated with this block of code:

```
1451 unsigned int objnr = (objp-slabp->s_mem)/cachep->objsize;
1452
1453 slab_bufctl(slabp)[objnr] = slabp->free;
1454 slabp->free = objnr;
```

The pointer objp is the object about to be freed and objnr is its index. kmem_bufctl_t[objnr] is updated to point to the current value of slabp->free, effectively placing the object pointed to by free on the pseudo linked list. slabp->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

Slabs, number of objects

During cache creation, the function 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 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 wastage to be the total size of the slab i.e. PAGE_SIZE^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 kmem_bufctl_t if the slab descriptor is stored on the slab. Keep increasing the size of i until the slab is filled;
- Return the number of objects and bytes wasted.
8.2.8 Slab Destroying

Slab deletion

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.

---

Figure 8.13: Call Graph: `kmem_slab_destroy()`

8.3 Objects

Slab objects, see 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

Object initialisation

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

Object allocation Slab 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.

![Figure 8.14: Call Graph: kmem_cache_alloc()](image)

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 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

Object freeing Slab object allocation

`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.
8.4 Sizes Cache

Small allocation caches

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, typed to cache_sizes_t, which is defined in mm/slab.c as:

```c
331 typedef struct cache_sizes {
332    size_t cs_size;
333    kmem_cache_t *cs_cachep;
334    kmem_cache_t *cs_dmacachep;
335 } 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.
static cache_sizes_t cache_sizes[] = {
#if PAGE_SIZE == 4096
    { 32, NULL, NULL},
#else
    { 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()](image-url)
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

Per-CPU cache Slab caches, per-CPU

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
#define cc_data(cachep) ((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

```c
cc_entry(cc)[cc->avail++] = obj;
```

To remove an object

```c
obj = cc_entry(cc)[--cc->avail];
```

There is a helper macro called `cc_entry()` which gives a pointer to the first object in the cpucache. It is defined as

```c
#define cc_entry(cpucache) ((void **)(((cpucache_t*)(cpucache))+1))
```

This takes a pointer to a cpucache, 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 be 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

```c
typedef struct ccupdate_struct_s
{
    kmem_cache_t *cachep;
    cpucache_t *new[NR_CPUS];
} ccupdate_struct_t;
```

cachep is the cache being updated and new is the array of the cpucache descriptors for each CPU on the system. The function smp_function_all_cpus() is used to get each CPU to call the do_ccupdate_local() function which swaps the information from ccupdate_struct_t with the information in the cache descriptor.

Once the information has been swapped, the old data can be deleted.

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

Slab initialisation Initialising Slab Allocator

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
static kmem_cache_t cache_cache = {
    slabs_full: LIST_HEAD_INIT(cache_cache.slabs_full),
    slabs_partial: LIST_HEAD_INIT(cache_cache.slabs_partial),
    slabs_free: LIST_HEAD_INIT(cache_cache.slabs_free),
    objsize: sizeof(kmem_cache_t),
    flags: SLAB_NO_REAP,
    spinlock: SPIN_LOCK_UNLOCKED,
    colour_off: L1_CACHE_BYTES,
    name: "kmem_cache",
};
```

This code statically initialised the kmem_cache_t struct as follows:

357-360 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;

363 Initialise the spinlock unlocked;

364 Align the objects to the L1 cache;

365 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

Slabs, buddy interaction

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 \texttt{kmem\_getpages()} and \texttt{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 \texttt{cachep->gfpflags} and the order is taken from \texttt{cachep->gfporder} where \texttt{cachep} is the cache requesting the pages. When freeing the pages, \texttt{PageClearSlab()} will be called for every page being freed before calling \texttt{free\_pages()}.

8.8 Whats New in 2.6

The first obvious change is that the version of the \texttt{/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 \texttt{#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, \texttt{kmem\_cache\_estimate()} is now called \texttt{cache\_estimate()} even though they are identical in every other respect.

Cache descriptor  The changes to the \texttt{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. \texttt{slabs\_full}) and statistics related to them have been moved to a separate \texttt{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. 
CFLGS_OPTIMIZE no longer exists but its usage in 2.4 was non-existent. Two new 
flags have been introduced which are:

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.

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 slab_reclaim_pages to record how many pages are used in slabs allocated to these caches. This counter is later used in vm_enough_memory() to help determine if the system is truly out of memory.

Cache Reaping This is one of the most interesting changes made to the slab 
allocator. 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 set_shrinker() 
for the intelligent aging and shrinking of slabs. This simple function populates a 
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 shrinker_list.

During page reclaim, the function shrink_slab() is called which steps through 
the full 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 reclaim_state. When the slab allocator frees pages, this field is updated with the number of pages that is freed. Before calling shrink_slab(), this field is set to 0 and then read again after 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 `struct slab` instead of `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.
8.8. What's New in 2.6

```c
kmem_cache_t
*(kmem_cache_create)
const char *
name,
size_t size,
size_t offset,
unsigned long flags,
void (*ctor)(void*,
  kmem_cache_t *
  , unsigned long),
void (*dtor)(void*,
  kmem_cache_t *
  , unsigned long)
Creates a new cache and adds it to the cache chain

int(kmem_cache_reap)
int gfp_mask
Scans at most REAP_SCANLEN caches and selects one for reaping all per-cpu objects and free slabs from. Called when memory is tight

int(kmem_cache_shrink)
kmem_cache_t *
cachep 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.

void *
*(kmem_cache_alloc)
kmem_cache_t *
cachep, int flags
Allocate a single object from the cache and return it to the caller

void(kmem_cache_free)
```
### 8.8. What's New in 2.6

#### Table 8.2: Internal cache static flags

<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 8.3: Cache static flags set by caller

<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 8.4: Cache static debug flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLAB_DEBUG_FREE</td>
<td>Perform expensive checks on free</td>
</tr>
<tr>
<td>SLAB_DEBUG_INITIAL</td>
<td>On free, call the constructor as a verifier to ensure the object is still initialised correctly</td>
</tr>
<tr>
<td>SLAB_RED_ZONE</td>
<td>This places a marker at either end of objects to trap overflows</td>
</tr>
<tr>
<td>SLAB_POISON</td>
<td>Poison objects with a known pattern for trapping changes made to objects not allocated or initialised</td>
</tr>
</tbody>
</table>

#### Table 8.5: Cache Allocation Flags

<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 8.6: Cache Constructor Flags

<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>
Chapter 9

High Memory Management

High Memory

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

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

High memory mapping
The API for mapping pages from high memory is described in Table 9.1. The main function for mapping a page is \texttt{kmap()}. For users that do not wish to block, \texttt{kmap\_nonblock()} is available and interrupt users have \texttt{kmap\_atomic()}. The kmap pool is quite small so it is important that users of \texttt{kmap()} call \texttt{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.

![Figure 9.1: Call Graph: kmap()](image)

The \texttt{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 \texttt{out\_of\_line\_bug()} if true. An interrupt handler calling \texttt{BUG()} would panic the system so \texttt{out\_of\_line\_bug()} prints out bug information and exits cleanly. The second check is that the page is below \texttt{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 \texttt{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, \texttt{kmap\_high()} is called to begin the real work.

The \texttt{kmap\_high()} function begins with checking the \texttt{page\_virtual} field which is set if the page is already mapped. If it is \texttt{NULL}, \texttt{map\_new\_virtual()} provides a mapping for the page.
Creating a new virtual mapping with map_new_virtual() is a simple case of linearly scanning pkmap_count. The scan starts at last_pkmap_nr instead of 0 to prevent searching over the same areas repeatedly between kmap()s. When last_pkmap_nr wraps around to 0, 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 pkmap_map_wait wait queue until it is woken up after the next kunmap().

Once a mapping has been created, the corresponding entry in the pkmap_count array is incremented and the virtual address in low memory returned.

```c
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 9.1: High Memory Mapping API

9.2.1 Unmapping Pages

The API for unmapping pages from high memory is described in Table 9.2. The kunmap() function, like its complement, performs two checks. The first is an identical check to kmap() for usage from interrupt context. The second is that the page is below 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, kunmap_high() is called to perform the unmapping.
Figure 9.2: Call Graph: 

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.

```c
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 9.2: High Memory Unmapping API

---

9.3 Mapping High Memory Pages Atomically

High memory atomic mappings

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 \texttt{KM\_TYPE\_NR} entries per processor are reserved at boot time for atomic mapping at the location \texttt{FIX\_KMAP\_BEGIN} and ending at \texttt{FIX\_KMAP\_END}. Obviously a user of an atomic kmap may not sleep or exit before calling \texttt{kunmap\_atomic()} as the next process on the processor may try to use the same entry and fail.

The function \texttt{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 \texttt{kunmap\_atomic()} is interesting as it will only clear the PTE with \texttt{pte\_clear()} if debugging is enabled. It is considered unnecessary to bother unmapping atomic pages as the next call to \texttt{kmap\_atomic()} will simply replace it making TLB flushes unnecessary.

## 9.4 Bounce Buffers

### Bounce buffers High Memory IO

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 \texttt{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 \texttt{buffer\_head\rightarrow 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 \texttt{bounce\_end\_io\_write()}. 
9.4.2 Creating Bounce Buffers

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 \texttt{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 (\texttt{rw}) and the template buffer head to use (\texttt{bh\_orig}).

![Figure 9.3: Call Graph: \texttt{create\_bounce()}

A page is allocated for the buffer itself with the function \texttt{alloc\_bounce\_page()} which is a wrapper around \texttt{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 \texttt{alloc\_bounce\_bh()} which, similar in principle to \texttt{alloc\_bounce\_page()}, calls the slab allocator for a \texttt{buffer\_head} and uses the emergency pool if one cannot be allocated. Additionally, \texttt{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 \texttt{buffer\_head} have been allocated, information is copied from the template \texttt{buffer\_head} into the new one. Since part of this operation may use \texttt{kmap\_atomic()}, bounce buffers are only created with the IRQ safe \texttt{io\_request\_lock} held. The IO completion callbacks are changed to be either \texttt{bounce\_end\_io\_write()} or \texttt{bounce\_end\_io\_read()} depending on whether this is a read or write buffer so the data will be copied to and from high memory.

The most important aspect of the allocations to note is that the GFP flags specify
9.4.3 Copying via bounce buffers

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-url)

### 9.6 What’s New in 2.6

**Memory Pools** 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 \(\text{min\_nr}\), an allocator function for the object type \(\text{alloc\_fn()}\), a free function \(\text{free\_fn()}\) 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 \text{mempool\_alloc\_slab()} and \text{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 \text{ZONE\_DMA}. The allocate function is \text{page\_pool\_alloc()} and the private data parameter passed indicates the GFP flags to use. The free function is \text{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 \text{mempool\_alloc()} and \text{mempool\_free()} are provided. Memory pools are destroyed with \text{mempool\_destroy()}.  

Mapping High Memory Pages In 2.4, the field \text{page\_virtual} was used to store the address of the page within the \text{pkmap\_count} array. Due to the number of \text{struct page}s 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 \text{ZONE\_NORMAL}. 2.6 still has this \text{pkmap\_count} array but it is managed very differently.

In 2.6, a hash table called \text{page\_address\_htable} is created. This table is hashed based on the address of the \text{struct page} and the list is used to locate \text{struct page\_address\_slot}. This struct has two fields of interest, a \text{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 \text{page\_virtual} is no longer required.

Performing IO The last major change is that the \text{struct bio} is now used instead of the \text{struct buffer\_head} when performing IO. How \text{bio} structures work is beyond the scope of this book. However, the principle reason that \text{bio} structures were introduced is so that IO could be performed in blocks of whatever size the un-
derlying 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

Page reclaimation

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 \texttt{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 \textit{Swap cache}.

Finally, this chapter will cover the page replacement daemon \texttt{kswapd}, how it is implemented and what it's responsibilities are.

## 10.1 Page Replacement Policy

\textbf{Page replacement policy}

During discussions the page replacement policy is frequently said to be a \textit{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 \textit{active\_list} and \textit{inactive\_list}. The objective is for the \textit{active\_list} to contain the \textit{working set} \cite{Den70} of all processes and the \textit{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 \cite{JS94} where two lists called \textit{Am} and \textit{A1} are maintained. With LRU 2Q, pages when first allocated are placed on a FIFO queue called \textit{A1}. If they are referenced while on that queue, they are placed in a normal LRU managed list called \textit{Am}. This is roughly analogous to using \texttt{lru\_cache\_add()} to place pages on a queue called \textit{inactive\_list} (\textit{A1}) and using \texttt{mark\_page\_accessed()} to get moved to the \textit{active\_list} (\textit{Am}). The algorithm describes how the size of the two lists have to be tuned but Linux takes a simpler approach by using \texttt{refill\_inactive()} to move pages from the bottom of \textit{active\_list} to \textit{inactive\_list} to keep \textit{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 \texttt{refill\_inactive()}.

The lists described for 2Q presumes \textit{Am} is an LRU list but the \textit{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 reclaim-
mation are principally anonymous pages. In this case, Linux will keep examining
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 “laundrying” 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

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;

- 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

**Page 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`:

```c
atomic_t page_cache_size = ATOMIC_INIT(0);
unsigned int page_hash_bits;
struct page **page_hash_table;
```

**Initialising 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 \(2^{order} > htable\_size\). This may be roughly expressed as the integer component of the following simple equation.

\[
order = \log_2((htable\_size \times 2) - 1))
\]

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:

\[
page\_hash\_bits = \log_2 \left( \frac{\text{PAGE\_SIZE} \times 2^{order}}{\text{sizeof(struct page *)}} \right)
\]

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

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 with the inode; *dirty_pages* which have been written to since the list sync to disk; and *locked_pages* which are those currently locked. These three lists in combination are considered to be the inode queue for a given mapping and the *page→list* field is used to link pages on it. Pages are added to the inode queue with *add_page_to_inode_queue()* which places pages on the *clean_pages* lists and removed with *remove_page_from_inode_queue()*.

10.2.3 Adding Pages to the Page Cache

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()* function. The shared memory filesystem, which is covered in Chapter ??, 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()* which 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

\(^1\)Direct IO is handled differently with *generic_file_direct_IO()*.
Figure 10.2: Call Graph: `generic_file_read()`

to swap them out, they have no `address_space` acting as a mapping or any offset 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()`

10.3 LRU Lists

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

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

\[ \text{pages} = \text{nr\_pages} \times \frac{\text{nr\_active\_pages}}{2 \times (\text{nr\_inactive\_pages} + 1)} \]

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 LRU rotation 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

LRU list page reclaim

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 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 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 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 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 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 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:

\text{Page is mapped by a process. This jumps to the page_mapped label which we will meet again in a later case. The max_mapped count is decremented. If it reaches 0, the page tables of processes will be linearly searched and swapped out by the function swap_out()}

\text{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 com-}
Reclaiming Pages from the LRU Lists

A reference to the page is taken with `page_cache_get()` so that the page will not be freed prematurely and `wait_on_page()` is called which sleeps until the IO is complete. Once it is completed, the reference count is decremented with `page_cache_release()`. When the count reaches zero, the page will be reclaimed.

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 `writepage()` function available via `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 `page_cache_get()` before calling the `writepage()` function to synchronise the page with the backing file before dropping the reference with `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 `swapper_space` as a `page->mapping`. The page remains on the LRU. When it is found again, it will be simply freed if the IO has completed and the page will be reclaimed. If the IO has not completed, the kernel will wait for the IO to complete as described in the previous case.

Page has buffers associated with data on disk. A reference is taken to the page and an attempt is made to free the pages with `try_to_release_page()`. If it succeeds and is an anonymous page (no `page->mapping`, the page is removed from the LRU and `page_cache_released()` called to decrement the usage count. There is only one case where an anonymous page has associated buffers and that is when it is backed by a swap file as the page needs to be written out in block-sized chunk. If, on the other hand, it is backed by a file or device, the reference is simply dropped and the page will be freed as usual when the count reaches 0.

Page is anonymous and is mapped by more than one process. The LRU is unlocked and the page is unlocked before dropping into the same `page_mapped` label that was encountered in the first case. In other words, the `max_mapped` count is decremented and `swap_out` called when, or if, it reaches 0.

Page has no process referencing it. This is the final case that is “fallen” into rather than explicitly checked for. If the page is in the swap cache, it is removed from it as the page is now synchronised with the backing storage and has no process referencing it. If it was part of a file, it is removed from the inode queue, deleted from the page cache and freed.
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.

Figure 10.4: Call Graph: `shrink_caches()`

10.5 Swapping Out Process Pages

Process pageoutPageout of 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.

![Call Graph: `swap_out()`](image)

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)

**Pageout Daemon**

*Paging out*

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;
- 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.

### 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 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` structures. 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 removing 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 perform most of its 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`. 
### void(add_to_page_cache)
```
struct page * page, struct address_space * mapping, unsigned long oset
```
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 oset,
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()`, effectively 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 |
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
Chapter 11
Swap Management

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 coping 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 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

\footnote{Not to mention the affluent reader.}
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 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

Swap area, describing

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 };
```
11.1. Describing the Swap Area

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()`;

**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
struct swap_list_t {
    int head;  /* head of priority-ordered swapfile list */
    int next;  /* swapfile to be used next */
};
```

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];
    }
};
```
11.1. Describing the Swap Area

```c
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;

**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

Swap entries PTE to Swap Entry Mapping

When a page is swapped 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>`

\[
\begin{align*}
16 \textbf{typedef struct} & \{ \\
17 \quad \text{unsigned long val;} & \\
18 \} \textbf{swp\_entry\_t};
\end{align*}
\]

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 implemented 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.

To encode a type and offset into a `swp_entry_t`, the macro `SWP\_ENTRY()` is available which simply performs the relevant bit shifting operations. The relationship between all these macros is illustrated in Figure 11.1.

It should be noted that the six bits for “type” should allow up to 64 swap areas to exist in a 32 bit architecture instead of the `MAX\_SWAPFILES` restriction.
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 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

---

A Sun E450 could have in the region of 20 disks in it for example.
11.3. Allocating a swap slot

count does not overflow. If the entry is \texttt{SWAP\_MAP\_BAD}, the slot is unusable.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{call_graph}
\caption{Call Graph: \texttt{get\_swap\_page()}}
\end{figure}

The task of finding and allocating a swap entry is divided into two major tasks. The first performed by the high level function \texttt{get\_swap\_page()}. Starting with \texttt{swap\_list}→\texttt{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 \texttt{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 \textit{clusters} on disk of size \texttt{SWAPFILE\_CLUSTER}. It allocates \texttt{SWAPFILE\_CLUSTER} number of pages sequentially in swap keeping count of the number of sequentially allocated pages in \texttt{swap\_info\_struct}→\texttt{cluster\_nr} and records the current offset in \texttt{swap\_info\_struct}→\texttt{cluster\_next}. Once a sequential block has been allocated, it searches for a block of free entries of size \texttt{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 \texttt{swap\_info\_struct}→\texttt{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

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 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
39 struct address_space swapper_space = {
40   LIST_HEAD_INIT(swapper_space.clean_pages),
41   LIST_HEAD_INIT(swapper_space.dirty_pages),
42   LIST_HEAD_INIT(swapper_space.locked_pages),
43   0,
44   &swap_aops,
45  };
```

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
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](image)

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, 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

Swap reading

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.

![Figure 11.5: Call Graph: read_swap_cache_async()](image)

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

Swap writing

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.

![Call Graph: sys_writepage()](image)

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.

11.7 Reading/Writing Swap Area Blocks

Swap reading

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
11.8. Activating a Swap Area

Initialising swap areas

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

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 up to date flag with `ClearPageUptodate()` as the page is obviously not up to data 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 inode 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
11.9 Deactivating a Swap Area

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 `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.

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

Deactivating swap area Swap area deactivating

\[3\text{See the Code Commentary for the comprehensive reason for this.}\]
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 swapon 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

Out Of Memory Management

OOM 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.

12.1 Checking Available Memory

OOM Prevention

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 mount 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
12.2 Determining OOM Status

**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.

**Total pages used by the inode cache** as they are easily reclaimed.

If the total number of pages added here is sufficient for the request, `vm_enough_memory()` returns true to the caller. If false is returned, the caller knows that the memory is not available and usually decides to return `-ENOMEM` to userspace.

## 12.2 Determining OOM Status

**OOM Detection**

When the machine is low on memory, old page frames will be reclaimed (see Chapter 10) but despite reclaiming pages it 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.
12.3 Selecting a Process

OOM Killing

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():

\[
\text{badness\_for\_task} = \frac{\text{total\_vm\_for\_task}}{\sqrt{\text{cpu\_time\_in\_seconds}}} \cdot \sqrt[4]{\frac{\text{cpu\_time\_in\_minutes}}{4}}
\]

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.
12.4 Killing the Selected Process

Process killing

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.

12.5 Is That It?

Yes, that's it, out of memory management touches a lot of subsystems otherwise, there is not much to it.

12.6 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 `TotalRam * (OverCommitRatio/100) + TotalSwapPage`, where `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 13
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”¹ and the lack of documentation on practical VMs is not just confined to Linux. Matt Dillon, one of the principal developers of the FreeBSD VM² and considered a “VM Guru” stated in an interview³ that documentation can be “hard

¹http://mail.nl.linux.org/linux-mm/2002-05/msg00035.html
²His past involvement with the Linux VM is evident from http://mail.nl.linux.org/linux-mm/2000-05/msg00419.html
³http://kernel.org.com/node.php?id=8
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 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 13.1 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!
Bibliography


32, 5 of *ACM SIGPLAN Notices*, pages 171–182, New York, June 15–18

[JS94] Theodore Johnson and Dennis Shasha. 2q: a low overhead high per-
formance buffer management replacement algorithm. In *Proceedings of
the Twentieth International Conference on Very Large Databases*, pages

problem: solved? In *Proceedings of the first international symposium on

[KB85] David G. Korn and Kiem-Phong Bo. In search of a better malloc. In
*Proceedings of the Summer 1985 USENIX Conference*, pages 489–506,
1985.

memories. Technical Report CS-TR-1991-1032, University of Wisconsin,

[KMC02] Scott Kaplan, Lyle McGeoch, and Megan Cole. Adaptive caching for de-
mand prepaging. In David Detlefs, editor, *ISMM’02 Proceedings of the
Third International Symposium on Memory Management*, ACM SIG-


[Lev00] Check Lever. *Linux Kernel Hash Table Behavior: Analysis and Improve-
ments.*

[McK96] Marshall Kirk McKusick. *The design and implementation of the 4.4BSD
operating system*. Addison-Wesley, 1996.


