Can you handle the pressure? Making Linux bulletproof under load

Martin J. Bligh, mbligh@mbligh.org
Badari Pulavarty, pbadari@us.ibm.com
Andy Whitcroft, apw@shadowen.org
Darren Hart, dvhltc@us.ibm.com
IBM Linux Technology Center

Abstract

Operating under memory pressure has been a persistent problem for Linux customers. Despite significant work done in the 2.6 kernel to improve its handling of memory, it is still easy to make the Linux kernel slow to a crawl or lock up completely under load.

One of the fundamental sources for memory pressure is the filesystem pagecache usage, along with the `buffer_head` entries that control them. Another problem area is inode and dentry cache entries in the slab cache. Linux struggles to keep either of these under control. Userspace processes provide another obvious source of memory usage, which are partially handled by the OOM killer subsystem, which has often been accused of making poor decisions on which process to kill.

This paper takes a closer look at various scenerios causing of memory pressure and the way VM handles it currently, what we have done to keep the system for falling apart. This paper also discusses the future work that needs to be done to improve further, which may require careful re-design of subsystems.

This paper will try to describe the basics of memory reclaim in a way that is comprensible. In order to achieve that, some minor details have been glossed over; for the full gore, see the code. The intent is to give an overview first to give the reader some hope of understanding basic concepts and precepts.

As with any complex system, it is critical to have a high-level broad overview of how the system works before attempting to change anything within. Hopefully this paper will provide that skeleton understanding, and allow the reader to proceed to the code details themselves. This paper covers Linux® 2.6.11.

1 What is memory pressure?

The Linux VM code tries to use up spare memory for cache, thus there is normally little free memory on a running system. The intent is to use memory as efficiently as possible, and that cache should be easily recoverable when needed. We try to keep only a small number of pages free for each zone—usually between two watermarks: `zone->pages_low` and `zone->pages_high`. In practice, the interactions between zones make it a little more complex, but that is the basic intent. When the

system needs a page and there are insufficient available the system will trigger reclaim, that is it will start the process of identifying and releasing currently in-use pages.

Memory reclaim falls into two basic types:

- per-zone general page reclaim: `shrink_zone()`
- slab reclaim: `shrink_slab()`

both of which are invoked from each of 2 places:

- kswapd—background reclaim daemon; tries to keep a small number of free pages available at all times.
- direct reclaim—processes freeing memory for their own use. Triggered when a process is unable to allocate memory and is willing to wait.

2 When do we try to free pages?

The normal steady state of a running system is for most pages to be in-use, with just the minimum of pages actually free. The aim is to maintain the maximum working set in memory whilst maintaining sufficient truly empty pages to ensure critical operations will not block. The only thing that will cause us to have to reclaim pages is if we need to allocate new ones. In the diagram below are the watermarks that trigger reclaim activities.

Caption: For highmem zones, `pages_min` is normally 512KB. For lowmem, it is about `4*sqrt(low_kb)`, but spread across all low zones in the system. For an ia32 machine with 1GB or more of memory, that works out at about 3.8MB.

The memory page allocator (`__alloc_pages`) iterates over all the allowable zones for a given allocation (the zonelist) and tries to find a zone with enough free memory to take from. If we are below `pages_low`, it will wake up kswapd to try to reclaim more. If kswapd is failing to keep up with demand, and we fall below `pages_min`, each allocating process can drop into direct reclaim via `try_to_free_pages` ... searching for memory itself.

3 What pages do we try to free?

The basic plan is to target the least useful pages on the system. In broad terms the least recently used pages (LRU). However, in practice it is rather more complex than this, as we want to apply some bias over in which pages we keep, and which we discard (e.g. keeping a balance of anonymous (application) memory vs filebacked (pagecache) memory).

Some pages (e.g. slabcache and other kernel control structures) are not kept on the LRU lists at all. Either they are not reclaimable, or require special handling before release (we will cover these separately below).

3.1 The LRU lists

We keep memory on two lists (per zone)—the active and inactive lists. The basic premise is that pages on the active list are in active use, and pages on the inactive lists are not. We monitor the hardware pagetables (on most architectures) to detect whether the the page is being actively referenced or not, and copy that information down into the struct page in the form of the `PG_referenced` flag.

When attempting to reclaim pages, we scan the LRU lists; pages that are found to be active will

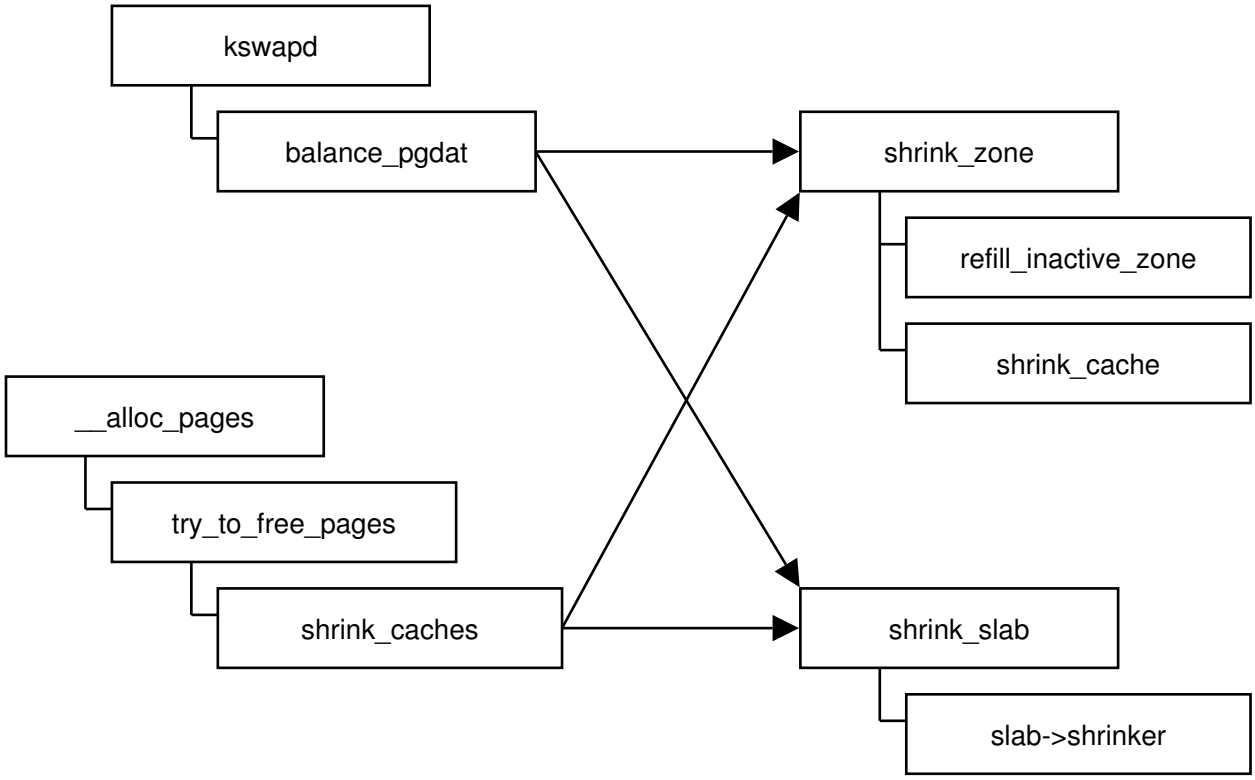

Figure 1: Memory Reclaimers

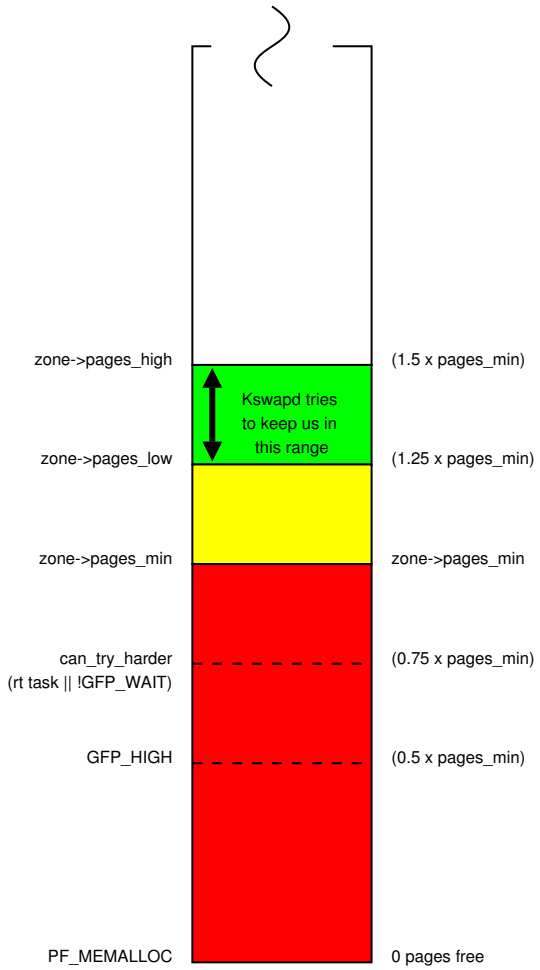

Figure 2: Zone Reclaim Watermarks

be moved to the head of the active list, pages that are found to be inactive will be demoted:

- If they were on the active list, they will be moved to the inactive list.
- If they were on the inactive list, we will try to discard them

3.2 Discarding pages

Reclaiming an in-use page from the system involves 5 basic steps:

- free the pagetable mappings (`try_to_unmap()`)
- clean the page if it is dirty (i.e. sync it to disk)
- release any `buffer_heads` associated with the page (explained in section below)
- Remove it from the pagecache
- free the page

3.3 Freeing the pagetable mappings

Freeing the pagetable mappings uses the rmap (reverse mapping) mechanism to go from a

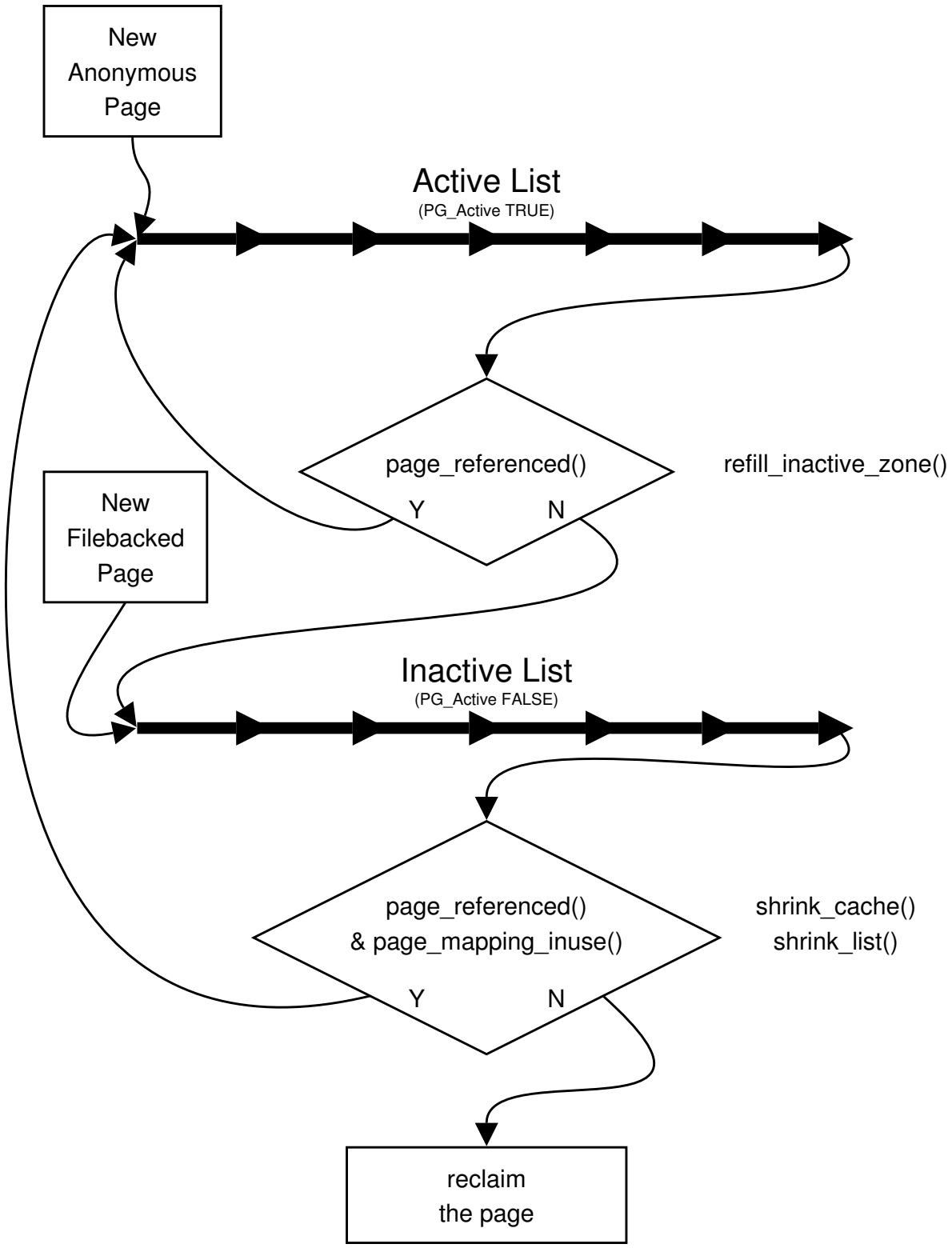

Figure 3: LRU lists

physical page to a list of the pagetable entries mapping it. The mechanism for how this works depends on whether the page is anonymous, or filebacked

- Anonymous page (`try_to_unmap_anon()`) use the `anon_vma` structure to retrieve the list of `vma`s mapping the page
- Filebacked page (`try_to_unmap_file()`) Go via the `address_space` structure (the file's controlling object) to retrieve a list of `vma`s mapping the page.

From the `vma`, combined with the offset information in the struct page, we can find the virtual address within the process, and walk the pagetables to the PTE entry.

4 Buffer heads

A `buffer_head` is a control structure for a page in the pagecache, but are not required for all pages. Their basic usage is to cache the disk mapping information for that pagecache page.

4.1 Why are bufferheads used?

- to provide support for filesystem blocksizes not matching system pagesize. If the filesystem blocksize is smaller than the system pagesize, each page may end up belonging to multiple physical blocks on the disk. Buffer heads provide a convenient way to map multiple blocks to a single page.
- To cache the page to disk block mapping information. All the pages belong to a file/inode are attached to that inode using the logical offset in the file and they are represented by a radix tree. This will significantly reduce the search/traversal times to map from a given file offset to the backing pagecache page. However, the filesystem can map these pages on the disk whatever way it wants. So every time, we need disk block mapping, we need to ask the filesystem to give us physical block number for the given page. Bufferheads provide a way to cache this information and there by eliminates an extra call to filesystem to figure out the disk block mapping. Note that figuring out the disk block mapping could involve reading the disk, depending on the filesystem.
- In order to provide ordering guarantees in case of a transaction commit. Ext3 ordered mode guarantees that the file data gets written to the disk before the metadata gets commited to the journal. In order to provide this guarantee, bufferheads are used as mechanism to link the pages belong to a transaction. If the transaction is getting commited to the journal, the `buffer_head` makes sure that all the pages attached to the transaction using the bufferhead are written to the disk.
- as meta data cache. All the meta data (superblock, directory, inode data, indirect blocks) are read into the buffer cache for a quick reference. Bufferheads provide a way to access the data.

4.2 What is the problem with bufferheads?

- Lowmem consumption: All bufferheads come from `buffer_head` slab cache (see later section on slab cache). Since all the slabs come from `ZONE_NORMAL`, they all consume lowmem (in the case of ia32). Since there is one or more `buffer_head` structures for each filesystem pagecache page, the `buffer_head` slab

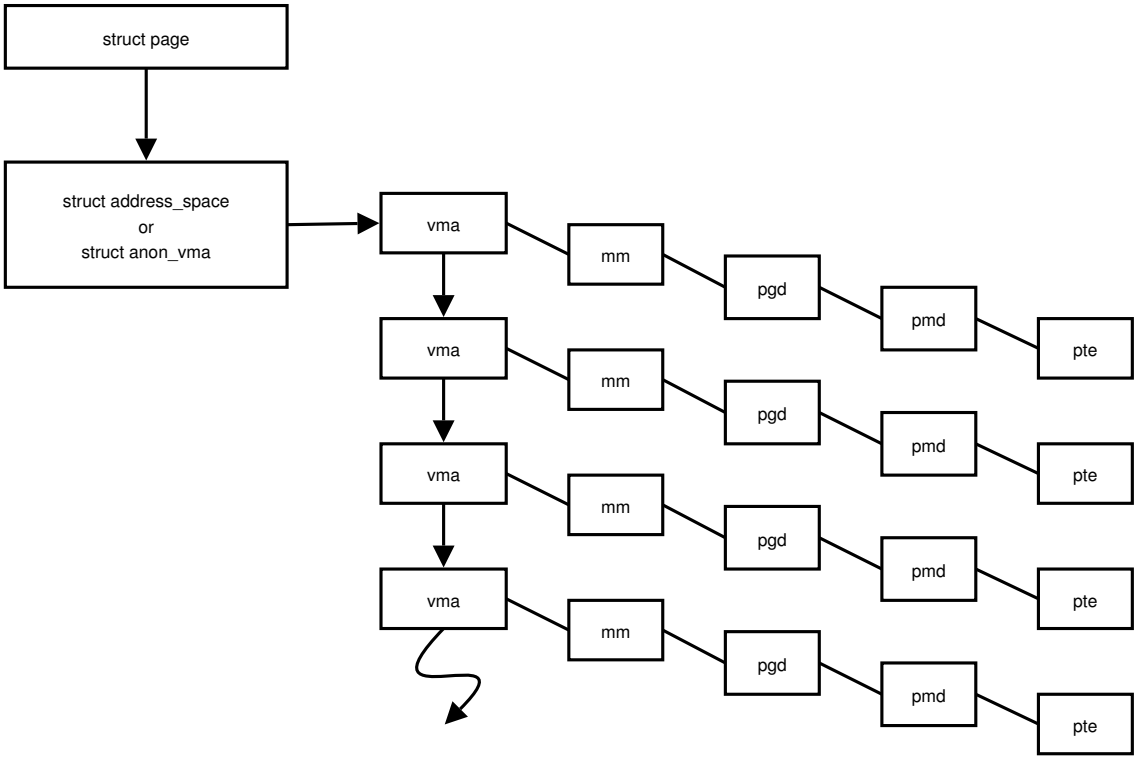

Figure 4: Object based rmap

grows very quickly and consumes lots of lowmem. In an attempt to address the problem, there is a limit on how much memory "bh" can occupy, which has been set to 10% of `ZONE_NORMAL`.

- page reference handling: When buffer-heads get attached to a page, they take a reference on the page, which is held until the VM tries to release the page. Typically, once the page gets flushed to disk it is acceptable to release the bufferhead. However, there is no clean way to drop the `buffer_head`, since the completion of the page being flushed is done in interrupt context. Thus we leave the buffer-heads around attached to the page and release them as and when VM decides to reuse the page. So, its normal to see lots of bufferheads floating around in the system. The `buffer_head` structures are allocated via `page_cache_get()`, and freed in `try_to_free_buffers()`.

- TLB/SLB/cache efficiency: Everytime we reference the `buffer_head`'s attached to page, it might cause a TLB/SLB miss. We have observed this problem with a large NFS workload, where ext3 `kjournald()` goes through all the transactions, all the journal heads and all the bufferheads looking for things to flush/clean. Eliminating bufferheads completely would be the best solution.

5 Non-reclaimable pages

Memory allocated to user processes are generally reclaimable. A user process can almost always be stopped and its memory image pushed out onto swap. Not all memory in the system can be so easily reclaimed: for example, pages allocated to the kernel text, pagetable pages, or those allocated to non-cooperative slab caches (as we will see later) may not be readily freed. Such memory is termed non-reclaimable—the ultimate owner of the allocation may not even be traceable.

5.1 Kernel Pages

By far the largest source of non-reclaimable allocations come from the kernel itself. The kernel text, and all pagetables are non-reclaimable. Any allocation where the owner is not easily determined will fall into this category. Often this occurs because the cost of maintaining the ownership information for each and every allocation would dominate the cost of those allocations.

5.2 Locked User Pages

The mlock system call provides a mechanism for a userspace process to request that a section of memory be held in RAM. mlock operates on the processes' reference to the page (e.g. the `vma` and pagetables), not the physical page controlling structure (e.g. the struct page). Thus the lock is indicated within the vma by the `VM_LOCKED` flag.

Whilst it would be useful to track this state within the struct page, this would require another reference count there, for something that is not often used. The `PG_locked` flag is sometimes confused with mlock functionality, but is not related to this at all; `PG_LOCKED` is held whilst the page is in use by the VM (e.g. whilst being written out).

5.3 Why are locked pages such an issue?

Locked pages in and of themselves are not a huge issue. There will always be information

which must remain in memory and cannot be allowed to be 'moved' out to secondary storage. It is when we are in need of higher order allocations (physically contiguous groups of pages) or are attempting to hotplug a specific area of physical memory that such 'unmovable' memory becomes an issue.

Taking a pathological example (on an ia32 system), we have a process allocating large areas of anonymous memory. For each 1024 4k pages we will need to allocate a page table page to map it, which is non-reclaimable. As allocations proceed we end up with a non-reclaimable page every 1025 pages, or one per `MAX_ORDER` allocation. As those unreclaimable pages are interspersed with the reclaimable pages, if we now need to free a large physically contiguous region we will find no fully reclaimable area.

6 Slab reclaim

The slab poses special problems. The slab is a typed memory allocator and as such takes system pages and carves them up for allocation. Each system page in the slab is potentially split into a number of separate allocations and owned by different parts of the operating system. In order to reclaim any slab page we have to first reclaim each and every one of the contained allocations.

In order to be reclaimable a slab must register a reclaim method—each slab can register a callback function to ask it to shrink itself, known as a "shrinker" routine. These are registered with `set_shrinker()` and unregistered with `remove_shrinker()`, held on `shrinker_list`, and called from `shrink_slab()`. Note that most slabs do NOT register a shrinker, and are thus non-reclaimable, the only ones that currently do are:

- directory entry cache (dentry)
- disk quota subsystem (dquot)
- inode cache (icache)
- filesystem meta information block cache (mbcache)

6.1 The Blunderbuss Effect

Taking the dentry cache as an example, we walk an LRU-type list of dentries—but note this is *entries*, not of *pages*. The problem with this is that whilst it will get rid of the best dcache entries it may not get rid of any whole pages at all. Imagine the following situation, for example:

Each row represents a page of dentrys, each box represents in individual dentry. Whilst many entries have been freed, no pages are reclaimable as a result—I call this the blunderbuss effect. We are suffering from internal fragmentation, which is made worse by the fact that some of the dentries (e.g. directories) may be locked. We actually have a fairly high likelihood of blowing away a very significant portion of the cache before freeing any pages at all. So whilst the shrink routine is good for keeping dcache size in check, it is not effective at shrinking it.

Dentry holds a reference to the inode as well. When we decrement the reference count to the dentry, the inode entry count is decremented as well. If the inode refcount is decremented to 0, we will call `truncate_inode_pages()` which will write back the pages for that file to disk. That could take a *very* long time to complete. This means that slab reclaim can cause very high latencies in order to allocate a page.

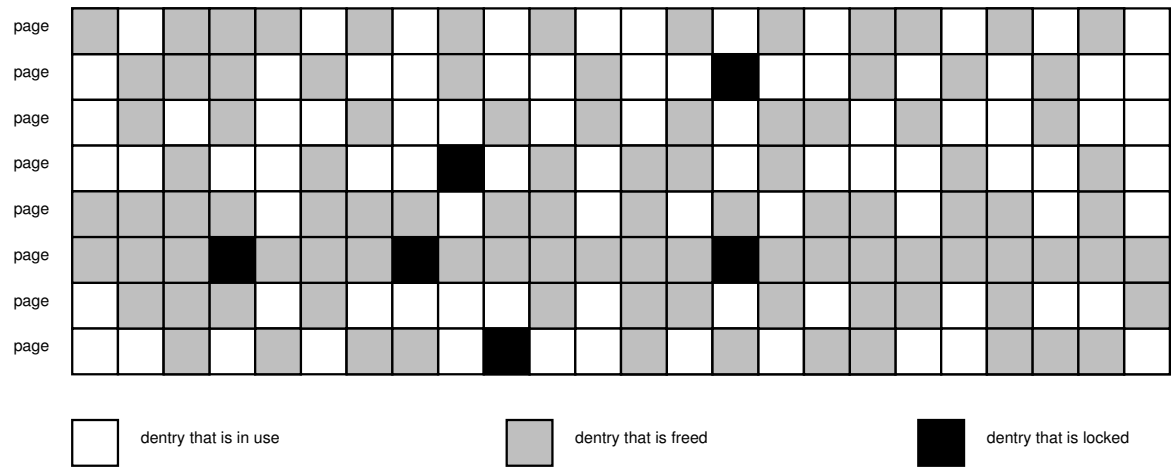

Figure 5: dentry slab

7 Diagnosing OOM situations

When the system runs out of memory, you will typically see messages either from the OOM killer, or "page allocation failures." These are typically symptoms that either:

- Your workload is unreasonable for the machine
- Something is wrong

If the workload does not fit into RAM + SWAP, then you are *going* to run out of memory. If it does not fit into RAM, then it will probably perform badly, but should still work.

7.1 Examining error messages

When `__alloc_pages` can not allocate you the memory you asked for, it prints something like this:

`%s: page allocation failure.`
`order:%d, mode:0x%x`

If the order was 0, the system could not allocate you 1 single page of memory. Examine the flags for the allocation carefully, and match them up to the `GFP_` ones in `include/linux/gfp.h`. Things like `GFP_HIGH`, and not having `GFP_WAIT` and/or `GFP_IO` set are brutal on the allocator. If you do such things at a high rate then, yes, you will exhaust the system of memory. Play nice!

If it was a normal alloc (e.g. `__GFP_WAIT` | `__GFP_IO` | `__GFP_FS`), then you have no memory, and we could free no memory to satisfy your request. Your system is in deep trouble.

If the order was say 3 (or even larger) you probably have a slightly different problem. Order n means trying to allocate 2^n pages. For example, order 3 means $2^3 = 8$ pages. Worse, these cannot be any old 8 pages, but 8 physically contiguous pages, aligned on an boundary of 8 pages. Systems that have been running for a while inevitably get fragmented to the point

where large allocs are inevitably going to fail (i.e. we have lots of smaller blocks free, but none big enough for that). Possible fixes are:

- Wait for the VM to get better at dealing with fragmentation (do not hold your breath).
- See if the caller can do without physically contiguous blocks of RAM.
- Make the caller operate out of a reserved mempool

Use the printed stack trace to find the associated caller requesting the memory. CIFS, NFS, and gigabit ethernet with jumbo frames are known offenders. /proc/buddyinfo will give you more stats on fragmentation. Adding a `show_mem()` to `__alloc_pages` just after the `printk` of the failure is often helpful.

7.2 So who ate all my memory then?

There are two basic answers, either the kernel ate it, or userspace ate it. If the userspace ate it, then hopefully the OOM killer will blow them away. If it was kernel memory, we need two basic things to diagnose it, */proc/meminfo* and */proc/slabinfo*.

If your system has already hung, Alt+Sysrq+M may give you some some of the same information.

If your system has already OOM killed a bunch of stuff, then it is hard to get any accurate output. Your best bet is to reproduce it, and do something like this:

```
while true
do
  date
  cat /proc/meminfo
  cat /proc/slabinfo
  ps ef -o user,pid,rss,command
  echo ------------------------
  sleep 10
done
```

From a script, preferably running that from a remote machine and logging it, i.e.:

```
script log
ssh theserverthatkeepsbreaking
./thatstupidloggingscript
```

Examination of the logs from the time the machine got into trouble will often reveal the source of the problem.

8 Future Direction

Memory reclaim is sure to be a focus area going forward—the difference in access latencies between disk and memory make the decisions about which pages we select to reclaim critical. We are seeing ever increasing complexity at the architectural level: SMP systems are becoming increasingly large at the high end and increasingly common at the desktop, SMT (symmetric multi-threading) and multi-core CPUs are entering the market at ever lower prices. All of these place new constraints on memory in relation to memory contention and locality which has a knock on effect on memory allocation and thereby memory reclaim. There is already much work in progress looking at these issues.

Promoting Reclaimabilty: work in the allocator to try and group the reclaimable and non-reclaimable allocations with allocations of the same type at various levels. This increases the chance of finding contigious allocations and

when they are not available greatly improves the likelihood of being able to reclaim an appropriate area.

Promoting Locality: work is ongoing to better target allocations in NUMA systems when under memory pressure. On much NUMA hardware the cost of using non-local memory for long running tasks is severe both for the performance of the affected process and for the system as a whole. Promoting some reclaim for local allocations even when remote memory is available is being added.

Hotplug: hot-removal of memory requires that we be able to force reclaim the memory which is about to be removed. Work is ongoing to both increase the likelyhood of being able to reclaim the memory and how to handle the case where it cannot be reclaimed thorough remapping and relocation.

Targeted Reclaim: memory reclaim currently only comes in the form of general pressure on the memory system. The requirements of hotplug and others brings a new kind of pressure, pressure over a specific address range. Work is ongoing to see how we can apply address specific pressure both to the normal memory allocator and the slab allocators.

Active Defragmentation: as a last resort, we can re-order pages within the system in order to free up physically contiguous segments to use.

9 Conclusion

As we have shown memory reclaim is a complex subject, something of a black art. The current memory reclaim system is extremely complex, one huge heuristic guess. Moreover, it is under pressure from new requirements from big and small iron alike. NUMA architectures are moving to the desktop. Hotplug memory is becoming the norm for larger machines, and is increasingly important for virtualization. Each of these requirements brings its own issues to what already is a difficult, complex subsystem.

Proceedings of the Linux Symposium

Volume One

July 20nd–23th, 2005
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

Review Committee

Gerrit Huizenga, *IBM*
Matthew Wilcox, *HP*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Matt Domsch, *Dell*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*