USENIX Association

Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference

Boston, Massachusetts, USA
June 25–30, 2001

Page replacement in Linux 2.4 memory management

Rik van Riel
Conectiva Inc.
riel@conectiva.com.br, http://www.surriel.com/

Abstract

While the virtual memory management in Linux 2.2 has decent performance for many workloads, it suffers from a number of problems. The first part of this paper contains a description of how the Linux 2.2 VMM works and an analysis of why it has bad behaviour in some situations.

The way in which a lot of this behaviour has been fixed in the Linux 2.4 kernel is described in the second part of the paper. Due to Linux 2.4 being in a code freeze period while these improvements were implemented, only known-good solutions have been integrated. A lot of the ideas used are derived from principles used in other operating systems, mostly because we have certainty that they work and a good understanding of why, making them suitable for integration into the Linux codebase during a code freeze.

1 Linux 2.2 memory management

The memory management in the Linux 2.2 kernel seems to be focussed on simplicity and low overhead. While this works pretty well in practice for most systems, it has some weak points left and simply falls apart under some scenarios.

Memory in Linux is unified, that is all the physical memory is on the same free list and can be allocated to any of the following memory pools on demand. Most of these pools can grow and shrink on demand. Typically most of a system's memory will be allocated to the data pages of processes and the page and buffer caches.

- The slab cache: this is the kernel's dynamically allocated heap storage. This memory is unswappable, but once all objects within one (usually page-sized) area are unused, that area can be reclaimed.
- The page cache: this cache is used to cache file data for both mmap() and read() and is indexed by (inode, index) pairs. No dirty data exists in this cache; whenever a program writes to a page, the dirty data is copied to the buffer cache, from where the data is written back to disk.
- The buffer cache: this cache is indexed by (block device, block number) tuples and is used to cache raw disk devices, inodes, directories and other filesystem metadata. It is also used to perform disk IO on behalf of the page cache and the other caches. For disk reads the pagecache bypasses this cache and for network filesystems it isn't used at all.
- The inode cache: this cache resides in the slab cache and contains information about cached files in the system. Linux 2.2 cannot shrink this cache, but because of its limited size it does need to reclaim individual entries.
- The dentry cache: this cache contains directory and name information in a filesystem-independent way and is used to lookup files and directories. This cache is dynamically grown and shrunk on demand.
- SYSV shared memory: the memory pool containing the SYSV shared memory segments is managed pretty much like the page cache, but has its own infrastructure for doing things.
- Process mapped virtual memory: this memory is administrated in the process page tables. Processes can have page cache or SYSV shared memory segments mapped, in which case those pages are managed in both the page tables and the data structures used for respectively the page cache or the shared memory code.

1.1 Linux 2.2 page replacement

The page replacement of Linux 2.2 works as follows. When free memory drops below a certain threshold, the pageout daemon (kswapd) is woken up. The pageout daemon should usually be able to keep enough free memory, but if it isn't, user programs will end up calling the pageout code itself.

The main pageout loop is in the function try_to_free_pages, which starts by freeing unused slabs from the kernel memory pool. After that, it calls the following functions in a loop, asking each of them to scan a small part of their part of memory until enough memory has been freed.

- shrink_mmap is a classical clock algorithm, which loops over all physical pages, clearing referenced bits, queueing old dirty pages pages for IO and freeing old clean pages. The main disadvantage it has compared to a clock algorithm, however, is that it isn't able to free pages which are in use by a program or a shared memory segment. Those pages need to be unmapped by swap_out first.
- shm_swap scans the SYSV shared memory segments, swapping out those pages that haven't been referenced recently and which aren't mapped into any process.
- swap_out scans the virtual memory of all processes in the system, unmapping pages which haven't been referenced recently, starting swapout IO and placing those pages in the page cache.
- shrink_dcache_memory recaims entries from the VFS name cache. This is not directly reusable memory, but as soon as a whole page of these entries gets unused we can reclaim that page.

Some balancing between these memory freeing function is achieved by calling them in a loop, starting of by asking each of these functions to scan a little bit of their memory, as each of these funnctions accepts a priority argument which tells them how big a percentage of their memory to scan. If not enough memory is freed in the first loop, the priority is increased and the functions are called again. The idea behind this scheme is that when one memory pool is heavily used, it will not give up its resources lightly and we'll automatically fall through to one of the other memory pools. However, this scheme relies on each of the memory pools to react in a similar way to the priority argument under different load conditions. This doesn't work out in practice because the memory pools just have fundamentally different properties to begin with.

1.2 Problems with the Linux 2.2 page replacement

- Balancing between evicting pages from the file cache, evicting unused process pages and evicting pages from shm segments. If memory pressure is "just right" shrink_mmap is always successful in freeing cache pages and a process which has been idle for a day is still in memory. This can even happen on a system with a fairly busy filesystem cache, but only with the right phase of moon.
- Simple NRU[Note] replacement cannot accurately identify the working set versus incidentally accessed pages and can lead to extra page faults. This doesn't hurt noticably for most workloads, but it makes a big difference in some workloads and can be fixed easily, mostly since the LFU replacement used in older Linux kernels is known to work.
- Due to the simple clock algorithm in shrink_mmap, sometimes clean, accessed pages can get evicted before dirty, old pages. With a relatively small file cache that mostly consists of dirty data, eg unpacking a tarball, it is possible for the dirty pages to evict the (clean) metadata buffers that are needed to write the dirty data to disk. A few other corner cases with amusing variations on this theme are bound to exist.
- The system reacts badly to variable VM load or to load spikes after a period of no VM activity. Since kswapd, the pageout daemon, only scans when the system is low on memory, the system can end up in a state where some pages have referenced bits from the last 5 seconds, while other pages have referenced bits from 20 minutes ago. This means that on a load spike the system has no clue which are the right pages to evict from memory, this can lead to a swapping storm, where the wrong pages are evicted and almost immediately af-

terwards faulted back in, leading to the page-out of another random page, etc...

- Under very heavy loads, NRU replacement of pages simply doesn't cut it. More careful and better balanced pageout eviction and flushing is called for. With the fragility of the Linux 2.2 pageout framework this goal doesn't really seem achievable.

The facts that shrink_mmap is a simple clock algorithm and relies on other functions to make process-mapped pages freeable makes it fairly unpredictable. Add to that the balancing loop in try_to_free_pages and you get a VM subsystem which is extremely sensitive to minute changes in the code and a fragile beast at its best when it comes to maintenance or (shudder) tweaking.

2 Changes in Linux 2.4

For Linux 2.4 a substantial development effort has gone into things like making the VM subsystem fully fine-grained for SMP systems and supporting machines with more than 1GB of RAM. Changes to the pageout code were done only in the last phase of development and are, because of that, somewhat conservative in nature and only employ known-good methods to deal with the problems that happened in the page replacement of the Linux 2.2 kernel. Before we get to the page replacement changes, however, first a short overview of the other changes in the 2.4 VM:

- More fine-grained SMP locking. The scalability of the VM subsystem has improved a lot for workloads where multiple CPUs are reading or writing the same file simultaneously; for example web or ftp server workloads. This has no real influence on the page replacement code.
- Unification of the buffer cache and the page cache. While in Linux 2.2 the page cache used the buffer cache to write back its data, needing an extra copy of the data and doubling memory requirements for some write loads, in Linux 2.4 dirty page cache pages are simply added in both the buffer and the page cache. The system does disk IO directly to and from the page cache page. That the buffer cache is still maintained separately for filesystem metadata and the caching of raw block devices. Note that the cache was already unified for reads in Linux 2.2, Linux 2.4 just completes the unification.
- Support for systems with up to 64GB of RAM (on x86). The Linux kernel previously had all physical memory directly mapped in the kernel's virtual address space, which limited the amount of supported memory to slightly under 1GB. For Linux 2.4 the kernel also supports additional memory (so called "high memory" or highmem), which can not be used for kernel data structures but only for page cache and user process memory. To do IO on these pages they are temporarily mapped into kernel virtual memory and the data is copied to or from a bounce buffer in "low memory".

  At the same time the memory zone for ISA DMA (0 - 16 MB physical address range) has also been split out into a separate page zone. This means larger x86 systems end up with 3 memory zones, which all need their free memory balanced so we can continue allocating kernel data structures and ISA DMA buffers. The memory zones logic is generalised enough to also work for NUMA systems.
- The SYSV shared memory code has been removed and replaced with a simple memory filesystem which uses the page cache for all its functions. It supports both POSIX SHM and SYSV SHM semantics and can also be used as a swappable memory filesystem (tmpfs).

Since the changes to the page replacement code took place after all these changes and in the (one and a half year long) code freeze period of the Linux 2.4 kernel, the changes have been kept fairly conservative. On the other hand, we have tried to fix as many of the Linux 2.2 page replacement problems as possible. Here is a short overview of the page replacement changes: they'll be described in more detail below.

- Page aging, which was present in the Linux 1.2 and 2.0 kernels and in FreeBSD has been reintroduced into the VM. However, a few small changes have been made to avoid some artifacts of virtual page based aging.

- To avoid the eviction of "wrong" pages due to interactions from page aging and page flushing, the page aging and flushing has been separated. There are active and inactive page lists.
- Page flushing has been optimised to avoid too much interference by writeout IO on the more time-critical disk read IO.
- Controlled background page aging during periods of little or no VM activity in order to keep the system in a state where it can easily deal with load spikes.
- Streaming IO is detected; we do early eviction on the pages that have already been used and reward the IO stream with more agressive readahead.

3 Linux 2.4 page replacement changes in detail

The development of the page replacement changes in Linux 2.4 has been influenced by two main factors. Firstly the bad behaviours of Linux 2.2 page replacement had to be fixed, using only known-good strategies because the development of Linux 2.4 had already entered the "code freeze" state. Secondly the page replacement had to be more predictable and easier to understand than Linux 2.2 because tuning the page replacement in Linux 2.2 was deserving of the proverbial label "subtle and quick to upset". This means that only VM ideas that are well understood and have little interactions with the rest of the system were integrated. Lots of ideas were taken from other freely available operating systems and literature.

3.1 Page aging

Page aging was the first easy step in making the bad border-case behaviour from Linux 2.2 go away, it works reasonably well in Linux 1.2, Linux 2.0 and FreeBSD. Page aging allows us to make a much finer distinction between pages we want to keep in memory and pages we want to swap out than the NRU aging in Linux 2.2.

Page aging in these OSes works as follows: for each physical page we keep a counter (called age in Linux, or act_count in FreeBSD) that indicates how desirable it is to keep this page in memory. When scanning through memory for pages to evict, we increase the page age (adding a constant) whenever we find that the page was accessed and we decrease the page age (substracting a constant) whenever we find that the page wasn't accessed. When the page age (or act_count) reaches zero, the page is a candidate for eviction.

However, in some situations the LFU[Note] page aging of Linux 2.0 is known to have too much CPU overhead and adjust to changes in system load too slowly. Furthermore, research[Smaragdis, Kaplan, Wilson] has shown that recency of access is a more important criteria for page replacement than frequency.

These two problems are solved by doing exponential decline of the page age (divide by two instead of substracting a constant) whenever we find a page that wasn't accessed, resulting in page replacement which is closer to LRU[Note] than LFU. This reduces the CPU overhead of page aging drastically in some cases; however, no noticable change in swap behaviour has been observed.

Another artifact comes from the virtual address scanning. In Linux 1.2 and 2.0 the system reduces the page age of a page whenever it sees that the page hasn't been accessed from the page table which it is currently scanning, completely ignoring the fact that the page could have been accessed from other page tables. This can put a severe penalty on heavily shared pages, for example the C library.

This problem is fixed by simply not doing "downwards" aging from the virtual page scans, but only from the physical-page based scanning of the active list. If we encounter pages which are not referenced, present in the page tables but not on the active list, we simply follow the swapout path to add this page to the swap cache and the active list so we'll be able to lower the page age of this page and swap it out as soon as the page age reaches zero.

3.2 Multiple page lists

The bad interactions between page aging and page flushing, where referenced clean pages were freed before old dirty pages, is fixed by keeping the pages which are candidates for eviction separated from the

pages we want to keep in memory (page age zero vs. nonzero). We separate the pages out by putting them on various page lists and having separate algorithms deal with each list.

Pages which are not (yet) candidate for eviction are in process page tables, on the active list or both. Page aging as described above happens on these pages, with the function refill_inactive() balancing between scanning the page tables and scanning the active list.

When the page age on a page reaches zero, due to a combination of pageout scanning and the page not being actively used, the page is moved to the inactive_dirty list. Pages on this list are not mapped in the page tables of any process and are, or can become, reclaimable. Pages on this list are handled by the function page_launder(), which flushes the dirty pages to disk and moves the clean pages to the inactive_clean list.

Unlike the active and inactive_dirty lists, the inactive_clean list isn't global but per memory zone. The pages on these lists can be immediately reused by the page allocation code and count as free pages. These pages can also still be faulted back into where it came from, since the data is still there. In BSD this would be called the "cache" queue.

3.3 Dynamically sized inactive list

Since we do page aging to select which pages to evict, having a very large statically sized inactive list (like FreeBSD has) doesn't seem to make much sense. In fact, it would cancel out some of the effects of doing the page aging in the first place: why spend much effort selecting which pages to evict[Dillon] when you keep as much as 33% of your swappable pages on the inactive list? Why do careful page aging when 33% of your pages end up as candidates for eviction at the same priority and you've effectively undone the aging for those 33% of pages which are candidates for eviction?

On the other hand, having lots of inactive pages to choose from when doing page eviction means you have more chances of avoiding writeout IO or doing better IO clustering. It also gives you more of a "buffer" to deal with allocations due to page faults, etc.

Both a large and a small target size for the inactive page list have their benefits. In Linux 2.4 we have chosen for a middle ground by letting the system dynamically vary the size of the inactive list depending on VM activity, with an artificial upper limit to make sure the system always preserves some aging information.

Linux 2.4 keeps a floating average of the amount of pages evicted per second and sets the target for the inactive list and the free list combined to the free target plus this average number of page steals per second. Not only does this second give us enough time to do all kinds of page flushing optimisations, it also is small enough to keep page age distribution within the system intact, allowing us to make good choices on which pages to evict and which pages to keep.

3.4 Optimised page flushing

Writing out pages from the inactive_dirty list as we encounter them can cause a system to totally destroy read performance because of the extra disk seeks done. A better solution is to delay writeout of dirty pages and let these dirty pages accumulate until we can do better IO clustering so that these pages can be written out to disk with less disk seeks and less interference with read performance.

Due to the development of the page replacement changes happening in the code freeze, the system currently has a rather simple implementation of what's present in FreeBSD 4.2. As long as there are enough clean inactive pages around, we keep moving those to the inactive_clean list and never bother with syncing out the dirty pages. Note that this catches both clean pages and pages which have been written to disk by the update daemon (which commits filesystem data to disk periodically).

This means that under loads where data is seldom written we can avoid writing out dirty inactive pages most of the time, giving us much better latencies in freeing pages and letting streaming reads continue without the disk head moving away to write out data all the time. Only under loads where lots of pages are being dirtied quickly does the system suffer a bit from syncing out dirty data irregularly.

Another alternative would have been the strategy used in FreeBSD 4.3, where dirty pages get to stay

in the inactive list longer than clean pages but are synced out before the clean pages are exhausted. This strategy gives more consistent pageout IO in FreeBSD during heavy write loads. However, a big factor causing the irregularities in pageout writes using the simpler strategy above may well be caused because of the huge inactive list target in FreeBSD (33It is not at all clear what this more complicated strategy would do when used on the dynamically sized inactive list on Linux 2.4, because of this Linux 2.4 uses the better understood strategy of evicting clean inactive pages first and only after those are gone start syncing the dirty ones.

3.5 Background page aging

On many systems the normal operating mode is that after a period of relative activity a sudden load spike comes in and the system has to deal with that as gracefully as possible. Linux 2.2 has the problem that, with the lack of an inactive page list, it is not clear at all which pages should be evicted when a sudden demand for memory kicks in.

Linux 2.4 is better in this respect, with the reclaim candidates neatly separated out on the inactive list. However, the inactive list could have any random size the moment VM pressure drops off. We'd like get the system in a more predictable state while the VM pressure is low. In order to achieve this, Linux 2.4 does background scanning of the pages, trying to get a sane amount of pages on the inactive list, but without scanning agressively so only truly idle pages will end up on the inactive list and the scanning overhead stays small.

3.6 Drop behind

Streaming IO doesn't just have readahead, but also its natural complement: drop behind. After the program doing the streaming IO is done with a page, we depress its priority heavily so it will be a prime candidate for eviction. Not only does this protect the working set of running processes from being quickly evicted by streaming IO, but it also prevents the streaming IO from competing with the pageouts and pageins of the other running processes, which reduces the number of disk seeks and allows the streaming IO to proceed at a faster speed. Currently readahead and drop-behind only work for read() and write(); mmap()ed files and swap-backed anonymous memory aren't supported yet.

4 Conclusions

Since the Linux 2.4 kernel's VM subsystem is still being tuned heavily, it is too early to come with conclusive figures on performance. However, initial results seem to indicate that Linux 2.4 generally has better performance than Linux 2.2 on the same hardware.

Reports from users indicate that performance on typical desktop machines has improved a lot, even though the tuning of the new VM has only just begun. Throughput figures for server machines seem to be better too, but that could also be attributed to the fact that the unification of the page cache and the buffer cache is complete.

One big difference between the VM in Linux 2.4 and the VM in Linux 2.2 is that the new VM is far less sensitive to subtle changes. While in Linux 2.2 a subtle change in the page flushing logic could upset page replacement, in Linux 2.4 it is possible to tweak the various aspects of the VM with predictable results and little to no side-effects in the rest of the VM.

The solid performance and relative insensitivity to subtle changes in the environment can be taken as a sign that the Linux 2.4 VM is not just a set of simple fixes for the problems experienced in Linux 2.2, but also a good base for future development.

5 Remaining issues

The Linux 2.4 VM mainly contains easy to implement and obvious to verify solutions for some of the known problems Linux 2.2 suffers from. A number of issues are either too subtle to implement during the code freeze or will have too much impact on the code. The complete list of TODO items can be found on the Linux-MM page[Linux-MM]; here are the most important ones:

- Low memory deadlock prevention: with the arrival of journaling and delayed-allocation

filesystems it is possible that the system will need to allocate memory in order to free memory; more precisely, to write out data so memory can become freeable. To remove the possibility for deadlock, we need to limit the number of outstanding transactions to a safe number, possibly letting each of the page flushing functions indicate how much memory it may need and doing bookkeeping of these values. Note that the same problem occurs with swap over network.

- Load control: no matter how good we can get the page replacement code, there will always be a point where the system ends up thrashing to death. Implementing a simple load control system, where processes get suspended in round-robin fashion when the paging load gets too high, can keep the system alive under heavy overload and allow the system to get enough work done to bring itself back to a sane state.

- RSS limits and guarantees: in some situations it is desirable to control the amount of physical memory a process can consume (the resident set size, or RSS). With the virtual address based page scanning of Linux' VM subsystem it is trivial to implement RSS ulimits and minimal RSS guarantees. Both help to protect processes under heavy load and allow the system administrator to better control the use of memory resources.

- VM balancing: in Linux 2.4, the balancing between the eviction of cache pages, swap-backed anonymous memory and the inode and dentry caches is essentially the same as in Linux 2.2. While this seems to work well for most cases there are some possible scenarios where a few of the caches push the other users out of memory, leading to suboptimal system performance. It may be worthwhile to look into improving the balancing algorithm to achieve better performance in "non-standard" situations.

- Unified readahead: currently readahead and drop-behind only works for read() and write(). Ideally they should work for mmap()ed files and anonymous memory too. Having the same set of algorithms for both read()/write(), mmap() and swap-backed anonymous memory will simplify the code and make performance improvements in the readahead and drop-behind code immediately available to all of the system.

6 Acknowledgements

The author would like to thank, in no particular order: Stephen Tweedie, for taking care of memory management in Linux 1.2, 2.0 and 2.2 and also for his help with this paper; Matt Dillon, for taking the time to explain the rationale behind every little piece of the FreeBSD VM; Conectiva Inc, who employ the author to hack the Linux kernel and the wonderful crowd testers from #kernelnewbies[Kernelnewbies] and elsewhere who have helped flesh out the bugs in the Linux 2.4 VM.

References

[de Castro] Rodrigo S. de Castro
Linux 2.4 Virtual Memory Overview (2001)
`http://linuxcompressed.sourceforge.net/vm24/`

[Dillon] Matthew Dillon
Design Elements of the FreeBSD VM System (2000)
`http://www.daemonnews.org/200001/freebsd_vm.html`

[Kernelnewbies] Kernelnewbies
`http://kernelnewbies.org/`

[Linux-MM] The Linux Memory Management home page
`http://linux-mm.org/`

[Smaragdis, Kaplan, Wilson] Yannis Smaragdakis, Scott F. Kaplan and Paul R. Wilson
EELRU: Simple and Effective Adaptive Page Replacement,
SIGMETRICS '99
`http://www.cs.amherst.edu/~sfkaplan/papers/index.html`

[Note] Extensive documentation about page replacement algorithms is available practically everywhere. The 3 algorithms discussed in this paper are:

- NRU: Not Recently Used, we scan through memory and evict every page that wasn't accessed since we last scanned it.

- LRU: we evict those pages that haven't been accessed for the longest time.
- LFU: we evict those pages that have been accessed least frequently in recent times.