Reprinted from the

Proceedings of the Linux Symposium

July 23th–26th, 2003
Ottawa, Ontario
Canada

Conference Organizers

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

Review Committee

Alan Cox, *Red Hat, Inc.*
Andi Kleen, *SuSE, GmbH*
Matthew Wilcox, *Hewlett-Packard*
Gerrit Huizenga, *IBM*
Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Martin K. Petersen, *Wild Open Source, Inc.*

Proceedings Formatting Team

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

Towards an O(1) VM:

Making Linux virtual memory management scale towards large amounts of physical memory

Rik van Riel
Red Hat, Inc.
riel@surriel.com

Abstract

Linux 2.4 and 2.5 already scale fairly well towards many CPUs, large numbers of files, large numbers of network connections and several "other kinds of big." However, the VM still has a few places with poor worst case (or even average case) behavior that needs to be improved in order to make Linux work well on machines with many gigabytes of RAM.

1 Introduction

In this paper I will explore the problem spaces and algorithmic complexities of the virtual memory subsystem. This paper will focus mostly on the page replacement code, which by definition has all of physical memory and parts of virtual memory as its search space. The following aspects of page replacement will be discussed:

- Page launder, the reclaiming of pages that are selected for pageout.
- Page aging, how to select which pages to evict.
- Balancing filesystem cache vs. anonymous memory.
- Reverse mapping, pte based vs. object based.

2 Page launder

Traditionally the virtual memory management subsystems in Unix and Linux systems have had either a clock algorithm or Mach-style active and inactive lists to do both LRU aging and eviction of pages. Linux 2.4 and 2.5 have what amounts to simple Mach-style active and inactive lists (Figure 1), at least when it comes to the writeout and reclaiming of pages that aren't mapped in processes. In this paper, the Mach VM pageout algorithm is used as an example because it is a decent approximation of what the different Linux VMs have done and the Mach VM is quite possibly the best documented virtual memory subsystem.

In the Mach VM, pages get recycled once they reach the end of the inactive list and are clean, meaning they do not need to be written to disk. If the page needs to be written to disk, a so-called dirty page, disk IO is started and the page is moved to the beginning of the inactive list. Presumably the disk IO will have finished and the page will be clean by the time it gets to the end of the inactive list again.

This organisation works reasonably well when dealing with filesystem cache pages, since

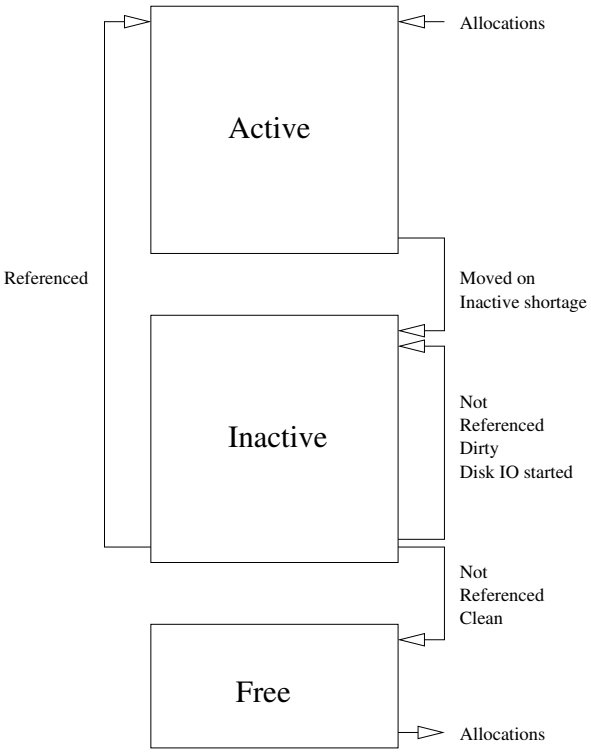

Figure 1: Mach pageout lists

those are usually clean pages which can be reclaimed the moment they reach the end of the inactive list. However, when the filesystem cache is small and the system is dealing mostly with dirty, swap or mmap backed pages from processes, this strategy has a big drawback on modern, large memory computers.

2.1 The problem with Mach-style page laundering

Memory used by processes is often dirty, meaning it needs to be written back to disk. The problem with this becomes obvious when we look at exactly what happens when all of the pages on the inactive list are dirty:

- The pageout code encounters a dirty page.
- Disk IO is started, the page is written to disk.
- The page is moved to the far end of the inactive list.
- The page reclaiming code encounters the next dirty page, starts writeout, etc.. . .
- Since only a finite number of disk IO operations can be underway at any time, the page reclaiming code needs to wait for current IO operations to finish once it has started writeout on a certain number of pages.
- IO on the pages that were written out first finishes, meaning the pages are now clean and reclaimable.
- The page reclaiming code continues with the write out of the other dirty pages on the inactive list.

Of course, this has a number of serious drawbacks. The most obvious one is that on large memory systems the system will wait for most pageout IO to have finished before it can even start the last IO. Worse yet, it won't be able to free a page before all IO has been submitted.

In the early 1990s, when the Mach VM was popular, systems had up to a few megabytes of memory, with maybe a few hundred kilobytes of inactive pages, which could be written to disk in one or at most a few seconds. Modern systems, on the other hand, often have multiple gigabytes of memory. Since the speed of hard disks hasn't increased nearly as much as the size of memory, the time needed to write out all of the inactive list can be unacceptably high, up to dozens of seconds.

2.2 Solutions

One obvious solution is to only write out part of the pages on the inactive list. After all, if the system needs to free ten megabytes of memory, there is little reason to write out one gigabyte of

data. The implementation of this solution is a little less obvious, since there are various ways to approach this goal and there is a tradeoff to make between CPU usage and page freeing latency.

The first solution would be to simply write out a limited number of pages and skip the dirty pages on the list, scanning the list like usual and freeing all the clean pages encountered. In situations where the inactive list has both clean and dirty pages this tactic will allow you to always free the clean pages, reaching your free target and allowing allocations to go on with as little latency as possible. Of course, if the list only has dirty pages, then the system could end up spending a lot of CPU time scanning the list over and over again.

For the rmap VM a different, hopefully more predictable and CPU friendly solution (Figure 2) has been chosen. Instead of just one inactive list, there are various lists for the different stages of the pageout process a page can be in. Initially all rarely used pages are placed on the inactive_dirty list, regardless of whether or not they need to be written back to disk.

When a page reaches the end of the inactive_dirty list and wasn't referenced, the VM will move it to the inactive_laundry list, starting disk IO if the page was dirty. Referenced pages get moved back to the active list.

On the other end of the inactive_laundry list the VM removes clean pages, until the system has enough immediately freeable and free pages. Referenced pages are moved back to the active list; cleaned pages are moved on to the inactive_clean list, from where they can be immediately reused by the page allocation code.

The inactive_clean list is just an extension of the free page list. It contains clean pages that were not referenced and can be immediately reclaimed by the page allocation code. The reason for having an inactive_clean list is that the free page list in a VM is never the right size. The list should be as large as possible in order to be able to satisfy allocations with low latency, but at the same time the list should be as small as possible so almost all of memory can be used for processes and the cache. Having a list of immediately reclaimable pages with useful data in them avoids most of this dilemma.

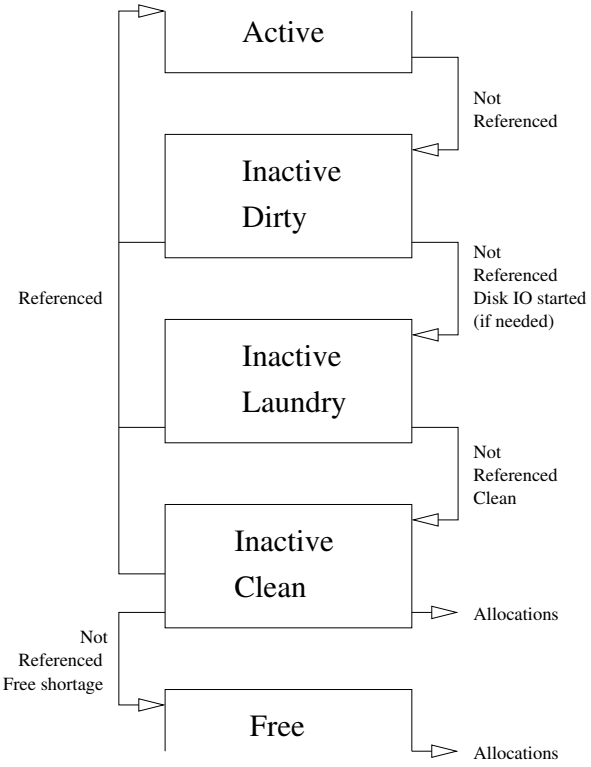

Figure 2: O(1) page launder

3 Page aging

Since the performance penalty of evicting the wrong page from memory is so high, due to the enormous speed differential between memory and disk, any virtual memory subsystem needs to take great care in selecting which pages to evict and which pages to keep in memory. On the other hand, on systems with more than a few megabytes of memory you do not want to

scan all the active pages every time the system is short on inactive memory.

While it is impossible to ensure this situation will never happen, because some applications just have access patterns you cannot tune a page replacement algorithm for, we can improve the situation a lot by pre-sorting the active pages in various lists (Figure 3), according to activity.

The pageout code will only look at the pages that most likely aren't very active, meaning it has a better chance of finding the proper pages for eviction without needing to resort to a full scan of memory. If the list with least used pages is empty, the pageout code simply shifts down all of the active lists and starts looking at the pages that came from the next list up.

The page aging (sorting) code scans the active lists periodically and moves the pages that were accessed to higher lists. It only needs to age pages upwards, because the downwards movement is done by the pageout code shifting down whole lists at a time. The period with which the page aging code scans the active lists is varied in reaction to the amount of pageout activity. Ideally the system would do a similar number of up aging scans as the number of times it shifts down active lists. The scan interval of the up aging code is reduced if the VM did too many down shifting of active pages and increased if the VM was quiet in-between two aging scans. The page aging interval has both a lower and an upper bound, to keep the overhead under control and to have some background aging in an otherwise idle system. The only time the page aging doesn't run is when there are more active pages on the higher lists than on the lower lists.

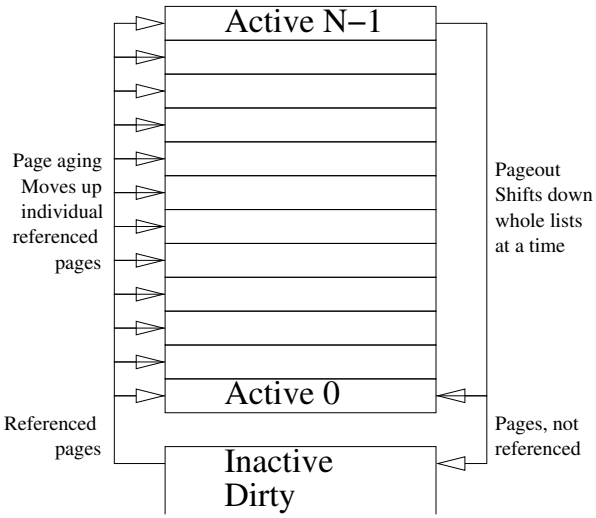

Figure 3: Multi list page aging

4 Balancing cache vs program memory

LRU style page replacement algorithms have well-documented, known problems. There are several replacement algorithms available that improve the replacement of pages within one set of data, e.g. EELRU, SEQ and LRFU; however none of these address the problem of balancing replacement between various sets of data. Since all currently implemented page replacement algorithms for Linux have this problem, the replacement algorithm needs some help balancing the file cache with memory used for programs.

The rmap VM borrows a common trick from other systems here. There are separate active lists for file cache memory and program memory, active_cache and active_anon, respectively. While the cache is larger than a certain percentage of active memory only the cache pages are a candidate for pageout, this value is the so-called borrow percentage and is 15 by default. Below the borrow percentage the VM will move both cache pages and pages belonging to processes to the inactive

list, reclaiming the pages that haven't been referenced again by the time they reach the far end of the inactive list. This gives cache and processes a chance to balance against each other by referencing pages. If the cache takes less than a predetermined minimum of the active list, one percent by default, the VM will only reclaim pages from processes.

The deeper reasons behind the need for these balancing hints are a little more complex than the reasons behind other design choices in the VM. One of the factors is that the amount of data on the filesystems tends to be several magnitudes larger than the amount of memory taken by the processes in the system. This means that the number of accesses to pages from the file cache could overwhelm the total number of accesses to the pages of the processes, even though the individual pages of the processes get accessed more frequently than most file cache pages. In other words, the system can end up evicting frequently accessed pages from memory in favor of a mass of recently but far less frequently accessed pages.

A replacement algorithm like LIRS, Low Interreference Recency Set, would probably do the right thing since it replaces pages with a higher interval between references before pages that have a lower interval between references. However, for LIRS to work properly the VM would need to keep track of pages that have already been evicted from memory. Since Linux does not have an infrastructure to keep track of those, the rmap VM uses an LRFU style page replacement algorithm with cache size hints.

Even if the direct value of LIRS over LRU/LFU for use as a primary cache wouldn't be big enough to offset the overhead of the needed infrastructure, the facts that LIRS would make the file cache vs process memory balancing automatic and that LIRS would also do the right thing as a second level cache (e.g. an NFS server, page cache on a web proxy where squid itself has the first level cache) make the implementation of LIRS for Linux a promising future experiment.

5 Reverse mapping

Reverse mappings provide an inverse to the page tables of the processes; that is, they keep track of which processes are using the physical pages, at which virtual addresses. Using reverse mappings, the pageout code can:

- Unmap a page from all processes using it, without needing to search the virtual memory of all processes.
- Unmap only those pages it really wants to evict, instead of scanning the virtual memory of all processes and unmapping more pages than it wants to evict in order to be on the safe side. This could reduce the number of minor page faults.
- Evict pages in a certain physical address range, which is useful since Linux divides physical memory into various zones.
- Scan only the virtual mappings of known inactive pages, which means the pageout code has a smaller search space in virtual memory. Combined with smarter page aging and page laundering, this results in a smaller overall search space for the pageout code.

5.1 Page based vs object based

There are pros and cons to doing reverse mapping on a per-page or a per-object basis. Reverse mapping on a per-page basis is more efficient for the pageout code, but the reverse

mapping code affects more than just the page-out code path. The page fault, fork, exit, and mmap paths all modify the reverse mappings, so doing reverse mappings on objects larger than a page (like a vma) would reduce the reverse mapping overhead in those code paths, at the cost of the pageout code needing to search more space.

The big question here is how much the overhead and algorithmic complexities would change, especially under larger workloads. A quadratic increase in complexity in the pageout path is almost certainly more expensive than what could be offset by a linear speedup in the other code paths, even though the pageout path is rarely run.

Large workloads, with many gigabytes of memory and hundreds or thousands of large, active processes are certainly able to bring out the worst of any VM; with the current implementations it doesn't even matter which style of reverse mapping is used. Bad behaviour can be triggered in either case.

It appears that for both object-based and page-based reverse mappings, Linux is in need of smarter data structures that aren't susceptible to quadratic algorithmic complexities anywhere. Once those are written we will be able to make a proper comparison between both methods of reverse mapping. It is conceivable that Linux would end up using a hybrid of object-based and page-based reverse mapping, with each type being used where it is most appropriate.

6 Conclusions

Linux memory management has come a long way in the last few years, but at the same time users have deployed Linux in more and more demanding environments. In fact, demand always seems to be one step ahead of whatever stage kernel development is at.

Users have shown beyond any doubt that there are legitimate workloads that bring out the worst case behaviour in any VM; because of this there is a constant need to bring the algorithmic complexity of any part of the virtual memory management subsystem closer to the holy grail of constant-time, or O(1) complexity. The author expects development of the Linux virtual management subsystem to remain challenging for years to come.

7 References

Draves, Richard P. *Page Replacement and Reference Bit Emulation in Mach.* In Proceedings of the USENIX Mach Symposium, Monterey, CA, November 1991.

Y. Smaragdakis, S. Kaplan, and P. Wilson, *EELRU: Simple and Effective Adaptive Page Replacement* in Proceeding of the 1999 ACM SIGMETRICS Conference, 1999.

Gideon Glass and Pei Cao. *Adaptive Page Replacement Based on Memory Reference Behavior.* In Proceedings of ACM SIGMETRICS 1997, June, 1997.

D. Lee, J. Choi, J.-H. Kim, S.H. Noh, S.L. Min, Y. Cho, and C.S. Kim, *LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies* IEEE Trans. Computers, vol. 50, no. 12, pp. 1352–1360, 2001.

S. Jiang and X. Zhuang. *LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance.* In Proc. of SIGMETRICS 2002.