

Virtual Memory Behavior in Red Hat Linux Advanced Server 2.1

Bob Matthews

Red Hat, Inc.

Kernel Development Team

Norm Murray

Red Hat, Inc.

Client Engineering Team

This is an explanation of the virtual memory behavior of the Red Hat Linux Advanced Server 2.1 kernel. Where relevant, this paper also makes reference to the virtual memory behavior of the 2.4.18 based kernels released by Red Hat.

The behavior of the virtual memory system (VM) in Red Hat Linux Advanced Server 2.1 is different from that seen in other Unix(tm) operating systems, and is also different than previous and subsequent Red Hat Linux kernels.

Table of Contents

Introductory Terms	3
Inspecting Process Memory Usage and Tuning the VM.....	4
Linux Caching.....	6
Questions about the Linux Virtual Memory Management	8

Introductory Terms

For those new to kernel virtual memory managers, or to the Linux virtual memory manager in particular, we start by defining some terms which will later be used in building an understanding of the various kernel subsystems which influence VM performance.

- *Virtual Memory:* A system that combines physical memory along with some secondary storage device to give the appearance that a computer system has more physical memory than is actually installed. A virtual memory system tries to efficiently allocate physical memory (RAM) among many competing demands, including: kernel code, kernel global data, dynamically allocated kernel memory, kernel caches (buffer, page, swap, and slab), application code, application stack space, static application memory, and application heap.
- *Page:* Kernel memory management works on top of the computer system hardware which manipulates memory in units called pages. Page size is determined solely by the underlying hardware. The page size is 4096 bytes on IA32 hardware platforms.
- *Buffer Cache:* The buffer cache holds filesystem metadata, such as the inode tables, direct blocks, indirect blocks, journals, superblocks and filesystem bitmaps. Buffer replacement is triggered by new buffer allocations which cause the VM to evict old buffers from the system's inactive list.
- *Page Cache:* This is a read/write cache for files stored on a filesystem. It is reported as `Cached` when the file `/proc/meminfo` is consulted. It includes regular files opened for reading and writing, along with mmaped files, and pages of executables currently in memory. (In 2.4.18 and later kernels, the page cache also contains filesystem directories.) In addition, objects which appear in the file system space but have no associated backing store (such as `/proc` files, pipes and FIFOs) use memory in the page cache.
- *Swap Cache:* This is a read/write cache for process data and stack pages that have been written to the swap device. It is reported as `SwapCached` when the file `/proc/meminfo` is consulted. The swap cache should be considered a virtual cache, since there is no separate bookkeeping for it in the kernel. It is, however, a convenient concept, and we shall refer to it in subsequent sections.
- *Active List:* This is a collection of pages which the kernel has determined to be in active use. The size of this list is reported as `Active` when the file `/proc/meminfo` is consulted.
- *Inactive List:* This set of pages resides in memory, but the kernel has determined that they are not in active use. These pages are candidates for eviction should the system come under memory pressure. There are several fields which describe the inactive list in the file `/proc/meminfo`:
 - `Inact_dirty`: This is memory on the inactive list which may have been modified since it was last written to disk. These pages may need to be written to disk before the associated page frames can be reallocated. All pages are categorized as `inact_dirty` when they are placed on the inactive list. The kernel will eventually inspect the pages on this list, write them out to disk if necessary, and then reclassify them as `inact_clean`.
 - `Inact_clean`: This is memory on the inactive list which can be reclaimed immediately since a valid copy exists on secondary storage.
 - `Inact_target`: This is a user tunable parameter specifying the target size of the inactive list. If the kernel detects a memory shortfall, and the inactive list is smaller than the target, the kernel will attempt to move pages from the active list to the inactive list by aging them. The default value for `inactive_target` is one-quarter of physical memory. This parameter can be changed by modifying `/proc/sys/vm/static_inactive_target`.

- *Out of Memory Killer (OOM)*: The OOM is an algorithm which is invoked when the system senses a potentially fatal shortage of memory. The kernel attempts to kill processes on a 'most good for least harm' basis. This algorithm is only invoked when the system is truly out of memory and swap, and more memory is needed by the kernel to avoid a deadlock situation.
- *VM Killer*: This algorithm is invoked after a critical shortage of memory has been detected. It indiscriminately kills the process whose request for memory was immediate and critical to the process, and where the request was impossible to satisfy. In 2.4.9 and later kernels, this algorithm should never trigger. Such a trigger is considered a kernel bug. (In experimental kernels, the most commonly seen causes of the VM killer invocation are poorly written filesystems and pathological VM implementations.)

Inspecting Process Memory Usage and Tuning the VM

Virtual memory characteristics for individual processes can be found by inspecting the status file underneath the `/proc` directory corresponding to a process's PID.

- `/proc/$PID/status vmSize`: The total size of the process's memory footprint. This includes the text segment, stack, static variables, data segment, and pages which are shared with other processes.
- `/proc/$PID/status vmLck`: The amount of the process's memory which is currently locked by the kernel. Locked memory cannot be swapped out.
- `/proc/$PID/status vmRSS`: The kernel's estimate of the resident set size for this process.
- `/proc/$PID/status vmData`: The amount of memory used for data by the process. It includes static variables and the data segment, but excludes the stack.
- `/proc/$PID/status vmStk`: The amount of memory consumed by the process's stack.
- `/proc/$PID/status vmExe`: The size of the process's executable pages, excluding shared pages.
- `/proc/$PID/status vmLib`: The size of the shared memory pages mapped into the process's address space. This excludes pages shared using System V style IPC.

The following files found beneath `/proc/sys/vm` are tunable by `sysctl.conf` or by simply writing values into the appropriate file.

- `/proc/sys/vm/freepages`: This file contains the values `freepages.min`, `freepages.low` and `freepages.high`.
 - `freepages.high` is the high memory watermark for the kernel. If the number of available pages in the system drops below this value, the system will begin to execute its normal background page eviction algorithm. This may involve swapping pages to disk, evicting clean pages from the system caches, or cleaning dirty pages in the system caches.
 - `freepages.low` is the low memory watermark for the kernel. If the number of available pages in the system drops below this value, the kernel will enter an aggressive swapping mode in an attempt to bring the freepage count back above this value.
 - `freepages.min` is the number of pages reserved for kernel allocation. If the number of allocable pages in the system drops below this value, any applications requesting memory will block. The kernel will then begin to reclaim pages until the amount of free memory in the system is again above `freepages.min`. Use caution when adjusting this value. If the value is set too low, the kernel may

not have enough memory to execute the victim selection and page replacement algorithm, resulting in a deadlock.

- `/proc/sys/vm/min-readahead`: The minimum number of disk blocks to read-ahead when performing file reads.
- `/proc/sys/vm/max-readahead`: The maximum number of disk blocks to read-ahead when performing file reads. If memory is allocable, the kernel will read up to this number of blocks, but a process will not block on a low memory condition solely to perform the read-ahead.

Read-ahead applies only to files being read sequentially. It does not apply to mmaped files or files being read non-sequentially. Initially, the kernel sets the read-ahead window size to `min-readahead`. If subsequent reads are determined to be sequential, the window size is slowly increased, up to `max-readahead`. The kernel tracks a different read-ahead window for each file descriptor.

- `/proc/sys/vm/buffermem`: The values in this file control the amount of memory which the kernel allocates to the buffer cache.
 - `buffermem.min_percent` is the minimum percentage of memory that should ever be allocated to disk buffers.
 - `buffermem.borrow_percent` can indicate to the kernel that the buffer cache should be pruned more aggressively than other system caches. This aggressive pruning will begin when the buffer cache grows above this percentage of memory and the system subsequently comes under memory pressure.
 - `buffermem.max_percent` is the maximum percent of memory the buffer cache will occupy.
- `/proc/sys/vm/pagecache`: The values in this file control the amount of memory which should be used for page cache. The three fields are `pagecache.min_percent`, `pagecache.borrow_percent` and `pagecache.max_percent`, and are defined as in the `buffermem` above.
- `/proc/sys/vm/kswapd`: The values in this file control the kernel swap daemon, **kswapd**.
 - `kswapd.tries_base` controls the number of pages **kswapd** tries to free in one round. Increasing this number can increase throughput between memory and the swap device, at the cost of overall system performance.
 - `kswapd.tries_min` controls the minimum number of pages **kswapd** tries to free each time it is called.
 - `kswapd.swap_cluster` is the number of pages **kswapd** writes to the swap device in one chunk. Larger cluster sizes can increase the throughput on the swap device, but if the number becomes too large, the device request queue will become swamped, slowing overall system performance.
- `/proc/sys/vm/bdflush`: The values in this file control the operation of the buffer cache flushing daemon, **bdflush**. There are nine values in this file. Six are used by 2.4.9 based kernels.
 - `bdflush.nfract` gives the percentage of blocks in the cache which, when dirty, cause the kernel to arouse the **bdflush**.
 - `bdflush.ndirty` is the maximum number of blocks to write out to disk each time **bdflush** is called.
 - `bdflush.nrefill` is the number of buffers the kernel tries to obtain each time the kernel calls **refill()**.

- `bdfflush.nref_dirt` is the number of dirty buffers which will cause `bdfflush` to wake up when we try to refill the buffer list.
- dummy
- `bdfflush.age_buffer` is the maximum amount of time, in seconds, that a dirty buffer may remain in memory without being flushed.
- `bdfflush.age_super` is the maximum amount of time, in seconds, that a dirty superblock may remain in memory without being flushed.
- dummy
- dummy

With these terms and tunable parameters in mind, we now turn to explaining the algorithms controlling Linux VM behavior.

Linux Caching

The motivation behind using caches is to speed access to frequently used data. In early versions of Linux, only disk blocks were cached. The idea has expanded as the Linux kernel has matured. For example, the kernel can now cache pages which are queued up to be written to the swap device. The kernel can also read-ahead when disk block requests are made, on the assumption that the extra data will eventually be requested by some process. Writes are also cached before being written to disk. (And indeed, sometimes information will never hit the disk before being deleted. Common examples are the intermediate files produced by the GNU C compiler. These files often have a very short existence and typically spend their entire lifetime in the cache.)

The system benefits from a judicious use of cache, but when applications or the kernel need memory, kernel caches are the natural choice for memory reclamation.

When a Linux system is not under memory pressure, all of the kernel's caches will slowly grow over time as larger and larger parts of the executing processes' locality are captured in memory. This behavior illustrates a fundamental in Linux design philosophy: Any resource that is otherwise unused should be exploited to minimize the time a process spends executing or waiting in the kernel. It is better for a resource to be allocated and possibly unused than for the resource to be idle. This is often distilled into the phrase *make the common case fast*.

The Slab Cache

Because the kernel does not (and in fact, cannot) use the C standard library, many of the library functions that Unix developers regularly use have been reimplemented within the kernel. These include things like string manipulation routines (such as `strlen()` and `ffs()`), file system and network access routines (such as `fopen()` and `inet_addr()`), and dynamic memory management routines (`malloc()` and `free()`).

Because efficient and fast memory allocation and deallocation routines are so critical to kernel performance, there has been much work done over many years to optimize the performance of these routines. In fact, these routines are considered so critical that many of them have been hand coded in assembly language for each architecture on which Linux runs. For a segment of code which may execute several hundred-thousand times a second, a savings of two or three machine instructions can be substantial.

The original kernel dynamic memory management algorithm was based on the Buddy System, and was implemented by Linus in very early (0.x) versions of Linux. Under the buddy system, free kernel memory is partitioned into lists. Each list contains blocks of a fixed size, ranging from 32 to 131056 bytes. Kernel memory

allocations retrieved blocks from the list appropriate for the requested memory size. As lists were depleted, blocks could be borrowed from other lists and split or coalesced as necessary. Similarly, if a list contained too many blocks, blocks could be split or coalesced and moved to other lists.

The current kernel dynamic memory management algorithm still uses a descendant of the original *buddy system* allocator. But it also contains a much more sophisticated allocator referred to as the *slab cache allocator*. In the 2.4 series of kernels, the buddy system allocator has been subsumed by the slab cache allocator.

In a slab cache allocator, many of the structures which are repeatedly created and destroyed by the kernel (such as inodes, buffer heads, vm_area structs, and dentries) have their own separate caches. New objects are allocated from a dedicated cache, and returned to the same cache when deallocated. The slab cache is a very fast allocator because it minimizes the initialization which must be done when an allocated object is reused, and it avoids the serious twin problems of cache-line contention and cache-line underutilization found in power-of-two allocators like the buddy system.

The slab cache allocator receives its name from the fact that each of the caches it manages is comprised of one or more areas of memory, called *slabs*. Each slab contains one or more objects of the specified type.

The file `/proc/slabinfo` contains information about the slab caches in the system. The file is divided in to two parts. The first part gives information about the slab caches in the system which are dedicated to particular objects. The second part of this file reports on fixed size general purpose caches. Objects which do not have a dedicated slab cache receive allocations from these general purpose caches.

Each line in `/proc/slabinfo` contains six fields. Machines running SMP kernels will contain an additional two fields per line. The fields are defined as follows:

- **cname**: The name of the cache. The cache name is typically comprised of the name of the object served by the cache, followed by the string `_cache`, although the kernel does not enforce this naming convention.
- **num-active-objs**: The number of objects which have been allocated from this cache and which have not yet been released by the kernel.
- **total-objs**: the total number of objects this cache can allocate before it must grow itself.
- **object-size**: the size of the object, in bytes.
- **num-active-slabs**: the number of active slabs in this cache.
- **total-slabs**: the total number of slabs in this cache.
- **num-pages-per-slab**: the number of pages needed to store a single slab in this cache.

On SMP systems, the kernel maintains per-CPU slab caches in addition to the global cache. These per-CPU caches eliminate most cache contention among processes, at the cost of an increase in allocator overhead. The file `/proc/slabinfo` contains two additional fields concerning the per-CPU caches on SMP systems.

- **batchcount**: the number of objects that get transferred between the global pool and the per-CPU pools in one transaction when the per-CPU pools grow or shrink.
- **limit**: the maximum number of unused objects in a per-CPU pool before the pool starts returning memory to the global pool.

Zones

Early versions of Linux mapped all physical memory into the kernel's address space. Because of the limited size of the x86 address bus, early Linux systems were restricted to accessing 1GB of physical memory.

With the introduction of the 2.4 kernel series, Linux is able to take advantage of the Intel *Physical Address Extension* (PAE) hardware available on high-end workstations and servers. This hardware allows the kernel to access up to 64GB of physical memory, but it requires that different areas of physical memory be treated differently by the kernel.

The kernel divides the x86 address space into three areas, called *zones*. Memory in the 0 to 16MB-1 physical address range falls into the ISA DMA zone. Older I/O cards and some modern cards with unusual hardware limitations must do all of their DMA I/O from this zone. (Common examples of modern cards which must do DMA I/O from the ISA zone include some PCI sound cards.)

Memory from 16MB to just slightly under 1GB is referred to as the *normal kernel zone*. Memory in this zone can be used by the kernel for any purpose other than ISA DMA.

Memory above 1GB is referred to as the *high memory* (or *himem*) zone. Memory in this zone can be used by userspace applications. The kernel can also make limited use of this memory. In stock 2.4.x kernels, the kernel can only store pages from the page cache in himem.

One of the largest potential causes of deadlock in a himem enabled kernel is caused by a condition known as *zone memory imbalance*. This condition occurs when a particular memory zone consumes almost all of its free memory, but adequate physical memory in other zones indicate to the kernel that it can safely increase the system workload. The system itself may have plenty of free memory, but that memory is not suitable for some task which the kernel is required to perform.

A common situation seen during the development stage of the Advanced Server kernel occurred on machines with large amounts of physical RAM running processes with large memory footprints. These processes had plenty of memory in which to execute, but the processes' pages tables would not fit into the normal kernel zone. The situation resulted in either livelock or deadlock, depending upon the exact job mix presented to the machine. It was this observation which led the Red Hat kernel development team, along with members of the open source community, to develop a patch which allowed the kernel to migrate pagetables to himem if necessary.

The Red Hat Linux Advanced Server kernel also adds the ability to do *per-zone balancing*. In per-zone balancing, the kernel can move pages between physical memory zones in order to better balance used and free space in each zone. In addition, the Red Hat Linux Advanced Server kernel implements per zone lists. With per zone lists, each memory zone contains separate active and inactive lists, which help support the per zone balancing algorithm.

Questions about the Linux Virtual Memory Management

- *How is memory allocated to a process?*

The typical method for an application to allocate memory is via one of the standard library dynamic memory allocation calls (**malloc(3)**, **calloc(3)**, and others). When one of these functions is invoked, the allocation is handled by the standard library. If the allocation succeeds, the process may receive memory which it had previously used and returned to the allocator, or the allocation request may have caused the allocator to request additional memory from the kernel via the **brk()** system call.

When the allocator requests memory from the kernel, the kernel creates the page table entries for the requested pages. All entries initially point to a special page in

the kernel called the `ZERO_PAGE`. Each entry is then marked as copy-on-write, and the associated page is referred to as a COW page. Only when the process actually writes to these pages are new physical page frames allocated for pages.

Thus, for a particular process, an allocation may succeed, but when the process begins to write to the newly allocated pages, memory pressure may lead to delays as the kernel tries to find physical memory in which to place the newly allocated pages.

- *How is data chosen to be dropped from the cache(s) and/or moved out to swap?*

The page replacement policy in the Red Hat Linux 2.4.18 kernel is based on a modified LRU algorithm. The two main LRU lists maintained by the kernel are the active and inactive page lists. Pages on the active list are scanned and aged up or down according to their referenced bits. The referenced bits include those in the page tables (in the case of a mapped page), and the page's referenced bit (which is set when the page cache is accessed, or any buffer cache entries on the page are accessed). Once a page has not been accessed for a period of time, the age of the page hits zero and the page is moved from the active list to the inactive list. Memory is then reclaimed from the inactive list and released into the free pool.

In the 2.4.9-e kernels, the kernel scans page tables as a separate step from scanning the active list. This scanning is done using the traditional Unix clock algorithm. Pages that have not been accessed since the last scan and have an age of 0 are swapped out. The rest of the process is similar to 2.4.18.

- *What is the difference in the way the kernel treats mmaped pages, disk blocks, text pages, shared text pages, etc., when evicting pages from memory?*

When memory is reclaimed from the LRU lists, all pages are treated identically. Any differences in reclaim are the result of access causing the page's age to increase, and subsequently remain on the active list for a longer period of time.

In the 2.4.9-e kernel, pages mapped into a process's address space must be evicted from all pages tables which reference them before they are candidates for reclamation. This is because a single physical page may be mapped into the address space of multiple processes. For example, the pages from the standard library are mapped into the address space of almost every userspace process. The 2.4.18 kernel simplifies this process by removing a page from all page tables at the same time, thus avoiding a complete second pass of the clock algorithm over the page tables.

- *What do the terms major page fault and minor page fault mean?*

A major page fault occurs when a process must access the disk or swap device to satisfy the page request. A minor page fault indicates that, although the page table entry for a page was not valid, the page request can be satisfied without accessing the disk. This can occur when, for example, the page was found in the swap cache, or when a new page is created and the page table entry points to the `ZERO_PAGE`.

- *What is the overhead, in terms of space and time, of using a PAE enabled kernel as compared to an SMP kernel?*

PAE doubles the size of page table entries from 32 bits to 64 bits, as well as adding a small third level to the page tables. This means the maximum amount of kernel memory consumed by page tables per process is doubled to slightly more than 6MB. In the 2.4.18 kernels shipped by Red Hat, this memory comes out of the ~700MB of available memory in the normal kernel zone. In the Red Hat Linux Advanced Server series of kernels, page tables can be located anywhere in physical memory.

The performance impact is highly workload dependent, but on a fairly typical kernel compile, the PAE penalty works out to be around a 1% performance hit on Red Hat's test boxes. Testing with various other workload mixes has given performance hits ranging from 0% to 10%.

- *What symptoms are there when the system is coming under memory pressure? What tools can be used to monitor the system to watch for this?*

The **vmstat(8)** program can be used to monitor system virtual memory activity. Among other things, it shows the amount of space allocated on the swap device, the number of unused pages in RAM, the sizes of the buffer and page cache, and the number of blocks swapped in and out over a given period of time.

When both free and swap free are very low the OOM algorithm will be invoked in an attempt to keep everything except the killed process(es) running.

The 2.4.18 kernels provided by Red Hat contain a line tagged as `Committed_AS` in `/proc/meminfo`. This line indicates the amount of memory which applications have allocated. When this number exceeds the amount of installed RAM, you should assume that the system will eventually come under memory pressure. When this number is larger than the total of RAM and swap on the system, you should assume that the system may eventually consume all memory resources and thus may need to invoke the OOM killer.

- *Why is my system using swap if I have memory free?*

In the 2.4.9.e kernels, processes reserve space on the swap device before their constituent pages are written out. After the reservation takes place, the kernel may determine that no swapping is in fact needed. These pages still show up in the `swap used` entry in the output of **top(1)**, however. To determine actual swapping activity, the system administrator should consult the `si` and `so` columns in the output of **vmstat(8)**.

In addition, once the kernel begins to sense memory pressure, it may preemptively wake up `kswapd` and begin low priority swapping of inactive pages in an attempt to free pages before the memory situation becomes critical. If pages are swapped out, they will not be swapped back in until they are referenced, even if there is sufficient physical memory to hold them.

Finally, when a page is written to swap, the swap space it occupies is not recovered until swapspace itself is nearly full. This is done because a page that is swapped out and then back in might not be written to again. Keeping a copy on the swap disk allows us to avoid writing the page out a second time.