Understanding Virtual Memory In Red Hat Enterprise Linux 3

Norm Murray and Neil Horman

Version 1.6

December 13, 2005

Contents

List of Figures

1 Introduction

One of the most important aspects of an operating system is the Virtual Memory Management system. Virtual Memory (VM) allows an operating system to perform many of its advanced functions, such as process isolation, file caching, and swapping. As such, it is imperative that an administrator understand the functions and tunable parameters of an operating system's virtual memory manager so that optimal performance for a given workload may be achieved. This article is intended to provide a system administrator a general overview of how a VM works, specifically the VM implemented in Red Hat Enterprise Linux 3 (RHEL3). After reading this document, the reader should have a rudimentary understanding of the data the RHEL3 VM controls and the algorithms it uses. Further, the reader should have a fairly good understanding of general Linux VM tuning techniques. It is important to note that Linux as an operating system has a proud legacy of overhaul. Items which no longer server useful purposes or which have better implementations as technology advances are phased out. This implies that the tuning parameters described in this article may be out of date if you are using a newer or older kernel. Fear not however! With a well grounded understanding of the general mechanics of a VM, it is fairly easy to convert ones knowledge of VM tuning to another VM. The same general principles apply, and documentation for a given kernel (including its specific tunable parameters), can be found in the corresponding kernel source tree under the file Documentation/sysctl/vm.txt.

2 Definitions

To properly understand how a Virtual Memory Manager does its job, it helps to understand what components comprise a VM. While the low level details of a VM are overwhelming for most, a high level view is nonetheless helpful in understanding how a VM works, and how it can be optimized for various workloads. A high level overview of the components that make up a Virtual memory manager is presented in Figure 1 below:

2.1 What Comprises a VM

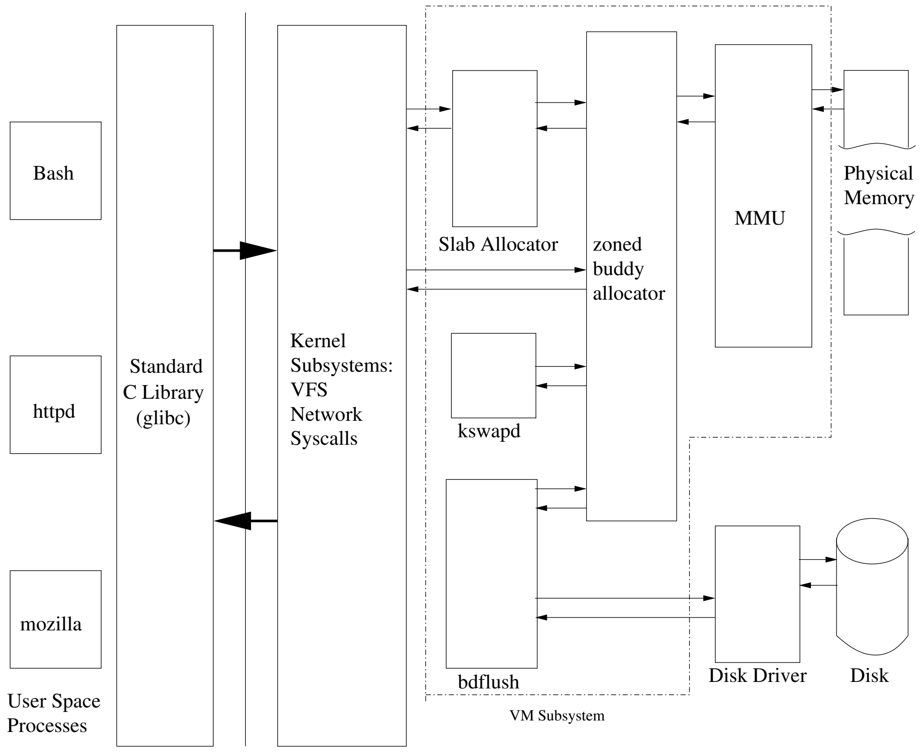

Figure 1: High level overview of VM subsystem

While a VM is actually far more complicated than illustrated in Figure 1, the high level function of the system is accurate. The following sections describe each of the listed components in the VM:

2.2 MMU

The Memory Management Unit (MMU) is the hardware base that make a Virtual Memory system possible. The MMU allows software to reference physical memory by aliased addresses, quite often more than one. It accomplishes this through the use of *pages* and *page tables*. The MMU uses a section of memory to translate virtual addresses into physical addresses via a series of table lookups. Various processor architectures preform this function is slightly different ways, but in general figure 2 illustrates how a translation is preformed from a virtual address to a physical address:

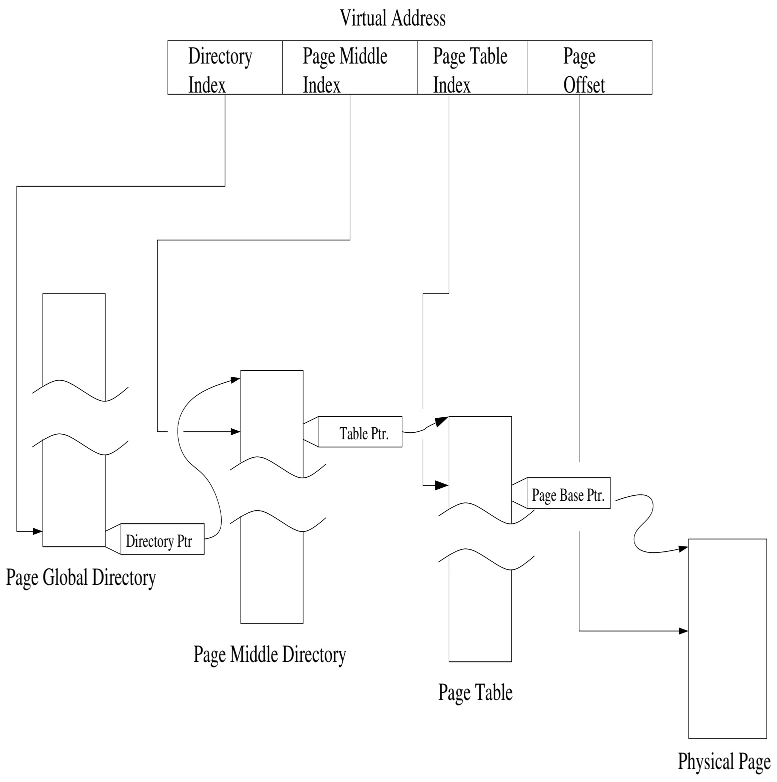

Figure 2: Illustration of a virtual to physical memory translation

Each table lookup provides a pointer to the base of the next table, as well as a set of extra bits which provide auxiliary data regarding that page or set of pages. This information typically includes the current page status, access privileges, and size. A separate portion of the virtual address being accessed provides an index into each table in the lookup process. The final table provides a pointer to the start of the physical page corresponding to the virtual address in RAM, while the last field in the virtual address selects the actual word in the page being accessed. Any one of the table lookups during this translation, may direct the lookup operation to terminate and drive the operating system to preform another action. Some of these actions are somewhat observable at a system level, and have common names or references

- **Segmentation Violation** - A user space process requests a virtual address, and during the translation the kernel is interrupted and informed that the requested translation has resulted in a page which it has not allocated, or which the process does not have permission to access. The kernel responds by signaling the process that it has attempted to access an invalid memory region, after which it is terminated.

- **Swapped out** - During a translation of an address from a user space process, the kernel was interrupted and informed that the page table entry lists the page as accessible, but not present in RAM. The kernel interprets this to mean that the requested address is in a page which has been swapped to disk. The user process requesting the address is put to sleep and an I/O operation is started to retrieve the page.

2.3 Zoned Buddy Allocator

The Zoned Buddy Allocator is responsible for the management of page allocations to the entire system. This code manages lists of *physically contiguous* pages and maps them into the MMU page tables, so as to provide other kernel subsystems with valid *physical* address ranges when the kernel requests them (Physical to Virtual Address mapping is handled by a higher layer of the VM and is collapsed into the kernel subsystems block of Figure 1). The name Buddy Allocator is derived from the algorithm this subsystem uses to maintain it free page lists. All physical pages in RAM are cataloged by the buddy allocator and grouped into lists. Each list represents clusters of 2^n pages, where n is incremented in each list. There is a list of single pages, a list of 2 page clusters, a list of 4 page cluster, and so on. When a request comes in for an amount of memory, that value is rounded up to the nearest power of 2, and a entry is removed from the appropriate list, registered in the page tables of the MMU and a corresponding physical address is returned to the caller, which is then mapped into a virtual address for kernel use. If no entries exist on the requested list, an entry from the next list up is broken into two separate clusters, and 1 is returned to the caller while the other is added to the next list down. When an allocation is returned to the buddy allocator, the reverse process happens. The allocation is returned to the requisite list, and the list is then examined to determine if a larger cluster can be made from the existing entries on the list which was just updated. This algorithm is advantageous in that it *automatically returns pages to the highest order free list possible.* That is to say, as allocations are returned to the free pool, they automatically form larger clusters, so that when a need arises for a large amount of physically contiguous memory (i.e. for a DMA operation), it is more likely that the request can be satisfied. Note that the buddy allocator allocates memory in page multiples only. Other subsystems are responsible for finer grained control over allocation size. For more information regarding the finer details of a buddy allocator, refer to [1]. Note that the Buddy allocator also manages memory *zones*, which define pools of memory which have different purposes. Currently there are three memory pools which the buddy allocator manages accesses for:

- **DMA** - This zone consists of the first 16 MB of RAM, from which legacy devices allocate to perform direct memory operations
- **NORMAL** - This zone encompasses memory addresses from 16 MB to 1 GB[1] and is used by the kernel for internal data structures, as well as other system and user space allocations.
- **HIGHMEM** - This zone includes all memory above 1 GB and is used exclusively for system allocations (file system buffers, user space allocations, etc).

2.4 Slab Allocator

The Slab Allocator provides a more usable front end to the Buddy Allocator for those sections of the kernel which require memory in sizes that are more flexible than the standard 4 KB page. The Slab Allocator allows other kernel components to create *caches* of memory objects of a given size. The Slab Allocator is responsible for placing as many of the caches objects on a page as possible and monitoring which objects are free and which are allocated. When allocations are requested and no more are available, the Slab Allocator requests more pages from the Buddy Allocator to satisfy the request. This allows kernel components to use memory in a much simpler way. This way components which make use of many small portions of memory are not required to individually implement memory management code so that too many pages are not wasted[2].

2.5 Kernel Threads

The last component in the VM subsystem are the active tasks: kscand, kswapd, kupdated, and bdflush. These tasks are responsible for the recovery and management of in use memory. All pages of memory have an associated state (for more info on the memory state machine, refer to section 3). Freshly allocated memory initially enters the *active* state. As the page sits in RAM, it is periodically visited by the kscand task, which marks the page as being *inactive*. If the page is subsequently accessed again by a task other than kscand, it is returned to the active state. If a page is modified during its allocation it is additionally marked as being dirty. The kswapd task scan pages periodically in much the same way, except that when it finds a page which is inactive it considers it a candidate to be returned to the free pool of pages for later allocation. If kswapd selects a page for freeing, it examines its dirty bit. In the event the dirty bit

[1]The hugemem kernel extends this zone to 3.9 GB of space

[2]The Slab Allocator may only allocate from the DMA and NORMAL zones

is set, it locks the page, and initiates a disk I/O operation to move the page to swap space on the hard drive. When the I/O operation is complete, kswapd modifies the page table entry to indicate that the page has been migrated to disk, unlocks the page and places it back on the free list, available for further allocations.

2.6 Components that use the VM

It is worth mentioning here the remaining components which sit on top of the VM subsystem. These components actively use the VM to acquire memory and presents it to users, providing the overall 'feel' of the system:

- **Network Stack -** The Network Stack is responsible for the management of network buffers being received and sent out of the various network interfaces in a system
- **Standard C Library -** Via various system calls the standard C library manages pages of virtual memory and presents user applications with a an API allowing fine grained memory control
- **Virtual File System -** The Virtual file system buffers data from disks for more rapid file access, and holds pages containing file data which has been memory mapped by an application

3 The Life of A Page

All of the memory managed by the VM is labeled by a *state*. These states help let the VM know what to do with a given page under various circumstances. Dependent on the current needs of the system, the VM may transfer pages from one state to the next, according to the state machine diagrammed in figure 3 below: Using these states, the VM can determine what is being done with a page by the system at a given time and what actions it (the VM) may take on the page. The states that have particular meanings are as follows:

- **FREE** - All pages available for allocation begin in this state. This indicates to the VM that the page is not being used for any purpose and is available for allocation.
- **ACTIVE** - Pages which have been allocated from the Buddy Allocator enter this state. It indicates to the VM that the page has been allocated and is actively in use by the kernel or a user process.
- **INACTIVE DIRTY** - Th state indicates that the page has fallen into disuse by the entity which allocated it, and as such is a candidate for removal from main memory. The kscand task periodically sweeps through all the pages in memory Taking note of the amount of time the page has been in memory since it was last accessed. If kscand finds that a page has been accessed since it last visited the page, it increments the pages age counter, otherwise, it decrements that counter. If kscand happens on a page which has its age counter at zero, then the page is moved to the inactive dirty state. Pages in the inactive dirty state are kept in a list of pages to be laundered.
- **INACTIVE LAUNDERED** - This is an interim state in which those pages which have been selected for removal from main memory enter while their contents are being moved to disk. Only pages which were in the inactive dirty state enter this state. When the disk I/O operation is complete the page is moved to the inactive clean state, where it may be deallocated, or overwritten for another purpose. If, during the disk operation, the page is accessed, the page is moved back into the active state.
- **INACTIVE CLEAN** - Pages in this state have been laundered. This means that the contents of the page are in sync with they backing data on disk. As such they may be deallocated by the VM or overwritten for other purposes.

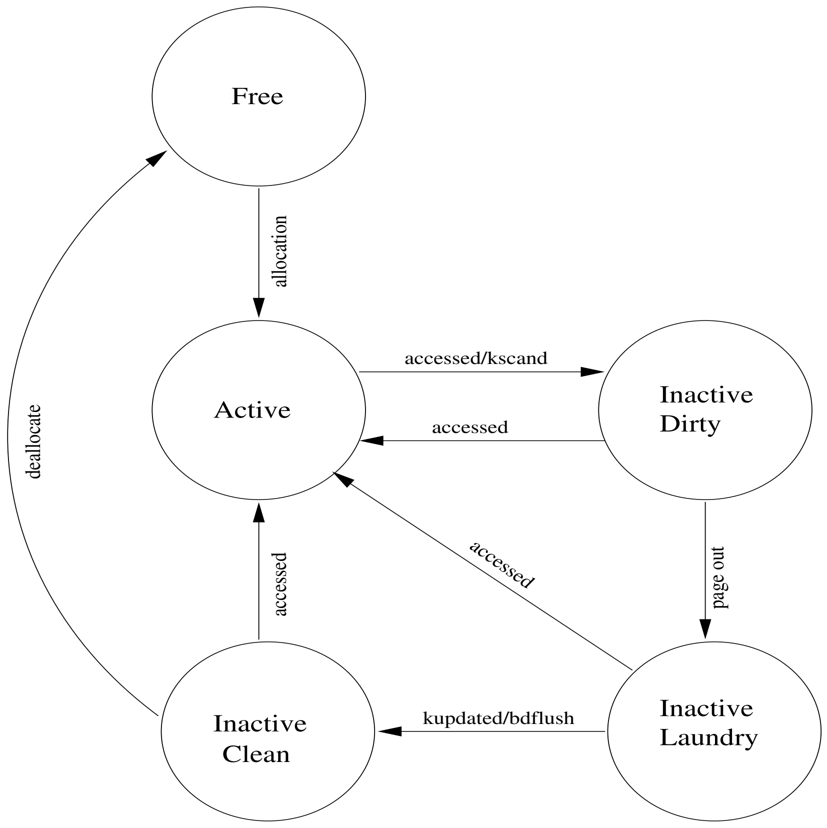

Figure 3: Diagram of the VM page state machine

4 Tuning the VM

Now that the picture of the VM mechanism is sufficiently illustrated, how is it adjusted to fit certain workloads? There are two methods for changing tunable parameters in the Linux VM. The first is the sysctl interface. The sysctl interface is a programming oriented interface, which allows software programs to directly modify various tunable parameters. The sysctl interface is exported to system administrators via the sysctl utility, which allows an administrator to specify a specific value for any of the tunable VM parameters on the command line, as in the following example:

```
sysctl -w vm.max_map_count=65535
```

The sysctl utility also supports the use of a configuration file (/etc/sysctl.conf), in which all the desirable changes to a VM can be recorded for a system and restored after a restart of the operating system, making this access method suitable for long term changes to a system VM. The file is straightforward in its layout, using simple key-value pairs, with comments for clarity, as in the following example:

```
#Adjust the min and max read-ahead for files vm.max-readahead=64
vm.min-readahead=32

#turn on memory over-commit vm.overcommit_memory=2

#bump up the percentage of memory in use to #activate bdflush
vm.bdflush="40 500 0 0 500 3000 60 20 0"
```

The second method of modifying VM tunable parameters is via the proc file system. This method exports every group of VM tunables as a file, accessible via all the common Linux utilities used for modify file contents. The VM tunables are available in the directory /proc/sys/vm, and are most commonly read and modified using the Linux cat and echo utilities, as in the following example:

```
# cat kswapd
512 32 8

echo 511 31 7 > /proc/sys/vm/kswapd

# cat kswapd
511 31 7
```

The proc filesystem interface is a convenient method for making adjustments to the VM while attempting to isolate the peak performance of a system. For convenience, the following sections list the VM tunable parameters as the filenames they are exported as in /proc/sys/vm. Please note that unless otherwise noted, these tunables apply to the RHEL3 2.4.21-4 kernel.

4.1 bdflush

The bdflush file contains 9 parameters, of which 6 are tunable. These parameters affect the rate at which pages in the buffer cache[3] are freed and returned to disk. by adjusting the various values in this file, a system can be tuned to achieve better performance in environments where large amounts of file I/O are preformed. The following parameters are defined in the order they appear in the file

Parameter	**Description**
nfract	The percentage of of dirty pages in the buffer cache required to activate the bdflush task
ndirty	The maximum number of dirty pages in the buffer cache to write to disk in each bdflush execution
reserved1	Reserved for future use
reserved2	Reserved for future use
interval	the number of jiffies (10ms periods) to delay between bdflush iterations
age_buffer	The time for a normal buffer to age before it is considered for flushing back to disk
nfract_sync	The percentage of dirty pages in the buffer cache required to cause the tasks which are dirtying pages of memory start writing those pages to disk themselves, slowing the dirtying process
nfract_stop_bdflush	The percentage of dirty pages in buffer cache required to allow bdflush to return to idle state
reserved3	Reserved for future use

Generally, systems that require more free memory for application allocation will want to set these values higher (except for the age buffer, which would be moved lower), so that file data is sent to disk more frequently, and in greater volume, thus freeing up pages of ram for application use. This of course comes at the expense of CPU cycles, as the system processor spends more time moving data to disk, and less time running applications. Conversely, systems which are required to preform large amounts of I/O would want to do the opposite to these values, allowing more ram to be used to cache disk file, so that file access is faster.

4.2 dcache_priority

This file controls the bias of the priority for caching directory contents. When the system is under stress, it will selectively reduce the size of various file system caches in an effort to reclaim memory. By adjusting this value up, memory reclamation bias is shifted away from the dirent cache. By reducing this amount bias is shifted toward reclaiming dirent memory. This is not a particularly useful tuning parameter, but it can be helpful in maintaining the interactive response

[3]The subset of pagecache which stores files in memory

time on an otherwise heavily loaded system. If you experience intolerable delays in communicating with your system when it is busy preforming other work, increasing this parameter may help you.

4.3 hugetlb_pool

This file is responsible for recording the number of megabytes used for 'huge' pages. Huge pages are just like regular pages in the VM, only they are an order of magnitude larger[4]. Hugepages are both beneficial and detrimental to a system. They are helpful in that each hugepage takes only one set of entries in the VM page tables, which allows for a higher degree of virtual address caching in the TLB[5] and a requisite performance improvement. On the downside, they are very large and can be wasteful of memory resources for those applications which do not need large amounts of memory. Some applications however, do require large amounts of memory and if they are written to be aware of them, can make good use of hugepages. If a system run applications which requires large amounts of memory and is aware of this feature, then it is advantageous to increase this value to an amount satisfactory to that application or set of applications.

4.4 inactive_clean_percent

This control specifies the minimum percentage of pages in each page zone that must be in the clean or laundered state. If any zone drops below this threshold, and the system is under pressure for more memory, then that zone will begin having its inactive dirty pages laundered. Note that this control is only available on the 2.4.21-5EL kernels forward. Raising the value for the corresponding zone which is memory starved will cause pages to be paged out more quickly, eliminating memory starvation, at the expense of CPU clock cycles. Lowering this number will allow more data to remain in ram, increasing the system performance, but at the risk of memory starvation.

4.5 kswapd

While this set of parameters previously defined how frequently and in what volume a system moved non buffer cache pages to disk, in Red Hat Enterprise Linux 3, these controls are unused.

4.6 max_map_count

This file allows for the restriction of the number of VMAs[6] that a particular process can own. A Virtual memory area is a contiguous area of virtual address

[4]Note also that hugepages are not swappable

[5]Translation Look-aside Buffer: A device which caches virtual address translations for faster lookups

[6]Virtual Memory Areas

space. These areas are created during the life of the process when the program attempts to memory map a file, link to a shared memory segment, or simply allocates heap space. Tuning this value limits the amount of these VMA's that a process can own. Limiting the amount of VMA's a process can own can lead to problematic application behavior, as the system will return out of memory errors when a process reaches its VMA limit, but can free up lowmem for other kernel uses. If your system is running low on memory in the ZONE_NORMAL zone, then lowering this value will help free up memory for kernel use.

4.7 max-readahead

This tunable affects how early the Linux VFS[7] will fetch the next block of a file from memory. File readahead values are determined on a per file basis in the VFS and are adjusted based on the behavior of the application accessing the file. Anytime the current position being read in a file plus the current read ahead value results in the file pointer pointing to the next block in the file, that block will be fetched from disk. By raising this value, the Linux kernel will allow the readahead value to grow larger, resulting in more blocks being prefetched from disks which predictably access files in uniform linear fashion. This can result in performance improvements, but can also result in excess (and often unnecessary) memory usage. Lowering this value has the opposite affect. By forcing readaheads to be less aggressive, memory may be conserved at a potential performance impact.

4.8 min-readahead

Like max_readahead, min-readahead places a floor on the readahead value. Raising this number forces a files readahead value to be unconditionally higher, which can bring about performance improvements, provided that all file access in the system is predictably linear from the start to the end of a file. This of course results in higher memory usage from the pagecache. Conversely, lowering this value, allows the kernel to conserve pagecache memory, at a potential performance cost.

4.9 overcommit_memory

Overcommit_memory is a value which sets the general kernel policy toward granting memory allocations. If the value in this file is 0, then the kernel will check to see if there is enough memory free to grant a memory request to a malloc call from an application. If there is enough memory then the request is granted. Otherwise it is denied and an error code is returned to the application. If the setting in this file is 1, the kernel will allow all memory allocations, regardless of the current memory allocation state. If the value is set to 2, then the kernel will grant allocations above the amount of physical ram and swap in the system, as

[7] Virtual file System

defined by the overcommit_ratio value (defined below). Enabling this feature can be somewhat helpful in environments which allocate large amounts of memory expecting worst case scenarios, but do not use it all.

4.10 overcommit_ratio

This tunable defines the amount by which the kernel will overextend its memory resources, in the event that overcommit_memory is set to the value 2. The value in this file represents a percentage which will be added to the amount of actual ram in a system when considering whether to grant a particular memory request. For instance, if this value was set to 50, then the kernel would treat a system with 1GB of ram and 1GB of swap as a system with 2.5GB of allocatable memory when considering weather to grant a malloc request from an application. The general formula for this tunable is:

$$memory_{allocatable} = (sizeof_{swap} + (sizeof_{ram} * overcommit_{ratio}))$$

Use these previous two parameters with caution. Enabling memory overcommit can create significant performance gains at little cost, but only if your applications are suited to its use. If your applications use all of the memory they allocate, memory overcommit can lead to short performance gains followed by long latencies as your applications are swapped out to disk frequently when they must compete for oversubscribed ram. Also ensure that you have at least enough swap space to cover the overallocation of ram (meaning that your swap space should be <u>at least</u> big enough to handle the percentage if overcommit, in addition to the regular 50 percent of ram that is normally recommended).

4.11 pagecache

The pagecache file adjusts the amount of ram which can be used by the pagecache. The pagecache holds various pieces of data, such as open files from disk, memory mapped files and pages of executable programs. Modifying the values in this file dictates how much of memory is used for this purpose

Parameter	**Description**
min	The minimum amount of memory to reserve for pagecache use
borrow	kswapd balances the reclaiming of pagecache pages and process memory to reach this percentage of pagecache pages
max	If more memory than this percentage is taken by the pagecache, kswapd will only evict pages from the pagecache. Once the amount of memory in pagecache is below this threshold, then kswapd will again allow itself to move process pages to swap.

Adjusting these values upward allows more programs and cached files to stay in memory longer, thereby allowing applications to execute more quickly. On memory starved systems however, this may lead to application delays, as processes must wait for memory to become available. Moving these values downward swaps processes and other disk-backed data out more quickly, allowing for other processes to obtain memory more easily, increasing execution speed. For most workloads the automatic tuning works fine. However, if your workload suffers from excessive swapping and a large cache, you may want to reduce the values until the swapping problem goes away

4.12 page-cluster

The kernel will attempt to read multiple pages from disk on a page fault, in order to avoid excessive seeks on the hard drive. This parameter defines the number of pages the kernel will try to read from memory during each page fault. The value is interpreted as $2^{page-cluster}$ pages for each page fault. A page fault is encountered every time a virtual memory address is accessed for which there is not yet a corresponding physical page assigned, or for which the corresponding physical page has been swapped to disk. If the memory address has been requested in a valid way (i.e. the application contains the address in its virtual memory map) then the kernel will associate a page of ram with the address, or retrieve the page from disk and place it back in ram, and restart the application from where it left off. By increasing the page-cluster value, pages subsequent to the requested page will also be retrieved, meaning that if the workload of a particular system accesses data in ram in a linear fashion, increasing this parameter can provide significant performance gains (much like the file read-ahead parameters described earlier). Of course if your workload accesses data discreetly in many separate areas of memory, then this can just as easily cause performance degradation.

4.13 kscand_work_percent

This parameter was introduced in Red Hat Enterprise Linux version 3 Update 6. This corresponds to kernel versions 2.4.21-37 and later. This parameter controlls how much of the active page list should be aged each time kscand does work. It defaults to 100 (percent) of the pages on the active page list. Since many memory operations are not possible while the pages on the active list are being aged the system may appear to be 'hung' until kscand finishes its job. This will only be apparent on systems with very large amounts of memory with most of that memory in use. An example could be a very large database which can have gigs of active memory. By tuning this parameter down it is possible to age less than all of the pages at once thus reducing the length of time that work cannot be accomplished. However kscand will run more often to make up for the lost work. As an example by tuning this to 50 we will only age half of the pages thus reducing the pause seen by the user by half. But, we will run kscand twice thus doubling the total number of pauses. Remember the system must pause for the entire time it takes to walk the active list and this parameter will not significantly change that total time.

4.14 oom-kill

In kernel versions 2.4.21-37 and later a new parameter call oom-kill will be found. This parameter controls the maximum number of processes that bave been killed due to an out of memory situation but have not terminated yet. The default is 1 process at a time can be in the state of terminating after being OOM killed. In RARE situations it may be appropriete to set this value to 0. A value of 0 will disable the OOM killer entirely. At first glance this might seem good to users who are experiencing OOM kills. But remember that the OOM killer only kills a process when the system is very close to becoming completely and totally unusable. If the machine actually did exhaust all pages of memory it would be unable to complete ANY new work. The system will just hang and be absolutely unrecoverable. An example of when this might be reasonbly disabled is in a compute job or scientific application where it is known that almost every bit of memory will be used but all processes running can be absolutly trusted to never use it all. An example when tuning this parameter up might be appropriete is when you often run into OOM situations and you also are known to have may processes spend lots of time in uninteruptible wait states (like never returning IO) since they won't be able to handle a kill signal you might have better chances of successfully killing a process and saving the system from a true out of memory state. This parameter should rarely be tuned.

4.15 numa_memory_allocator

The new sfsctl tunable numa_memory_allocator is introduced in Red Hat Enterprise Linux version 3 Update 7. This parameter controls memory allocation policy for x86_64 systems. By default a numa system will choose the local node to allocate memory from and totally exhaust all of the memory from that node before deciding to allocate memory from other remote numa nodes. While this results in holding an object that will fit in a numa node within a single node, it can also result in swapping on one node while there is plenty of free memory on other nodes. Changing the numa_memory_allocator to 1 will prevent total memory exhaustion on the local node before allocating memory from other remode nodes. While this might result in objects not properly fitting in a node, it will also prevent the exhaustion from one node and the resulting swapping while there is plenty of memory on other nodes.

5 Example Scenarios

Now that we have covered the details of kernel tuning, let's look at some example workloads and the various tuning parameters that may improve system performance.

5.1 File (IMAP, Web, etc.) Server

This workload is geared towards performing a large amount of I/O to and from the local disk, thus benefiting from an adjustment allowing more files to be maintained in RAM. This would speed up I/O by caching more files in RAM and eliminating the need to wait for disk I/O to complete. A simple change to sysctl.conf as follows usually benefits this workload:

```
#increase the amount of RAM pagecache is allowed to use
#before we start moving it back to disk
vm.pagecache="10 40 100"
```

5.2 General Compute Server With Many Active Users

This workload is a very general kind of configuration. It involves many active users who will likely run many processes, all of which may or may not be CPU intensive or I/O intensive or a combination thereof. As the default VM configuration attempts to find a balance between I/O and process memory usage, it may be best to leave most configuration settings alone in this case. However, this environment will also likely contain many small processes, which regardless of workload, consume memory resourses, particularly lowmem. It may help, therefore, to tune the VM to conserve low memory resources when possible:

```
#lower the pagecache max to keep from eating all our memory up
with cache
vm.pagecache=10 25 50

#lower max_readahead to reduce the amount of unneeded IO
vm.max-readahead=16
```

5.3 Non interactive (Batch) Computing Server

A batch computing server is usually the exact opposite of a file server. Applications run without human interaction, and they commonly perform very little I/O. The number of processes running on the system can be very closely controlled. Consequently this system should be tuned exclusively to allow for maximum throughput:

```
#Reduce the amount of pagecache normally allowed
vm.pagecache="1 10 100"

#don't worry about conserving lowmem, not that many processes
vm.max_map_count=128000
```

```
#crank up overcommit, processes can sleep as they are not interac-
tive
vm.overcommit=2
vm.overcommit_ratio=75
```

References

[1] Bovet, Daniel & Cesati, Marco, Understanding the Linux Kernel. *Oreilly & Associates*. Sebastopol, CA, 2001.

[2] Matthews, Bob & Murray, Norm
Virtual Memory Behavior in Red Hat Linux A.S. 2.1.
Red Hat whitepaper, Raleigh, NC, 2001.

[3] Van Riel, Rik Towards an O(1) VM.2003.
http://surriel.com/lectures/ols2003/.

[4] Various. The Linux Kernel Source Tree. Versions
2.4.21-4EL & 2.4.21-5EL.
http://www.kernel.org/pub/linux/kernel/v2.4/linux-2.4.21.tar.bz2
Red Hat, Inc. 2003