Understanding Virtual Memory In Red Hat Enterprise Linux 4

Neil Horman

Version 0.1 - DRAFT EDIT

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 4 (RHEL4). After reading this document, the reader should have a rudimentary understanding of the data the RHEL4 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 serve 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 This implies that the tuning parameters described in this article may be out of date if you are using a newer or older kernel. This is particularly true of this release of Red Hat Enterprise Linux, as it is the first RHEL release making use of the 2.6 kernel series. 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

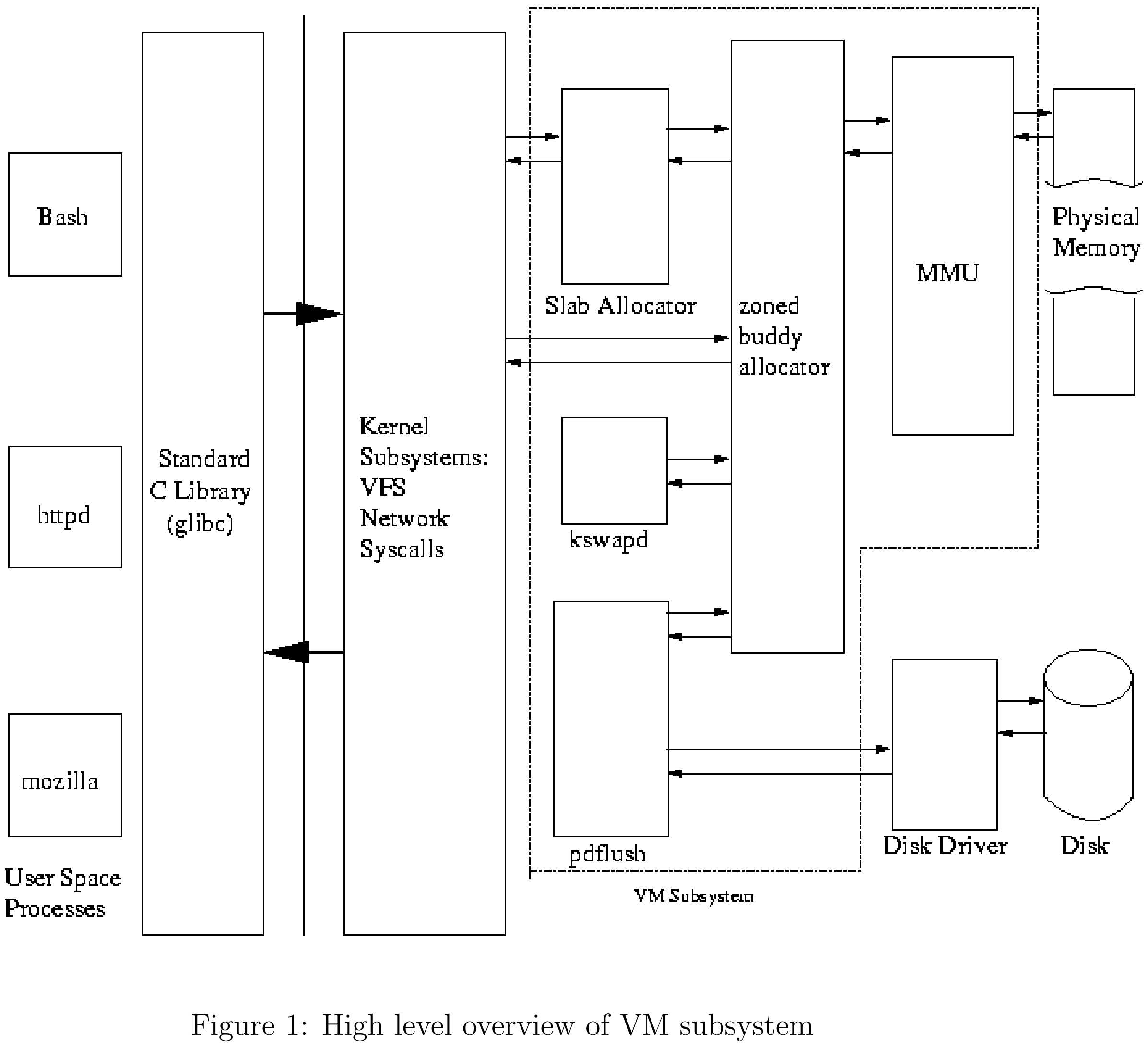

Figure 1: High level overview of VM subsystem

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: [1]

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:

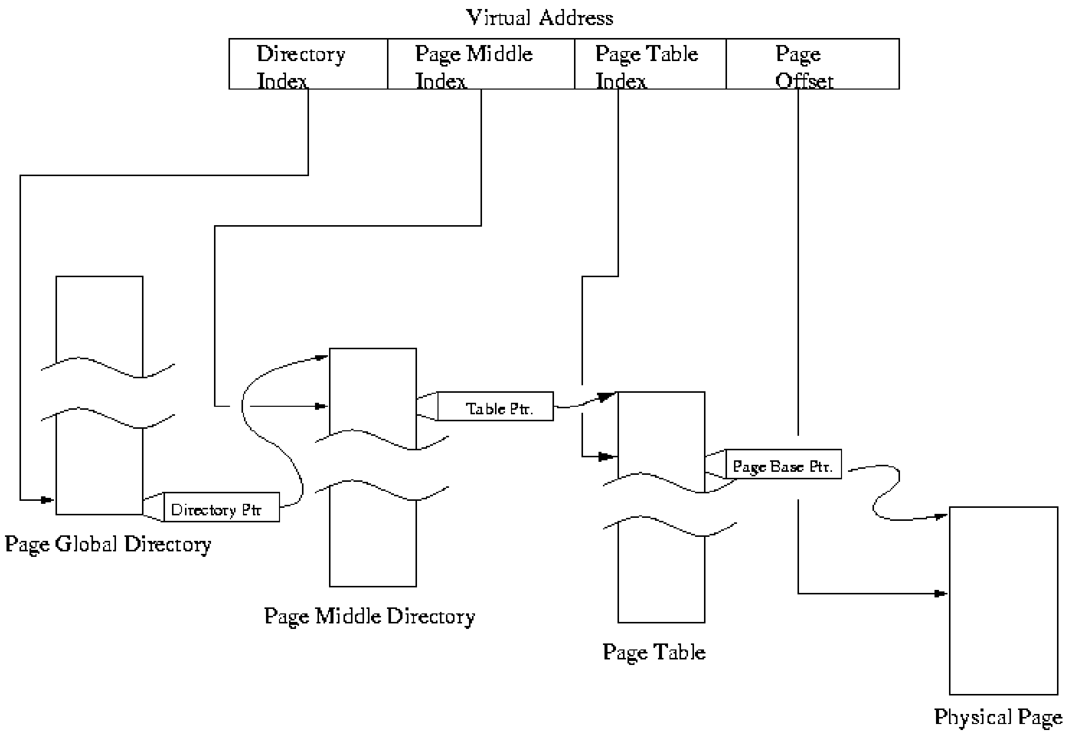

Figure 2: Illustration of a virtual to physical memory translation

Each table lookup provides a pointer to the base of the next table, as

[1]**AUTHORS NOTE: The Overview figure may need updating to bring it into line w/ 2.6 kernel architecture**

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[2] 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.

[2]The RHEL4 kernel extends this zone to 3.9 GB of space

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[3].

2.5 Kernel Threads

The last component in the VM subsystem are the active tasks: kswapd, and pdflush. 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). These two tasks are responsible for the management of swap and pagecache respectively. Kswapd periodically scans all pgdat[4]. structures in the kernel, looking for dirty pages to write out to swap space. It does this in an effort to keep the number of free and clean pages above pre-calculated ¨safe¨thresholds. Likewise the pdflush daemon is responsible for managing the migration of cached file system data to its backing store on disk. As system load increases, the effective priority of the pdflush daemons likewise increase to continue to keep free memory at safe levels, subject to the VM tunables described in section [5]

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 Slab Allocator may only allocate from the DMA and NORMAL zones

[4]A pgdat is the data structure that describes one node in a Non Uniform Memory Access Architecture (NUMA) machine

[5]**AUTHORS NOTE: ADD REFERENCE TO SECTION FOR VM TUNABLES HERE**

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: [6] 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 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.

[6]**AUTHORS NOTE: I Believe that the InactiveLaundry state needs to be removed from the figure**

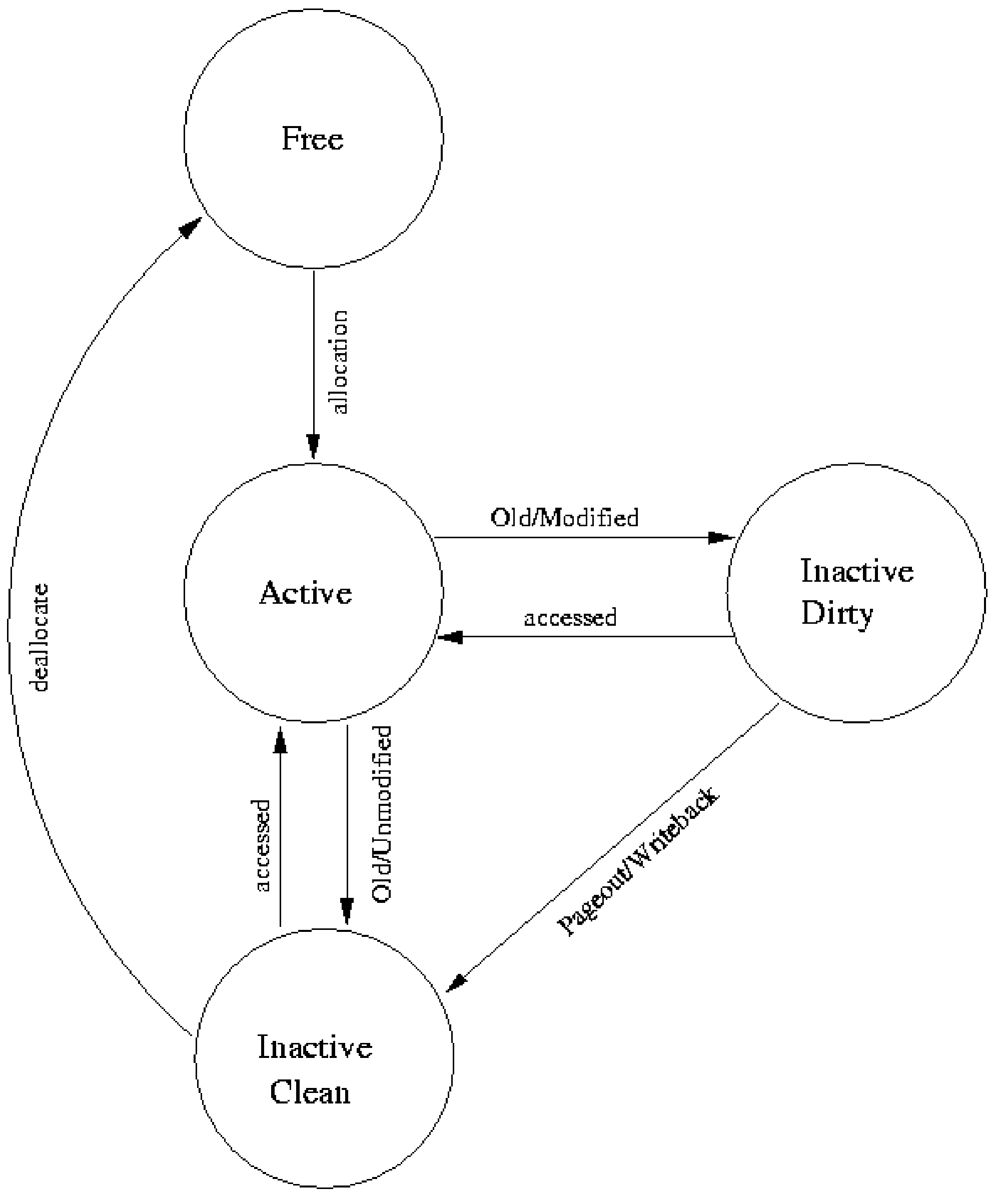

Figure 3: Diagram of the VM page state machine

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 number of available hugepages vm.nr_hugepages=64

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

#bump up the number of pdflush threads vm.nr_pdflush_threads=2
```

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 swappiness
60

echo 70 > /proc/sys/vm/swappiness

# cat kswapd
70
```

The proc file system 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 RHEL4 2.6.9-11 kernel.

4.1 block_dump

The block_dump parameter is a boolean value which is used to enable message logging of submitted I/O requests and page writes. It is not a vm tunable per se, but it is a handy tool to help identify which processes are causing large amounts of disk access. Set it to a non-zero value and configure syslog to record debug messages from the kernel to have this debugging information recorded to /var/log/messages

4.2 laptop_mode

Laptop Mode is an umbrella setting designed to increase battery life in laptops. By enabling laptop mode the VM makes decisions regarding the write-out of pages in such a way as to attempt to minimize high power operations. Specifically, enabling laptop mode does the following:

- modifies the behavior of kswapd to allow more pages to dirty before swapping
- modified the behavior of pdflush to allow more buffers to be dirty before writing them back to disk
- coordinates the activities of kswapd and pdflush such that they write to disk when the disk is active to avoid unneeded disk spin up activity, which wastes battery power.

4.3 legacy_va_layout

This tunable provides, for architectures which support it, a switch which allows the kernel to allocate memory address space in a processes virtual memory map either using the current 2.6 layout algorithm, or using the legacy layout algorithm which was provided with the 2.4 kernel. While this is not strictly speaking a performance tunable, it does allow older applications which push the usage envelope of their process address space a greater degree of compatibility with 2.6 kernels without requiring excess modification.

4.4 nr_pdflush_threads

This is a read-only value. It indicates how many pdflush threads are running at any one time. It is a useful metric for determining how much disk buffer work is being done on the system. While it is not adjustable, it is usefully for telling an admin how much disk activity a certain workload is driving. It does not correlate directly to any amount of I/O or I/O rate, but it does indicate within a certain range that when a pdflush thread wrote to disk, it detected there was additional work to do, and no available thread to do it. This value will fluctuate between **MIN_PDFLUSH_THREADS** and **MAX_PDFLUSH_THREADS** which are current defined to 2 and 8 respectively.

4.5 dirty_background_ratio

This is the percentage of memory that needs to be dirty[7] before one of the pdflush threads will begin writing out the dirty data in the background.

4.6 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 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.

4.7 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:

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

[7]modified, but not written to disk

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).

4.8 dirty_expire_centisecs

This tunable, expressed in 100^{ths} of a second, defines who long a disk buffer can remain in ram in a dirty state. If a buffer is dirty, and has been in ram longer than this amount of time, it will be written back to disk when next one of the pdflush daemons runs.

4.9 dirty_writeback_centisecs

This tunable, also expressed in 100^{ths} of a second, defines the poll interval between iterations of any one of the pdflush daemons. Lowering this value causes a pdflush task to wake up more often, decreasing the latency between the time a buffer is dirtied, and the time it is written back to disk, while lowering it increases the poll interval and the sync-to-disk latency. Decreasing the sync-to-disk latency of course potentially trades of system responsiveness, since time that a processor is running a pdflush daemon is time that it may be able to spend doing user application work.

4.10 lower_zone_protection

This tunable provides a level of protection against applications inadvertently allocating memory from a memory zone lower than what the allocation requires. In some workloads, where large quantities of memory are required, some allocations may be drawn from ZONE_NORMAL when they could have used ZONE_HIGHMEM, or from ZONE_DMA when they could have used ZONE_NORMAL or ZONE_HIGHMEM. Under large amounts of memory pressure, satisfying allocation requests from lower zones may occur if higher zones are sufficiently depleted. If, when these allocations are granted, the requester pins them (for example via an mlock call), the kernel may find itself in a state in which it is unable to satisfy requests from allocators who do need memory from a particular zone, potentially resulting in system failure. The protection operates as a multiplier on the page_low value computed for each zone. If the number of free page in a given zone is lower than $lower\_zone\_protection * zone \rightarrow page\_low$ it will wake up kswapd before attempting the allocation, so that an allocation request will be more likely to obtain its memory from the highest memory zone possible.

4.11 dirty_ratio

This value, expressed as a percentage of total system memory, defines the limit at which processes which are generating dirty buffers will begin to synchronously write out data to disk, rather than relying on the pdflush daemons to do it. Increasing this value tends to make disk write access faster for a process, but at the expense of a larger workload presented to pdflush, should that memory be required for other uses later.

4.12 max_map_count

This file allows for the restriction of the number of VMAs[8] that a particular process can own. A Virtual memory area is a contiguous area of virtual address 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.

4.13 page-cluster

This tunable defines how many pages of data are read into memory on a page fault. In an effort to decrease disk I/O, the Linux VM reads pages beyond the page faulted on into memory, on the assumption that the pages of data beyond the page being accessed will soon be accessed by the same task. Depending on how accurate and predictable this is on a given workload, telling the kernel to read ahead in larger or smaller clusters can reduce disk I/O and increase system performance. If your workload tends to access data in

[8]Virtual Memory Areas

large sequential segments, then increasing this value may increase application performance, while for a workload which makes small random memory access will benefit from a decrease in this value. This value is interpreted by the kernel as an indicator that on a page fault, $2^{page-cluster}$ pages should be read ahead.

4.14 min_free_kbytes

This value defines the number of kilobytes the VM must keep as free in the ZONE_NORMAL zone of each node in a system. Keeping sufficient memory free in lowmem is crucial to preventing acute degradation in system performance. Some workloads may have a sufficiently high demand for resources that the VM will become "cornered". The VM will need to do extra work to gain the resources it need to satisfy the resources of the user workload, resulting in a sharp decline in performance. In those situations, increasing this number may help give the kernel the resources it needs to properly juggle the resource demands of the user workload, without the VM itself becoming to resource constrained.

4.15 swappiness

Swappiness lets an admin decide how quickly they want the VM to reclaim mapped pages, rather than just try to flush out dirty pagecache data. The algorithm for deciding weather to reclaim mapped pages is based on a combination of the percentage of the inactive list we are scanning in an effort to reclaim pages, the amount of total system memory we have mapped, and this swappiness value. The values defined are:

- **Distress** - This is a measurement of how much difficulty the VM is having reclaiming pages. Each time the VM tries to reclaim memory, it scans $1/n^{th}$ of the inactive lists in each zone in an effort to reclaim pages. Each time a pass over the list is made, if the number of inactive clean + free pages in that zone is not over the low water mark, n is decreased by one. Distress is measured as $100 >> n$

- **Mapped_percent** - This is a measure of the percentage of total system memory that is taken up with mapped pages. It is computed as $(number_{mappedpages})/(totalpages) * 100$

- **Swappiness** - This is exactly the value entered in the sysctl parameter

Should $Distress + (Mapped_{percent}/2) + Swappiness >= 100$, then the VM will try to unmap pages in an effort to reclaim memory, rather than just attempting to expunge pagecache.

4.16 hugetlb_shm_group

This is not a tunable per-se, but is worth mentioning in that the 2.6 VM allows certain users to create SysV shared memory segments using hugepages[9]. Users belonging to the group id specified here may created shared memory segments using hugepages.

4.17 nr_hugepages

This tunable specifies the amount of memory (in pages) to reserve for allocation as hugepages. If an application makes use of very large amounts of memory, the application can save kernel resources by allocating that memory as hugepages, which reduce the VM's need for PTE's[10]. Applications not specifically written to use hugepages however, cannot use this memory, so this value should be set such that it is just enough for the applications running on the system which can take advantage of it.

4.18 vfs_cache_pressure

This tunable adjusts the bias on reclaiming inodes and dentries vs. reclaimation via swap and pagecache. The default setting of 100 provides a "fair" balance between the reclamaition of the often used data structures and swap/pagecache. Reducing this value biases the VM to prefer reclaimation of memory using swap and pagecache, while increasing it biases the VM to reclaim memory by flushing inodes and dentry structures.

[9]Hugepages are extra large pages of ram, usually multiple MB in size, which reduce the number of page table entries required by the hardware to map them

[10]Page Table Entries

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. Version
2.6.11
http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.12.5.tar.bz2
Red Hat, Inc. 2005

2.3