

FINE-TUNE YOUR STORAGE CONFIGURATION FOR MAXIMUM PERFORMANCE

by Hydra Systems, LLC

Document Author: Petros Koutoupis

Document Version: v1.01

19 July 2008



Table of Contents

Introduction.....	3
A General Guide Through the I/O Subsystem.....	5
What is I/O?.....	5
The Basic Function of a Kernel.....	6
What is a Process and How Does the Kernel Handle it?.....	6
Virtual Memory Addressing.....	7
Page Caching Vs Direct I/O.....	7
More on the Swap File.....	9
How It All Comes Together.....	10
A Closer Look at the SCSI Layer.....	11
Tunable Components.....	12
Optimizing the Disk Device Variables.....	12
Optimizing the Host Bus Adapter Variables.....	14
Additional Performance Gains.....	14
Customizing Applications to Achieve Better Performance.....	15
Storage Access.....	15
I/O Methods.....	15
Seeking Methods.....	16
Conclusion.....	16

Introduction

Believe it or not but Operating Systems utilizing the Linux kernel have been gaining significant popularity on the enterprise market. From a majority of my personal experiences both Linux and UNIX run the “behind-the-scenes” to everyday computing. More so than Microsoft's series of Windows Operating Systems.¹ And why not? Since the year 2001 I have been watching the Linux kernel grow in stability and as a community; and the kernel has matured up to the point where it has been ready for enterprise use.

It is not to say that a Linux Operating System is far superior than its Microsoft counterpart. Over the years I have learned that no Operating System is perfect and each serves its purpose. It just so happens to be a mere coincidence that out of my 7 servers and personal computers at home, none of them run a Windows Operating System; just Linux and Sun's Solaris UNIX. After their attempt at bringing XENIX (Microsoft's first Operating System; yes a UNIX Operating System) to the market for general computing and not wanting to license an Operating System core from AT&T, Microsoft later focused on their version of the DOS operating system which would eventually be followed by Windows and making the PC easily accessible and extremely user friendly to the everyday individual. They showed the world that through simplicity you did not need to be an engineer or a hobbyist to have, own or work on a computer. So what are we giving up for this simplicity. The answer is functionality, customizable options/features and in most cases, stability. It is a result of “keeping it simple” that Microsoft has been able to succeed in the end-user desktop and small business market. I have been paying attention to how the newer generations of the Linux Operating Systems are getting more and more user-friendly and this can pose as serious competition because on top of its stability you are starting to get a lot of the necessary simplicity (with a lot of complexity for the more advanced user). It has gotten to the point that anybody from any background can install and run Linux without direction and have access to all necessary applications and the Internet as soon as they first turn on the PC. What more can you ask for with a price of zero dollars?

In the end what a Linux Operating System brings to the table is more choices and what is wrong with being able to choose? What if you had only one option for the type of automobile you can drive? What if you had no choices of a meal for the rest of your life? Our world thrives on choices. In fact, if you have gotten this far down in this article, you have chosen to read this far and hopefully you will choose to continue on reading. It is by choice that entire European countries run only on the Linux Operating System and open source software in their enterprise environments (France, Germany, etc.). It is also by choice that the New York Stock Exchange and Chicago Mercantile Exchange run entirely on Linux; the latter I can testify to from experience.

What is holding others back from thinking that they have choices? The answer is: the misinformation fed to the industry that you will not obtain the solid support that you may require for your environment. A type of support, it is believed, that only Microsoft and other closed source vendors can offer. This is completely untrue and for years companies such as Redhat, Novell, Canonical, etc. have been offering

¹ Gould, Jeff. Linux and the rise of the dual stack datacenter. <http://www1.interopsystems.com/news/linux-and-the-rise-of-the-dual-stack-datacenter.html>

the best of support services on top of professional training courses so that they may educate the world of their distributions of the Linux Operating System. Some of which are only a fraction of the price of what you would pay for the support of other companies. Even Sun Microsystems offers full support for all their open source projects. The support is there and has been there. On top of these companies, you also have an entire community as a resource that can be found all over the world through whatever channel you choose to communicate with them.

This now brings me to the point of this article; understanding proper storage administration of a Linux Operating System running on the 2.6 Linux kernel. Now since more and more companies are choosing to utilize a Linux Operating System in their environment, are they fine-tuning it to achieve the best performance? With my experience in the storage industry, despite what platform you run on (Windows, Linux or UNIX) there has never been a proper method for enlightening individuals on how to fine-tune their storage equipment for maximum performance. A lot of this knowledge comes more from trial and error; even after the configuration is out in production. The training courses for networking administration, volume management, etc. are there but I have yet to see anything on tuning the SCSI layer (all SCSI and HBA drivers), or at least training on how the Operating System interacts with the storage. Unfortunately not everyone contains enough development knowledge to even be able to dive into open source code and attempt to trace through all actions taken when a User Mode process initiates a simple function such as a read or a write and how it travels into Kernel Mode until it reaches the end disk device and back to the calling process. One major problem to this is that a lot of these fine-tunings are unique to specific vendors who manufacture a lot of the equipment for the low-level protocol communication layer between the host and the target(s). As I start speaking of an Operating System's I/O Subsystem, the queue depth and timeout values of a disk device and how to load balance across multiple node ports, some may be sitting there and scratching their heads. That is the purpose of this document. To educate and to inform that despite what Operating System you use, nothing is perfect "out of the box." The "out of the box" experience does not and should not exist when working with enterprise level equipment. It is a myth and if a vendor leads you to believe otherwise, It would extremely concern me to use their product lines. There is too much at stake when putting into production your storage solution.

Now while the rest of the document focuses in on the 2.6 Linux kernel, please note that the ideas and the mentality presented do not differ that much from other Operating Systems.

A General Guide Through the I/O Subsystem

What is I/O?

In general, I/O (or Input/Output) is the ability to perform an input and output operation between a computer and an output device. For example, a keyboard or a mouse is considered to be an input device and the actions you take affect an output device such as what you view on the monitor after moving your mouse around on the screen. Now, when someone in the storage industry speaks of I/O they are usually referring to the input and output of data to a disk device.

There are numerous ways to initiate an I/O process to a storage device. To keep this as simple as possible we will not get into great depth in those details. Just note that the basic steps an application usually uses to generate I/O between the application layer of the Operating System and the end storage device are:

1. Open the device or file.
2. Set the location from which to read or write.
3. Execute the read or write operation.
4. Repeat steps 2 and 3 as needed.
5. Close the device or file.

Although these steps may seem a little simplistic, note that many variables are used to define how an I/O operation is performed. These variables are sometimes referred to as the *I/O profile*. Some of the parameters that define a configuration's I/O profile are:

- Transfer size – The number of bytes or blocks transferred.
- Seeking method – That is sequential, random or mixed.
- Range – The area on which to execute the I/O and its size. A couple of examples are: the first 100 blocks of the disk device or creating a file with a file length of 1 GBs.
- Processes – The amount of processes generating I/O operations that are running concurrently to a disk device. This includes the total number of hosts as well as the number of processes of I/O running to the end storage device.
- Data Pattern – The data pattern filled into the write/read buffers and being send to/from the disk device.
- Timing – The rate at which the I/Os are generated including the timing difference between read and write operations.

While I have listed the most general parameters in the makeup of an I/O profile, note that the profile also includes the host's HBA type, the throttling levels of the SCSI layer's Queue Depth, the SCSI Disk Timeout Value, and if the disk device is in an array you must also include the stripe/chunk size of the disk device to even its RAID type and more. Understanding the makeup of the I/O profile is extremely important for development and problem isolation but unfortunately it is beyond the scope of this article. The topic will just have to wait for another time.

The Basic Function of a Kernel

Massive details of the kernel including the different types are not going to be discussed here. That too is a topic beyond the scope of this document. For further information on the internals of a kernel it is advised to pick up a book on *kernel internals*, or specific *Operating System* materials.² This is more of a basic overview of what the average kernel is responsible for. The majority of the roles taken on by the average kernel are:

- *Process/Task Management*
The main task of a kernel is to allow the execution of applications. Note that I am using the terms process and application interchangeably to signify one and the same thing when in execution. That is because a process is an application in execution. This also includes the servicing of interrupt request (IRQ) routines. The kernel needs to perform context switching between hardware and software contexts.
- *Memory Management*
The kernel has complete access to the system's memory and must allow processes to safely access this memory as they need it.
- *Device Management*
When processes need to access peripherals connected to the computer, controlled by the kernel through device drivers, the kernel allows such access.
- *System Calls*
In order to accomplish task such as file I/O, the process is required to have access to memory, and that same process must be able to access various services that are provided by the kernel. The process will make a call for a function which in turn will invoke the related kernel function.

What is a Process and How Does the Kernel Handle it?

The kernel internally refers to these processes as tasks. Each process or task is issued an ID unique to that running process alone and managed in the application layer, the kernel is unaware of the ID. A process should not be confused with a thread. A kernel thread means something else so when you are running 12 instances of I/O, you are running 12 processes of I/O and not 12 threads. Threads of execution are the objects of activity within the process (i.e. instructions that a processor has to do).

A Process ID (PID) is an identifier to the execution of a binary, a living result of running program code. When the binary completes its execution, the process to that execution is wiped clean from memory. Here is an example from a system running Linux (on Windows you would invoke a *tasklist*):

```
# ps -ef|grep bash
root  12408  12406  0  07:43  pts/0  00:00:00  bash
```

This process holds the PID of 12408 and this ID will be cleared once this process is either completed or aborted (either by user or error). If you want to find out more information about an actively running process, both Linux and UNIX make its information, such as memory mappings, file descriptor usage and more through its virtual file system procfs mounted from the root path at /proc. In it all actively running processes are organized according to their respective PID number.

² Some excellent books come to mind:

1. Linux Kernel Development by Robert Love
2. Understanding the Linux Kernel by Daniel Bovet and Marco Cesati
3. Solaris Internals(TM): Solaris 10 and OpenSolaris Kernel Architecture by Richard McDougall, Jim Mauro

Virtual Memory Addressing

Virtual Memory Addressing (VMA) is a memory management technique commonly utilized in multitasking Operating Systems, wherein non-contiguous memory is presented to a User Space process (i.e. a software application) as contiguous memory, and referred to as the virtual address space. VMA is typically utilized in paged memory systems (read below).

Page Caching Vs Direct I/O

Paging memory allocation algorithms divide a specific region of computer memory into small partitions, and allocate memory using a page as the smallest building block. The memory access part of paging is done at the hardware level via page tables, and is handled by the Memory Management Unit (MMU) local to the kernel. As mentioned earlier, physical memory is divided into small blocks called pages (typically 4 KB in 32-bit architectures and 8KB in 64-bit architectures) in size, and each block is assigned a page number. The operating system may keep a list of free pages in its memory, or may choose to probe the memory each time a memory request is made (though most modern operating systems do the former). Whatever the case, when a program makes a request for memory, the operating system allocates a number of pages to the program, and keeps a list of allocated pages for that particular program in memory.

When paging is used alongside with virtual memory, the operating system has to keep track of pages in use and pages which will not be used or have not been used for some time. Then, when the operating system deems fit, or when a program requests a page that has been swapped out, the operating system swaps out a page to disk, and brings another page into memory. *Read next section for additional details regarding the swap file.* In this way, you can use more memory than your computer physically has.

A paging file should NOT be confused with a swap file. The swap file and paging file are two different entities. Although both are used to create virtual memory, there are subtle differences between the two. The main difference lies in their names. Swap files operate by swapping entire processes from system memory into the swap file on the physical disk. Your Operating System usually allocates a finite size of swap space during installation. This swapping immediately frees up memory for other applications to use.

In contrast, paging files function by moving "pages" of a program from system memory into the paging file. These pages are 4KB (again, usually determined by PC architecture) in size. The entire program does not get swapped wholesale into the paging file. For further reading onto the topic of the host side pages it is recommended to pick up a copy of a book discussing kernel internals. This topic is usually covered with greater detail.

When you open up a file "unbuffered" to perform direct I/O, you are not using these cache pages. So every time you work with that recently modified and "unbuffered" file, you are going straight to the location in disk as opposed to memory instead holding the altered data in pages. This feature can hurt overall performance of file I/O to disk. By relying on paging ("buffered I/O") you are in fact increasing

your performance and productivity because you are not constantly going straight to the hard disk (which is obviously MUCH slower than dynamic memory) to obtain your data. While paging your data, you are able to retrieve and send your data very quickly while moving on to the next step in the file I/O process. There are certain moments when the data from a cache page will be flushed to disk. When data is cached to this page, this is marked as dirty because it is data that has not been written to disk. These dirty pages get written to disk when a Synchronize Cache is invoked, or when new data belonging to the same file area over writes the older paged data. In the latter case the older paged data gets written to disk while the newer data gets paged to the same memory address marking that page as dirty once again.

To keep things simple I will detail the 32-Bit VMA Model. The Random Access Memory (RAM) is separated into zones on the O/S.

Zone 0: Direct memory Access (DMA) Region

Zone 1: is a Linear mapping of the Kernel Space (see figure below) while shared with User Space calculations.

Zone 2 (High Memory Zone): Is a non-linear mapping of the last 1/8GB of the VMA.

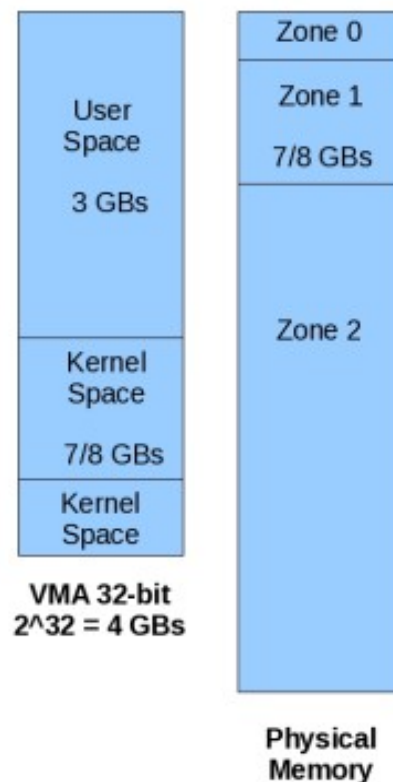


Illustration 1: 32-bit Virtual Address Model.

We obtain 4 GB of Virtual Memory addressable regions on a 32-bit Operating System because 2^{32} equals 4294967296 or 4 GB. As for the User-Space Virtual 3 GB, this is addressed wherever there is free memory. We find our cache pages in kernel space of the VMA model.

On a 64-bit architecture memory addressing performs much better. There is more linear mapping and less swapping in the high memory region.

More on the Swap File

Many thanks go to **Mark Lord** for his contribution in clarifying some of the details regarding the swap file on Linux (19 Jan 2008).

As you know, paging-in means reading data from storage into the page cache, and paging-out means evicting dirty (modified) pages from the page cache back to their underlying ("backing") storage.

When Linux needs to free up memory, it first discards clean (unmodified) pages which can always be re-read (paged-in) again from their source files on storage. It also tries to commit dirty pages back to storage, forcing writes out to their modified files on storage.

This works for memory contents which originate from a file on storage. But some memory, eg. stack space, malloc'd objects, etc.. does not originate from a file, and therefore cannot be paged-out to a file when the system needs to free up some memory. This is known as ANONYMOUS memory.

For dirty ANONYMOUS memory, there is no original file to write pages to. That is where swap comes into the picture -- it provides backing storage for ANONYMOUS memory objects, such as process stacks, malloc'd memory, or any other memory obtained with `brk()` or `mmap(.. MAP_ANONYMOUS ..)`.

How It All Comes Together

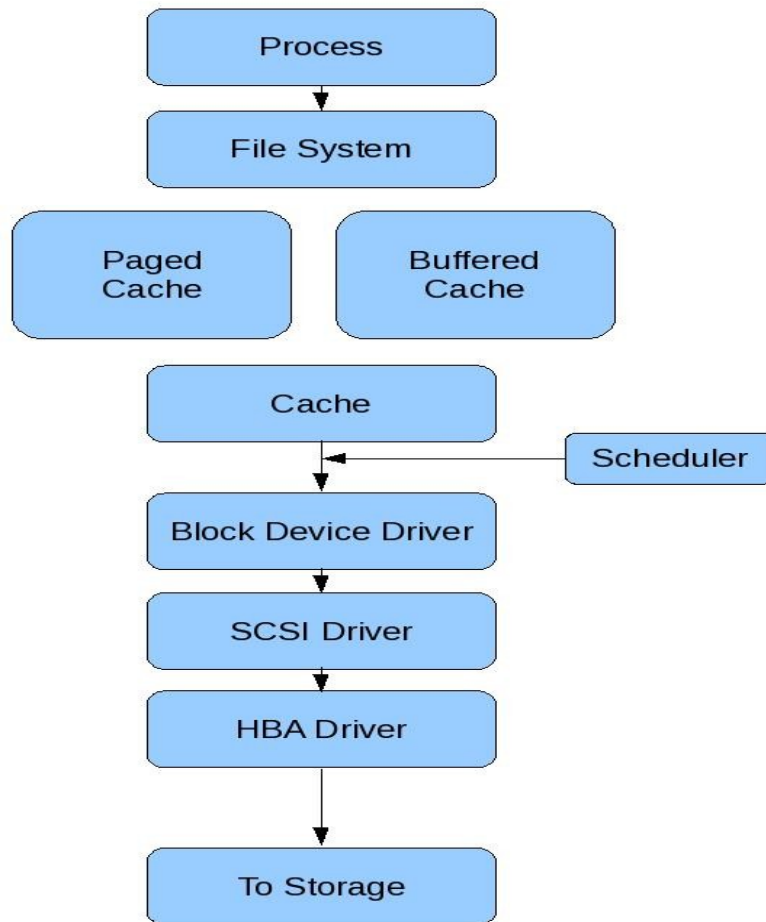


Illustration 2: The I/O Subsystem.

To briefly summarize, the above diagram does a pretty good job of displaying the general layout of the I/O subsystem on any OS. When an I/O process is initiated for a disk device it first travels through the file system layer (assuming that it is a file over a file system) to figure out where it needs to go. Once that is figured out it is either paged (enabled by default) or thrown into a temporary buffer cache (for *direct I/O*). When the I/O is ready to be written it is thrown into another cache waiting for the scheduler. It is up to the scheduler to intervene and prioritize and coalesce the transfer(s). *This occurs on both Synchronous and Asynchronous I/O*. The I/O then gets sent to the block device driver (`sd_mod`) and follows through to the many other layers involved after that block device driver (`scsi_mod` then the HBA module). These layers vary depending on the block device being written to. One last thing to note with this diagram is that the process is running in User Mode and it hits Kernel Mode between the Process block and the File system block (on the diagram). Everything else below is Kernel Mode. This

situation occurs when a user mode function or API is called which has a corresponding `__syscall` (system call) into Kernel Mode such as a `write()` or `read()` function. When writing to a physical device, the file system layer is skipped and the I/O requests are placed into the temporary buffer cache for *direct I/O*.

If you are transferring I/O to/from a SCSI based device (i.e. SCSI, SAS, fibre, etc.), it is when these transfers fall onto the SCSI driver that the CDB structure is populated and that the SCSI Disk Timeout Value initiates. If your I/O goes stale and you do not see any activity and it does not time out, chances are it never came to the SCSI layer. The transfer(s) are possibly stuck in one of the earlier layers.

A Closer Look at the SCSI Layer

In Linux, the SCSI Subsystem exists as a multi-layered interface divided into the Upper, Middle and Lower layers. The Upper Layer consists of device type identification modules (i.e. Disk Driver (`sd`), Tape Driver (`st`), CDROM Driver (`sr`) and Generic Driver (`sg`)). The Middle Layer's purpose is to connect both Upper and Lower Layers and in our case is the `scsi_mod.ko` module. The Lower Layer is for the device drivers for the physical communication interfaces between the host's SCSI Layer and end target device. Here is where we will find the device driver to the HBA.

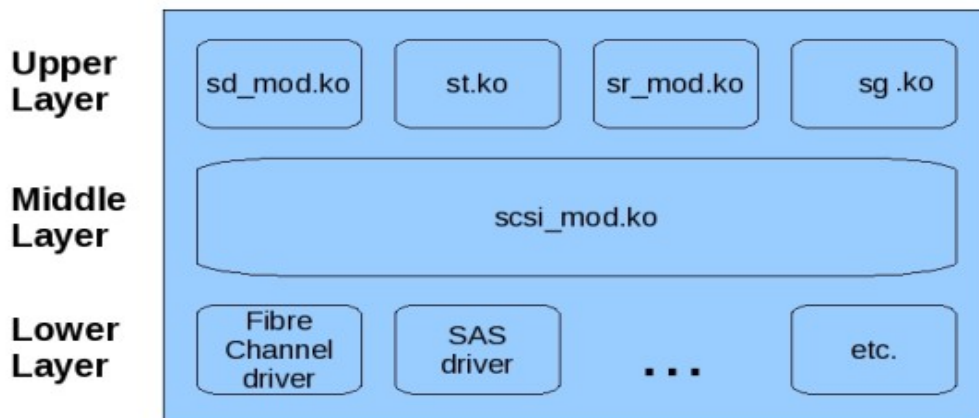


Illustration 3: The SCSI Subsystem.

Whenever the Lower Layer detects a newer SCSI device, it will then provide `scsi_mod.ko` with the appropriate host, bus (channel), target and LUN Ids. Depending on what type of media the devices are would determine what Upper Layer driver will be invoked. If you view `/proc/scsi/scsi` you can see what each SCSI device's type is:

```
[pkoutoupis@linuxbox3 ~]$ cat /proc/scsi/scsi
Attached devices:
Host: scsi0 Channel: 00 Id: 00 Lun: 00
```

Vendor: ATA Model: WDC WD800BEVS-08 Rev: 08.0
Type: Direct-Access ANSI SCSI revision: 05
Host: scsi3 Channel: 00 Id: 00 Lun: 00
Vendor: MATSHITA Model: DVD-RAM UJ-860 Rev: RB01
Type: CD-ROM ANSI SCSI revision: 05

The Direct-Access media type will utilize the `sd_mod.ko` while the CD-ROM media type will utilize the `sr_mod.ko`. Each respective driver will allocate an available major and minor number to each newly discovered and properly identified device and on the 2.6 kernel, `udev` will create an appropriate node name for each device. As an example, the Direct-Access media type will be accessible through the `/dev/sdb` node name.

When a device is removed, the physical interface driver will detect it from the Lower Layer and pass the information back up to the Upper Layer.

Tunable Components

There are multiple approaches to handling this and in the following two sections again I will keep things simple. Note that the more complex approach involves the editing of source code and recompiling the device driver to have these variables hard-coded during the lifetime of the utilized driver(s). That is not what we want, we want a more dynamic approach. Something that can be customized on-the-fly. One day it may be optimal to configure a driver one way and the next another.

Optimizing the Disk Device Variables

The 2.6 Linux kernel introduced a new virtual file system to help reduce the clutter that became `/proc` (for those not familiar with the traditional UNIX file system hierarchy, this was originally intended for process information) with a `sysfs` file system mounted at `/sys`. To summarize, `/sys` contains all registered components to the Operating System's kernel. That is, you will find block devices, networking ports, devices and drivers, etc. mapped from this location and easily accessible from user space for enhanced configuration(s). It is through `/sys` that we will be able to navigate to the disk device and fine tune it to how we wish to utilize it. After I explain `sysfs`, I will move onto to describing modules and how a module can be inserted with fine-tuned and pseudo-static parameters.

Let us assume that the disk device that we want to view the parameters to and possibly modify is `/dev/sda`. You would navigate your way to `/sys/block/sda`. All device details are stored or linked from this point for device node named `/dev/sda`. If you go to the device you can view time out values, queue depth values, current states, vendor information and more (below).

```
[pkoutoupis@linuxbox3 device]$ ls  
block:sda delete evt_media_change iodone_cnt modalias queue_depth rev scsi_generic:sg0  
subsystem uevent bsg:0:0:0:0 device_blocked generic ioerr_cnt model queue_type  
scsi_device:0:0:0:0 scsi_level timeout vendor bus driver iocounterbits iorequest_cnt power  
rescan scsi_disk:0:0:0:0 state type
```

To view a parameter value you can simply open the file for a read.

```
[pkoutoupis@linuxbox3 device]$ cat timeout
60
```

Here we can see that the timeout value for SCSI labeled device is 60 seconds. To modify the value you can *echo* the new value into it.

```
[root@linuxbox3 device]# echo 180 >> timeout
[root@linuxbox3 device]# cat timeout
180
```

You can perform the same task for the queue depth of the device along with the rest of the values. Approaching the disk device values this way are unfortunately not maintained statically. That means that every time the device mapping is refreshed (through a module removal/insertion, bus scan, or a reboot) the values restore back to their defaults. This can be both good and bad. A basic shell script can modify all values to all desired disk devices so that the user does not have to enter each device path and modify everything one by one. On top of the basic shell script a simple *cron* job can also validate that the values are maintained and if not it can rerun the original modifying shell script.

Another way to modify values and have them pseudo-statically maintained is by inserting those values within the module itself. For example if you do a *modinfo* on *scsi_mod* you will see the following dumped to the terminal screen.

```
[pkoutoupis@linuxbox3 device]$ modinfo scsi_mod
filename:    /lib/modules/2.6.25.10-47.fc8/kernel/drivers/scsi/scsi_mod.ko
license:    GPL
description: SCSI core
srcversion: E9AA190FE1857E8BB844015
depends:
vermagic:   2.6.25.10-47.fc8 SMP mod_unload 686 4KSTACKS
parm:      dev_flags:Given scsi_dev_flags=vendor:model:flags[,v:m:f] add black/white
list entries for vendor and model with an integer value of flags to the scsi device info list
(string)
parm:      default_dev_flags:scsi default device flag integer value (int)
parm:      max_luns:last scsi LUN (should be between 1 and 2^32-1) (uint)
parm:      scan:sync, async or none (string)
parm:      max_report_luns:REPORT LUNS maximum number of LUNS received (should
be between 1 and 16384) (uint)
parm:      inq_timeout:Timeout (in seconds) waiting for devices to answer INQUIRY.
Default is 5. Some non-compliant devices need more. (uint)
parm:      scsi_logging_level:a bit mask of logging levels (int)
```

The appropriate way to enable a pseudo-static value is to insert the module with that parameter:

```
[pkoutoupis@linuxbox3 device]$ modprobe scsi_mod max_luns=255
```

Or modify the */etc/modprobe.conf* (some platforms use an */etc/modprobe.conf.local*) file by appending an “options *scsi_mod* max_luns=255” and then reinsert the module. In both cases you must rebuild the RAM Disk so that when the host reboots it will load max_luns=255 into the insertion of the *scsi_mod*

module. This is what I meant by pseudo-static. The value is maintained only when it is inserted during the insertion of the module and must always be defined during its insertion to stay statically assigned.

Some may now be asking, well what the heck is a timeout value and what does queue depth mean? A lot of resources with some pretty good information can easily be found on the Internet but as far as basic explanations go, a SCSI timeout value is the maximum value to which an outstanding SCSI command has to completion on that SCSI device. So for instance, when `scsi_mod` initiates a SCSI command for the physical drive (the target) associated with `/dev/sda` with a timeout value of 60, it has 60 seconds to complete the command and if it doesn't, an ABORT sequence is issued to cancel the command.

The queue depth gets a little bit more involved in which it limits the total amount of transfers that can be outstanding for a device at a given point. If I have 64 outstanding SCSI commands that need to be issued to `/dev/sda` and my queue depth is set to 32, I can only service 32 at a time limiting my throughput and thus creating a bottleneck to slow down future transfers. On Linux, queue depth becomes a very hairy topic primarily because it is not adjusted only in the block device parameters but is also defined on the Lower Layer of the SCSI Subsystem where the HBA throttles I/O with its own queue depth values. This will be briefly explained in the next section.

Other limitations can be seen on the storage end. The storage controller(s) can handle only so many service requests and in most cases it may be forced to begin issuing ABORTs for anything above its limit. In turn the transfers may be retried from the host side and complete successfully, so a lot of this may not be that apparent to the administrator.

Again, it becomes necessary to familiarize oneself with these terms when dealing with mass storage devices.

Optimizing the Host Bus Adapter Variables

An HBA can also be optimized in pretty much the same fashion as the SCSI device. Although it is worth noting that the parameters that can be adjusted to an HBA are vendor specific. These are additional timeout values, queue depth values, port down retry counts, etc. Some HBAs come with volume and/or path management capabilities. Just simply identify the module name for the device by doing an `lsmod` (it may also be useful to first identify it usually attached to your PCI bus by executing an `lspci`). And from that point you should be able to either navigate the `/sys /module` path or just list all module parameters to that device with a `modinfo`.

Additional Performance Gains

First and foremost, dependent on the method of connection between host(s) to target(s), load balancing becomes a big problem. How do you balance the load to all Logical Units (LU) making sure that all can get serviced within an appropriate time frame? Fortunately enough, there exists some great tools for the Linux platform that many utilize for both volume management, multipathing/load balancing and failover/failback needs. One such tool is a device-mapper used in conjunction with `multipath-tools`. In

the past I have always used this and it has always served me extremely well, but be aware, that this set of modules must also be fine tuned to accommodate the type of storage you are utilizing.

It also becomes quite necessary to understand file system basics; that is basic structures and methodologies. For each file system is unique and can offer many positives and/or negatives in a production environment. Some things to consider with the file system are journaling methods, data allocation (block-based or extent-based on top of a B+ tree), write barriers and more. In regards, to journaling methods some file systems contain more than one method for journaling, some more reliable than others but they come at a cost with significant performance drops.

In order to fully and appropriately optimize all of these variables, the administrator must fully understand the I/O profile to which they are catering to. What limitations is our storage leaving us? Would I need to increase my SCSI timeout values in order to make sure that all I/O requests are fully serviced with little or no problems? What limitations does my HBA give me? How is the host accessing the end storage device (directly or in a SAN)? What about redundancy (path failovers), to make sure that there will be little to no down time on failures? How much traffic should I expect? How do the application work with the disk device(s)? What performance gains or losses do I obtain with Volume Management? These are just a few of many pressing question that an administrator must ask themselves when configuring and placing storage into production.

Customizing Applications to Achieve Better Performance

This section focuses on the development and customizable options/features of applications that may need to work directly with the storage. Concepts of cached and direct I/O, synchronous and asynchronous access and seeking methods are covered in this section.

Storage Access

There are a few situations where you would need to rely on methods of direct I/O, that is bypassing the cache paging layer of the I/O Subsystem, but again this comes at a tremendous cost to overall performance. If performance is what you desire, then your decision on this topic is already made, cache all operations and work entirely out of memory. If you need data flushed quickly to the disk device, invoke an *fsync()* to flush all pages of cached data to disk. Note that page caching is only utilized when writing to a file over a file system and not when directly accessing the physical device through its device or volume node (the volume manager may also employ its own method of enhanced caching).

I/O Methods

Which one should you choose, synchronous or asynchronous I/O methods? There are positives and negatives to using both. Asynchronous works quicker in which it dispatches a finite amount of commands under a given process and worries about their return status at a later point. Now if something did not go as anticipated, moving backwards to isolate the problem can become quite a

hassle. Let us say we dispatch 40 asynchronous write transfers and both my HBA and SCSI block device queue depth is set to 32, I will service 32 transfers at a time and when some of those transfers return then the rest of the 40 will get transferred. When some of the 40 original asynchronous transfers return, it will continue to dispatch more until I reach my program's defined limit of total outstanding asynchronous transfers (in this case I made it 40). Traditionally database applications fully utilize asynchronous access to the disk device. Good performance is usually a must for database applications.

With synchronous transfers, a process will issue one transfer at a time. It will not continue to send any additional transfers until, the one returns and is handled appropriately. It simplifies the program but you may not necessarily need anything more.

Seeking Methods

What can make or break the performance on a configuration can be the method the data gets read/written from disk. Traditionally you would develop an application where you either use a sequential seeking method starting from one block address and ending at another without a break in contiguous blocks or a random seeking method where all data blocks are accessed in more of a random pattern. In some cases one may use a staggered-sequential method or maybe others which are beyond the scope of this article. Come to understand how you want to present the data to file or disk. There is an advantage to performance when writing sequentially, whereas the I/O scheduler will coalesce sequential like operations into a single command. If I were to send 32 sequential write transfers to a file, the I/O scheduler can group them together into 2-3 write commands, simplifying the task(s) and providing faster results. In random, data writes/reads are truly random where if I had 32 random write transfers to a file, I will most likely have the scheduler queue 32 separate write commands. On top of that, there will be additional latency in waiting for the disk device to seek to the appropriate sector for each command. Doesn't sound so good, does it? So be extremely careful and understand what may be the best approach to access the data on the disk device.

Conclusion

Configuring your storage solution is obviously not as simple as most may think. Some of you may be reading this article and already are aware of these concepts and procedures while others may be completely new to the scene. Always know that if you have any questions or desire any aid to configuring and fine-tuning your storage environment there are plenty of good reading materials covering these topics with much more detail. The vendors of these Operating Systems and hardware devices also offer great support and then there is also Hydra Systems, LLC. We can provide not only the answers you need but also the customization and training if desired.

For more information, contact: pkoutoupis@hydrasystemsllc.com

If you find any errors or have any further questions with this reference material, please contact: pkoutoupis@hydrasystemsllc.com