Linux Storage Management

Petros Koutoupis

For years now I have been working in the data storage industry, offering various services which include development and consultation. With regards to consultation, the services extend to storage customization, configuration and performance tuning.

During these instances I have taken notice, that while I work with extremely intelligent individuals, the education is still somewhat lacking in how the operating system manages the storage volumes and how to tune all aspects relating to this topic for the ideal computing environment. This in turn can lead to disastrous results.

The purpose of this article is to define some of the basics of all the layers involved in Linux 2.6 storage management along with key concepts to be aware of with regards to decision making and performance tuning. The idea is to give the reader a better insight into analyzing the environment that they are configuring the storage for and to be able to understand the I/O profile that it is supposed to cater to. This article will in turn be partitioned into the following main topics: (1) the outline of the I/O and SCSI Subsystems, (2) followed by a summary of how the storage is presented to the host, (3) and finally, real world methods of configuring and tuning your environment.

What is I/O and How Do I Determine my I/O Profile?

In order to fully understand the following material, it becomes necessary to cover the basics of I/O and I/O management. In general the concept of I/O (or Input/Output) is the ability to perform an input and/or output operation between a computer and a device. That device could be a keyboard or mouse functioning as input devices. It could also be an output device such as a monitor updating the coordinates of the mouse cursor based on user input. With regards to data storage, when I/O is spoken of, it usually signifies the input and output streams of data to a disk device (a.k.a block device).

A block device is a device (i.e Hard Disk Drive, CD-ROM, floppy, et.c) that can host a file system and/or store non-volatile data. In most modern day computing systems, a block device can only handle I/O operations that transfer one or more whole blocks (or sectors), which are usually 512 bytes (or a larger power of two) in length. A block device can be accessed through a file system node mounted locally or networked and shared or through the physical device interface. The advantages to utilizing a file system is to add organization in the storing of data so that it can be readable by both the user and the OS (with a corresponding application). File systems have become so advanced that there are numerous features written into the modules enabling the user to manage a redundant and/or high performing environment. Without the file system, the physical device would be a series of unintelligible sequence of byte values. There wouldn't be any meta-data to identify the location of files including file specific information such as file size, permissions, etc.

Moving back to I/O. 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:

- Open the device or file.
- Set the location from which to read or write.
- Execute the read or write operation.
- Repeat steps 2 and 3 as needed.
- 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 *Host Bus Adapter* (hereafter, HBA) type and configuration, 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, design and problem isolation.

When a process is initiated in User Space and it needs to read or write from a device, it will need enter the Kernel Space in order to resume that process. Massive details of the kernel including the different types are not going to be discussed here. It is a topic beyond the scope of this article. 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 [1]. 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.

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*):

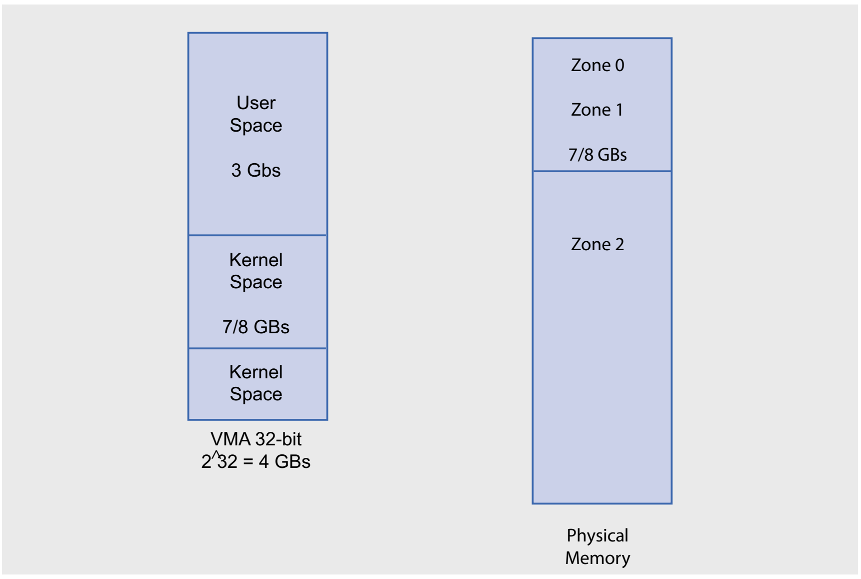

Figure 1. 32-bit VMA model

```
# 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.

Memory Management

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.

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.

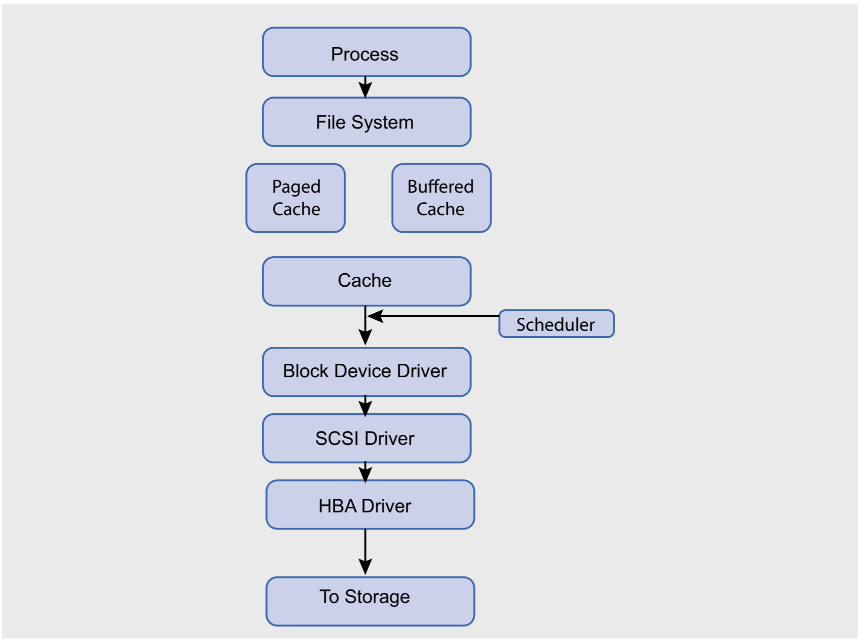

Figure 2. The I/O Subsystem

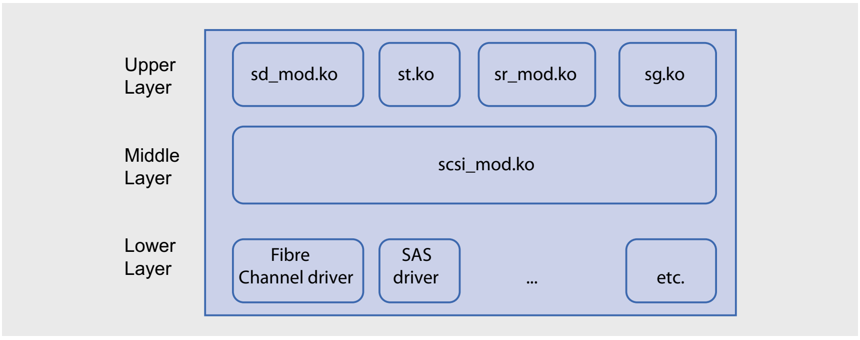

Figure 3. The SCSI Subsystem

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. 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 a processes' memory regions 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 1) while shared with User Space calculations.

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

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.

General Layout

Some of you may be wondering why I am discussing all this information for a topic on Linux Storage Management. This information becomes vital with regards to understanding how the entire I/O subsystem functions in Linux. When you tune on one portion of the subsystem, you also need to understand the ramifications it may have on another portion. Which is why I present the following diagram: see Figure 2

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 Channel, 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.

The SCSI Subsystems and Representation of Devices

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.

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: see Listing 1.

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.

There are multiple approaches to tuning the SCSI devices. 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.

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

Listing 1. Listing identified SCSI devices

```
[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
```

values, queue depth values, current states, vendor information and more (Listing 2).

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 (Listing 3).

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.The 2.6 Linux kernel introduced a new virtual

Listing 2. Listing device parameters in sysfs

```
[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
```

Listing 3. Listing module parameters

```
[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)
```

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.

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.

Methods of Storage Management

Again, the Linux 2.6 kernel has made great advancement in this area. With the introduction of newer file system to even methods of device and volume management, it makes it increasingly easy for a storage administrator to set up a Linux server to perform all necessary tasks. In my personal opinion, the best interface introduced in the 2.6 kernel was the device-mapper framework.

Device-mapper is one of the best collection of device drivers that I have ever worked with. It brings high availability, flexibility and more to the Linux 2.6 kernel. Device-mapper is a Linux 2.6 kernel infrastructure that provides a generic way to create virtual layers of a block device while supporting stripping, mirroring, snapshots, concatenation, multipathing, etc. Device-mapper multipath provides the following features:

- Allows the multivendor Storage RAID systems and host servers equipped with multivendor Host Bus Adapters (HBAs) redundant physical connectivity along the independent Fibre Channel fabric paths available
- Monitors each path and automatically reroutes (failover) I/O to an available functioning alternate path if an existing connection fails
- Provides an option to perform fail-back of the LUN to the repaired paths
- Implements failover or failback actions transparently without disrupting applications
- Monitors each path and notifies if there is a change in the path status
- Facilitates the load balancing among the multiple paths
- Provides CLI with display options to configure and manage Multipath features
- Provides all Device Mapper Multipath features support for any LUN newly added to the host
- Provides an option to have customized names for the Device Mapper Multipath devices
- Provides persistency to the Device Mapper Multipath devices across reboots if there are any change in the Storage Area Network
- Provides policy based path grouping for the user to customize the I/O flow through specific set of paths

Also built on top of the device mapper framework is LVM2 which allows the administrator to utilize those virtual layers of block devices in creating both physical and logical volumes, with snapshot capabilities. In LVM2 physical and logical volumes can be striped or mirrored to create volume groups. Volume groups can be dynamically resized while active and online. It is with these volume groups that you will write a file system to (if applicable) and mount locally to either manage locally or share within a network.

If you are utilizing an older version of the 2.6 kernel, you may be able to use mdadm as a volume and path manager.

The File System

The file system is one of the most important aspects to storage management. Deciding which file system is best suited for your environment rests solely on the type of computing that is to be done to the volume. Each file system has advantages and disadvantages in certain workloads. So before you tell yourself that you will use ZFS over FUSE, Ext3, XFS or anything else, take some time to understand the differences between the options. Some questions to be asking yourself are:

- What is relevent?
- What is to be the size of the file system (i.e. files and directories)?
- How will the file system be frequently accessed and how much load should I expect it to work with?
- How recoverable or redundant do I want the file system to be?
- If I need it to be 100% recoverable, how much performance am I willing to lose?
- Do I want the application(s) accessing the file system to handling caching or the OS?

Listing 4. Listing multipath devices and paths

```
$ multipath -ll
32002000bb55555cd
[size=92 GB][features="1 queue_if_no_path"][hwhandler="0"]
\_ round-robin 0 [active]
\_ 30:0:0:2 sdc 8:32 [active][ready]
\_ round-robin 0 [enabled]
\_ 31:0:0:2 sdg 8:96 [active][ready]
32001000bb55555cd
[size=27 GB][features="1 queue_if_no_path"][hwhandler="0"]
\_ round-robin 0 [enabled]
\_ 30:0:0:1 sdb 8:16 [active][ready]
\_ round-robin 0 [active]
\_ 31:0:0:1 sdf 8:80 [active][ready]
```

One thing worth noting is that there is no single *best* file system. Across all different types of file systems you will find differences in:

- the limitations of file system and file sizes
- number of files and directories it can store
- on-disk space efficiency and block(s) allocation methods
- journaling capabilities and options
- data consistency
- crash recovery methods (fsck, journaling, copy-on-write)
- special features such as direct I/O support, enabled compression and more

To quickly summarize some of your local file systems

Ext2 is a simple, fast and stable file system and while it is easy to repair, its recovery is extremely slow, which is why the developers introduced the journal in Ext3. Ext3 performs slower than Ext2 primarily because of the journal. Different journaling options are supported but by default the file system makes a copy of all meta data and file data to the journal just before committing them to the appropriate locations on the disk device. This method of double writing costs a lot of time in seek and write operations. It does have a speedy recovery though. Another problem with Ext3 is that its methods of block allocation consume too much space for meta data leaving less room for actual user data.

While ReiserFS is a journaled file system that performs much better and is more space efficient than Ext3, it is known for being a little less unstable and has also been known for poor repairing methods.

XFS has been known as one of the better file systems for high performance and high volume computing. It too supports a journal but unlike the Ext3 file system, XFS only journals the meta data of the operations to be performed. It is also an extent based file system that reserves data regions for a file, thus reducing the amount of space wasted in meta data. Data consistency is not as focused here as it is on Ext3.

Is this file system to run on an embedded device? For embedded devices, there are usually limitations to the amount of erase/write operations committed to a flash cell. Which why JFFS2 supports a built in method of wear leveling across the flash device. It is slow but does its job.

Performance Tuning

Much like anything else, file systems can also be tuned. As I had mentioned earlier things such as journaling options can altered. So can other parameters such as limiting write operations as much as possible, by mounting the device with the *mount -o noatime* switch. You can also limit the maximum size of read/write operations when mounting a device.

Other performance gains can be dependent on the method of connection between host(s) to target(s), where 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, this can be configured through device-mapper and multipath-tools. In device-mapper you can configure for the load to be distributed in a round-robin method of access across all available paths or if set up as failover, you can have I/O run on a single active path at a time and make sure that two separate volumes can travel across two separate paths: see Listing 4.

The the above example I have two separate paths for each volume. Each path is going through a Fibre Channel node with a node host number 30 and 31. The first volumes active path is set to pass through host 30 while in the second volume it is the opposite. This way I can have the I/O to both volumes balance across both paths as opposed to limiting traffic across a single path. If an active path were to fail, than I/O would resume in the other path.

As mentioned earlier, other performance gains can be achieved at the SCSI subsystem level by modifying specific disk device or HBA parameters. Again, any and all modifications made will play some sort of an impact to the rest of the I/O subsystem so please proceed with caution.

Conclusion

As one can see, Linux Volume Management is not a simple topic for discussion. A lot is involved when attempting to set up and configure an environment for computing across Linux hosted volumes. As always, before you proceed with any modification review all provided reading materials. It can save you from a lot of future headaches.

Further Reading

Some excellent books come to mind:

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

Resources

- Bar, Moshe. Linux File Systems
- Pate, Steven. UNIX Filesystems.
- Installation and Reference Guide Device Mapper Multipath Enablement Kit for HP StorageWorks Disk Arrays
- Wikipedia Articles on Linux File Systems:
- *en.wikipedia.org/wiki/Ext3*
- *en.wikipedia.org/wiki/ReiserFS*
- *en.wikipedia.org/wiki/XFS*

About the Author

Petros Koutoupis has been using Linux since 2001 and has been in software development and administration even longer. He has been involved with enterprise storage computing from 2005 to the present and currently offers consultation services in the same field. His Web Site is *www.hydrasystemsllc.com*. He can always be contacted at *pkoutoupis@hydrasystemsllc.com*.