CHAPTER 1: DISK AND STORAGE SYSTEM BASICS

by Volker Herminghaus

1.1 Overview

1.1.1 Storage Hardware Situation and Outlook

Disk media are the entities that all persistent user data is eventually stored on. Because the surface of a disk medium can be permanently magnetized, disks can store information across reboots and power failures, when data residing in the computer's internal volatile memory is lost. Disks can not be replaced by any amount of volatile memory. After all, where would you put all that data after a shutdown? But a transition is slowly getting under way: A few months before work on this book was begun, Apple Inc. released a notebook computer that did not have a disk drive but used flash memory instead. EMC, a vendor of mass storage systems, announced a storage array that used flash. These events marked the beginning of a trend away from moving macroscopic mechanical spindles for storing data - an incredibly arcane concept when compared to light-based fibre-channel communications and memory cells holding only a few dozen electrons per bit.

However, flash is still much more expensive than disk storage, and even with prices falling and some problematic properties of flash being alleviated, disk based storage systems will be here for a long time. They will eventually be found at the back end of the storage chain, similar to tape reels in the times of the old mainframe computers. Disk

V. Herminghaus and A. Sriba, *Storage Management in Data Centers*,
DOI: 10.1007/978-3-540-85023-6_1, © Springer-Verlag Berlin Heidelberg 2009

storage will still need to be managed, and volume management software will still do that job. Emphasis will likely be shifting from performance towards reliability, as more people become aware of the fact that with the amount of data processed today, data errors will be a frequent problem very soon. Error rates looked extremely low a few years ago, but when multi-terabyte databases are processed at high speed around the clock, the seemingly low probability for errors that slip through all error checking and prevention mechanisms soon turns to certainty.

Rock Bottom Basics of Hard Disks

You probably know most of this already, but a little walk-through still makes sense because a lot of the terminology introduced here will be used in later chapters. You can skip this if you are very familiar with the interior of hard disks.

A hard disk consists of one or more flat, round platters covered with magnetic material and fixed to a spindle rotating at around 5,000 to 15,000 rpm. At an extremely short distance (about 20nm or 1/30th the wavelength of visible light!) above the platters there are one or more arms ("actuators") moving perpendicular to the rotation if the disks. These arms carry (usually) one tiny solenoid called the read/write head that serves two purposes: When a current is sent through it, then it creates a magnetic field that permanently magnetizes the surface of the disk platter. This is called the write cycle. In the read cycle, no current is sent to the solenoid and the magnetic field rushing along below it induces a tiny current which is then caught by appropriate circuitry and ultimately converted to a binary value, 0 or 1. This bit is shifted into a register while the next bit is read. When a full byte is assembled, the byte is put into a buffer while the next byte is read and so on.

Current 1TB 3.25" disks have bit densities of more than 100 GBit per square inch.

Data on a disk is organized in blocks (also called sectors), and a sector or block is the smallest addressable entity in disk input/output (or I/O). That means that a disk will always transfer whole blocks to the host computer. The length of a block is usually 512 bytes, although some disks use 1024 byte blocks. Each block is protected by a checksum that is written behind its usable contents and is not accessible at the user level. Because of the layout of the disk data hard disks are so-called "**block addressed devices**". I.e. it is not possible to directly change a certain byte or bit on a disk, but the whole sector must be read from the host, modified, and written back. This alone makes access to a disk very different from access to random access memory (RAM).

Furthermore disk data is organized into tracks (all sectors on a surface that are located at the same distance from the center) and cylinders (the same track each across all platters). Fortunately, both of these can be considered irrelevant today; they are mere remnants of past physical qualities and are merely emulated for backwards compatibility. A disk is now simply a device that can read and write blocks of data, linearly addressed by the block number.

From here on we will use the term "**extent**" to specify a stretch of magnetic storage that starts at a certain block number and is a given number of blocks long. It is the most convenient data structure when discussing block addressed devices.

1.1.2 Physical Limits

Whenever one has to deal with physical entities one has to deal with the limits of same. In contrast to objects of the virtual world, physical objects are rigid, inflexible, error-prone and generally undesirable. The purpose of a great amount of software in data centers is mostly to replace all physical objects with virtual counterparts, which can then be used instead of the physical entities. Physical disks are among the most limiting entities nowadays, because they are still dominated by mechanical access methods. This is extremely arcane in comparison to almost all other computer system's components, which are based on electrical or optical components.

Let us look at the limiting physical qualities of physical hard disks:

Performance

Imagine the mechanical overhead that a disk read or write incurs. First of all, the arm assembly carrying the read/write heads has to be moved to the correct cylinder and the correct read/write head is electronically selected. Moving the actuators takes several milliseconds, roughly between 1ms for a close track to 10ms for one that is far away. Next, it must settle on the track (i.e. stop vibrating from the sudden rapid movement). Then the disk electronics must wait for the appropriate sector to actually fly by under the read/write head. This takes, on average, half a disk rotation (the probability for the sector being "close" vs. being "far" is 50%).

Very clever algorithms in the disk's on board controller, like tagged command queuing and elevator sorting try to minimize the effects of the mechanical nature of the device, but after all there remains an average latency of about 5 ms even for very good disks. That means that on average, we can get no more than 200 independent operations per second to or from a disk device. Also, just for comparison, or a bit of data travelling inside a computer would have travelled 1000 km in the same time that the disk read/write moved those 5 mm!

Reliability

If a hard disk fails your data is likely lost forever. Occasionally you may only have a faulty on-board controller which you might replace but more likely the mechanical or magnetic parts have suffered damage. Basically, if you are using hard disks without some kind of redundancy layer on top of it, your only hope is a really good backup system.

Size and Performance per Size

Size is less of a problem now than it used to be, but no matter how many disks you have attached to your system, storing a file that is larger than your disks will simply fail. Now of course you can get Terabyte-sized hard disks, but they are still limited to around 200 I/Os per second. It is hard to imagine a TB in an enterprise database idling around at no more than 200 accesses per second While that sounds reasonable to the average home computer user, data centers handle thousands of users per server concurrently. The real performance measure we need is not size. It is not performance, as measured in transactions per second. It is performance per size! How many transactions per second can be done per GB of data-

base, is the question. A TB of data is never just sitting around except maybe as a database export file destined for a backup device. This kind of file is read and written sequentially so there is less of a bottleneck. But for general-purpose, especially for database volumes, the most important question is how many TX/s/GB can the volume deliver to the database. Due to the exponential increase in hard disk size, the ratio of performance per GigaByte has dropped to abysmal levels in recent years.

Flexibility

To put it shortly: disks are not flexible. You cannot change their size nor their speed nor their reliability. The only flexible thing about hard disks is the wire that attaches it to the computer.

Manageability

Managing a disk that is directly attached to a server means physically going there and plugging or unplugging it from the server or power supply. No remote management is usually possible.

Moore's Law and the Problem with Mechanics

Hard disks became anachronistic in the 1980's when computers started outpacing disks by a bigger margin every year. Unfortunately they are still anachronistic today and we are stuck with them. A well-known fact known as "Moore's law" states that the density of microelectronics doubles every 18 months (or gains a factor of ten every five years). This is basically true for hard disks as well. However, while in computers denser structures on chips increase their processing speed, for disks the increasing density led merely to three things:

1) Increased processing speed in the on board controller of the disk (which never was much of a problem anyway)
2) Increased speed of sequential read/write operations because more data is packed onto each track and is read in the same revolution. This is an advantage only in large sequential transfers, which are not typical for data center usage (databases)
3) Increased capacity of the disk, meaning more accesses per second are directed to the same hard disk

What Moore's law of exponential growth did not help was the rotational speed of the platters, which is limited by the centrifugal force exerted on the platters, and the speed at which the read/write heads are moved by the actuator. The latter is limited by the amount of heat that is generated and must be dissipated from the device. There were efforts to put several actuators into the same housing as well as several read-write heads per platter onto each actuator but for various reasons they all failed in the long run. So in the end Moore's law ran away generating gigantic amounts of storage space, bandwidth and processing power while leaving the hard disks' transactions per second sadly behind, forever tied to their mechanical internals. That is still the situation we are facing today.

It is also the reason why hard disks destined for private or SOHO use are usually larger than "server-grade" disks. It just does not make a bit of sense putting 1 TB onto a single

spindle with one actuator if you want multi-user access on the database that resides on it. But it does make sense to have a few TB on your desktop to use for streaming media and backing up your data. Both are highly sequential types of access, and definitely not multi-user so they can be satisfied with a single, large disk.

Consider the following data points. In 1988, the typical hard disk was 20 MB, cost US$ 1000.- transferred 0.5 MB/s and allowed about 20 random access operations per second. Twenty years later, in 2008, disks are ten times faster and ten times cheaper: 200 random access operations per second at around US$ 100.-. That sounds like a big improvement, but bandwidth has increased even more: 50 MB/s or 100 MB/s are easily reached; an improvement by a factor of one or two hundred! But now take a deep breath and look at the size of the disk: Its capacity has increased from 20 MB to one TeraByte. That is a factor of 50,000 (fifty-thousand)! That means that even though disk mechanics are now ten times faster than they used to be, a very large database based on modern disks is five thousand times slower (measured in accesses per second) than the same database based on old disks. It is also half a million times cheaper.

The point is that you must never base your volume or LUN requirements on size alone, but always mostly on the number of physical disks you need in order to handle the load. Size is irrelevant. Size is basically free. Disks cost money, but it's the physical disks heads that you need in order to perform actual work. Ignore your storage array sales representatives when they talk about capacity in terms of size. They are fooling you. You get much more space per physical disk than you can put to reasonable use. The last type of disk that could efficiently handle enterprise database traffic was the 9 GB 10.000 RPM disk. Current disks only deliver about 1/100th the performance per GB as those 9 GB ones did.

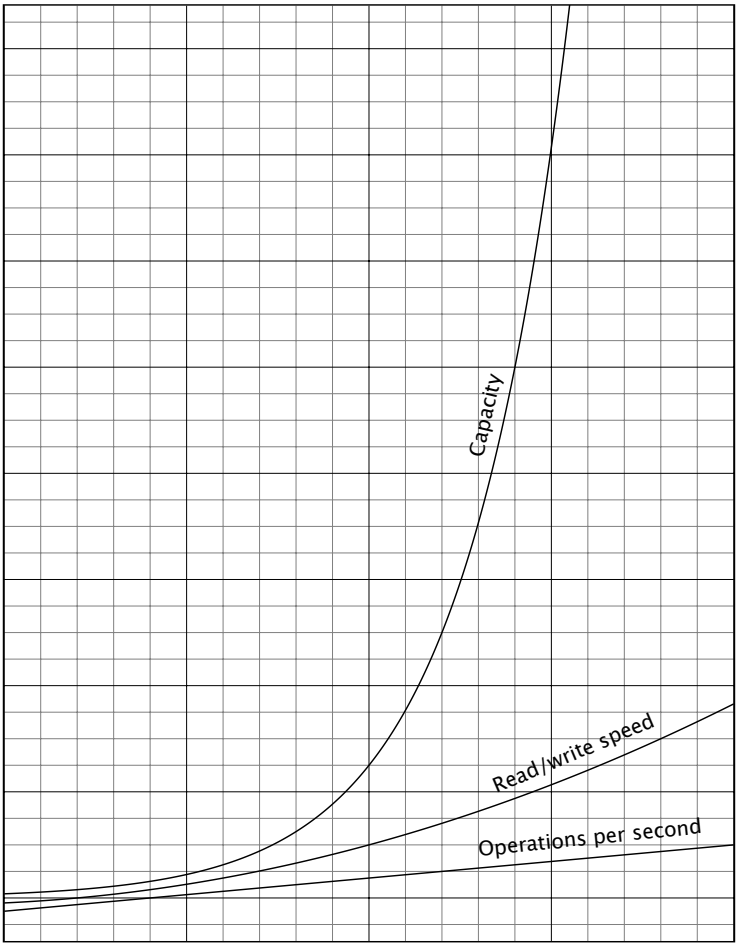

Figure 1-1: Non-quantitative graph showing how Moore's law only applies to disk capacity, while bandwidth and particularly operations per second are left behind. The X axis shows the years, while the Y axis shows the criterion that the curves are labelled with. Note that for the most importance today, namely TX/sec/GB, the capacity (which grows exponentially) is in the denominator, leading to exceedingly poor random read performance!

1.1.3 Trying to Fix the Problems - and Failing!

A number of approaches have been executed trying to get rid of or at least abate the problems caused by the mechanical heritage of physical hard disk drives. They have been successful in the past, sometimes yielding surprising performance benefits at their times and in their area of application. However, all the while Moore's law has been stomping on and on, grinding away any improvement that even the smartest software engineers came up with. Moore's law being exponential in nature has long since destroyed all attempts by storage providers to keep their disk drives' image as a fast, convenient and reliable storage medium. Let us have a look at the various attempts in a little more detail.

RAID Software

In order to alleviate the performance and reliability problems the University of California in Berkeley in the 1980s developed a software solution that allowed to group disks together and distribute and/or multiplex I/Os across all members of the group. They called the software RAID, for Redundant Array of Inexpensive Disks. This was later changed to Redundant Array of Independent Disks by people who wanted to make money off the concept and did not like the term "Inexpensive".

RAID introduced the idea of inserting a virtual device called a "volume" between the application (usually a file system) and the physical disks, thus making it possible to circumvent the restrictions and limits of physical disks to a certain degree. There were different approaches to circumvent the various limitations, each with its own merits and drawbacks. They were called RAID levels. You have probably heard about RAID software and what it does, so this will only be a short introduction into the various RAID levels in use today.

1) **RAID-0 concat** concatenates disks so that when one disk fills up the next one in the chain is used. The capacity of the volume equals the sum of the capacities of the individual disks, and the disks can vary in size. Due to the way most modern file systems are organized, losing any one of the disks means that the volume is no longer usable although one may get lucky occasionally trying to restore that one important file before giving up the volume.

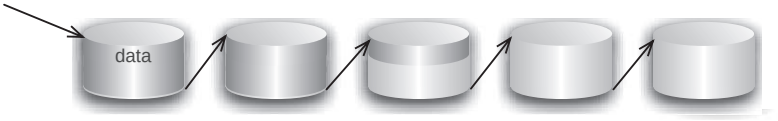

Figure 1-2: A RAID level 0 (concat) volume's block numbers are counted from beginning to end of the first disk, then skip to the next disk. In effect, storage on all disks is appended in a linear fashion. Disks of different sizes and types can be mixed freely.

2) **RAID-0 stripe** interleaves disks with what is called the striping factor, stripe width or **stripe size**. The volume's address space is logically chopped into extents the size of

the stripe width. These are then mapped to the individual **columns** of the stripe set, one column usually consisting of one disk. The first extent is mapped to the beginning of the first column, the second extent to the beginning of the second column and so on, up to the number of columns in the stripe set. The next extents are then mapped behind the first extent on the first column, then the second, and so on. The size of the volume is equal to the size of the smallest disk multiplied by the number of disks in the stripe set.

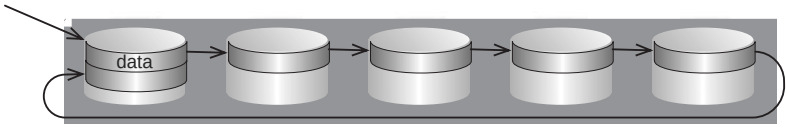

Figure 1-3: A RAID level 0 (stripe) volume's block numbers are combined into chunks of blocks. The number of blocks in a chunk is called the stripe unit size, or stripe size. Each chunk maps to a disk linearly before mapping skips to the chunk in the next so-called "column". A column could be a single disk, a slice (or partition) of a disk, or any concatenation of such. All columns are necessarily the same size. The disks underlying each column must be on separate physical spindles for performance reasons.

3) **RAID-1 mirror** writes data to more than one disk. Each block is written to all disks in the mirror (usually two). Data that is flagged to be flushed to disk synchronously must be persistently written to all members of the mirror set before control is returned to the writing process, while normal, buffered I/O may leave the mirror in an inconsistent state for a while. **Note that this is not a problem** because buffered I/O does not guarantee data persistence to the user anyway!

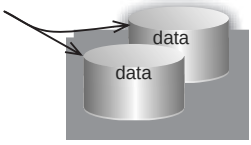

Figure 1-4: A RAID level 1 volume's blocks map to more than one disk. Writes are flushed to all members, while reads are generally read in a round-robin fashion for load balancing. Some low-end RAID solutions try to increase speed by issuing read requests to all members and only processing the first one. While that optimizes single-threaded performance, multi-threaded performance is lost because disk queues become longer and disks are overloaded with unnecessary redundant traffic.

4) **RAID-4 parity** maps extents in the same way that a normal stripe does. But RAID-4 adds an extra column for a special checksum extents created by combining the values of all corresponding extents of the data columns using the lossless exclusive-OR (XOR) operator. Thus, if any of the disks in the stripe set fails, the data for each extent can be recovered by reading the extents from all the remaining disks including the parity disk and recombining them with another exclusive-OR operation. Of course, these operations take time and there are many problems including write consistency and performance especially in degraded mode (when a disk has failed) or with multi-user access. I will not go into great detail about the many performance penalties incurred when doing RAID-4 in software. In short, doing it in hardware is OK, in software it is close to a nightmare.

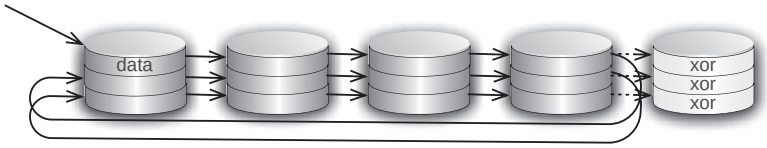

Figure 1-5: **A RAID level 4 volume's blocks map onto the backing store in the same way as a common RAID level 0 stripe, except that one column is excluded from data I/O. Instead, whenever a row of the stripe is changed, RAID-4's special write policy generates an extra block containing a checksum over all data blocks in that row, and writes it to the excluded column. The checksum is based on the lossless bitwise exclusive-OR (or XOR) operation the result of which is `0` if the sum of all input bits is even, and `1` if it is uneven. Therefore, the checksum is also called the parity.**

5) **RAID-5 distributed parity** is similar to RAID-4 but distributes the parity blocks across all columns thus improving RAID-4's performance problem when handling multi-threaded writes. Multi-threaded writes used to be one of the worst flaws of RAID-4 because they overloaded the dedicated parity-disk.

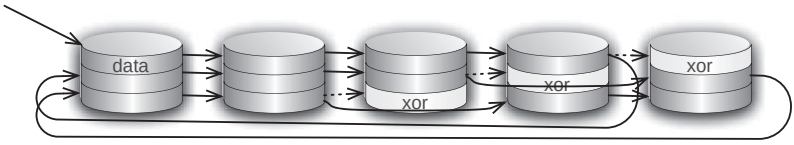

Figure 1-6: **A RAID level 5 volume maps its blocks like a RAID level 4 volume, but parity distribution requires skipping parity blocks during reads and writes. The checksum itself is calculated in the same way as with RAID level 4.**

1.1.4 SAN-Attached Hard Disks

In order to increase performance and manageability the second step the industry took was to introduce a fibre-channel based network for storage devices called a SAN (storage area network). Devices were thus accessible from more than one server and could be managed by programming the SAN switches accordingly. SANs introduced a whole set of problems into the administrator's world, many of which still are not solved. Attaching the disks to a SAN did not help the performance very much, although vendors like to boast about their multi-Gigabit/s connections. Unfortunately the speed of the channel is not the problem, as will become obvious later (beginning on page 214). But I have yet to meet a sales representative that is willing to understand the problem **and** advise the customers appropriately.

Initially SAN disks were packaged in boxes with little if any internal intelligence or caching, thus exposing the physical features of the disks to the outside (so called JBODs, for "Just a Bunch Of Disks"). The first devices of this sort used a rather broken transport protocol called FC-AL (for fibre-channel arbitrated loop) that was designed to make a cheap fibre connection to disks possible without having to buy expensive switches. FC-AL had and still has lots of problems and you do not want to use it except in very price-sensitive environments that do not require good resilience or performance. I.e. not in your typical data center.

1.1.5 Storage Arrays and LUNs

The third step in the scramble to alleviate the problems introduced by the hard disk's mechanical legacy was to bundle groups of disks together into a chassis with relatively large amounts of battery–backed RAM. These assemblies are manufactured and sold by many vendors, e.g. HP, Hitachi, IBM, Sun microsystems, EMC and are called **storage arrays**, **cache machines**, **SAN boxes** or similar. Layout and feature set of all of these devices is similar: They consist of one or more chassis holding the disk units and a central control unit containing back-end controllers that connect to the disks, front-end controllers that connect to the SAN. They may employ interconnects that connect to another box of the same vendor for remote replication or mirroring, and they usually have a large battery–backed RAM as a read– and write–buffer and lots of CPUs that control access from and to disks (back end) and hosts (front end). The disks are grouped into internal RAID groups of some sort (often some variant of RAID-4). In this step, care is usually taken to achieve a good load balance and throughput by applying knowledge about the internals of the storage array's architecture.

Now the RAID group, consisting of several multi-hundred GB disks, is usually much too large for a given problem so it is split into logical units (also called LUNs because they correspond to the logical unit numbers in the SCSI addressing scheme). These LUNs are then mapped to the appropriate front-end controllers via which the host can access them as if they were physical hard disks. To the host computer, there is no obvious difference between a LUN and a physical hard disk.

Usually more than one path is provided by the storage array by mapping the same LUN to more than one front-end controller. The host runs some variant of multi-pathing software at the driver level to make use of this redundancy. The paths are either used one

at a time and only switched when a path fails. This is called an active-passive configuration. Alternatively, the paths can be used in a round-robin manner for load balancing. This configuration is called active-active.

The storage arrays use advanced algorithms to do both read-ahead and write-behind caching, they allow LUNs of various sizes, remote copying, instant snapshots and a lot more. That is why storage array vendors often claim no software volume management is necessary if the customer uses their box.

However, as usual, things are much more complicated than what the sales reps say.

Figure 1-7: A storage array's architecture in principle: Physical disks reside in trays, each of which is controlled by a back-end controller. Slices of these physical disks are combined via RAID logic and create a virtual object inside the storage array called a LUN (Logical Unit Number, from the SCSI addressing parlance). The LUN is mapped onto one or more front end controllers (or "service processors"), from which they can be accessed by the host machines. Each connection from a host to a LUN via a front end controller is called a path. If a host accesses the same LUN via more than one path then multipathing software is required in order to coordinate access and to make use of the extra redundancy. The RAID level used inside the storage array is often a variant of RAID-5.

What LUNs Can Do

- **They can** take lot of writes per second and acknowledge them to the host OS very rapidly, then flush them to disk asynchronously when load permits. This is possible because their RAM is battery–backed. As soon as the data is in the storage array's cache RAM, it can be considered safely written. In case of a power failure the array logic will use battery power to flush the data to the disk drives.
- **They can** deliver pretty high throughput in sequential I/O, both read and write, due to their smart read-ahead and write-behind caching.
- **They can** balance I/O automatically if you let them - they observe usage patterns and move data if necessary to enable more rapid access
- **They can** replicate data to a remote site, but this is only useful in special cases, like short distances or flat file systems.

What LUNs Cannot Do

- **They cannot** offer you the flexibility of storage objects that a software volume management offers. This is due to several reasons:
  1) Most organizations will not allow a UNIX administrator to log into the storage array. Usually someone from the SAN group makes the storage objects (LUNs) for the UNIX admins and that's it.
  2) The granularity of the storage array's external objects is, of course, the LUN. Freeing up bit of space from one volume by reducing its size, then moving the freed space to another volume works on the server, not in the storage array.
  3) Backing up your volume configuration every night and being able to restore it on a per–volume basis and thus recover from all kinds of outages is easy in VxVM. I know of no way to do this in any storage array.
- **They cannot** increase **scattered read** (also known as random read) performance by giving you "cache hits". The myth about cache hits was introduced a long time ago, when storage arrays were sold mainly to the IBM mainframe market. It had some validity in those days but it does not any more. Unfortunately it has not disappeared since. The mainframes of that time used 31-bit addressing (yes, 31 **is** indeed a prime number. Remember we are talking about IBM mainframes here...). So all they could address directly was 2 GB of RAM. Storage arrays have been more free in implementing their internals and they used block addressing, so they could address 32-bit times 512 byte blocks. Having a lot of RAM in the mass storage system made some sense in those days, especially when the disks were smaller than they are now. Total RAM would be, say, 64 GB, and total disk capacity maybe one TB, which yielded about a 6% cache rate (see picture). Together with the OS's limited address space there was actually a pretty good chance for cache hits, especially because the storage array was often dedicated to a single mainframe. Nowadays however servers use 64-bit addressing. They tend to have much more RAM than they used to, easily going into the hundreds of GBs. What's more, many servers usually share a single storage array. And the disks inside the storage array are much larger. All these factors together distort the magnitudes enough to make the cache completely irrelevant for scattered

reads. And even if we were lucky and the desired block actually was in the storage array's cache, then it is almost guaranteed to be in the server's cache anyway. Because the amount of cache that the storage array allocates **per server** is usually much smaller than that server's file system buffer cache. So you can safely forget about speeding up random read access using storage arrays. The only thing the cache does effectively is read ahead and write behind and thus speed up sequential read and all write transfers.

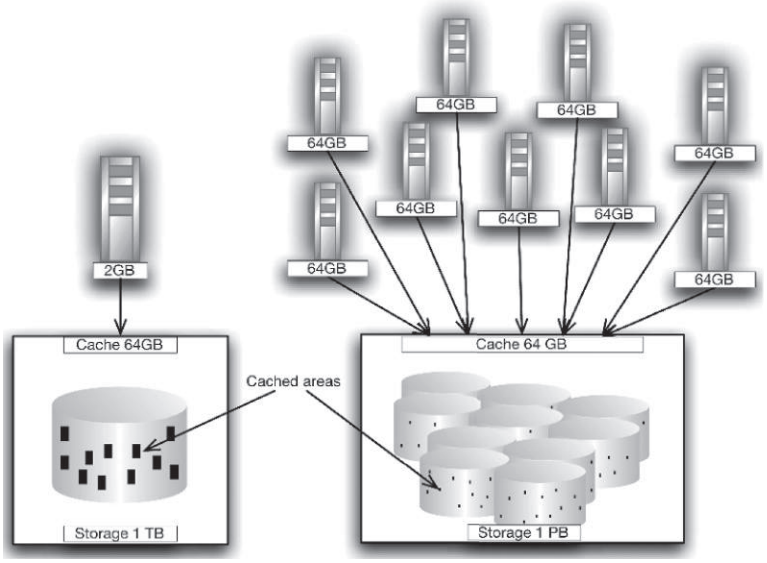

Figure 1-8: While ten years ago storage arrays may have offered some cache hits due to the ratios of OS memory, number of machines per storage array, and cache/disk inside the storage array (left), today a cache hit for a scattered read is almost like winning the lottery (right).

- **They cannot** replicate online database traffic across great distances. Replication means storing an consistent copy of the data in a remote location for disaster recovery. It is different from a mirror in that updates to the replica may be delayed somewhat while updates to a mirror must not be. In addition, the replica is not accessible to the user at the remote site until - usually while testing or after a disaster - the direction of replication has been reversed. While some storage array vendors will claim that their hardware replicates databases quickly and consistently across great distances, this is not true. They may not know they are not telling the truth, but the simple fact is that physics makes it impossible. If you are interested in why the speed of light is too slow, and why 4 GBit/second are not helpful when it comes to long distances, read the discussion about light speed and protocols in the chapter about

dual data centers (page 214). The conclusion is that in order to replicate database traffic quickly and consistently the replicating agent needs information about the write sequence that only the operating system has. Therefore it is not possible to achieve this goal with a purely external solution but you need a special device driver. Veritas Volume Manager comes with a built-in solution for replication that is both fast and consistent. It is called VVR, or Veritas Volume Replicator. Unfortunately there is no free lunch and VxVM-based replication is not easy to learn or administer.

1.1.6 Common Problems

Scattered Read Latency

All the technological advances of the last thirty years have failed to fix one basic yet crucial problem: access to a random data block generally incurs a relatively long latency consisting of positioning the actuator and waiting for the right sector to fly by. In fact, the problem got worse and worse. It is still getting worse with every new generation of hard disks. You need to understand what exactly the problem is so that you can make smart decisions about the layout of your storage. Let us look at the problem, at how bad it already is and why it keeps getting worse.

When RAID was conceived the usual hard disks were a few dozen megabyte in size, let's say 20MB. A typical disk of that era had a rotational speed of 3600rpm, a data transfer rate of half an MB per second and an a seek time of about 60ms. It used an interleaving factor of 3 to 5 because the interface between disk controller and host computer could not transfer the data at the full speed of the rotating platter. so you had to read the same track several times in order to transfer all of it to or from the computer. Let's look at a sample data transfer from one of these disks:

1) Position the actuator - 35ms
2) Wait for head to settle down - 7ms
3) Wait for first sector under read/write head - 8ms
4) Read track four times to gather all the sectors, and transfer - 70ms

That's a total of 120ms. You could do eight of these large I/Os in a second, or twice as many if you only read one sector per I/O. Now what can we do today? In 2008 hard disks have much faster access times: a good average value is 6ms; ten times faster than in the days of the first RAID concepts. The also boast transfer speeds beyond 50MB/second, that is one hundred times the bandwidth we used to have. They also store one terabyte instead of 20MB. A database of 1GB that migrated from fifty old 20MB disks with their 60ms latency to fifty of today's 6ms hard disks would be ten times faster in random access, and a hundred times faster in sequential access! It could accommodate ten or twenty times more users than the old setup, depending on the I/O mix! It even turns out it would be about twenty times cheaper even with the same number of disks because hard disk prices have dropped a lot since the 1980s.

So what is the problem?

The problem is that, while access times have improved by a factor of ten, and band-

width has improved by a factor of one hundred, size has increased by a factor of 50,000 (fifty thousand, from 20MB to 1000 GB)! So nobody is going to put that 1GB database on fifty individual spindles because that would waste fifty terabyte minus 1 GB of hard disk space. Try explaining to procurement why you need 50 disks and only use 0.002% of their space. Well, let us be fair: you may not need the tenfold performance increase; you may just stick with the performance you used to have, so you can actually use ten times as much space: 0.02%! Try explaining that. Good luck!

The problem is that disks have so incredibly much space that you are tempted to use it, but because the mechanics basically haven't changed since 1980 you cannot use it for data that is accessed in a random fashion.

Yet people do it and that is what causes many of today's performance bottlenecks. How come most people do not recognize the problem? It is hard to say, but one thing is probably that benchmarking is often done the wrong way. Many times people benchmark only the "classic worst case" scenario, namely: scattered write I/O. This kind of I/O is the worst case for **physical** hard disks because not only does a write access incur the seek latency and write time, but the disk controller also has to verify that the data was properly written, which means that another revolution of the disk has to be waited for. A **physical** hard disk in a data center should never acknowledge a write I/O as soon as it has received the data from the host computer and put it into its cache (write back mode, the default on many personal computer systems). The data is only safe when it is actually persisted onto backing store, i.e. on its magnetic media, and that is when the I/O can be safely acknowledged to the host (write through).

The second worst case is scattered read I/O, but why measure the second worst if you can measure the worst case, right? Wrong!

A storage array buffers all writes, both scattered and sequential, and acknowledge them to the host as soon as the data is in its cache. It does write back instead of write through, and in the case of the storage array, that is OK. Remember that the storage array has internal batteries that keep the array running in case of a power failure. Additionally, data in their cache is usually organized with enough metadata so that even when the storage array CPU fails it will replay the data from its cache onto the backing store when control is regained by the CPU.

So if scattered write performance is measured on LUNs the result is hugely distorted due to the storage array's caching effects. The write benchmark merely measures the speed of the channel and the controller, which is fair by itself because he storage array will actually deliver that performance in real life, too. But what people tend to forget is that what used to be the second worst case is now by far the worst case: scattered reads.

Another weak point in benchmarking today is that benchmarks (and optimization runs) are usually run on a single machine, while in fact the storage array is (or will be) connected to dozens if not hundreds of machines, each of which will put some load on the array. Because a combined benchmark is normally impossible due to logistical limitations (who could afford to shut down the whole data center just for benchmarking?) people limit themselves to benchmarking a single machine. But the results of such benchmarks are usually invalid because they do not replicate the real world in any significant way.

1.1.7 Physical Disks vs. LUNs

Advantage LUN

Being the more recent and modern type of storage, LUNs have a number of advantages over physical disks. In particular there are the following advantages to LUNs:

- **Write Performanc**e: Storage arrays are at an the advantage when it comes to writing (both random and sequential) because of their write behind caching strategy. Because the storage array is built to survive power outages and other mishaps, a data block received from the host can be acknowledged as soon as it arrives at the storage array. There is no need to wait until the data has been persisted to magnetic storage. The storage array's cache memory is already persistent, and disks are just the backing store into which the cache contents are flushed for long-term storage. Because the storage array acknowledges received blocks immediately, the time waiting for the mechanical components of the disks is saved.
- **Sequential Read Performance**: Sequential (or "streaming") reads are also served very well by storage arrays because due to their vast caches they read ahead many more data blocks than an individual disk with its limited memory could. When the prefetched data is subsequently requested by the host the storage array can deliver it much more rapidly than a disk could. This is not because the storage array has more powerful CPUs. In fact, its I/O controllers are industry standard components. It is purely because the storage array has read so far ahead that no actual disk head movement needs to be done and thus the crucial bottleneck is mostly avoided when doing sequential I/O.
- **Reliability**: LUNs are almost always based on some kind of redundant internal construct. In many cases, some variant of RAID–5 is used. As a result, read/write errors from head crashes, power supply or even logic board failures on disk spindles, simply do not happen. LUNs are like disks that cannot break - unless the SAN admin inadvertently breaks them by misconfiguration. You may still lose part or all of your connectivity to the storage array, and you may still lose a LUN due to administrator error. But having defective sectors on a LUN is next to impossible.
- **Management**: Apart from performance and reliability they also have an advantage when it comes to (remote) management. Most large storage arrays include fibre-channel switch hardware that is integrated with the array logic to facilitate easier zoning, masking and mapping of LUNs to hosts. While this could also be done using external SAN switches and fibre-channel JBODs it is generally easier to use the integrated approach.

Advantage Disk

What, using physical disks has advantages over using LUNs? Yes, it does, and probably more than you would think possible:

- **Lower latency**: While writes and sequential reads are usually better served by a storage array, the additional overhead created by talking to the front-end controller

which puts the request into a queue from whence it is passed to the appropriate back-end controller which then talks to the disk, and the whole way back introduces significant extra latency. This aggravates the scattered read latency problem, which is already the ultimate bottleneck in today's storage systems.

- **Transparency and Dedication**: When you encounter a performance problem on a database the classical approach is to consult your sysadmin tools to find out which disk gets the most I/O and then balance it using the appropriate tools When you do this, then if you are running from physical hard disks you can be rather confident to get the expected result. But if you are running on a storage array chances are very high that you are not alone on the array, or on the physical spindles inside it. What appears to your sysadmin tools as a spindle is in fact a complicated construct comprised of several spindles, each of which may be part of more such constructs (see picture page 12). These are likely in use by some other machines in your data center that you may not even be aware of. It is a frequent complaint that storage arrays give relatively good average performance but can suffer unpredictable and severe performance degradations occasionally. This is often because some other machines peak at that time because they are doing disk-intensive tasks, like database imports, full table scans, export or backups, taking away all the IOPS (I/Os Per Second) which you thought were exclusively yours.
- **Price**: JBODs are typically much cheaper per GB than storage arrays. This is not only because JBODs need less components, but also because the target market for storage arrays is mostly medium-sized to large enterprises who are expected to put their most mission-critical data onto storage arrays. Therefore, the array vendors must offer thorough, worldwide support on a 365/24 basis. They must also test their equipment very thoroughly to exclude bugs as much as possible, and work together with server, HBA and SAN hardware makers on interoperability issues. All this consumes a lot of financial resources which must be recovered by the higher equipment price.

Other features like snapshots, redundancy etc. can all be done in software with physical hard disks and Volume Manager. There is just one special thing that cannot easily be done in software, and that is creating an incremental snapshot of a volume (a snapshot based on the differences from the original) and moving it to another host for offhost processing. This is logically impossible for a host based approach because both hosts would need to have read and write access to the volume, and only one of them would maintain the snapshot. This would not work without a lock manager and a modified file system. Veritas has actually implemented that with a product called a "Volume Server" but that product is not widely used and it may never be.

1.2 Disk Addressing and Layout

Blocks and Extents

Every object that the host system sees as a disk, i.e. a LUN, a physical disk inside a JBOD, or a single disk (e.g. the boot disk) must be addressable in the same way or else there would be different code paths for different media types. Both developing and debugging for the programmer as well as administration and fault analysis for the administrator would be unnecessarily complicated by this. Fortunately, it is a universal truth that LUNs, JBODs and physical disks are indeed addressed in the same way and share the same layout.

As discussed before, disks are basically addressed as a one-dimensional address space segmented into blocks of BLKSIZE bytes (usually 512, sometimes 1024). The tracks and cylinders do not actually play a role any more except to shoot yourself in the foot if you mess them up. They are purely legacy information. In former times this information was used to optimize disk access, but it is irrelevant today. The SCSI device driver expects every hard disk to report them and so they diligently do, but the actual physical layout of the disk uses a variable amount of sectors per track (outside cylinders are longer so they can hold more sectors than inside cylinders), and the number of cylinders and even heads is emulated by the disk's firmware. Only when mirroring boot disks the cylinder information plays a little role, because the Solaris VTOC entries are required to start on cylinder boundaries.

In order to illustrate disk addressing we will use several variations of the following graphical element for the disk address space, in which each of the little rectangles stands for one disk block or sector:

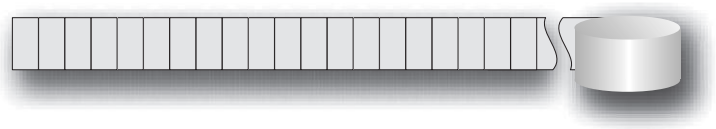

Figure 1-9: A disk is addressed as a sequence of blocks or sectors of the same length, usually 512 bytes (1024 bytes on some HP-UX machines)

The next step up from disk blocks towards something more usable is the **extent**. An extent is a data structure which is widely used in both Volume Manager and File System. Once you get used to extents you begin to wonder why everybody does not use them since they are such a useful abstraction. What is an extent? An extent is a contiguous range of blocks starting somewhere at a given block number and stretching over a number of blocks. Veritas File System uses only powers of two for extent sizes, but VxVM uses a less stringent definition of extent: an extent in VxVM is simply a sequence of disk blocks defined by a starting block and a number of blocks (or a beginning and a length, if you prefer that notion).

A partition (another word for "slice") can also be seen as an extent. It starts at an offset (given as a block number) and extends across a number of disk blocks. An extent is anything that is given as an offset and a length. The disk itself is an extent, beginning at block zero and extending across the whole disk. A file could – almost – be an extent if the file system was smart enough to recognize what it is doing and allocate contiguous storage space. It is not really an extent, even if it is contiguously allocated, because it is not block-addressed but byte-addressed, i.e. it could end anywhere inside a disk block. By the way, VxFS actually does a very good job at allocating blocks contiguously, as does UFS, the common UNIX file system. Current implementations of UFS are derived from BSD's Fast File System (FFS); the original UFS from AT&T was very poor at allocating contiguously.

VTOC, Partition Table, Volume Label

At some fixed location on whatever the operating system identifies as a disk (i.e. a real hard disk or LUN) there needs to be some meta information that describes that medium. Such metadata include the length of the medium and the location of some critical extents like boot code or root file system etc. In principle it would be possible to derive much of the required metadata via I/O control system calls (ioctls) from the device driver (e.g. the length and blocksize of the device could be determined this way). However it makes a lot of sense to do this only once, then store the results along with other metadata in some fixed location on the device so it can be read and written without hassle.

In Solaris and several other operating systems, the extent that holds the metadata starts at block zero and has a length of one block. In other words, block number zero holds the metadata for the device. The Solaris name for the metadata extent is VTOC, which stands for Volume Table Of Contents. Other names from other operating system parlances are "partition table" or "volume label".

That metadata contains additional information about the usable extents residing on the medium. These extents are commonly known as "partitions" or "slices". They start at cylinder boundaries and are well enough integrated into the host system that they can be directly addressed, even before the boot process has started. For instance, a disk with a Solaris VTOC contains descriptions of eight extents. An extent of zero length that starts at offset zero is considered invalid (but still exists in the VTOC). Extent number two is by default initialized as an extent that contains the whole disk and is called the "backup slice". (A long time ago, when Moore's Law was not yet applicable to hard disks and you could actually buy the same type of hard disk for a few years, system administrators liked to make backups of their system disks by copying data from the "backup slice" onto tape. Note that the backup slice also contained the metadata extent (block zero) so when a disk was terminally broken one could install a new disk, copy the backup from tape to the new disk device's backup slice and thus recover both metadata (VTOC, slices, boot code) as well as user data of all file systems in one step. Hence the term "backup slice".)

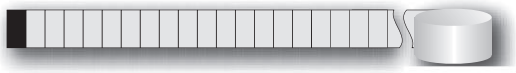

Figure 1-10: The Solaris VTOC (black) is located in a one-block extent starting at offset zero. It points to up to eight extents (on SPARC systems) which are also called slices or partitions. Slice 2 is an extent that covers the whole disk.

In Solaris, the extents known as slices or partitions carry a tag in the VTOC that is used to identify the partition's purpose. For instance, there is a special tag for the swap slice which is used by the Solaris installation program to find an extent into which a "mini-root" file system can be written without possibly overwriting important user data.

Boot Code

The boot code, also called boot block, is (again, in Solaris systems) located in an extent that is 15 blocks in size and starts at an offset of 1 block from the root slice. Why is there an offset of 1 block? It is there simply in case the root slice starts at cylinder zero of the disk. In that case, if the boot code actually started at block zero it would overwrite the VTOC (which is located at block zero of the disk) and render it useless. So to accommodate for this special case, block zero of every extent on a Solaris disk is unused by any higher-level code.

The boot code contained in these fifteen blocks is smart enough to read UFS file system structures and load the kernel from its subdirectory path in e.g. /kernel/sparcv9/.... There will be more on booting and the boot process later in the discussion of making a third boot disk mirror for maintenance purposes.

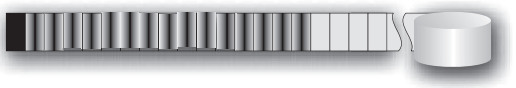

Figure 1-11: The boot code (grey) on a Solaris disk resides in an extent that starts at offset one from any slice and is 15 blocks long. This slice is tagged as the `boot` or `root` slice, which contains the root file system.

Similar to the gap left for the VTOC "just in case" the slice happens to start on cylinder zero of the disk there is another gap that the file system leaves for the boot code "just in case" it happens to be located on the root partition: The first sector, block zero, is skipped because we do not want to overwrite the VTOC. The following fifteen blocks, block 1-15,

are also skipped by the file system in order to prevent overwriting the boot block if this slice happens to be the root slice that contains the boot code.

In summary, a Solaris slice or partition may start at any cylinder boundary but the file systems that reside in it will always skip the first 16 blocks "just in case" they happen to be on the root slice so they will not overwrite the VTOC or boot code. This is why the super block for a Solaris file system always resides at block 16 instead of block 0 of a slice. The same holds true even for database using raw devices: their access methods also skip the first 16 blocks for the same reason.

Figure 1-12: The superblock, the entry point for a file system, is contained in an extent that starts at offset 16 and has a length of one block.

Slices or Partitions

Slices, also called partitions, are container extents into which a file system or a database writes and from which it retrieves data that was previously written. They are discussed here because of the implications they have for volume management. A volume is, after all, a logical construct that ultimately serves as a backing store for a file system or a database raw device. To paraphrase: a volume is an extent that must act as an **exact equivalent** to a slice under all conditions. If a volume would behave in even a slightly different way from a partition the file system or database accessing the volume could run into situations it is not prepared for, and crash.

What is the most critical nature of a disk extent that must be emulated by its logical equivalent, the volume? Well, of course it must be able to store data in a persistent way. It must also adhere to exactly the same semantics on the driver level; the file system or data base driver must not be forced to use a different paradigm for accessing a volume than the one it uses for accessing a slice. The most crucial part is, however, that under all circumstances the virtual equivalent of a slice – the volume – must deliver one and only one set of data contents for any specific block until that block's contents are changed by that same device driver. It is not at all obvious that this is always the case. For instance, consider a volume that is a three-way mirror. If during an update to this mirror the host loses power, then because not all extents are written at exactly the same time you may have up to three different contents of any data block that was being written to when the outage occurred. Which one is "the right copy"? Should the write be transactional so that this cannot happen? Should we always refer to a "MASTER" copy, a preferred mirror side that is always up-to-date? Then what if the disk holding that mirror fails?

These questions will be answered beginning on page 132 and you will be surprised at how sophisticated the problem, yet how simple the solution is.

1.3 Paths and path redundancy

A disk is not worth much if there is no way to access it. In order to access a disk there must be one or more I/O paths to it that the operating system can use. Over the course of time many types of paths have been developed. The ones most commonly used in data centers will be discussed here. They are: SCSI, fibre-channel (FC) and iSCSI. Two of these, FC and iSCSI, are network protocols, while SCSI is point-to-point and has been abandoned in most data centers by now. The SCSI protocol is still discussed here because its command set is the foundation for most block-addressed storage today. The SCSI heritage turns out to become a big problem occasionally, as we will later see.

SCSI and SCSI Addressing

We will not go into the historical details about how Alan Shugart invented SCSI as a network protocol for hard disk access. There are better sources for that kind of information. But it is important to know the naming conventions and some protocol intricacies in order to understand the later chapters, especially latency concerns and system deadlocks.

First of all SCSI is a stateful protocol that uses commands sent from the initiator (usually the host computer) to initiate data block transfers between the initiator and the target. A transaction consists roughly of the following steps (for a data read): first the initiator selects the target which the target acknowledges. The initiator then sends the command, e.g. a read command, which is again acknowledged. The target can then choose to decouple from the initiator and fetch the data. When the target has retrieved the data from its storage medium it reconnects to the initiator who then fetches the data from the target and ultimately deselects the target.

How is the target addressed?

The target address used to be an integer number between 0-7, later 0-15, that was put onto the three (later four) address lines of the parallel SCSI interface. The operating systems had internal logic to translate names like **/dev/sd4**, **/dev/sd5c** etc. to the appropriate counterparts on the SCSI bus, in this case SCSI-ID 4 and SCSI-ID 5 slice 3 (the **c** in **sd5c**) etc. Later, when multi-instance devices appeared and systems with several controllers became more common this naming scheme became very inconvenient and had to be extended by what has become known as the LUN address.

The SCSI LUN

Imagine a device that houses more than one actual media. A good example is a CD-ROM changer. Such a device consists of just one controlling unit and therefor occupies just one SCSI target. But it is able to address more than one logical device, namely the individual CDs in the slots that the changer provides. In order to address such multi-instance devices an extension to the SCSI protocol was provided called a "Logical Unit Number". Does that sound familiar? It is the term we use when talking about virtual hard disks acquired from a storage array, the LUNs. Remember that storage arrays are multi-instance devices, too. They consist basically of a control unit with a mass-storage back end and can deliver multiple instances of block-addressed storage, so it makes sense to apply the same addressing scheme to them as with other multiple-instance devices. Initially there was a maximum of

16 LUNs due to limitations of the number of wires on the parallel SCSI bus. That limitation does not exist any more with serial interfaces (the limit was due to the number of address lines on the parallel SCSI cable) so it is up to the device driver how many LUNs it can address. Usually 256 LUNs per target is the limit.

Modern UNIX Device Naming Convention

Using names like **/dev/sd4** or **/dev/hdisk5** does not work very well when hundreds of disks need to be addressed. First of all it clutters the **/dev** directory. Then, all the names may change when a disk is added or removed and the system is reinitialized. That makes it very hard to keep the file system tree organized in a system with frequent device changes. So a more clever naming scheme was conceived, which identifies a device by the various entities on the path to that device: Controller or host-bus-adapter (HBA), Endpoint (Target), Disk (or LUN), and Slice. Typically, a path to a disk block device would be /dev/dsk/c#t#d#s#, with the # standing for the corresponding object number. The controller is the operating system's internal controller number that has been enumerated upon boot by the hardware integration layer. The endpoint or target number is the SCSI-ID in case of JBODs or – in case of a storage array – the fibre–channel port in the array to which the controller is connected. The disk number is the port-specific number of the array-internal volume. Each port of a storage array gets a number of array–internal volumes (the "LUN" in the storage array picture on page 12) for each connecting host bus adapter. That internal number is passed on the SCSI bus to the host and turns into the disk number in the device tree.
To prevent clutter in the **/dev** directory, the device nodes are put into separate subdirectories for block and raw addressing called **/dev/dsk** and **/dev/rdsk**. A device name like **/dev/dsk/c4t9d2s0** identifies the block device for controller 4 -> target 9 -> disk 2 -> slice 0, and **/dev/rdsk/c8t15d7s2** identifies the raw device for the whole disk on controller 8 -> target 15 -> disk 7 (remember that slice 2 addresses the whole disk in Solaris).

When disks are added or removed on one controller or target, this does no longer change the names of all the other entities on different controllers or targets. This naming scheme is very convenient since it is immediately obvious which disks are connected to a certain controller (c4) or a certain storage array port (t9). There are other naming conventions but they will not be used in this book.

Fibre-Channel

Fibre-channel is currently the most widely used interface for disks in data centers. Fortunately, switched fibre-channel fabrics have displaced the previous, rather unreliable and slow FC-AL (Fibre-Channel Arbitrated Loop) architectures. The physical layers of fibre-channel are not too interesting in this context and are not covered in this book; there are many good books that explain FC very well (I very much recommend Tom Clark's "Designing Storage Area Networks"). But what you need to know are the following facts:

- Fibre-channel can use both copper or light as the physical transport medium. Copper is used for short distances only (usually inside the machine or array) while several variants of glass fibre are used for cheap medium-range connections (multi-mode) or more expensive long-range connections (mono-mode). It is quite common to multiplex several light connections onto a single physical channel to increase bandwidth without increasing cost at the same rate. This is done by a technique called [D]WDM

for [Dense] Wavelength Division Multiplexing. This type of multiplexing means no more than using lasers of different wavelengths (i.e. colors) in parallel on the same fibre.

- Fibre-channel uses a variant of the SCSI protocol command set called FC-3 (for Fibre-Channel based on SCSI-3). In fact, the standard SCSI protocol driver is used on top of the FC host bus adapter (HBA) driver. This can cause great problems as you will see toward the end of this chapter.
- Fibre-Channel is a network architecture. It was originally designed to replace ethernet for high-end applications but that failed due to the high cost of recabling, and because Ethernet developed very quickly to Gigabit versions. Fibre-Channel works with multiple initiators, multiple targets, switches and routers. It also has the usual set of network problems like missing or wrong routes, nodes that fail to answer or answer late, buffer overflows etc.

iSCSI

This protocol is gaining momentum because it allows to use a storage area network to be installed over an existing ethernet infrastructure. The expenses for fibre-channel components are saved, and administration is simplified. Similar to FC, iSCSI uses the existing SCSI protocol driver on top of the TCP driver to directly address iSCSI devices as block devices. (This is in contrast to NFS where the server does not serve data blocks but file semantics. NFS and other file servers are outside the scope of this book.)

Multipathing

No matter which protocol is used it is always a good idea to have redundant paths to the disks. SCSI, Ethernet and FC connectors are not perfect and they can only withstand a very limited amount of force. In addition, especially in the case of network based storage it is always possible that an intermediate node loses power or crashes. If there was no path redundancy you would lose disk connectivity immediately and the data would no longer be accessible. If you were lucky, then you would lose only one side of each mirrored volume, but you would have to resynchronize all mirrored volumes after such an event. All this can be prevented by having redundant paths. But there are more reasons for having path redundancy:

- You can switch one of the paths off and upgrade the firmware on your HBA or on the storage array's controller to which this path is connected. After the upgrade is successful, you switch the path back on and repeat the procedure with the other path. This enables online upgrades with no downtime.
- The load is distributed across more than one controller so that peak loads do not run into a bottleneck.

Multipathing drivers come in a variety of flavors. Some are provided by the operating system vendor, like Sun microsystems' MPXIO driver. Others are provided by the storage vendor, like EMC's PowerPath driver. And some come with the volume management software, like Veritas' Dynamic Multi Pathing (DMP). There are two places in the driver stack where they fit in: below the SCSI protocol driver or above it.

Sun vHCI Driver

The vHCI (virtual Host Controller Interface) driver that implements Sun's MPXIO is an example for a nexus driver. A nexus driver is a driver that is part of a driver chain, i.e. one driver accessing or being accessed by another driver. In this case it operates between the SCSI protocol driver and the HBA (host bus adapter or pHCI, for physical Host Controller Interface) driver. The SCSI driver therefore only sees a single path which never seems to fail because the software driver below uses two or more redundant paths to do the I/O. That makes it easier for software that sits on top of the SCSI protocol because that software will rarely have to deal with path errors. It also makes it easier for system administrators to identify multipathed devices because each device only appears once in the device tree, only has one device node etc. More about the vHCI device driver can be found here: http://www.patentstorm.us/patents/6904477-description.html and, of course, later in this book. Here is a somewhat simplistic graphical depiction of where the vHCI driver resides in the driver hierarchy of a system running Veritas Volume Manager:

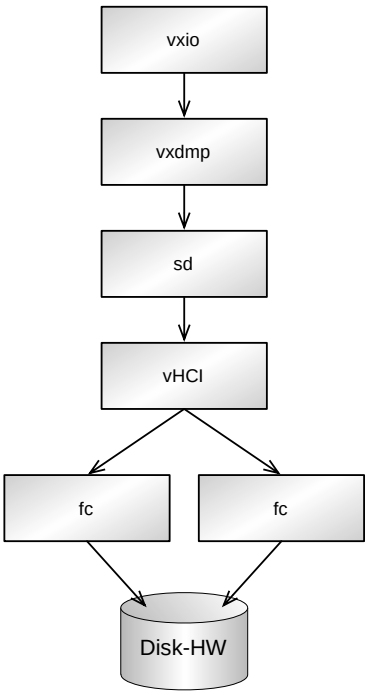

Figure 1-13: The Sun vHCI driver is located below the sd driver. The operating system only offers the DMP driver a single path to each disk

Veritas DMP Driver

The Veritas DMP driver is a layered driver that sits on top of the SCSI protocol and bundles several standard SCSI paths to one redundant path. The **/dev** directory keeps the device nodes for all SCSI devices after DMP is installed, but a **/dev/dmp** directory is added, which contains meta-nodes for devices accessed via the redundant paths. This has the advantage of making online installation and deinstallation of the driver possible because devices can still be accessed via their original device nodes. But this can be both positive and negative for the system administrator because common operating system commands and utilities continue to work on the original, individual paths, while Volume Manager and its tools use the redundant paths. For instance the Solaris **format** command may display two or four times as many disks as are actually connected to the system if they are connected via two controllers or two controllers and two switches, respectively. This can be confusing to the uninitiated. The following graphic illustrates where DMP resides:

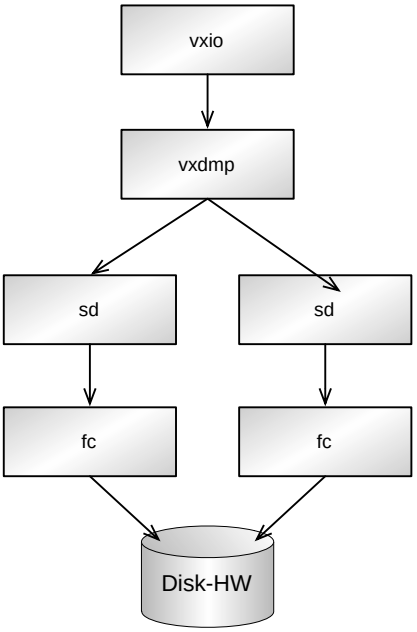

Figure 1-14: The Veritas DMP driver is located above the sd driver. The operating system offers the DMP driver several paths to each disk

How Does the DMP Driver Work?

This is obviously not a discussion of actual DMP implementation details, but one can get a good idea of how DMP works by using the following simplified description:

Whenever a device discovery is started (e.g. upon system boot or when the administrator asks VxVM to scan for new disks) DMP reads the disk's unique ID (UID) from all devices

in **/dev/rdsk/c*t*d*s2**, i.e. from all disk devices that are known to the system (CD-ROMs etc. are skipped). It then builds a list of all UIDs and adds to each UID all paths via which that particular UID was found. This is called building the DMP device tree. The result of this is that each disk device is mapped to all paths that reach that particular device.

During operation DMP issues I/Os for a certain UID in a round-robin fashion across all its paths. If a path encounters an error it is marked bad and henceforth skipped for I/Os. In order to regain that path when it comes back online a kernel thread called **restored** reads the list of bad paths from the dmp driver at a configurable interval (default: 300 seconds). It then issues a small read-I/O to test if the path has come back online. If it has, then that path is reactivated in DMP and taken off the bad path list and thus returns to normal operation.

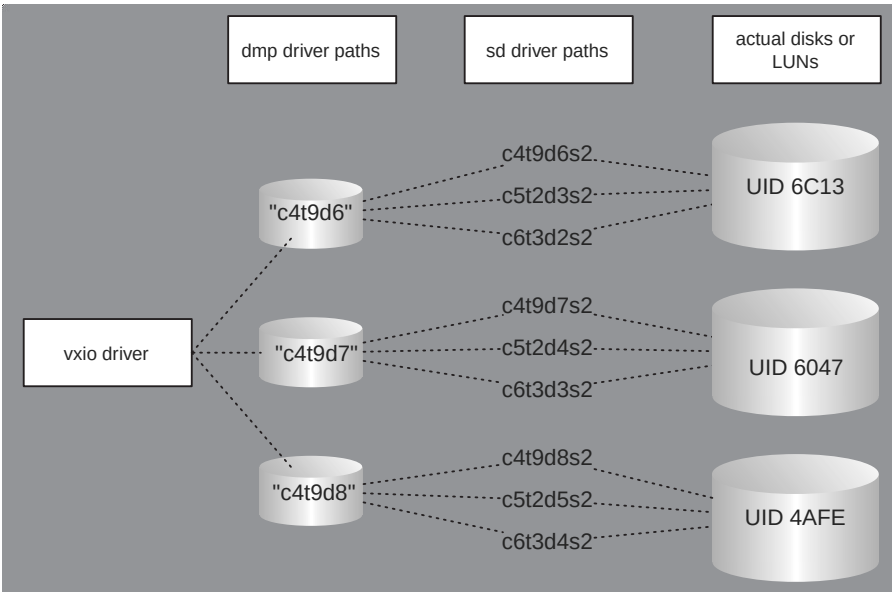

Figure 1-15: Upon initialization, DMP reads all the SCSI disk device nodes and finds their universal ID. It maps all nodes that return the same UID to a logical device and gives it a unique "access name". Subsequently, accesses to that UID are routed over all available paths that originally returned the same UID. The standard sd driver remains functional, but is unaware of the fact that many of its paths are actually the same device.

EMC Powerpath

Powerpath is a commercial product which is sold by EMC's strong sales force. It resides on the same layer as the **dmp** driver, i.e. on top of the **sd** driver. But it does something that should be considered at least problematic: Besides routing I/O to the powerpath drivers it also intercepts calls to the normal **sd** drivers underlying powerpath, and reroutes them via powerpath's internal logic to any of the paths via which the desired target can be reached. It therefore interferes with the operating system's own drivers as well as with other multipathing drivers.

Using Powerpath and DMP Together

Powerpath **cannot** be used in conjunction with DMP without a major hassle or risking data loss under certain circumstances. If a path fails, DMP and powerpath will not typically recognize the failure at the exact same point in time, so write I/Os that are pending via DMP may be rerouted via powerpath to a path that has already been found faulted by DMP, or vice versa. The result is that those I/Os may never be flushed to persistent storage and/or DMP may never find out about this and notify the **vxio** driver (which could then signal the application to retry the I/O). The result is occasional data loss in case of path failure.

Because powerpath does not seem to solve any problems that DMP hasn't solved we strongly advise against using them both at the same time. The illustration depicting the combination of DMP and powerpath hopefully speaks for itself:

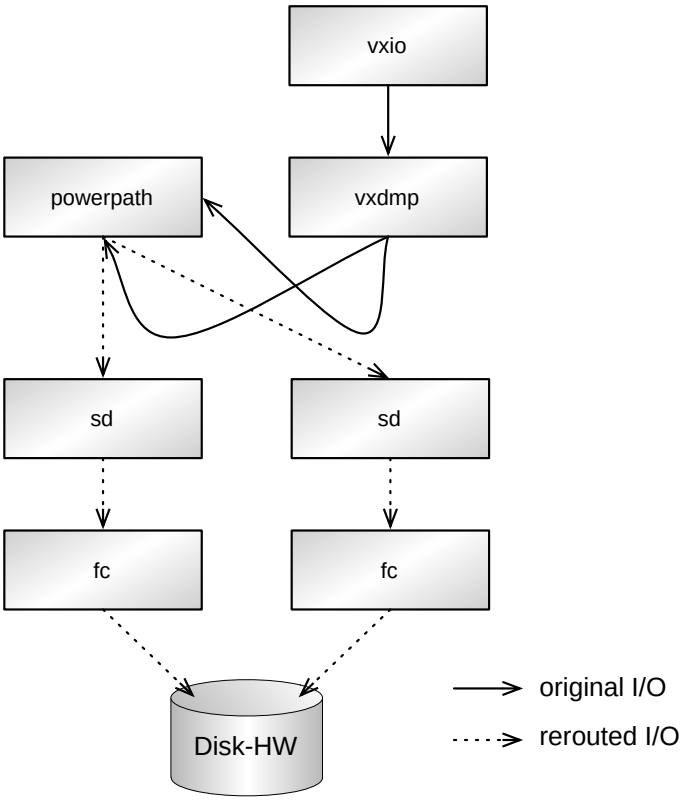

Figure 1-16: The EMC powerpath driver is located at the same level as the DMP driver. The operating system offers the DMP driver several paths to each disk, but usage of these paths is caught and redirected by powerpath

1.4 The Trouble with Networked Disk Access

Process Sleep Behavior

Using networks as a transport medium for SCSI-addressed devices is - from a technical perspective - not a particularly good idea. The reason is a hierarchy of limitations and intricacies of the UNIX kernel, and of networked systems in general, which may cause disastrous results under some circumstances. The most common of these circumstances is an overloaded machine on a poorly performing SAN. Let us look at the individual points whose combination may cause a machine to hang or panic if you use a SAN:

The first problem is that the UNIX device driver model works in the following way

(roughly; the details are much more complicated):

A process that requests an input or output operation will go to sleep and will receive a wakeup call when the I/O request is ready to be served. For instance, a process that reads from a terminal, like an interactive UNIX shell or command interpreter, will go to sleep when the read system call is made. It will wake up when the user has completed his input by pressing the carriage return key or, if the process uses the terminal in raw mode in order to interpret the cursor and other special keys, upon any keypress. Since it is not clear when or if the user will actually create any input (he may have gone to lunch) the device driver sleeps in an interruptible way. That means that if the user decides to stop the program, leave the session, turn off his terminal etc., the process will be interrupted or terminated in a controlled way by the kernel jumping into the process' signal handler code, where exception processing can be handled by the user process. The drivers for devices that typically exhibit indefinite or at least long response times, and especially the application that use these drivers (in this case, anything that uses the terminal, like the shell etc.) take great care to avoid blocking system resources, i.e. they will typically not lock any data structures or allocate a lot of kernel memory and so on. After all, these valuable resources could be blocked for a long time.

Slow Devices vs. Fast Devices

Networks, too, are considered slow devices, just like terminals or serial lines. Due to the intricacies of network architectures, all sorts of delays are possible and must be reckoned with. Router reboots, slow or unresponsive nodes, network congestion and so on can all cause delays. This is represented by the typical timeout values for networks being in the minute range (usually between 1 and 10 minutes). Of course, network based device drivers and applications will use kernel resources sparingly and avoid locking any data structures while waiting for their data to arrive or their timeouts to occur.

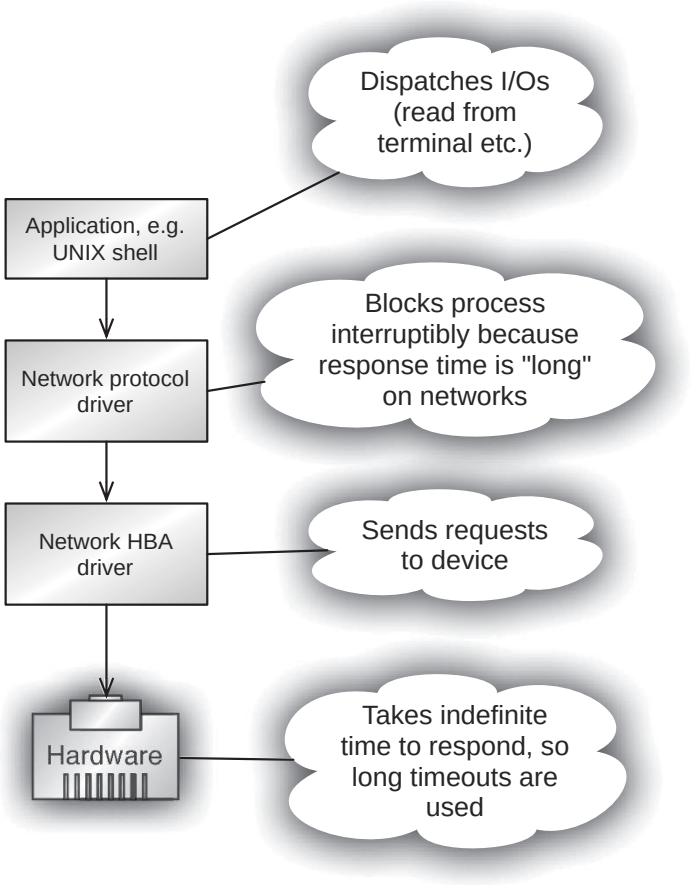

Figure 1-17: Drivers for devices which are known to exhibit potentially slow response times sleep interruptibly. They also typically do not require a lot of resources, and the applications that use them will not keep important data structures locked while they do I/O.

The opposite example is a SCSI disk driver. The classic SCSI driver issues an I/O request to a disk that is attached via a parallel data cable to a hard disk drive. The driver expects the I/O to be served in a very short period of time, since the SCSI command and data exchange protocol is fairly quick. For that reason it need not be interruptible, which simplifies the context switch and thus improves performance. The device driver and especially the applications that use it – the file system or database – can also keep some kernel data structures or code regions locked, or keep large buffers allocated – the I/O response is due in very short order anyway, so there is no need to go through the trouble of making the system call interruptible.

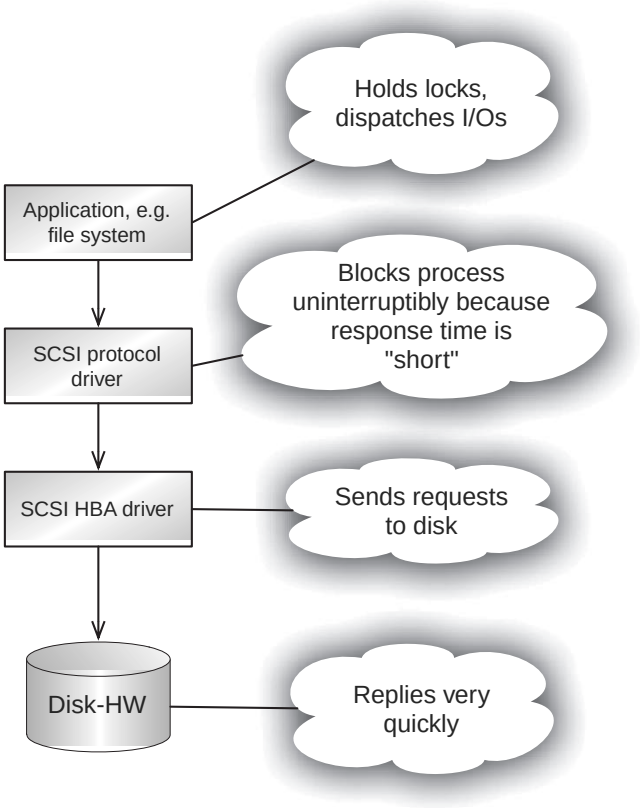

Figure 1-18: **Drivers for devices which are considered to respond quickly will allocate resources much more aggressively, since the response is due in very short order anyway. The I/O is made uninterruptible for performance and other reasons.**

The Problem of Mixing Slow and Fast Devices

The problem of mixing fast and slow devices in the same nexus hierarchy is the following: Remember that the SCSI protocol driver sits on top of the HBA driver, i.e. a device driver for a quick device calls the device driver for a networked, or very slow, device. What happens is that the SCSI driver, depending on how well it is written and how much it sacrifices reliability to performance, might allocate or lock quite a lot of kernel resources while it is doing its presumably quick I/O. But then it calls the networked device driver to actually serve the physical I/O. Everything works out if the device responds quickly, which it usually does, but if the SAN infrastructure experiences problems because of congestion, not enough buffer credits, network reconfiguration, slow disk response due to scattered read latency or otherwise (see page 4), all the resources allocated by either of the device drivers

will remain allocated for a long time. This is not a problem by itself, but if this happens on a heavily loaded database server that does massively parallelized I/O, the results may render the system unresponsive or even cause a kernel panic.

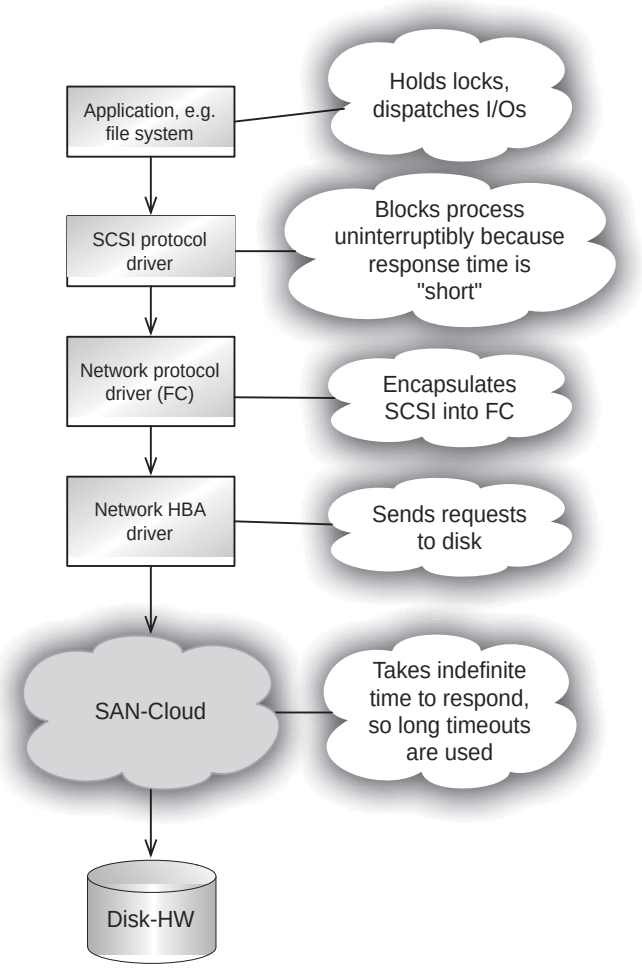

Figure 1-19: Layered device drivers become a big problem if the upper layers expect a rapid response, but the lower layers do not deliver as rapidly. Locks, memory, and process contexts get stuck for an extended time, eventually leading to deadlock situations.

Why Systems Hang – the Deadlock Situation

Because now another problem becomes visible: deadlock avoidance algorithms are not perfect. Deadlock avoidance belongs to a class of problems known to mathematicians and computer scientists as NP problems. NP is an abbreviation of non-polynomial [time] and refers to a complexity class whose problems cannot be solved in a time that is a mere polynomial function of the problem's complexity. What that means is that while a simple case of an NP problem could be solved relatively quickly, as the problem grows larger the time required to find the optimal solution grows at a much faster rate than the problem size. And very quickly the time becomes so long that the optimal solution to the problem cannot be guaranteed to be found any more in any reasonable time.

Travelling Salesman Problem

You may be familiar with one example of this class of problems, the travelling salesman problem. Imagine a salesman (and by this and all other similar occurrences of "man" in the text I mean "man" in the sense of "human"; please forgive me for sacrificing political correctness for legibility) who needs to visit his clients at several places and is looking for the best route to visit them all. If there is just one client, the problem is easily solved: drive to the client, and drive back. If there are two clients the problem is still easy. With five clients, there may be several orders and routes that look good, and it will be quite hard to figure out which is the best one. But now imagine a thousand clients and the poor salesman needs to find the **best** route. The problem would likely be unsolvable in any reasonable time.

Finding a Practical Solution

On the other hand, exactly this problem - the problem of finding the best route through several points - is addressed very effectively on a daily basis by cheap electronic devices called GPS navigation systems. Is that not a contradiction? No, actually it is not. The solution that these devices offer is not generally **optimal**. It is merely **very good** and can therefore be calculated in a rather short time. So the programmers of navigation system devices are trading off precision for speed. Finding the **optimal** route would still take forever, and that would not be useful. So they do try to **approach** the optimal solution but then stick with just a **very good** one to make it practically useful.

Deadlock Avoidance Problem

Operating systems programmers face the same dilemma when trying to avoid deadlock situations. If one process or kernel thread holds resource A locked and then requires resource B, but another process or thread holds resource B and needs resource A, they get into what is called a deadlock, i.e. they are both stuck until either one releases his lock to the other one. Of course, operating systems programmers have for decades been able to handle deadlocks by implementing deadlock avoidance strategies into their kernels. But unfortunately, deadlock avoidance is also a member of the NP problem class. What that means is that while there are algorithms that work under all circumstances to resolve any deadlock situation, those algorithms my take literally forever and are therefore never implemented in any OS. As is the case with navigation systems however, there are

very practical algorithms that avoid deadlocks under **almost** all circumstances which are deemed reasonable and that have a very short runtime. Only the really pathological cases are not covered by these algorithms. This kind of algorithm is now found in all modern operating systems.

Identifying Pathological Cases

What are the pathological cases I mentioned? They are cases in which a large amount of processes or kernel threads exist, and many locks are held for a long time.

As the number of CPU cores per system grows and I/O parallelism especially in databases increases in order to make use of these cores, the number of processes and threads that an operating system has to deal with threatens to grow out of bounds of what the practical deadlock avoidance algorithms can handle.

If in addition to the large number of threads that hold locks in an uninterruptible context, the underlying network does not function perfectly, then a deadlocked system becomes a very likely result. This was actually the reason for many of the outages that we analyzed during the past five years, and the frequency of such events seems to increase.

You need to be aware that large systems doing massively parallel I/O to a SAN will tend to deadlock when the SAN response time increases. The result will be an unresponsive system that will eventually either call a system panic and dump its core to the dump device, or one that will have to be manually revived by causing an NMI (non maskable interrupt) on the console and rebooting it.

Keep this in mind for later chapters, where we will deal with latency and the problem sets that arise from it. We will see that light speed is not fast enough in many common cases, which in the end leads to what looks like an unresponsive SAN. Under these circumstances systems will suffer similar deadlock crashes even if the SAN is actually in perfect working condition. This has indeed become a major problem in many data centers, because storage is typically bought by size rather than the more appropriate "transactions per second per gigabyte" metric. Additionally, SANs are still considered to be very fast, and people tend to overload their machines.

More on this in the chapter on dual data centers, page 213.

1.4.1 Summary

This chapter provided an overview of disk hardware and its development through the last twenty or so years. It showed what Moore's Law is and how it makes the speedy hard disks of yesteryear turn into really slow devices if proper care is not taken to balance the amount of IOPS (I/Os per second) against the number of physical hard disk spindles available. It introduces the most important, yet largely unknown, measure required for storage systems today: TX/sec/GB, or performance per gigabyte. Particular emphasis was put on the unsolvable problem of scattered reads, which cannot be tackled with any conventional approach no matter what the storage vendors may try to argue. The chapter also gave an overview of RAID software and RAID hardware-assisted systems, also known as "storage arrays", and where either storage arrays or disks have an advantage over the other.

In addition you got an insight into the basic data structure of "extent", into how addressing both logical and physical disk blocks work and how the data is laid out on a disk.

We further delved into the device drivers used to talk to disks, SAN basics and UNIX device driver specifics that may sound complicated but the knowledge of which is mandatory in order to understand and fix problems should they arise. You should now also be familiar with one of the main reasons for deadlocked systems and why these situations are almost inevitable unless your SAN is working perfectly – and sometimes even then.