Chapter 5

I/O, Drivers, and DMA

This chapter covers (a) the memory and I/O bus architecture of modern computers, (b) programmed I/O and direct-memory access, (c) disk drive components and how they influence performance, and (d) logical block addressing and the SATA and SCSI buses.

5.1 Introduction

Input/Output (I/O) devices are crucial to the operation of a computer. The data that a program processes — as well as the program binary itself — must be loaded into memory from some I/O device such as a disk, network, or keyboard. Similarly, without a way to output the results of a computation to the user or to storage, those results would be lost. One of the primary functions of the operating system is to manage these I/O devices. It should control access to them, as well as providing a consistent programming interface across a wide range of hardware devices with similar functionality but differing details. This chapter describes how I/O devices fit within the architecture of modern computer systems, and the role of programmed I/O, interrupts, direct memory access (DMA), and device drivers in interacting with them. In addition, you will examine one device, the hard disk drive and its corresponding controller, which is the source and destination of most I/O on typical systems.

5.2 PC architecture and buses

In Figure 5.1 you see the architecture of a typical Intel-architecture computer from a few years ago. Different parts of the system are connected by buses, or communication channels, operating at various speeds. The

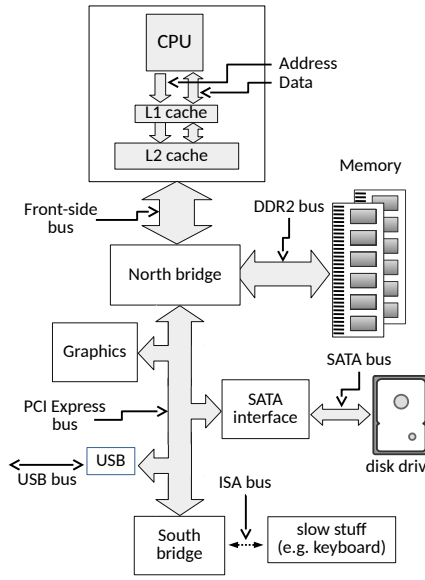


Figure 5.1: Standard Intel PC Architecture

Front-Side Bus carries all memory transactions which miss in L1 and L2 cache, and the North Bridge directs these transactions to memory (DDR2 bus) or I/O devices (PCIe bus) based on their address. The PCI Express (PCIe) is somewhat slower than the front-side bus, but can be extended farther; it connects all the I/O devices on the system. In some cases (like USB and SATA), a controller connected to the PCIe bus (although typically located on the motherboard itself) may interface to a yet slower external interface. Finally, the ISA bus is a vestige of the original IBM PC; for some reason, they've never moved some crucial system functions off of it, so it's still needed.[1]

Simple I/O bus and devices

The fictional computer system described in earlier chapters included a number of memory-mapped I/O devices, which are accessible at particular physical memory addresses. On early computers such as the Apple II and the original IBM PC this was done via a simple I/O bus as shown in Figure 5.2 and Figure 5.3. Address and data lines were extended across

[1]The primary difference between this figure and contemporary (2016) systems is that (a) the memory bus is DDR3 or DDR4, and (b) the north bridge is located on the CPU chip, with no external front-side bus.

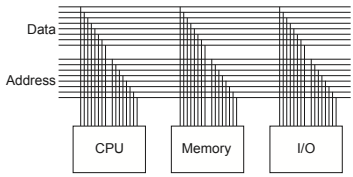


Figure 5.2: Simple memory/IO bus using shared address and data lines

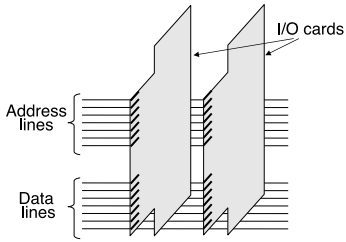


Figure 5.3: Simple memory/IO bus with extension cards

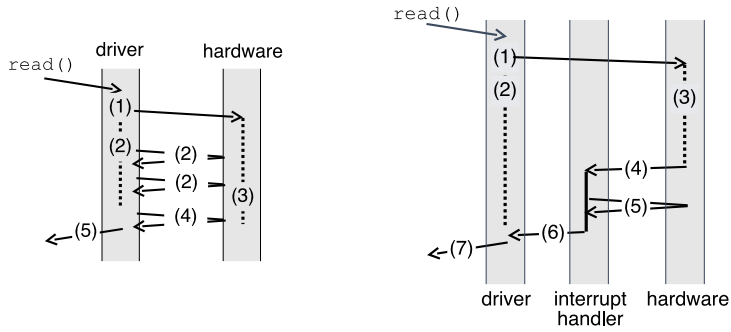
a series of connectors, allowing hardware on a card plugged into one of these slots to respond to read and write requests in much the same way as memory chips on the motherboard would. (This required each card to respond to a different address, no matter what combination of cards were plugged in, typically requiring the user to manually configure card addresses with DIP switches.)

The term "bus" was taken from electrical engineering; in high-power electric systems a *bus bar* is a copper bar used to distribute power to multiple pieces of equipment. A simple bus like this one distributes address and data signals in much the same way.

I/O vs. memory-mapped access: Certain CPUs, including Intel architecture, contain support for a secondary I/O bus, with a smaller address width and accessed via special instructions. (e.g. "IN 0x100" to read a byte from I/O location 0x100, which has nothing to do with reading a byte from memory location 0x100)

Memory-mapped I/O: like in our fictional computer, devices can be mapped in the physical memory space and accessed via standard load and store instructions. In either case, I/O devices will have access to an interrupt line, allowing interrupts to be raised for events like I/O completion.

Device selection: Depending on the system architecture, the device may be responsible for decoding the full address and determining when it has been selected, or a select signal may indicate when a particular slot on the bus is being accessed. Almost all computers today use a version of the PCI bus, which uses memory-mapped access, and at boot time, assigns each I/O device a physical address range to which it should respond.



(a) polled (b) interrupt-driven

Figure 5.4: Polled and interrupt-driven I/O

Polled vs. Interrupt-driven I/O

Polled (or "programmed") I/O: As described in earlier chapters, the simplest way to control an I/O device is for the CPU to issue commands and then wait, polling a device status register until the operation is complete. In Figure 5.4(a) an application requests I/O via e.g. a `read` system call; the OS (step 1) then writes to the device command register to start an operation, after which (step 2) it begins to poll the status register to detect completion. Meanwhile (step 3) the device carries out the operation, after which (step 4) polling by the OS detects that it is complete, and finally (step 5) the original request (e.g. `read`) can return to the application.

Interrupt-driven I/O: The alternate is interrupt-driven I/O, as shown in Figure 5.4(b). After (step 1) issuing a request to the hardware, the OS (step 2) puts the calling process to sleep and switches to another process while (step 3) the hardware handles the request. When the I/O is complete, the device (step 4) raises an interrupt. The interrupt handler then finishes the request. In the illustrated example, the interrupt handler (step 5) reads data that has become available, and then (step 6) wakes the waiting process, which returns from the I/O call (step 7) and continues.

Latency and Programmed I/O

On our fictional computer the CPU is responsible for copying data between I/O devices and memory, using normal memory load and store instructions. Such an approach works well on computers such as the Apple II or the original IBM PC which run at a few MHz, where the address and data buses can be extended at full speed to external I/O cards. A modern CPU

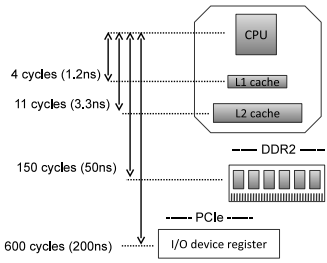


Figure 5.5: Latency between CPU and various levels of memory/IO hierarchy

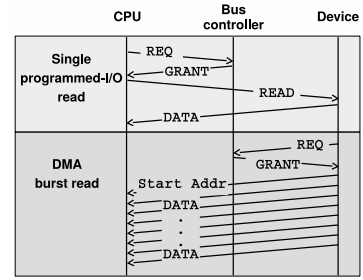


Figure 5.6: DMA access for high-speed data transfer

runs at over 3 GHz, however; during a single clock cycle light can only travel about 4 inches, and electrical signals even less. Figure 5.5 shows example latencies for a modern CPU (in this case an Intel i5, with L3 cache omitted) to read a data value from L1 and L2 cache, a random location in memory (sequential access is faster), and a register on a device on the PCIe bus. (e.g. the disk or ethernet controller) In such a system, reading data from a device in 4-byte words would result in a throughput of 5 words every microsecond, or 20MB/s — far slower than a modern network adapter or disk controller.

Review questions

5.2.1. Buses which extend farther from the CPU are usually:

a) Faster than those closer to the CPU
b) Slower than those closer to the CPU

5.2.2. Memory-mapped I/O is when the CPU reads from RAM: *True / False*

As CPU speeds have become faster and faster, RAM and I/O devices have only slowly increased in speed. The strategies for coping with the high relative latency of RAM and I/O are very different, however—caching works quite well with RAM, which stores data generated by the CPU, while I/O (at least the input side) involves reading new data; here latency is overcome by pipelining, instead.

The PCIe Bus and Direct Memory Access (DMA)

Almost all computers today use the PCIe bus. Transactions on the PCIe bus require a negotiation stage, when the CPU (or a device) requests

access to bus resources, and then is able to perform a transaction after being granted access. In addition to basic read and write requests, the bus also supports Direct Memory Access (DMA), where I/O devices are able to read or write memory directly without CPU intervention. Figure 5.6 shows a single programmed-I/O read (top) compared to a DMA burst transfer (bottom). While the read request requires a round trip to read each and every 4-byte word, once the DMA transfer is started it is able to transfer data at a rate limited by the maximum bus speed. (For an 8 or 16-lane PCIe card this limit is many GB/s)

DMA Descriptors

A device typically requires multiple parameters to perform an operation and transfer the data to or from memory. In the case of a disk controller, for instance, these parameters would include the type of access (read or write), the disk locations to be accessed, and the memory address where data will be stored or retrieved from. Rather than writing each of these parameters individually to device registers, the parameters are typically combined in memory in what is called a *DMA descriptor*, such as the one shown in Figure 5.7. A single write is then used to tell the device the address of this descriptor, and the device can read the entire descriptor in a single DMA read burst. In addition to being more efficient than multiple programmed I/O writes, this approach also allows multiple requests to be queued for a device. (In the case of queued disk commands, the device may even process multiple such requests simultaneously.) When an I/O completes, the device notifies the CPU via an interrupt, and writes status information (such as success/failure) into a field in the DMA descriptor. (or sometimes in a device register, for simple devices which do not allow multiple outstanding requests.) The interrupt handler can then determine which operations have completed, free their DMA descriptors, and notify any waiting processes.

Cache-coherent I/O: The PCIe bus is *cache-consistent*; many earlier I/O buses weren't. Consider what would happen if the CPU wrote a value to location 1000 (say that's the command/status field of a DMA descriptor), then the device wrote a new value to that same location, and finally the CPU tried to read it back?

Device Driver Architecture

Figure 5.8 illustrates the I/O process for a typical device from user-space application request through the driver, hardware I/O operation, interrupt, and finally back to user space.

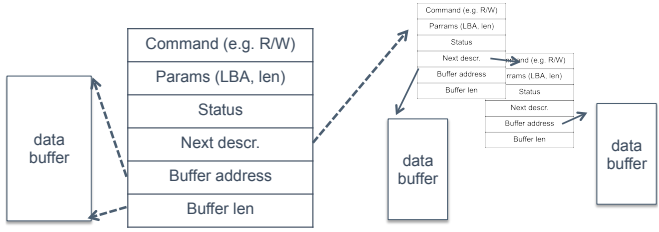


Figure 5.7: List of typical DMA descriptors

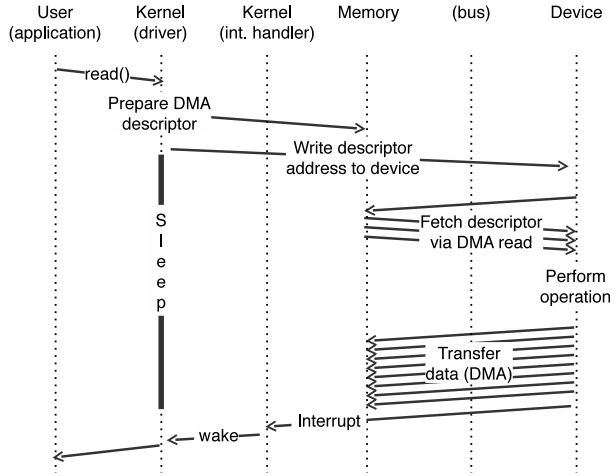


Figure 5.8: Driver Architecture

In more detail:

- The user process executes a `read` system call, which in turn invokes the driver `read` operation, found via the `read` method of the file operations structure.
- The driver fills in a DMA descriptor (in motherboard RAM), writes the physical address of the descriptor to a device register (generating a Memory Write operation across the PCIe bus), and then goes to sleep.
- The device issues a PCIe Memory Read Multiple command to read the DMA descriptor from RAM.
- The device does some sort of I/O. (e.g. read from a disk, or receive a network packet)
- A Memory Write and Invalidate operation is used to write the received data back across the PCIe bus to the motherboard RAM, and

to tell the CPU to invalidate any cached copies of those addresses.
- A hardware interrupt from the device causes the device driver interrupt handler to run.
- The interrupt handler wakes up the original process, which is currently in kernel space in the device driver read method, in a call to something like `interruptible_sleep_on`. After waking up, the read method copies the data to the user buffer and returns.

Review questions

5.2.1. High I/O latency can be compensated for by the use of CPU caches, so that almost all accesses complete at cache speeds instead of going over the bus: *True / False*

5.2.2. Direct Memory Access (DMA) refers to a class of CPU instructions which bypass the cache and access memory directly:
True / False

5.2.3. A device driver:

a) Is software which is part of the application
b) Is software which is part of the kernel
c) Is part of the hardware device

5.3 Hard Disk Drives

The most widely used storage technology today is the hard disk drive, which records data magnetically on one or more spinning platters, in concentric circular tracks. The performance of a hard drive is primarily determined by physical factors: the size and geometry of its components and the speeds at which they move:

Platter: the platter rotates at a constant speed, typically one of the following:

Speed	Rotations/sec	ms/rotation
5400 RPM	90	11
7200 RPM	120	8.3
10,000 RPM	167	6
15,000 RPM	250	4

Head and actuator arm: these take between 1 and 10 ms to move from one track to another on consumer disk drives, depending on the distance between tracks, and between 1 and 4 ms on high-end enterprise drives. (at the cost of higher power consumption and noise)

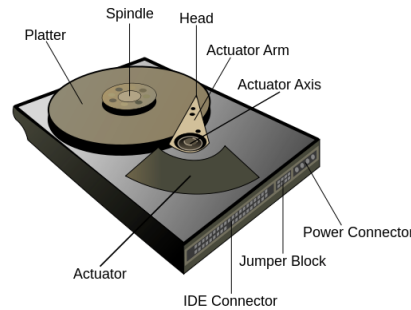


Figure 5.9: Hard Disk Drive (HDD) components

Bits and tracks: on modern drives each track is about 3 micro-inches (75nm) wide, and bits are about 1 micro-inch (25nm) long; with a bit of effort and knowing that the disk is 3.5 inches at its outer diameter you could calculate the maximum speed at which bits pass under the head.

Electronics and interface: the drive electronics are responsible for controlling the actuator and transferring data to and from the host. On a consumer drive this occurs over a SATA interface, which has a top speed of 150, 300, or 600MB/s for SATA 1, 2, or 3.

Hard Drive Performance

Data on a drive can be identified by the platter surface it is on, the track on that surface, and finally the position on that track. Reading data from a disk (or writing to it) requires the following steps:

- Switching the electronics to communicate with the appropriate head. (we'll ignore this, as it's fast)
- Moving the head assembly until the head is positioned over the target track. (*seek time*)
- Waiting for the platter to rotate until the first bit of the target data is passing under the head (*rotational latency*)
- Reading or writing until the last bit has passed under the head. (*transfer time*)

Geometric disk addressing

Unlike memory, data on a disk drive is read and written in fixed-sized units, or sectors, of either 512 or 4096 bytes. Thus small changes (e.g. a single byte) require what is known as a read/modify/write operation — a full sector is read from disk into memory, modified, and then written back to disk. These sectors are arranged in concentric tracks on each platter surface; a sector may thus be identified by its geometric coordinates:

- **Cylinder:** this is the track number; for historical reasons the group formed by the same track on all disk platters has been called a cylinder, as shown in the figure. Early disk drives could switch rapidly between accesses to tracks in the same cylinder; however this is no longer the case with modern drives.
- **Head:** this identifies one of the platter surfaces, as there is a separate head per surface and the drive electronics switches electrically between them in order to access the different surfaces.
- **Sector:** the sector within the track.

Performance examples

The overhead of seek and rotational delay has a major effect on disk performance. To give an example, consider randomly accessing a data

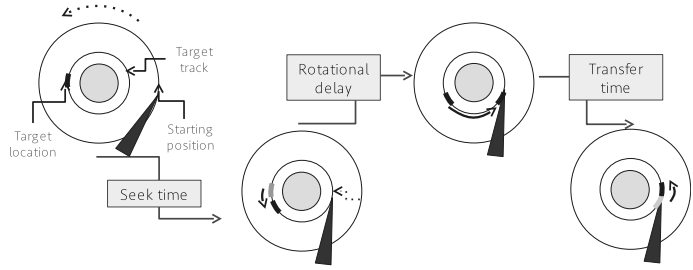


Figure 5.10: Hard disk latency

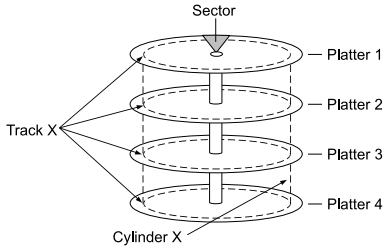


Figure 5.11: Why a track is also called a *cylinder* — the same track on each surface forms a "virtual" cylinder.

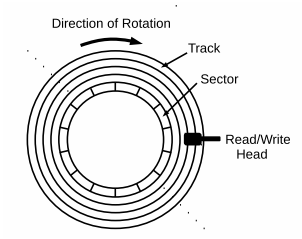


Figure 5.12: Disk access diagram

block on a 7200 RPM disk with the following parameters:

- Average seek time: 8 ms.
- Average rotational delay: 4 ms. (i.e., 1/2 rotation — after seeking to a track, the rotational delay for sectors on that track will be uniformly distributed between 0 and 1 rotation)
- Transfer rate: 200 MB/s. (outer tracks on disks available in 2017)

On average, reading a random 4KB block (i.e. one not immediately following the previous read) requires:

$$8 + 4 + 0.02 = 12ms$$

for an average throughput of 34 KB/s. (0.02 is 4KB / 200KB per ms) Random access to a 5 MB block, or over 1000 times more data, requires:

$$8 + 4 + 25 = 37ms$$

for an average throughput of 134MB/s. (25ms is obtained by dividing 5000KB by a rate of 200KB/ms)

In other words, although disks are random-access devices, random access is expensive. To achieve anywhere near full bandwidth on a modern disk drive you need to read or write data in large contiguous blocks; in our random access example, for instance, a 2 MB transfer would require 22 ms, or less than twice as long[2]as the smallest transfer.

[2]For system operations such as this where performance has a fixed and a variable component, you can think of the point where the two costs are equal as the "knee" in the curve, where you switch from the region where performance is dominated by the fixed cost to where it is dominated by the variable cost. To get high throughput you want to be firmly in the variable-cost region, where the fixed-cost effects are relatively minor.

Disk scheduling

A number of strategies are used to avoid the full penalties of seek and rotational delay in disks. One of these strategies is that of optimizing the order in which requests are performed—for instance reading sectors 10 and 11 on a single track, in that order, would require a seek, followed by a rotational delay until sector 10 was available, and then two sectors of transfer time. However reading 11 first would require the same seek and about the same rotational delay (waiting until sector 11 was under the head), followed by a full rotation to get from section 12 all the way back to sector 10.

Changing the order in which disk reads and writes are performed in order to minimize disk rotations is known as *disk scheduling*, and relies on the fact that multitasking operating systems frequently generate multiple disk requests in parallel, which do not have to be completed in strict order. Although a single process may wait for a read or write to complete before continuing, when multiple processes are running they can each issue requests and go to sleep, and then be woken in the order that requests complete.

Primary Disk Scheduling Algorithms

The primary algorithms used for disk scheduling are:

- **first-come first-served (FCFS):** in other words no scheduling, with requests handled in the order that they are received.
- **Shortest seek time first (SSTF):** this is the throughput-optimal strategy; however it is prone to starvation, as a stream of requests to nearby sections of the disk can prevent another request from being serviced for a long time.
- **SCAN:** this (and variants) are what is termed the *elevator algorithm* — pending requests are served from the inside to the outside of the disk, then from the outside back in, etc., much like an elevator goes from the first floor to the highest requested one before going back down again. It is nearly as efficient as SSTF, while avoiding starvation. (With SSTF one process can keep sending requests which will require less seek time than another waiting request, "starving" the waiting one.)

More sophisticated disk head scheduling algorithms exist, and could no doubt be found by a scan of the patent literature; however they are mostly of interest to hard drive designers.

Implementing Disk Scheduling

Disk scheduling can be implemented in two ways — in the operating system, or in the device itself. OS-level scheduling is performed by keeping a queue of requests which can be re-ordered before they are sent to the disk. On-disk scheduling requires the ability to send multiple commands to the disk before the first one completes, so that the disk is given a choice of which to complete first. This is supported as Command Queuing in SCSI, and in SATA as Native Command Queuing (NCQ).

Note that OS-level I/O scheduling is of limited use today for improving overall disk performance, as the OS has little or no visibility into the internal geometry of a drive. (OS scheduling is still used to merge adjacent requests into larger ones and to allocate performance fairly to different processes, however.)

On-Disk Cache

In addition to scheduling, the other strategy used to improve disk performance is caching, which takes two forms—*read caching* (also called track buffering) and *write buffering*. Disk drives typically have a small amount of RAM used for caching data[3]. Although this is very small in comparison the the amount of RAM typically dedicated to caching on the host, if used properly it can make a significant difference in performance.

At read time, after seeking to a track it is common practice for the disk to store the entire track in the on-disk cache, in case the host requests this data in the near future. Consider, for example, the case when the host requests sector 10 on a track, then almost (but not quite) immediately requests sector 11. Without the track buffer it would have missed the chance to read 11, and would have to wait an entire revolution for it to come back around; with the track buffer, small sequential requests such as this can be handled efficiently.

Write buffering is a different matter entirely, and refers to a feature where a disk drive may acknowledge a write request while the data is still in RAM, before it has been written to disk. This can risk loss of data, as there is a period of time during which the application thinks that data has been safely written, while it would in fact be lost if power failed.

Although in theory most or all of the performance benefit of write buffering could be achieved in a safer fashion via proper use of command queuing, this feature was not available (or poorly implemented) in consumer drives

[3] 8-16MB two or three years ago; 128 MB is common today, probably in part because 128 MB chips are now cheaper than the old 16 MB ones.

until recently; as a result write buffering is enabled in SATA drives by default. Although write buffering can be disabled on a per-drive basis, modern file systems typically issue commands[4] to flush the cache when necessary to ensure file system data is not lost.

SATA and SCSI

Almost all disk drives today use one of two interfaces: SATA (or its precursor, IDE) or SCSI. The SATA and IDE interfaces are derived from an ancient disk controller for the PC, the ST-506, introduced in about 1980. This controller was similar to—but even cruder than—the disk interface in our fictional computer, with registers for the command to execute (read/write/other) and address (cylinder/head/sector), and a single register which the CPU read from or wrote to repeatedly to transfer data. What is called the ATA (AT bus-attached) or IDE (integrated drive electronics) disk was created by putting this controller on the drive itself, and using an extender cable to connect it back to the bus, so that the same software could still access the control registers. Over the years many extensions were made, including DMA support, logical block addressing, and a high-speed serial connection instead of a multi-wire cable; however the protocol is still based on the idea of the CPU writing to and reading from a set of remote, disk-resident registers.

> **Logical vs. CHS addressing:** For CHS addressing to work the OS (and bootloader, e.g. BIOS) has to know the geometry of the drive, so it can tell e.g. whether the sector following (cyl=1,head=1,sector=51) is (1,1,52) or (2,1,0). For large computers sold with a small selection of vendor-approved disks this was not a problem, but it was a major hassle with PCs—you had to read a label on the disk and set BIOS options. Then drive manufacturers started using "fake" geometries because there weren't enough bits in the cylinder and sector fields, making drives that claimed to have 255 heads, giving the worst features of both logical and CHS addressing.

In contrast, SCSI was developed around 1980 as a high-level, device-independent protocol with the following features:

- Packet-based. The initiator (i.e. host) sends a command packet (e.g. READ or WRITE) over the bus to the target; DATA packets are then sent in the appropriate direction followed by a status indication. SCSI specifies these packets over the bus; how the CPU interacts with the disk controller to generate them is up to the maker of the disk controller. (often called an HBA, or host bus adapter)

[4] In SATA the FLUSH command or the FUA (force unit attention) flag. Don't ask me what "force unit attention" means - I have no idea.

- Logical block addressing. SCSI does not support C/H/S addressing — instead the disk sectors are numbered starting from 0, and the disk is responsible for translating this logical block address (LBA) into a location on a particular platter. In recent years logical addressing has been adopted by IDE and SATA, as well.

SCSI over everything

SCSI (like e.g. TCP/IP) is defined in a way that allows it to be carried across many different transport layers. Thus today it is found in:

- USB drives. The USB storage protocol transports SCSI command and data packets.
- CD and DVD drives. The first CD-ROM and CD-R drives were SCSI drives, and when IDE CDROM drives were introduced, rather than invent a new set of commands for CD-specific functions (e.g. eject) the drive makers defined a way to tunnel existing SCSI commands over IDE/ATA (and now SATA).
- Firewire, as used in some Apple systems.
- Fibre Channel, used in enterprise Storage Area Networks.
- iSCSI, which carries SCSI over TCP/IP, typically over Ethernet

and no doubt several other protocols as well. By using SCSI instead of defining another block protocol, the device makers gained SCSI features like the following:

- Standard commands ("Mode pages") for discovering drive properties and parameters.
- Command queuing, allowing multiple requests to be processed by the drive at once. (also offered by SATA, but not earlier IDE drives)
- Tagged command queuing, which allows a host to place constraints on the re-ordering of outstanding requests.

Review questions

5.3.1. Since the platter spins while the head is seeking, rotational latency and seek time happen in parallel and the time until data can be accessed is the maximum of the two: *True / False*

5.3.2. Command queuing in SATA and SCSI will make which of the following workloads run faster:

a) Very large sequential reads and writes
b) A single process performing random reads and waiting for each read to complete before issuing the next one

c) Multiple processes performing random reads.

5.4 RAID and other re-mappings

In the previous section you learned about:

- Disk drives: how they work, and how that determines their performance
- SCSI and SATA buses, which carry block I/O commands between host controllers and disk drives
- The PCI bus, DMA, and device drivers which communicate between host controllers and the operating system

This section is about about *disk-like* devices, which behave like disks but aren't; this includes multi-disk arrays, solid-state drives (SSDs), and other block devices.

Early disk drives used cylinder/head/sector addressing, required the operating system to be aware of the exact parameters of each disk so that it could store and retrieve data from valid locations. The development of logical block addressing, first in SCSI, then in IDE and SATA drives, allowed drives to be interchangeable: with logical block addressing the operating system only needs to know how big a disk is, and can ignore its internal details.

This model is more powerful than that, however, as there is no need for the device on the other end of the SCSI (or SATA) bus to actually *be* a disk drive. (You can do this with C/H/S addressing, as well, but it requires creating a fake drive geometry, and then hoping that the operating system won't assume that it's the real geometry when it schedules I/O requests)

Instead the device on the other end of the wire can be an array of disk drives, a solid-state drive, or any other device which stores and retrieves blocks of data in response to write and read commands. Such disk-like devices are found in many of today's computer systems, both on the desktop and especially in enterprise and data center systems, and include:

- Partitions and logical volume management, for flexible division of disk space
- Disk arrays, especially RAID (redundant arrays of inexpensive disks), for performance and reliability
- Solid-state drives, which use flash memory instead of magnetic disks
- Storage-area networks (SANs)
- De-duplication, to compress multiple copies of the same data

Almost all of these systems look exactly like a disk to the operating system. Their function, however, is typically (at least in the case of disk arrays) an attempt to overcome one or more deficiencies of disk drives, which include:

- Performance: Disk transfer speed is determined by (a) how small bits can be made, and (b) how fast the disk can spin under the head. Rotational latency is determined by (b again) how fast the disk spins. Seek time is determined by (c) how fast the head assembly can move and settle to a final position. For enough money, you can make (b) and (c) about twice as fast as in a desktop drive, although you may need to make the tracks wider, resulting in a lower-capacity drive. To go any faster requires using more disks, or a different technology, like SSDs.
- Reliability: Although disks are surprisingly reliable, they fail from time to time. If your data is worth a lot (like the records from the Bank of Lost Funds), you will be willing to pay for a system which doesn't lose data, even if one (or more) of the disks fails.
- Size: The maximum disk size is determined by the available technology at any time—if they could build them bigger for an affordable price, they would. If you want to store more data, you need to either wait until they can build larger disks, or use more than one. Conversely, in some cases (like dual-booting) a single disk may be more than big enough, but you may need to split it into multiple logical parts.

In the rest of this section we will look at drive *re-mappings*, where a *logical volume* is created which is a different size or has different properties than the disk or disks it is built from. These mappings are not complex—in most cases a simple mathematical operation on a logical block address (LBA) within the logical volume will determine which disk or disks the operation will be directed to, and to what LBA on that disk. This translation may be done on an external device (a *RAID array*), within a host bus adaptor, transparently to the host (a *RAID adapter*), or within the operating system itself (*software RAID*), but the translations performed are the same in each case.

Partitioning

The first remapping strategy, partitioning, is familiar to most advanced computer users. A desktop or laptop computer typically has a single disk drive; however it is frequently useful to split that device into multiple logical devices via partitioning. An example is shown in Figure 5.1, where a single 250GB disk (named `sda`, SCSI disk a) has been split into three

sections for a Linux installation. A small partition ('sda1') is used by the boot loader, followed by a swap partition used for virtual memory, and then the remainder ('sda3') is used for the root file system. Another common use for partitioning is for dual-booting a machine, where e.g. Windows might be installed into one partition and Linux or Apple OS X installed in another. Note that unlike some of the other remappings we will examine, partitioning is almost always handled in the operating system itself, rather than in an external device.

```
   Device Boot     Start        End     Blocks  Id  System
/dev/sda1  *          63     208844     104391  83  Linux
/dev/sda2        208845    4401809    2096482+ 82  Linux swap
/dev/sda3       4401810  488392064  241995127+ 83  Linux
```

Listing 5.1: Example Linux partition map

There are two parts to disk partitioning: (a) a method for recording partition information in a *partition table* to be read by the operating system, and (b) translating in-partition logical block addresses (LBAs) into *absolute* LBAs (i.e. counting from the beginning of the entire disk) at runtime.

The first step is done via a *partition table* on the disk, which gives the starting logical block address (LBA), length, and type of each partition. On boot the operating system reads this table, and then creates virtual block devices (each with an LBA range starting at 0) for each partition. There are two partition table formats in wide use today — Master Boot Record (MBR) boot tables based on the original IBM PC disk format, and GUID Partition Table (GPT) tables used in new systems; for more detail see the following Wikipedia entries: `http://en.wikipedia.org/wiki/Master_Boot_Record`, `http://en.wikipedia.org/wiki/GUID_Partition_Table`

Address translation: Figure 5.13 shows a logical view of the translation between logical block addresses within a partition and physical addresses on the actual device.

Given a partition with start address S and length L and a block address A within that partition, the actual on-disk address A0 can be determined as follows:

```
if A > L:
   error
else:
   A0 = A + S
```

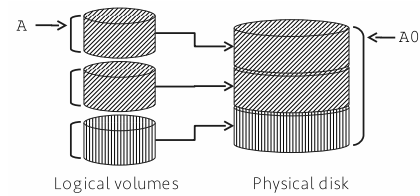


Figure 5.13: Partition layout and formula

Review questions

5.4.1. Which one of the following statements best describes what a disk partition is?

a) A set of files on a disk reserved for a specific purpose
b) A portion of the disk address space (LBA or logical block address space) which is treated as a separate virtual device
c) A type of disk drive

Concatenation

Concatenation means joining two things (like strings) end-to-end; a concatenated volume is the opposite of a partitioned disk, joining the LBA spaces of each disk, one after the other, into a single logical volume which is the sum of multiple physical disks.

Why would you do this? After all, you can just create separate file systems on multiple disks and use the mount command to join them into a single file system hierarchy, as shown in Figure 5.14.

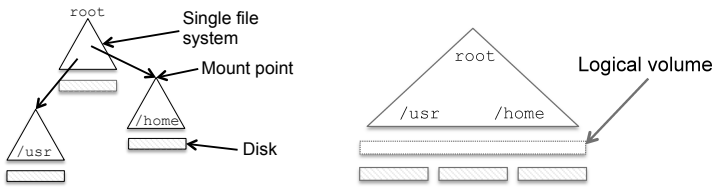


Figure 5.14: Multiple mounted file systems vs. single concatenated volume

This has disadvantages, though. What if you have 3 100GB disks, but 200GB of home directories? Now you're stuck with home directories that look like /home/disk1/joe and /home/disk2/jane, and no matter how you assign accounts, one of the disks is likely to fill up while there is still a lot of free space on the other one.

If you can paste all three disks together and create a single large volume, however, with a single file system on top, then you have a single large, flexible volume, and you don't need to guess how much space to allocate for different directories. (the most modern file systems — ZFS and Btrfs — will handle this for you, but widely-used file systems like NTFS and ext3 do not.)

In Figure 5.15 we see concatenation with three disks, D_1, D_2, D_3, of size S_1, S_2, S_3. The address A in the concatenated volume is translated to a physical disk D_0 and an address on that disk A_0, and (as for partitioning) the translation is very simple:

```
if A < S1 then
   D0 = D1
   A0 = A
else if A < S2 then
   D0 = D2
   A0 = A-S1
else if A < S3 then
   D0 = D3
   A0 = A-S1-S2
else
   error
```

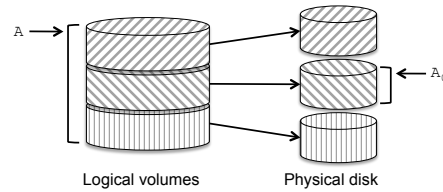


Figure 5.15: Concatenation layout and formula

Concatenation may be implemented in the OS (via the Logical Volume Manager in Linux, or as a type of "software RAID" in Windows) or in an external storage device. With the right tools for modifying the file system, it can even be used to add another disk to an existing file system.

Striping — faster concatenation

Although the size of a concatenated volume is the sum of the individual disk sizes, the performance is typically not. For instance, if you create a single large file, it will probably be placed on contiguous blocks on one of the disks, limiting read and write throughput to that of a single disk. If you've paid for more than one disk, it would be nice to actually get more than one disk's performance, if you can.

Isn't that RAID0? The term "RAID" was coined in a 1988 paper by Paterson, Gibson, and Katz, titled "A case for redundant arrays of inexpensive disks (RAID)", where they defined RAID levels 0 through 5—it turns out RAID0 and RAID1 were what everyone had been calling "striping" and "mirroring" for years, but no one had a name for the newer parity-based systems. RAID2 and 3 are weird and obsolete; no one talks about them.

If the file was instead split into small chunks, and each chunk placed on a different disk than the chunk before it, it would be possible to read and write to all disks in parallel. This is called *striping*, as the data is split into stripes which are spread across the set of drives.

In Figure 5.16 we see individual *strips*, or chunks of data, layed out in horizontal rows (called *stripes*) across three disks. In the figure, when writing strips 0 through 5, strips 0, 1, and 2 would be written first at the same time to the three different disks, followed by writes to strips 3, 4, and 5. Thus, writing six strips would take the same amount of time it takes to write two strips to a single disk.

How big is a strip? It depends, as this value is typically configurable—

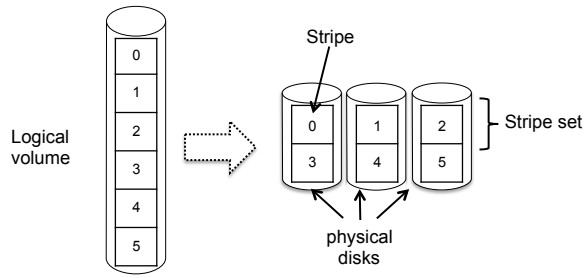


Figure 5.16: Striping across three disks

the RAID algorithms work with any strip size, although for convenience everyone uses a power of 2. If it's too small, the large number of I/Os may result in overhead for the host (software RAID) or for the RAID adapter; if it's too large, then large I/Os will read or write from individual disks one at a time, rather than in parallel. Typical values are 16 KB to 512 KB. (the last one is kind of large, but it's the default built into the `mdadm` utility for creating software RAID volumes on Linux. And the `mdadm` man page calls them "chunks" instead of "strips", which seems like a much more reasonable name.)

Striping data across multiple drives requires translating an address within the striped volume to an address on one of the physical disks making up the volume, using these steps:

1. Find the stripe set that the address is located in - this will give the stripe number within an individual disk.
2. Calculate the stripe number within that stripe set, which tells you the physical disk the stripe is located on.
3. Calculate the address offset within the stripe.

Note that—unlike concatenation—each disk must be of the same size for striping to work. (Well, if any disks are bigger than the smallest one, that extra space will be wasted.)

Given 3 disks d1, d2, d3 of the same size, with a strip size of N sectors, an address A in the striped volume is translated to a physical disk D_0 and an address on that disk A_0 as follows, assuming integer arithmetic:

Review questions

5.4.1. Which one of the following statements best describes the total storage capacity of a **striped** volume of equal-sized disks?

```
S = A / N
- strip # in volume
O = A % N
- offset in strip
case S % 3:
- disk is n1 mod 3
  0:  D0= d1
  1:  D0= d2
  2:  D0= d3
Sd = S / 3
- stripe # in disk
A0 = Sd*N + O
```

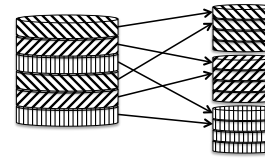


Figure 5.17: Striping layout and formula

a) the same as one of the disks in the volume
b) the sum of the capacity of the disks in the volume

5.4.2. The disks within a striped volume (or at least the portion used of each disk) must be the same size: *True / False*

Mirroring

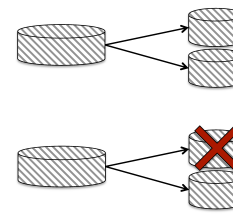


Figure 5.18: Failure of one disk in mirrored volume.

Disks fail, and if you don't have a copy of the data on that disk, it's lost. A lot of effort has been spent on creating multi-disk systems which are more reliable than single-disk ones, by adding *redundancy*—i.e. additional copies of data so that even if one disk fails completely there is still a copy of each piece of your data stored safely somewhere. (Note that striping is actually a step in the wrong direction - if *any one* of the disks in a striped volume fail, which is more likely than failure of a single disk, then you will almost certainly lose all the data in that volume.)

The simplest redundant configuration is *mirroring*, where two identical ("mirror image") copies of the entire volume are kept on two identical disks. In Figure 5.18 we see a mirrored volume comprising two physical disks; writes are sent to both disks, and reads may be sent to either one. If one disk fails, reads (and writes) will go to the remaining disk, and data is not lost. After the failed disk is replaced, the mirrored volume must be rebuilt (sometimes termed "re-silvering") by copying its contents from the other drive. If you wait too long to replace the failed drive, you risk having the second drive crash, losing your data.

Address translation in a mirrored volume is trivial: address A in the logical volume corresponds to the same address A on each of the physical disks. As with striping, both disks must be of the same size. (or any extra sectors in the larger drive must be ignored.)

Mirroring and Consistency

A mirrored volume can be temporarily inconsistent during writing. Consider the following case, illustrated in Figure 5.19:

1. a block in the logical volume contains the value X, and a write is issued changing it to Y, and
2. Y is successfully written to one disk but not the other, and then
3. the power fails

Now, when the system comes back up (step 4 in the figure) the value of this block will depend on which disk the request is sent to, and may change if a disk fails.

High-end storage systems typically solve this problem by storing a temporary copy of written data to non-volatile memory (NVRAM), either battery-backed RAM or flash. If power fails, on startup the system can check that all recent writes completed to each disk. Without hardware support, the OS can check on startup to see if it was cleanly shut down, and if not it may need to check both sides of the mirror and ensure they are consistent. (a lengthy process with modern disks)

When recovering an inconsistent mirrored volume, the value from either disk may be used. Why is this OK? (it helps to remember that from the point of view of the file system or application, a write to a mirrored volume does not complete until both sides have been successfully written to.)

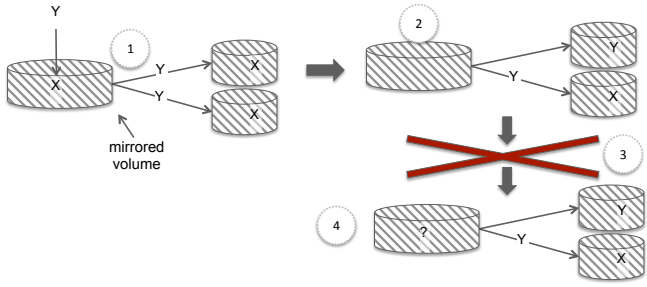
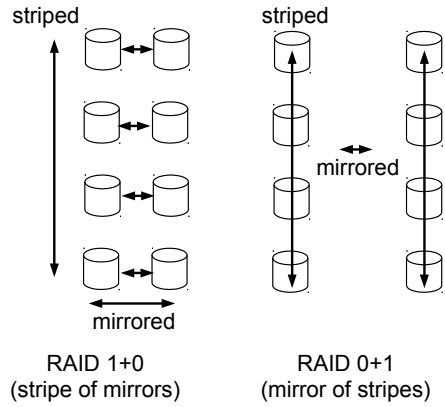


Figure 5.19: Failure during mirror write causing inconsistency



Striping + Mirroring (RAID 0+1, RAID 1+0)

Mirroring and striping can also be used to construct a logical volume out of other logical volumes, so you can create a mirrored volume consisting of two striped volumes, or a striped volume consisting of two mirrored volumes. In either case, a volume holding N drives worth of data will take 2N drives to hold (in this figure, that works out to eight drives) and will give N times the performance of a single disk.

Since striping is also known as RAID 0 and mirroring as RAID 1, these configurations are called RAID 0+1 and RAID 1+0, respectively. RAID 0+1 is less reliable, as if one disk fails in each of the two striped volumes the whole volume will fail. Interestingly enough, the disks contain exactly the same data in both cases; however, in the RAID 0+1 case the controller doesn't try as hard to recover it.

Review questions

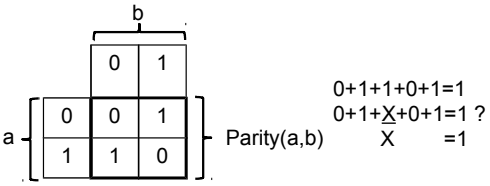
5.4.1. Which one of the following statements best describes the storage capacity of a mirrored volume?

a) It is the same as that of one of the disks making up the volume
b) It is equal to the sum of the capacities of the disks making it up
c) It is equal to the sum of the capacities of all disks, minus the capacity of the parity drive

RAID 4

Although mirroring and RAID 1+0 are good for constructing highly reliable storage systems, sometimes you don't want reliability bad enough to be willing to devote half of your disk space to redundant copies of data. This is where RAID 4 (and the related RAID 5) come in.

For the 8-disk RAID 1+0 volume described previously to fail, somewhere between 2 and 5 disks would have to fail (3.66 on average). If you plan on replacing disks as soon as they fail, this may be more reliability than you need or are willing to pay for. RAID 4 provides a high degree of reliability with much less overhead than mirroring or RAID 1+0.



RAID 4 takes N drives and adds a single parity drive, creating an array that can tolerate the failure of any single disk without loss of data. It does this by using the parity function (also known as exclusive-OR, or addition modulo 2), which has the truth table seen in the figure to the right. As you can see in the equation, given the parity calculated over a set of bits, if one bit is lost, it can be re-created given the other bits and the parity. In the case of a disk drive, instead of computing parity over N bits, you compute it over N disk blocks, as shown here where the parity of two blocks is computed:

```
  001010011101010010001 ... 001101010101 +
  011010100111010100100 ... 011000101010

= 010000111010000110101 ... 010101111111
```

RAID 4 - Organization: RAID 4 is organized almost exactly like a striped (RAID 0) volume, except for the parity drive. We can see this in Figure 5.20 — each data block is located in the same place as in the striped volume, and then the corresponding parity block is located on a separate disk.

Writing to a RAID 4 Volume: How you write to a RAID 4 volume depends on whether it is a small or large write. For large writes you can over-write a complete stripe set at a time, letting you calculate the parity before you write. Small writes are less efficient: you have to read back some amount of data in order to re-calculate the parity. There are two

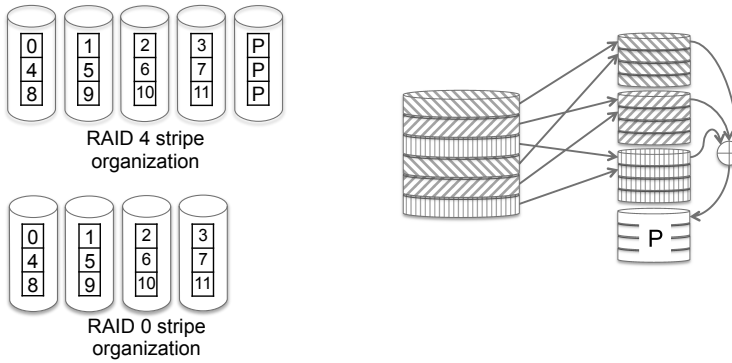
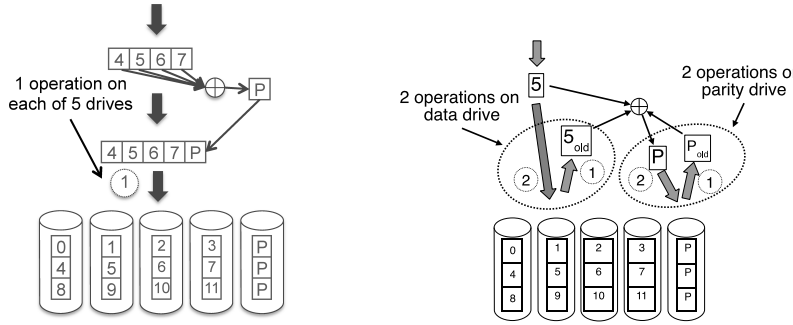


Figure 5.20: RAID 4 organization



Large write:

- Calculate parity (no I/O needed)
- (1) Write stripe set to disk

Small write:

- (1) read old data, parity
- Calculate new parity (no I/O)
- (2) write new data, parity

Figure 5.21: Large and small writes to RAID 4

options: you can either read the entire stripe set and calculate its parity after modifying it, or you can read the old data and parity, subtract the old data, and add in the new data, which is more efficient for larger RAID volumes (i.e. with more than 4 drives).

In Figure 5.21 you can see that a small write can take twice as long and require four times as many operations as the corresponding write to a striped volume, where no parity recalculation is needed.

A question for the reader: why does a small write to RAID 4 take twice as long, rather than four times as long, as a single disk write?

Reading from a RAID 4 Volume: There are two cases when reading from a RAID 4 volume: normal mode and *degraded mode*. In normal mode the data is available on the disk(s) it was written to, which is the case when no disks have failed, and for data on the remaining disks after one has failed. In *degraded* mode the data being read was written to the failed drive, and must be reconstructed from the remaining data and parity in the stripe set. (The actual reconstruction is quite simple, as the missing data stripe is just the exclusive OR of all the remaining data and parity in the stripe set.)

To write in degraded mode, parity is calculated and stripes are written to all but the failed disk. When the disk is replaced, its contents will be reconstructed from the other drives.

Review questions

5.4.1. A RAID 4 volume with five data drives and one parity drive can tolerate two disk failures without data loss: *True / False*

5.4.2. A RAID 4 volume with five data drives and one parity drive holds more data than three mirrored disk pairs (six disks total) assuming the disks are the same size in the two cases: *True / False*

5.4.3. After a disk fails on a RAID 4 volume, which statement is more correct?

a) It should be replaced quickly
b) It doesn't need to be replaced immediately, as the RAID controller will prevent data loss if another disk fails

5.4.4. Which one of the following statements best describes the efficiency of small writes on RAID 4?

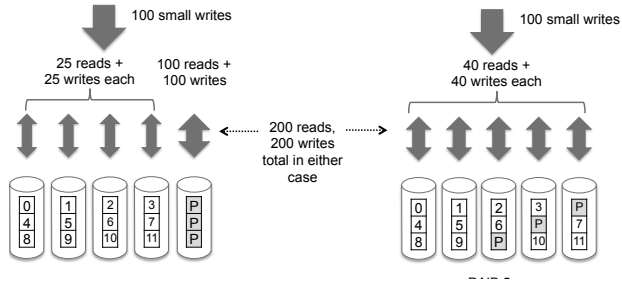
a) They are more efficient than large writes
b) They are less efficient than large writes

RAID 5

Small writes to RAID 4 require four operations: one read each for the old data and parity, and one write for each of the new data and parity. Two of these four operations go to the parity drive, no matter what LBA is being written, creating a bottleneck. If one drive can handle 200 random

operations per second, the entire array will be limited to a total throughput of 100 random small writes per second, no matter how many disks are in the array.

By distributing the parity across drives in RAID 5, the parity bottleneck is eliminated. It still takes four operations to perform a single small write, but those operations are distributed evenly across all the drives. (Because of the distribution algorithm, it's technically possible for all the writes to go to the same drive; however it's highly unlikely.) In the five-drive case shown here, if a disk can complete 200 operations a second, the RAID 4 array would be limited to 100 small writes per second, while the RAID 5 array could perform 250. (5 disks = 1000 requests/second, and 4 requests per small write)



RAID 6 - more reliability

RAID level 1 (including 1+0 and 0+1), and levels 4 and 5 are designed to protect against the total failure of any single disk, assuming that the remaining disks operate perfectly. However, there is another failure mode known as a *latent sector error*, in which the disk continues to operate but one or more sectors are corrupted and cannot be read back. As disks become larger these errors become more problematic: for instance, one vendor specifies their current desktop drives to have no more than 1 unrecoverable read error per 10^{14} bits of data read, or 12.5 TB. In other words, there might be in the worst case a 1 in 4 chance of an unrecoverable read error while reading the entire contents of a 3TB disk. (Luckily, actual error rates are typically much lower, but not low enough.)

If a disk in a RAID 5 array fails and is replaced, the "rebuild" process requires reading the entire contents of each remaining disk in order to reconstruct the contents of the failed disk. If any block in the remaining drives is unreadable, data will be lost. (Worse yet, some RAID adapters

and software will abandon the whole rebuild, causing the entire volume to be lost.)

RAID 6 refers to a number of RAID mechanisms which add additional redundancy, using a second parity drive with a more complex error-correcting code[5]. If a read failure occurs during a RAID rebuild, this additional protection may be used to recover the contents of the lost block, preventing data loss. Details of RAID 6 implementation will not be covered in this class, due to the complexity of the codes used.

Review questions

5.4.1. RAID 5 is less likely to lose data from disk failure than RAID 4: *True / False*

5.4.2. RAID 5 is faster for very large writes than RAID 4: *True / False*

5.4.3. RAID 5 is faster for small writes than RAID 4: *True / False*

5.4.4. Which one of the following statements best describes why RAID 6 has become important recently?

a) Because total failure is more common in modern disks
b) Because modern disks are bigger

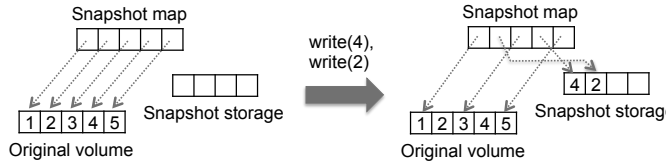
Logical Volume Management

If you have managed a Linux system (especially Fedora or Red Hat) you may have used the Logical Volume Manager (LVM), which allows disks on the system to be flexibly combined and split into different volumes; similar functionality is available on other operating systems, as well as on high-end storage arrays.

The volume types which can be created under LVM are those which have been described in this section: partitioned, concatenated, and the various RAID levels. In addition, however, logical volume managers typically offer functions to migrate storage contents and to create snapshots of a volume.

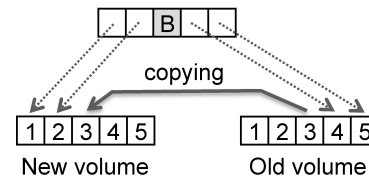
Volume snapshots rely on a copy-on-write mechanism almost identical to that used in virtual memory:

[5] Commonly a Reed-Solomon code; see Wikipedia if you want to find out what that is.



A snapshot is a "lazy copy" of a volume—it preserves the contents without immediately consuming any additional disk space, instead consuming space as the volume is written to. (It's also much faster than copying all the data) Why would you want to make a snapshot? Maybe you want to save the state of your machine before you make major changes, like installing new software and drivers, or upgrading the OS. If things don't work out, you can revert back to the snapshot and try again.

Snapshots are also frequently used for backing up a computer, because it takes so long to copy all the data from a modern disk. If you merely copied all the files off of the disk, the backed-up version of one file might be hours older than another file; this can be avoided by backing up a snapshot instead of the volume itself.



Live migration is a sort of magical operation, allowing you to switch from one disk drive to another while the machine continues to run. It works by using a map to direct individual requests to either the old volume or the new volume, with the dividing line moving as data is copied from one to the other. What happens if you try to write to the small section being copied in the middle? The write gets stalled until the copy is done, and then is directed to the new location.

Solid State Drives

Solid-state drives (SSDs) store data on semiconductor-based flash memory instead of magnetic disk; however by using the same block-based interface (e.g. SATA) to connect to the host they are able to directly replace disk drives.

SSDs rely on flash memory, which stores data electrically: a high programming voltage is used to inject a charge onto a circuit element (a *floating gate*—ask your EE friends if you want an explanation) that is isolated by insulating layers, and the presence or absence of such a stored charge can

be detected in order to read the contents of the cell. Flash memory has several advantages over magnetic disk, including:

- Random access performance: since flash memory is addressed electrically, instead of mechanically, random access can be very fast.
- Throughput: by using many flash chips in parallel, a consumer SSD (in 2018) can read speeds of 1-2 GB/s, while the fastest disks are limited to a bit more than 200MB/s.

Flash is organized in pages of 4KB to 16KB, which must be read or written as a unit. These pages may be written only once before they are erased in blocks of 128 to 256 pages, making it impossible to directly modify a single page. Instead, the same copy-on-write algorithm used in LVM snapshots is used internally in an SSD: a new write is written to a page in one of a small number of spare blocks, and a map is updated to point to the new location; the old page is now invalid and is not needed. When not enough spare blocks are left, a garbage collection process finds a block with many invalid pages, copies any remaining valid pages to another spare block, and erases the block.

When data is written sequentially, this process will be efficient, as the garbage collector will almost always find an entirely invalid block which can be erased without any copying. For very random workloads, especially on cheap drives with few spare blocks and less sophisticated garbage collection, this process can involve huge amounts of copying (called write amplification) and run very slowly.

SSD Wear-out: Flash can only be written and erased a certain number of times before it begins to degrade and will not hold data reliably: most flash today is rated for 3000 write/erase operations before it becomes unreliable. The internal SSD algorithms distribute writes evenly to all blocks in the device, so in theory you can safely write 3000 times the capacity of a current SSD, or the entire drive capacity every day for 8 years. (Note that 3000 refers to *internal* writes; random writes with high write amplification will wear out an SSD more than the same volume of sequential writes.)

For a laptop or desktop this would be an impossibly high workload, especially since they are typically used only half the hours in a day or less. For some server applications, however, this is a valid concern. Special-purpose SSDs are available (using what is called Single-Level Cell, or SLC, flash) which are much more expensive but are rated for as many as 100,000 write/erase cycles. (This capacity is the equivalent of overwriting an entire drive every 30 minutes for 5 years. For a 128GB drive, this would require continuously writing at over 70MB/s, 24 hours a day.)

New Disk Technologies

The capacity of a disk drive is determined by how many bits there are on a track (i.e. how *short* the bits are), how many tracks fit on each side of a platter (how *narrow* the bits are), and how many platters and sssociated heads fit into a drive enclosure. Since sometime around the late 90s most of the increase in drive density has come from making the tracks narrower; however this has hit a stumbling block recently. The narrower you make the write head, the weaker its magnetic field, until eventually it becomes too weak to magnetize bits on the platter. You can fix this for a while by making the platter easier to magnetize (lower *coercivity*), but if you go too far in that direction, the bits will flip spontaneously due to thermal noise. (There's a cure for that—make the bits bigger—but it obviously won't help.)

In the last few years disks have come perilously close to this limit. Much of the capacity growth in the last couple of years (2018) and most in coming years is expected to come from the following technologies:

- **Helium:** Filling the drive with helium[6] reduces the air turbulence around the heads and platters, allowing them to be thinner so you can cram more of them into a disk. (The highest capacity air-filled drives typically had 4 platters and 8 heads; the largest helium-filled drives today have 9 platters.)
- **Shingled magnetic recording (SMR):** Narrow tracks can be written with a wide (and thus high magnetic field) head by overlapping the wide tracks, like rows of shingles[7], and read back by a narrower read head. Unfortunately, overwriting a sector on an SMR disk will damage the neighboring sector, requiring a translation layer (much like a flash translation layer) in order to be used by a normal file system.
- **Heat-assisted Magnetic Recording (HAMR):** If you heat a magnetic material it becomes easier to magnetize. HAMR relies on narrow, weak write heads that shouldn't be able to write to the platter, and heats the surface with a laser just before writing to it.

Although the impending death of hard disk drives has been predicted many times—Google "bubble memory" for an example–technological breakthroughs have come through each time to keep them in the position

[6] Which is harder than it sounds, since helium will leak through cast aluminum, which is the preferred material for HDD enclosures.

[7] Really more like clapboards, but "clapboarded" just doesn't have the same ring to it.

of the most cost-effective bulk storage medium available. It remains to be seen whether high-density SSDs based on low-performance NAND flash are able to catch up to disk in cost per terabyte, or whether some technological advance will keep disk ahead for yet another decade.

Storage-area Networks

In enterprise environments it is typical to separate storage systems from the servers that use the storage. This allows tasks such as backup to be centralized, as well as simplifying the task of replacing or servicing hardware. (In fact, in a virtualized environment (covered in a later chapter) external storage allows running servers to be moved from one piece of hardware to another without interruption.)

Storage-Area Networks, or SANs, typically use the SCSI protocol and a transport which can be routed or switched as a network. The most common SAN technologies are Fibre Channel and iSCSI:

- Fibre Channel is a bizarre networking protocol used only in SANs; for historic reasons it is typically used with optical fiber cabling, which is expensive and unreliable for short connections.
- iSCSI is an encapsulation of SCSI within TCP/IP; it uses traditional ethernet cabling, switching, and IP routing, although an iSCSI deployment may use a separate network for storage.

"Disks" on a SAN are identified by a transport address (either an IP address or DNS name, for iSCSI, or a 64-bit World Wide Name (WWN) for Fibre Channel) plus a logical unit number (LUN), which identifies a specific volume on a target. In other words, an individual block of data on a SAN can be identified by address + LUN + LBA.

One of the key administrative features in a SAN is LUN masking, which determines which resources (LUNs) on the network may be seen by which hosts. This lets each server see only the LUNs which have been assigned to it, so that a misconfigured host cannot access or corrupt storage which it is not supposed to have access to. In addition to source-based access control, iSCSI also offers several authentication protocols, to prevent access to disk volumes from unauthorized hosts or applications.

De-duplication

Large enterprise storage systems typically store large amounts of similar data. As an example, your CCIS account stores your home directory on a central server; if you log onto a college Windows or Linux machine almost all the files you create and edit will be located on this server.

In a corporate environment this approach is frequently used with desktop machines, resulting in many copies of the same data (items like a spreadsheet or document sent to several people will be copied into each users' email inbox) In addition in such environments data is backed up frequently, creating even more copies. Very high compression ratios can be achieved by saving only a single copy of such data, using a process called (not surprisingly) *deduplication*. We see this in the figure below, where the data to be stored is 26 long, but only contains 9 unique blocks, giving a nearly 3:1 compression ratio if the 26 blocks of data are replaced by pointers to unique data blocks:

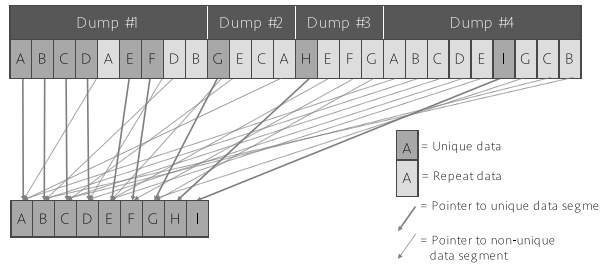


Figure 5.22: De-duplication

To perform deduplication, a cryptographic hash (a form of checksum) is calculated over each block to be written, and checked against a database. If the hash is found, then a block containing the same bits has already been written to storage, and we store a pointer to that block. If not—i.e. it is the first time we saw that particular data pattern—it is written to a new location on disk, and a pointer to that location is stored. By using this map we can then (somewhat slowly) retrieve the data later.

De-duplication is widely used for storing backups and retaining data for legal purposes, as it achieves very high compression (and thus lower cost) in many such cases. However, due to the overhead and non-sequential reads involved in retrieving data, it is typically much slower than normal storage.

5.5 Putting it all together

In our `ls` example the block layer and disk drive get used extensively by the file system. When the new process is created the kernel must read the first page from disk, to identify the type of executable, and then after the sections are mapped into memory, page faults will cause block read requests to be sent through the file system to the underlying device. Additional disk requests will come in response to the `readdir` system

call, as the file system reads directory and inode data to list the files in a directory.

We'll ignore the file system for now, as it is described in more detail in later chapters, and focus on the role of the Linux block layer, which sits between file systems and the physical devices[8]. The block layer is organized around the `struct bio` object, a typical Linux kernel object which is fantastically complicated in order to track lots of things we don't really care about. We'll ignore most of this complexity; the fields that we're concerned with are the command flag (indicating read or write), data pointer (points to one or more pages), a callback function and private data field provided by the subsystem which submitted the I/O (more on this below), and a pointer to the device to which the I/O has been issued. (actually a pointer to a `struct block_device`)

First, a note about the private data pointer and callbacks, which are a common design pattern in C. (at least in the Linux kernel) In a proper object-oriented language, if you want to specialize a class (e.g. a block I/O descriptor) by adding additional fields (e.g. for details like timers or queues needed by your device driver), you can create a derived class with these additional fields. You can't do that in C—you can allocate two structures, or embed an instance of the general structure within the specialized one, but there will be cases (like callback functions) where a function handling the general class will in turn invoke another function which needs to access the specialized structure.

The most straightforward way to do this is via a "private data" field in a object; this is a generic pointer which is set to point to a separate structure holding the specialized data. An example shown in the listing below is the bio callback function (called `bi_end_io`): this is a function pointer which is invoked when the I/O operation completes, which is given a pointer to the bio itself as an argument.

```
struct my_data {
  ... specific data ...
};

void my_end_io(struct bio *b)
{
    struct my_data *md = b->bi_private;
    ...
}

...
{
```

[8]For a more detailed description of the Linux block layer, see `https://lwn.net/Articles/736534/`.

```
        struct bio *b = ...
        struct my_data *priv = ...
        b->bi_private = priv;
        b->bi_end_io = my_end_io;
        submit_bio(b);
    }
```

Listing 5.2: Using the `bi_private` field to pass information to a callback function

Note that `struct bio` has no way to indicate the *type* of attached data; instead we need to be sure that functions which interpret `bi_private` as a pointer to `struct my_data` are only ever called on bios where the attached object actually *is* of that type. (e.g. in this case `bi_end_io` will only be set to `my_end_io` in cases where the attached object is of type `struct my_data`)

Turning our attention back to the block layer, let's trace the case where a file system submits a single page read or write to a old-fashioned programmed-IO IDE drive. If you remember the IDE drive is similar to the disk controller described earlier in the text, with a few registers to indicate the disk sector, command (read / write), and the number of sectors to transfer, as well as a register which the CPU reads or writes to transfer the data. For a write you push the command and data, then wait for an interrupt to indicate that it's done; for a read you wait until the interrupt before transferring the data.

Here we see the path for submitting a read request in ext2[9]:

```
fs/ext2/inode.c:
793 int ext2_readpage(struct file *file, struct page *page) {
795         return mpage_readpage(page, ext2_get_block);

fs/mpage.c:
398 int mpage_readpage(struct page *page, get_block_t get_block) {
408         bio = do_mpage_readpage(bio, page, 1, &last_block_in_bio,
411                 mpage_bio_submit(REQ_OP_READ, 0, bio);

143 struct bio *
144 do_mpage_readpage(struct bio *bio, struct page *page, ...) {
284                 bio = mpage_alloc(bdev, blocks[0] << (blkbits - 9),

68  struct bio *
69  mpage_alloc(struct block_device *bdev, ...) {
77          bio = bio_alloc(gfp_flags, nr_vecs);
85                  bio->bi_bdev = bdev;


59  struct bio *mpage_bio_submit(int op, int op_flags, ...
```

[9]Line numbers from Linux kernel 4.8.0

```
61          bio->bi_end_io = mpage_end_io;
64          submit_bio(bio);
```

Listing 5.3: Ext2 read bio submission

Ignoring all sorts of bookkeeping and optimizations, we have: a `bio` is allocated (`mpage_alloc` line 77) and a pointer is stored to the destination device (line 85), then a callback function is set (`mpage_bio_submit` line 61) and the I/O enters the block system via `submit_bio`.

From this point the block system generates a *request*[10] to the underlying device:

```
block/blk-core.c:
2067 blk_qc_t submit_bio(struct bio *bio) {
2099        return generic_make_request(bio);

1995 blk_qc_t generic_make_request(struct bio *bio) {
2036            struct request_queue *q = bdev_get_queue(bio->bi_bdev);
2039                        ret = q->make_request_fn(q, bio);
```

Listing 5.4: Submit_bio logic

We'll skip over the details of how I figured out what value `q->make_request_fn` has here; just trust me that in our case it's `blk_queue_bio:`

```
block/blk-core.c:
1663 blk_qc_t blk_queue_bio(struct request_queue *q, struct bio *bio)

1704        el_ret = elv_merge(q, &req, bio);
1705        if (el_ret == ELEVATOR_BACK_MERGE) {
1706              if (bio_attempt_back_merge(q, req, bio)) {
1710                        goto out_unlock;

1739        req = get_request(q, bio_data_dir(bio), rw_flags, bio, ...
1752        init_request_from_bio(req, bio);

1775               add_acct_request(q, req, where);
1776               __blk_run_queue(q);
```

Listing 5.5: `block/block-core.c`, `blk_queue_bio`

It first calls the "elevator" merge function (a reference to the classic disk scheduling algorithm) which tries to merge it with an existing queued I/O; if it can, then we return via `goto`[11] (lines 1704-1710). If not, we allocate a

[10] Unix block devices have always been different from normal files in that they have a single submission function for both reads and writes.

[11] The use of gotos to jump to cleanup code is a common design pattern in kernel coding, replacing the try/finally pattern in more civilized programming languages.

request structure (basically a bunch of information for queueing, hashing, accounting, sleeping, and stuff like that) and set up all its fields (lines 1739, 1752). Then we add the request to the elevator queue (line 1775, which in turn calls `__elv_add_request`, which has a lot of very complicated logic to figure out where to put the request in the queue) and then run a request from the queue:

```
block/block-core.c
311  inline void __blk_run_queue_uncond(struct request_queue *q)

324          q->request_fn(q);
```

Listing 5.6: Running a request from the queue

For a "legacy" (i.e. really old) IDE device the request function is `do_ide_request`. If you're looking at the code yourself, note that anything with `_pm_` in it is power management, that while `start_request` is important, `blk_start_request` doesn't do anything interesting, and that "plugging" refers to a complicated mechanism of delaying I/Os a short time to see if they'll be followed by additional requests that can be merged into one big request. You can skip over those parts; I did.

```
drivers/ide/ide-io.c:
456  void do_ide_request(struct request_queue *q)
517                          rq = blk_fetch_request(drive->queue);
551                  startstop = start_request(drive, rq);

block/blk-core.c:
2506 struct request *blk_fetch_request(struct request_queue *q)
2510         rq = blk_peek_request(q);

2349 struct request *blk_peek_request(struct request_queue *q)
2354          ... rq = __elv_next_request(q) ...
2399                 ret = q->prep_rq_fn(q, rq);

drivers/ide/ide-io.c
306  ide_startstop_t start_request (ide_drive_t *drive, ... *rq)

343                  if (rq->cmd_type == REQ_TYPE_ATA_TASKFILE)
344                        return execute_drive_cmd(drive, rq);
```

So in order of execution, we grab a request from the queue (`blk-core.c` 2354) and call the queue prep function (`idedisk_prep_fn`, which sets `rq->cmd_type` to `REQ_TYPE_ATA_TASKFILE` and does a lot of other things we ignore), and then we call `start_request` (`ide-io.c` line 551) which calls `execute_drive_cmd` (line 344).

```
drivers/ide/ide-io.c
253  ide_startstop_t execute_drive_cmd (ide_drive_t *drive, ... *rq)
```

```
259             if (cmd->protocol == ATA_PROT_PIO) {
260                     ide_init_sg_cmd(cmd, blk_rq_sectors(rq) << 9);
261                     ide_map_sg(drive, cmd);
264             return do_rw_taskfile(drive, cmd);
```

If the drive controller is in programmed I/O mode (PIO), `ide_init_sg_cmd` creates a "taskfile", the bytes that have to be written to the control registers of the device; `ide_map_sg` gets pointers to all the memory regions to transfer. *Now* we're finally ready to send a command to the disk controller.

We'll trace a write operation, since it's easier:

```
drivers/ide/ide-taskfile.c:
78  ide_startstop_t do_rw_taskfile(ide_drive_t *drive, ...

118           tp_ops->tf_load(drive, &cmd->hob, cmd->valid.out.hob);
119           tp_ops->tf_load(drive, &cmd->tf,
cmd->valid.out.tf);

122       switch (cmd->protocol) {
123       case ATA_PROT_PIO:
123            if (cmd->tf_flags & IDE_TFLAG_WRITE) {
125                    tp_ops->exec_command(hwif, tf->command);
126                    ndelay(400);  /* FIXME */
127                    return pre_task_out_intr(drive, cmd);
```

(Fun fact: that FIXME comment was there in kernel 2.4.31 in 2005. I don't think it will get fixed.)

First the taskfile (and extended taskfile, known as the HOB since it's valid when the High Order Bit is set somewhere in the basic taskfile) to the controller, using `ide_tf_load`, which uses the `outb` instruction to write the bytes to the appropriate control registers; e.g. the 3 bytes of LBA in each get written as so:

```
        ...
    if (valid & IDE_VALID_LBAL)
           tf_outb(tf->lbal, io_ports->lbal_addr);
    if (valid & IDE_VALID_LBAM)
           tf_outb(tf->lbam, io_ports->lbam_addr);
    if (valid & IDE_VALID_LBAH)
           tf_outb(tf->lbah, io_ports->lbah_addr);
        ...
```

Then `ide_exec_command` writes the command byte to the appropriate register, and calls `pre_task_out_intr`:

```
drivers/ide/ide-taskfile.c:
403 ide_startstop_t pre_task_out_intr(ide_drive_t *drive, ... cmd)
```

```
419         ide_set_handler(drive, &task_pio_intr, WAIT_WORSTCASE);
421         ide_pio_datablock(drive, cmd, 1);
```

which sets a handler (saved in `hwif->handler`, with a timer in case the disk hangs) to be called when the request completes, and then actually copies the data to the data register.

We're almost done; bear with me. When the drive finishes writing its data, the IDE interrupt handler is called, which invokes the handler we just registered above, and then through a long, complicated chain of calls invokes `bio->bi_end_io`, which is the `mpage_end_io` that we stuck in the `bio` structure way back up at the top:

```
drivers/ide/ide-io.c:
892 irqreturn_t ide_intr (int irq, void *dev_id)
793         handler = hwif->handler;
849         startstop = handler(drive);

drivers/ide/ide-taskfile.c:
344 ide_startstop_t task_pio_intr(ide_drive_t *drive)
348         u8 stat = hwif->tp_ops->read_status(hwif);
  ... handle partial transfers; if done:
396         ide_complete_rq(drive, 0, blk_rq_sectors(cmd->rq) << 9);

drivers/ide/ide-io.c:
115 int ide_complete_rq(ide_drive_t *drive, int error, ...
128         rc = ide_end_rq(drive, rq, error, nr_bytes);

57  int ide_end_rq(ide_drive_t *drive, struct request *rq, ...
70         return blk_end_request(rq, error, nr_bytes);

block/blk-core.c
2796 bool blk_end_request(struct request *rq, int error, ...
2798       return blk_end_bidi_request(rq, error, nr_bytes, 0);

2740 bool blk_end_bidi_request(struct request *rq, int error,
2746       if (blk_update_bidi_request(rq, error, nr_bytes, ...

2654 bool blk_update_bidi_request(struct request *rq, int error,
2658        if (blk_update_request(rq, error, nr_bytes))

2539 bool blk_update_request(struct request *req, int error, ...
2604               req_bio_endio(req, bio, bio_bytes, error);

142  void req_bio_endio(struct request *rq, struct bio *bio, ...
155               bio_endio(bio);

block/bio.c:
1742 void bio_endio(struct bio *bio)
1761        if (bio->bi_end_io)
1762              bio->bi_end_io(bio);
```

Listing 5.7: The home stretch: from IDE interrupt to invoking bio->bi_end_io

Review questions

5.5.1. SSDs wear out faster if you repeatedly write to the same file or logical block address: *True / False*

5.5.2. Which one of the following statements is correct?

a) Deduplication is faster than traditional RAID arrays, but requires more disk space to hold the same amount of data
b) Deduplication is slower than traditional RAID arrays, but can hold more data with the same amount of disk space

Answers to Review Questions

5.2.1 (2) In general, connections which span longer distances and connect more devices (such as those far from the CPU) will be slower.
5.2.2 False. RAM and I/O devices (even memory-mapped I/O devices) are separate parts of the system.
5.2.1 False. The whole idea of an I/O (input/output) device is that the CPU doesn't know what value will be returned when it reads it.
5.2.2 False. DMA is when a device on the PCIe (or similar) bus accesses memory directly, without CPU intervention.
5.2.3 (2), software in the kernel. A device driver is that part of the kernel code which reads from, writes to, and handles interrupts from one or more specific hardware devices.
5.3.1 False. Since the platter is constantly spinning, when the head reaches the right track it may still have to wait as much as a full rotation for the target block to come beneath the head.
5.3.2 (3), multiple processes performing simultaneous random reads. In this case the OS can issue multiple read commands which are queued by the drive and completed in the most efficient order.
5.4.1 (2), a portion of the disk LBA space. The partition boundary is specified in a partition table in the beginning of the disk, and the operating system treats each partition as if it were a separate device.
5.4.1 (2), the storage capacity of a striped volume is the sum of the capacity of the disks in the volume, since only one copy of data is stored.
5.4.2 True. Stripes from each disk are interleaved at a fine granularity, so when one disk comes to an end, the entire volume has to end.
5.4.1 (1), each disk holds a copy of each byte written to the volume. (and no, there's no parity drive in a mirrored volume.)

5.4.1 False. The RAID 4 parity code can only recover from one missing drive, no matter how many drives are in the volume.

5.4.2 True. Three mirrored pairs hold three disks worth of data, while the six-disk RAID volume contains five disks of data.

5.4.3 (1) Once a single disk fails in a RAID 4 (or 5) volume, the data is unprotected and will be lost if a second disk fails. The sooner the disk is replaced, the less likely this is to happen.

5.4.4 (2) Small writes require reading old data and parity, and then writing data and parity, requiring four operations for a one-block write. Writing a full stripe set allows parity to be calculated without reading any information from disk, adding only a single operation to the parity drive.

5.4.1 False. RAID 5 and RAID 4 can both tolerate only a single disk failure without data loss.

5.4.2 False. If an entire stripe set is written at once, the parity can be calculated and written with it, resulting in one write operation for each drive in the array, regardless of whether it is RAID 4 or RAID 5.

5.4.3 True. A small write requires four operations: read (1) old data, (2) old parity, write (3) new data, (4) new parity. In RAID 4, two of these always go to the same (parity) drive, which becomes a bottleneck.

5.4.4 (2) Modern disks do not seem to fail more or less frequently than those of several years ago. Similarly, the probability of losing a single block of data to an unrecoverable read error has stayed roughly the same (as of 2015); however, the number of data blocks on a single disk has grown hugely, making it far more likely that one of the data blocks on a disk will be lost.

5.5.1 False. SSD algorithms distribute writes evenly over the internal flash, whether writes are to the same or different block addresses.

5.5.2 (2) Writing to a de-duplicated volume is slower due to the need to search for possible duplicates. Reading is typically much slower, as well, because the fragments making up a file will not be sequential on the underlying disk. For many workloads, however, it may be possible to store 10 times as much data on the same number of disks.