Mass-Storage Structure

In this chapter, we discuss how mass storage—the nonvolatile storage system of a computer—is structured. The main mass-storage system in modern computers is secondary storage, which is usually provided by hard disk drives (HDD) and nonvolatile memory (NVM) devices. Some systems also have slower, larger, tertiary storage, generally consisting of magnetic tape, optical disks, or even cloud storage.

Because the most common and important storage devices in modern computer systems are HDDs and NVM devices, the bulk of this chapter is devoted to discussing these two types of storage. We first describe their physical structure. We then consider scheduling algorithms, which schedule the order of I/Os to maximize performance. Next, we discuss device formatting and management of boot blocks, damaged blocks, and swap space. Finally, we examine the structure of RAID systems.

There are many types of mass storage, and we use the general term ***nonvolatile storage*** (NVS) or talk about storage "drives" when the discussion includes all types. Particular devices, such as HDDs and NVM devices, are specified as appropriate.

CHAPTER OBJECTIVES

- Describe the physical structures of various secondary storage devices and the effect of a device's structure on its uses.
- Explain the performance characteristics of mass-storage devices.
- Evaluate I/O scheduling algorithms.
- Discuss operating-system services provided for mass storage, including RAID.

11.1 Overview of Mass-Storage Structure

The bulk of secondary storage for modern computers is provided by **hard disk drives** (**HDDs**) and **nonvolatile memory** (**NVM**) devices. In this section,

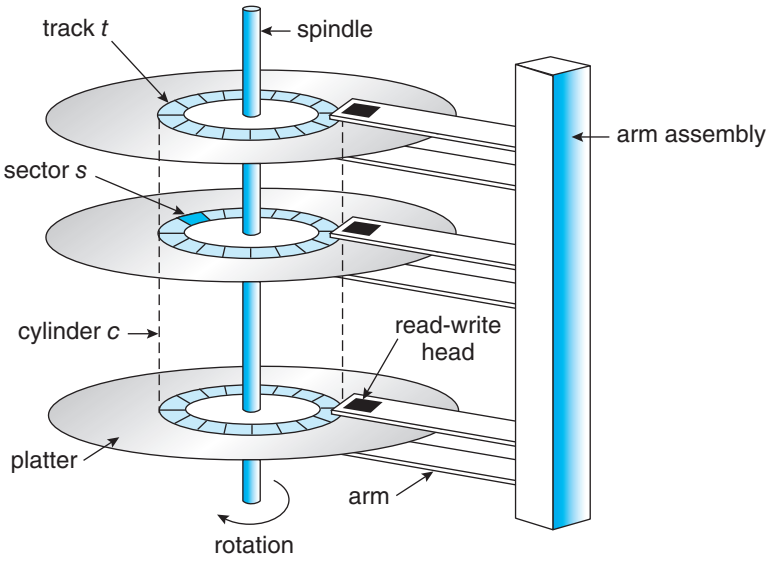


Figure 11.1 HDD moving-head disk mechanism.

we describe the basic mechanisms of these devices and explain how operating systems translate their physical properties to logical storage via address mapping.

11.1.1 Hard Disk Drives

Conceptually, HDDs are relatively simple (Figure 11.1). Each disk **platter** has a flat circular shape, like a CD. Common platter diameters range from 1.8 to 3.5 inches. The two surfaces of a platter are covered with a magnetic material. We store information by recording it magnetically on the platters, and we read information by detecting the magnetic pattern on the platters.

A read–write head "flies" just above each surface of every platter. The heads are attached to a **disk arm** that moves all the heads as a unit. The surface of a platter is logically divided into circular **tracks**, which are subdivided into **sectors**. The set of tracks at a given arm position make up a **cylinder**. There may be thousands of concentric cylinders in a disk drive, and each track may contain hundreds of sectors. Each sector has a fixed size and is the smallest unit of transfer. The sector size was commonly 512 bytes until around 2010. At that point, many manufacturers start migrating to 4KB sectors. The storage capacity of common disk drives is measured in gigabytes and terabytes. A disk drive with the cover removed is shown in Figure 11.2.

A disk drive motor spins it at high speed. Most drives rotate 60 to 250 times per second, specified in terms of rotations per minute (**RPM**). Common drives spin at 5,400, 7,200, 10,000, and 15,000 RPM. Some drives power down when not in use and spin up upon receiving an I/O request. Rotation speed relates to transfer rates. The **transfer rate** is the rate at which data flow between the drive and the computer. Another performance aspect, the **positioning time**, or **random-access time**, consists of two parts: the time necessary to move the disk arm to the desired cylinder, called the **seek time**, and the time necessary for the

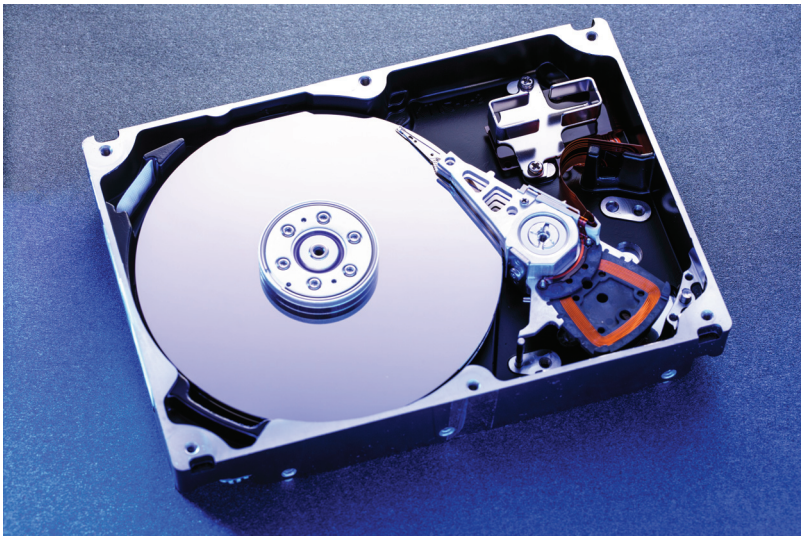


Figure 11.2 A 3.5-inch HDD with cover removed.

desired sector to rotate to the disk head, called the **rotational latency**. Typical disks can transfer tens to hundreds of megabytes of data per second, and they have seek times and rotational latencies of several milliseconds. They increase performance by having DRAM buffers in the drive controller.

The disk head flies on an extremely thin cushion (measured in microns) of air or another gas, such as helium, and there is a danger that the head will make contact with the disk surface. Although the disk platters are coated with a thin protective layer, the head will sometimes damage the magnetic surface. This accident is called a **head crash**. A head crash normally cannot be repaired; the entire disk must be replaced, and the data on the disk are lost unless they were backed up to other storage or RAID protected. (RAID is discussed in Section 11.8.)

HDDs are sealed units, and some chassis that hold HDDs allow their removal without shutting down the system or storage chassis. This is helpful when a system needs more storage than can be connected at a given time or when it is necessary to replace a bad drive with a working one. Other types of storage media are also **removable**, including CDs, DVDs, and Blu-ray discs.

DISK TRANSFER RATES

As with many aspects of computing, published performance numbers for disks are not the same as real-world performance numbers. Stated transfer rates are always higher than **effective transfer rates**, for example. The transfer rate may be the rate at which bits can be read from the magnetic media by the disk head, but that is different from the rate at which blocks are delivered to the operating system.

11.1.2 Nonvolatile Memory Devices

Nonvolatile memory (NVM) devices are growing in importance. Simply described, NVM devices are electrical rather than mechanical. Most commonly, such a device is composed of a controller and flash NAND die semiconductor chips, which are used to store data. Other NVM technologies exist, like DRAM with battery backing so it doesn't lose its contents, as well as other semiconductor technology like 3D XPoint, but they are far less common and so are not discussed in this book.

11.1.2.1 Overview of Nonvolatile Memory Devices

Flash-memory-based NVM is frequently used in a disk-drive-like container, in which case it is called a **solid-state disk** (**SSD**) (Figure 11.3). In other instances, it takes the form of a **USB drive** (also known as a thumb drive or flash drive) or a DRAM stick. It is also surface-mounted onto motherboards as the main storage in devices like smartphones. In all forms, it acts and can be treated in the same way. Our discussion of NVM devices focuses on this technology.

NVM devices can be more reliable than HDDs because they have no moving parts and can be faster because they have no seek time or rotational latency. In addition, they consume less power. On the negative side, they are more expensive per megabyte than traditional hard disks and have less capacity than the larger hard disks. Over time, however, the capacity of NVM devices has increased faster than HDD capacity, and their price has dropped more quickly, so their use is increasing dramatically. In fact, SSDs and similar devices are now used in some laptop computers to make them smaller, faster, and more energy-efficient.

Because NVM devices can be much faster than hard disk drives, standard bus interfaces can cause a major limit on throughput. Some NVM devices are designed to connect directly to the system bus (PCIe, for example). This technology is changing other traditional aspects of computer design as well.

Figure 11.3 A 3.5-inch SSD circuit board.

Some systems use it as a direct replacement for disk drives, while others use it as a new cache tier, moving data among magnetic disks, NVM, and main memory to optimize performance.

NAND semiconductors have some characteristics that present their own storage and reliability challenges. For example, they can be read and written in a "page" increment (similar to a sector), but data cannot be overwritten—rather, the NAND cells have to be erased first. The erasure, which occurs in a "block" increment that is several pages in size, takes much more time than a read (the fastest operation) or a write (slower than read, but much faster than erase). Helping the situation is that NVM flash devices are composed of many die, with many datapaths to each die, so operations can happen in parallel (each using a datapath). NAND semiconductors also deteriorate with every erase cycle, and after approximately 100,000 program-erase cycles (the specific number varies depending on the medium), the cells no longer retain data. Because of the write wear, and because there are no moving parts, NAND NVM lifespan is not measured in years but in **Drive Writes Per Day** (**DWPD**). That measure is how many times the drive capacity can be written per day before the drive fails. For example, a 1 TB NAND drive with a 5 DWPD rating is expected to have 5 TB per day written to it for the warranty period without failure.

These limitations have led to several ameliorating algorithms. Fortunately, they are usually implemented in the NVM device controller and are not of concern to the operating system. The operating system simply reads and writes logical blocks, and the device manages how that is done. (Logical blocks are discussed in more detail in Section 11.1.5.) However, NVM devices have performance variations based on their operating algorithms, so a brief discussion of what the controller does is warranted.

11.1.2.2 NAND Flash Controller Algorithms

Because NAND semiconductors cannot be overwritten once written, there are usually pages containing invalid data. Consider a file-system block, written once and then later written again. If no erase has occurred in the meantime, the page first written has the old data, which are now invalid, and the second page has the current, good version of the block. A NAND block containing valid and invalid pages is shown in Figure 11.4. To track which logical blocks contain valid data, the controller maintains a **flas translation layer** (**FTL**). This table maps which physical pages contain currently valid logical blocks. It also

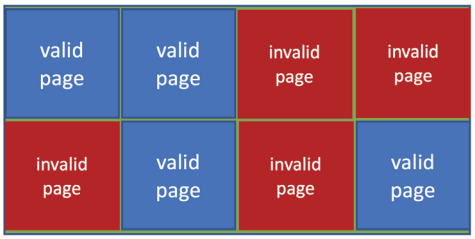


Figure 11.4 A NAND block with valid and invalid pages.

tracks physical block state—that is, which blocks contain only invalid pages and therefore can be erased.

Now consider a full SSD with a pending write request. Because the SSD is full, all pages have been written to, but there might be a block that contains no valid data. In that case, the write could wait for the erase to occur, and then the write could occur. But what if there are no free blocks? There still could be some space available if individual pages are holding invalid data. In that case, **garbage collection** could occur—good data could be copied to other locations, freeing up blocks that could be erased and could then receive the writes. However, where would the garbage collection store the good data? To solve this problem and improve write performance, the NVM device uses **over-provisioning**. The device sets aside a number of pages (frequently 20 percent of the total) as an area always available to write to. Blocks that are totally invalid by garbage collection, or write operations invalidating older versions of the data, are erased and placed in the over-provisioning space if the device is full or returned to the free pool.

The over-provisioning space can also help with **wear leveling**. If some blocks are erased repeatedly, while others are not, the frequently erased blocks will wear out faster than the others, and the entire device will have a shorter lifespan than it would if all the blocks wore out concurrently. The controller tries to avoid that by using various algorithms to place data on less-erased blocks so that subsequent erases will happen on those blocks rather than on the more erased blocks, leveling the wear across the entire device.

In terms of data protection, like HDDs, NVM devices provide error-correcting codes, which are calculated and stored along with the data during writing and read with the data to detect errors and correct them if possible. (Error-correcting codes are discussed in Section 11.5.1.) If a page frequently has correctible errors, the page might be marked as bad and not used in subsequent writes. Generally, a single NVM device, like an HDD, can have a catastrophic failure in which it corrupts or fails to reply to read or write requests. To allow data to be recoverable in those instances, RAID protection is used.

11.1.3 Volatile Memory

It might seem odd to discuss volatile memory in a chapter on mass-storage structure, but it is justifiable because DRAM is frequently used as a mass-storage device. Specifically, **RAM drives** (which are known by many names, including RAM disks) act like secondary storage but are created by device drivers that carve out a section of the system's DRAM and present it to the rest of the system as it if were a storage device. These "drives" can be used as raw block devices, but more commonly, file systems are created on them for standard file operations.

Computers already have buffering and caching, so what is the purpose of yet another use of DRAM for temporary data storage? After all, DRAM is volatile, and data on a RAM drive does not survive a system crash, shutdown, or power down. Caches and buffers are allocated by the programmer or operating system, whereas RAM drives allow the user (as well as the programmer) to place

MAGNETIC TAPES

Magnetic tape was used as an early secondary-storage medium. Although it is nonvolatile and can hold large quantities of data, its access time is slow compared with that of main memory and drives. In addition, random access to magnetic tape is about a thousand times slower than random access to HDDs and about a hundred thousand times slower than random access to SSDs so tapes are not very useful for secondary storage. Tapes are used mainly for backup, for storage of infrequently used information, and as a medium for transferring information from one system to another.

A tape is kept in a spool and is wound or rewound past a read–write head. Moving to the correct spot on a tape can take minutes, but once positioned, tape drives can read and write data at speeds comparable to HDDs. Tape capacities vary greatly, depending on the particular kind of tape drive, with current capacities exceeding several terabytes. Some tapes have built-in compression that can more than double the effective storage. Tapes and their drivers are usually categorized by width, including 4, 8, and 19 millimeters and 1/4 and 1/2 inch. Some are named according to technology, such as LTO-6 (Figure 11.5) and SDLT.

Figure 11.5 An LTO-6 Tape drive with tape cartridge inserted.

data in memory for temporary safekeeping using standard file operations. In fact, RAM drive functionality is useful enough that such drives are found in all major operating systems. On Linux there is `/dev/ram`, on macOS the `diskutil` command creates them, Windows has them via third-party tools, and Solaris and Linux create `/tmp` at boot time of type "tmpfs", which is a RAM drive.

RAM drives are useful as high-speed temporary storage space. Although NVM devices are fast, DRAM is much faster, and I/O operations to RAM drives are the fastest way to create, read, write, and delete files and their contents. Many programs use (or could benefit from using) RAM drives for storing temporary files. For example, programs can share data easily by writing and reading files from a RAM drive. For another example, Linux at boot time creates a temporary root file system (`initrd`) that allows other parts of the system to have access to a root file system and its contents before the parts of the operating system that understand storage devices are loaded.

11.1.4 Secondary Storage Connection Methods

A secondary storage device is attached to a computer by the system bus or an **I/O bus**. Several kinds of buses are available, including **advanced technology attachment** (**ATA**), **serial ATA** (**SATA**), **eSATA**, **serial attached SCSI** (**SAS**), **universal serial bus** (**USB**), and **fibr channel** (**FC**). The most common connection method is SATA. Because NVM devices are much faster than HDDs, the industry created a special, fast interface for NVM devices called **NVM express** (**NVMe**). NVMe directly connects the device to the system PCI bus, increasing throughput and decreasing latency compared with other connection methods.

The data transfers on a bus are carried out by special electronic processors called **controllers** (or **host-bus adapters** (**HBA**)). The **host controller** is the controller at the computer end of the bus. A **device controller** is built into each storage device. To perform a mass storage I/O operation, the computer places a command into the host controller, typically using memory-mapped I/O ports, as described in Section 12.2.1. The host controller then sends the command via messages to the device controller, and the controller operates the drive hardware to carry out the command. Device controllers usually have a built-in cache. Data transfer at the drive happens between the cache and the storage media, and data transfer to the host, at fast electronic speeds, occurs between the cache host DRAM via DMA.

11.1.5 Address Mapping

Storage devices are addressed as large one-dimensional arrays of **logical blocks**, where the logical block is the smallest unit of transfer. Each logical block maps to a physical sector or semiconductor page. The one-dimensional array of logical blocks is mapped onto the sectors or pages of the device. Sector 0 could be the first sector of the first track on the outermost cylinder on an HDD, for example. The mapping proceeds in order through that track, then through the rest of the tracks on that cylinder, and then through the rest of the cylinders, from outermost to innermost. For NVM the mapping is from a tuple (finite ordered list) of chip, block, and page to an array of logical blocks. A logical block address (**LBA**) is easier for algorithms to use than a sector, cylinder, head tuple or chip, block, page tuple.

By using this mapping on an HDD, we can—at least in theory—convert a logical block number into an old-style disk address that consists of a cylinder number, a track number within that cylinder, and a sector number within that track. In practice, it is difficult to perform this translation, for three reasons. First, most drives have some defective sectors, but the mapping hides this by substituting spare sectors from elsewhere on the drive. The logical block address stays sequential, but the physical sector location changes. Second, the number of sectors per track is not a constant on some drives. Third, disk manufacturers manage LBA to physical address mapping internally, so in current drives there is little relationship between LBA and physical sectors. In spite of these physical address vagaries, algorithms that deal with HDDs tend to assume that logical addresses are relatively related to physical addresses. That is, ascending logical addresses tend to mean ascending physical address.

Let's look more closely at the second reason. On media that use **constant linear velocity** (**CLV**), the density of bits per track is uniform. The farther a track is from the center of the disk, the greater its length, so the more sectors it can

hold. As we move from outer zones to inner zones, the number of sectors per track decreases. Tracks in the outermost zone typically hold 40 percent more sectors than do tracks in the innermost zone. The drive increases its rotation speed as the head moves from the outer to the inner tracks to keep the same rate of data moving under the head. This method is used in CD-ROM and DVD-ROM drives. Alternatively, the disk rotation speed can stay constant; in this case, the density of bits decreases from inner tracks to outer tracks to keep the data rate constant (and performance relatively the same no matter where data is on the drive). This method is used in hard disks and is known as **constant angular velocity** (**CAV**).

The number of sectors per track has been increasing as disk technology improves, and the outer zone of a disk usually has several hundred sectors per track. Similarly, the number of cylinders per disk has been increasing; large disks have tens of thousands of cylinders.

Note that there are more types of storage devices than are reasonable to cover in an operating systems text. For example, there are "shingled magnetic recording" hard drives with higher density but worse performance than mainstream HDDs (see http://www.tomsitpro.com/articles/shingled-magnetic-recoding-smr-101-basics,2-933.html). There are also combination devices that include NVM and HDD technology, or volume managers (see Section 11.5) that can knit together NVM and HDD devices into a storage unit faster than HDD but lower cost than NVM. These devices have different characteristics from the more common devices, and might need different caching and scheduling algorithms to maximize performance.

11.2 HDD Scheduling

One of the responsibilities of the operating system is to use the hardware efficiently. For HDDs, meeting this responsibility entails minimizing access time and maximizing data transfer bandwidth.

For HDDs and other mechanical storage devices that use platters, access time has two major components, as mentioned in Section 11.1. The seek time is the time for the device arm to move the heads to the cylinder containing the desired sector, and the rotational latency is the additional time for the platter to rotate the desired sector to the head. The device **bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer. We can improve both the access time and the bandwidth by managing the order in which storage I/O requests are serviced.

Whenever a process needs I/O to or from the drive, it issues a system call to the operating system. The request specifies several pieces of information:

- Whether this operation is input or output
- The open file handle indicating the file to operate on
- What the memory address for the transfer is
- The amount of data to transfer

If the desired drive and controller are available, the request can be serviced immediately. If the drive or controller is busy, any new requests for service will be placed in the queue of pending requests for that drive. For a multiprogramming system with many processes, the device queue may often have several pending requests.

The existence of a queue of requests to a device that can have its performance optimized by avoiding head seeks allows device drivers a chance to improve performance via queue ordering.

In the past, HDD interfaces required that the host specify which track and which head to use, and much effort was spent on disk scheduling algorithms. Drives newer than the turn of the century not only do not expose these controls to the host, but also map LBA to physical addresses under drive control. The current goals of disk scheduling include fairness, timeliness, and optimizations, such as bunching reads or writes that appear in sequence, as drives perform best with sequential I/O. Therefore some scheduling effort is still useful. Any one of several disk-scheduling algorithms can be used, and we discuss them next. Note that absolute knowledge of head location and physical block/cylinder locations is generally not possible on modern drives. But as a rough approximation, algorithms can assume that increasing LBAs mean increasing physical addresses, and LBAs close together equate to physical block proximity.

11.2.1 FCFS Scheduling

The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm (or FIFO). This algorithm is intrinsically fair, but it generally does not provide the fastest service. Consider, for example, a disk queue with requests for I/O to blocks on cylinders

98, 183, 37, 122, 14, 124, 65, 67,

in that order. If the disk head is initially at cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 124, 65, and finally to 67, for a total head movement of 640 cylinders. This schedule is diagrammed in Figure 11.6.

The wild swing from 122 to 14 and then back to 124 illustrates the problem with this schedule. If the requests for cylinders 37 and 14 could be serviced together, before or after the requests for 122 and 124, the total head movement could be decreased substantially, and performance could be thereby improved.

11.2.2 SCAN Scheduling

In the **SCAN algorithm**, the disk arm starts at one end of the disk and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk. The SCAN algorithm is sometimes called the **elevator algorithm**, since the disk arm behaves just like an elevator in a building, first servicing all the requests going up and then reversing to service requests the other way.

Let's return to our example to illustrate. Before applying SCAN to schedule the requests on cylinders 98, 183, 37, 122, 14, 124, 65, and 67, we need to know

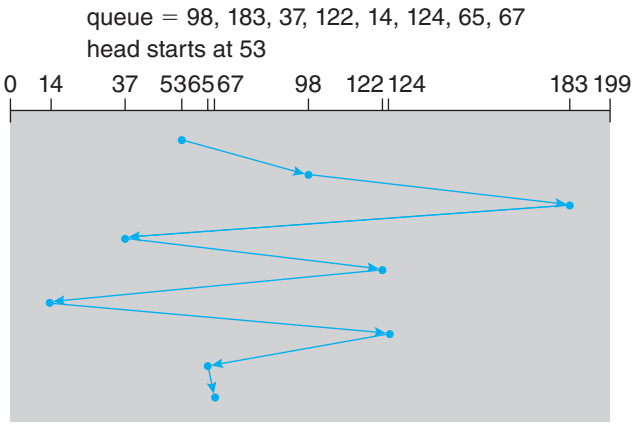


Figure 11.6 FCFS disk scheduling.

the direction of head movement in addition to the head's current position. Assuming that the disk arm is moving toward 0 and that the initial head position is again 53, the head will next service 37 and then 14. At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the requests at 65, 67, 98, 122, 124, and 183 (Figure 11.7). If a request arrives in the queue just in front of the head, it will be serviced almost immediately; a request arriving just behind the head will have to wait until the arm moves to the end of the disk, reverses direction, and comes back.

Assuming a uniform distribution of requests for cylinders, consider the density of requests when the head reaches one end and reverses direction. At this point, relatively few requests are immediately in front of the head, since these cylinders have recently been serviced. The heaviest density of requests is at the other end of the disk. These requests have also waited the longest, so why not go there first? That is the idea of the next algorithm.

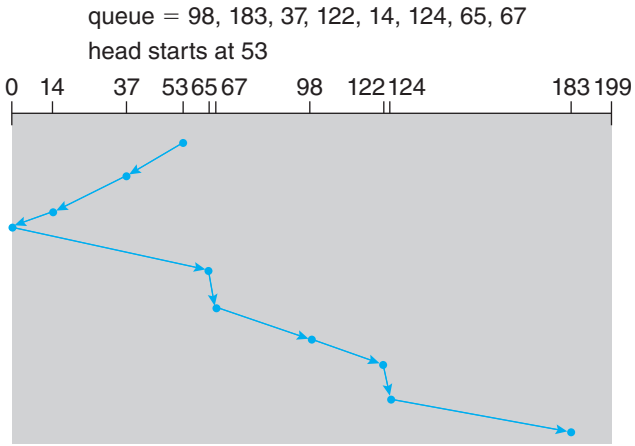


Figure 11.7 SCAN disk scheduling.

11.2.3 C-SCAN Scheduling

Circular SCAN (C-SCAN) scheduling is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk without servicing any requests on the return trip.

Let's return to our example to illustrate. Before applying C-SCAN to schedule the requests on cylinders 98, 183, 37, 122, 14, 124, 65, and 67, we need to know the direction of head movement in which the requests are scheduled. Assuming that the requests are scheduled when the disk arm is moving from 0 to 199 and that the initial head position is again 53, the request will be served as depicted in Figure 11.8. The C-SCAN scheduling algorithm essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one.

11.2.4 Selection of a Disk-Scheduling Algorithm

There are many disk-scheduling algorithms not included in this coverage, because they are rarely used. But how do operating system designers decide which to implement, and deployers chose the best to use? For any particular list of requests, we can define an optimal order of retrieval, but the computation needed to find an optimal schedule may not justify the savings over SCAN. With any scheduling algorithm, however, performance depends heavily on the number and types of requests. For instance, suppose that the queue usually has just one outstanding request. Then, all scheduling algorithms behave the same, because they have only one choice of where to move the disk head: they all behave like FCFS scheduling.

SCAN and C-SCAN perform better for systems that place a heavy load on the disk, because they are less likely to cause a starvation problem. There can still be starvation though, which drove Linux to create the **deadline** scheduler. This scheduler maintains separate read and write queues, and gives reads priority because processes are more likely to block on read than write. The queues are

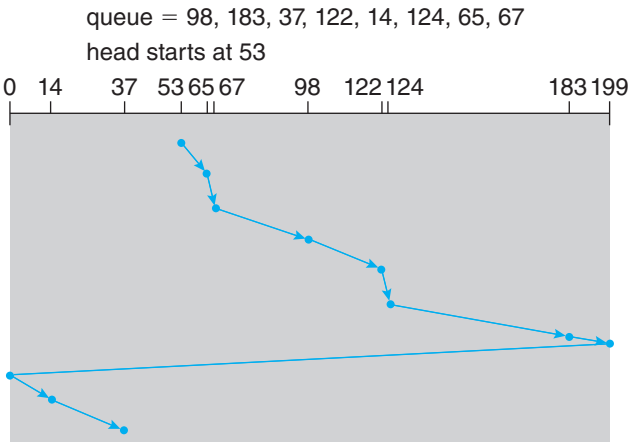


Figure 11.8 C-SCAN disk scheduling.

sorted in LBA order, essentially implementing C-SCAN. All I/O requests are sent in a batch in this LBA order. Deadline keeps four queues: two read and two write, one sorted by LBA and the other by FCFS. It checks after each batch to see if there are requests in the FCFS queues older than a configured age (by default, 500 ms). If so, the LBA queue (read or write) containing that request is selected for the next batch of I/O.

The deadline I/O scheduler is the default in the Linux RedHat 7 distribution, but **RHEL** 7 also includes two others. NOOP is preferred for CPU-bound systems using fast storage such as NVM devices, and the **Completely Fair Queueing scheduler** (**CFQ**) is the default for SATA drives. CFQ maintains three queues (with insertion sort to keep them sorted in LBA order): real time, best effort (the default), and idle. Each has exclusive priority over the others, in that order, with starvation possible. It uses historical data, anticipating if a process will likely issue more I/O requests soon. If it so determines, it idles waiting for the new I/O, ignoring other queued requests. This is to minimize seek time, assuming locality of reference of storage I/O requests, per process. Details of these schedulers can be found in https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Performance_Tuning_Guide/index.html.

11.3 NVM Scheduling

The disk-scheduling algorithms just discussed apply to mechanical platter-based storage like HDDs. They focus primarily on minimizing the amount of disk head movement. NVM devices do not contain moving disk heads and commonly use a simple FCFS policy. For example, the Linux **NOOP** scheduler uses an FCFS policy but modifies it to merge adjacent requests. The observed behavior of NVM devices indicates that the time required to service reads is uniform but that, because of the properties of flash memory, write service time is not uniform. Some SSD schedulers have exploited this property and merge only adjacent write requests, servicing all read requests in FCFS order.

As we have seen, I/O can occur sequentially or randomly. Sequential access is optimal for mechanical devices like HDD and tape because the data to be read or written is near the read/write head. Random-access I/O, which is measured in **input/output operations per second** (**IOPS**), causes HDD disk head movement. Naturally, random access I/O is much faster on NVM. An HDD can produce hundreds of IOPS, while an SSD can produce hundreds of thousands of IOPS.

NVM devices offer much less of an advantage for raw sequential throughput, where HDD head seeks are minimized and reading and writing of data to the media are emphasized. In those cases, for reads, performance for the two types of devices can range from equivalent to an order of magnitude advantage for NVM devices. Writing to NVM is slower than reading, decreasing the advantage. Furthermore, while write performance for HDDs is consistent throughout the life of the device, write performance for NVM devices varies depending on how full the device is (recall the need for garbage collection and over-provisioning) and how "worn" it is. An NVM device near its end of life due to many erase cycles generally has much worse performance than a new device.

One way to improve the lifespan and performance of NVM devices over time is to have the file system inform the device when files are deleted, so that the device can erase the blocks those files were stored on. This approach is discussed further in Section 14.5.6.

Let's look more closely at the impact of garbage collection on performance. Consider an NVM device under random read and write load. Assume that all blocks have been written to, but there is free space available. Garbage collection must occur to reclaim space taken by invalid data. That means that a write might cause a read of one or more pages, a write of the good data in those pages to overprovisioning space, an erase of the all-invalid-data block, and the placement of that block into overprovisioning space. In summary, one write request eventually causes a page write (the data), one or more page reads (by garbage collection), and one or more page writes (of good data from the garbage-collected blocks). The creation of I/O requests not by applications but by the NVM device doing garbage collection and space management is called **write amplificatio** and can greatly impact the write performance of the device. In the worst case, several extra I/Os are triggered with each write request.

11.4 Error Detection and Correction

Error detection and correction are fundamental to many areas of computing, including memory, networking, and storage. **Error detection** determines if a problem has occurred — for example a bit in DRAM spontaneously changed from a 0 to a 1, the contents of a network packet changed during transmission, or a block of data changed between when it was written and when it was read. By detecting the issue, the system can halt an operation before the error is propagated, report the error to the user or administrator, or warn of a device that might be starting to fail or has already failed.

Memory systems have long detected certain errors by using parity bits. In this scenario, each byte in a memory system has a parity bit associated with it that records whether the number of bits in the byte set to 1 is even (parity = 0) or odd (parity = 1). If one of the bits in the byte is damaged (either a 1 becomes a 0, or a 0 becomes a 1), the parity of the byte changes and thus does not match the stored parity. Similarly, if the stored parity bit is damaged, it does not match the computed parity. Thus, all single-bit errors are detected by the memory system. A double-bit-error might go undetected, however. Note that parity is easily calculated by performing an XOR (for "eXclusive OR") of the bits. Also note that for every byte of memory, we now need an extra bit of memory to store the parity.

Parity is one form of **checksums**, which use modular arithmetic to compute, store, and compare values on fixed-length words. Another error-detection method, common in networking, is a **cyclic redundancy check** (**CRCs**), which uses a hash function to detect multiple-bit errors (see http://www.mathpages.com/home/kmath458/kmath458.htm).

An **error-correction code** (**ECC**) not only detects the problem, but also corrects it. The correction is done by using algorithms and extra amounts of storage. The codes vary based on how much extra storage they need and how many errors they can correct. For example, disks drives use per-sector ECC and

flash drives per-page ECC. When the controller writes a sector/page of data during normal I/O, the ECC is written with a value calculated from all the bytes in the data being written. When the sector/page is read, the ECC is recalculated and compared with the stored value. If the stored and calculated numbers are different, this mismatch indicates that the data have become corrupted and that the storage media may be bad (Section 11.5.3). The ECC is error correcting because it contains enough information, if only a few bits of data have been corrupted, to enable the controller to identify which bits have changed and calculate what their correct values should be. It then reports a recoverable **soft error**. If too many changes occur, and the ECC cannot correct the error, a non-correctable **hard error** is signaled. The controller automatically does the ECC processing whenever a sector or page is read or written.

Error detection and correction are frequently differentiators between consumer products and enterprise products. ECC is used in some systems for DRAM error correction and data path protection, for example.

11.5 Storage Device Management

The operating system is responsible for several other aspects of storage device management, too. Here, we discuss drive initialization, booting from a drive, and bad-block recovery.

11.5.1 Drive Formatting, Partitions, and Volumes

A new storage device is a blank slate: it is just a platter of a magnetic recording material or a set of uninitialized semiconductor storage cells. Before a storage device can store data, it must be divided into sectors that the controller can read and write. NVM pages must be initialized and the FTL created. This process is called **low-level formatting**, or **physical formatting**. Low-level formatting fills the device with a special data structure for each storage location. The data structure for a sector or page typically consists of a header, a data area, and a trailer. The header and trailer contain information used by the controller, such as a sector/page number and an error detection or correction code.

Most drives are low-level-formatted at the factory as a part of the manufacturing process. This formatting enables the manufacturer to test the device and to initialize the mapping from logical block numbers to defect-free sectors or pages on the media. It is usually possible to choose among a few sector sizes, such as 512 bytes and 4KB. Formatting a disk with a larger sector size means that fewer sectors can fit on each track, but it also means that fewer headers and trailers are written on each track and more space is available for user data. Some operating systems can handle only one specific sector size.

Before it can use a drive to hold files, the operating system still needs to record its own data structures on the device. It does so in three steps.

The first step is to **partition** the device into one or more groups of blocks or pages. The operating system can treat each partition as though it were a separate device. For instance, one partition can hold a file system containing a copy of the operating system's executable code, another the swap space, and another a file system containing the user files. Some operating systems and file systems perform the partitioning automatically when an entire device is to

be managed by the file system. The partition information is written in a fixed format at a fixed location on the storage device. In Linux, the `fdisk` command is used to manage partitions on storage devices. The device, when recognized by the operating system, has its partition information read, and the operating system then creates device entries for the partitions (in `/dev` in Linux). From there, a configuration file, such as `/etc/fstab`, tells the operating system to mount each partition containing a file system at a specified location and to use mount options such as read-only. **Mounting** a file system is making the file system available for use by the system and its users.

The second step is volume creation and management. Sometimes, this step is implicit, as when a file system is placed directly within a partition. That **volume** is then ready to be mounted and used. At other times, volume creation and management is explicit—for example when multiple partitions or devices will be used together as a RAID set (see Section 11.8) with one or more file systems spread across the devices. The Linux volume manager `lvm2` can provide these features, as can commercial third-party tools for Linux and other operating systems. ZFS provides both volume management and a file system integrated into one set of commands and features. (Note that "volume" can also mean any mountable file system, even a file containing a file system such as a CD image.)

The third step is **logical formatting**, or creation of a file system. In this step, the operating system stores the initial file-system data structures onto the device. These data structures may include maps of free and allocated space and an initial empty directory.

The partition information also indicates if a partition contains a bootable file system (containing the operating system). The partition labeled for boot is used to establish the root of the file system. Once it is mounted, device links for all other devices and their partitions can be created. Generally, a computer's "file system" consists of all mounted volumes. On Windows, these are separately named via a letter (C:, D:, E:). On other systems, such as Linux, at boot time the boot file system is mounted, and other file systems can be mounted within that tree structure (as discussed in Section 13.3). On Windows, the file system interface makes it clear when a given device is being used, while in Linux a single file access might traverse many devices before the requested file in the requested file system (within a volume) is accessed. Figure 11.9 shows the Windows 7 `Disk Management` tool displaying three volumes (C:, E:, and F:). Note that E: and F: are each in a partition of the "Disk 1" device and that there is unallocated space on that device for more partitions (possibly containing file systems).

To increase efficiency, most file systems group blocks together into larger chunks, frequently called **clusters**. Device I/O is done via blocks, but file system I/O is done via clusters, effectively assuring that I/O has more sequential-access and fewer random-access characteristics. File systems try to group file contents near its metadata as well, reducing HDD head seeks when operating on a file, for example.

Some operating systems give special programs the ability to use a partition as a large sequential array of logical blocks, without any file-system data structures. This array is sometimes called the **raw disk**, and I/O to this array is termed ***raw I/O***. It can be used for swap space (see Section 11.6.2), for example, and some database systems prefer raw I/O because it enables them to control

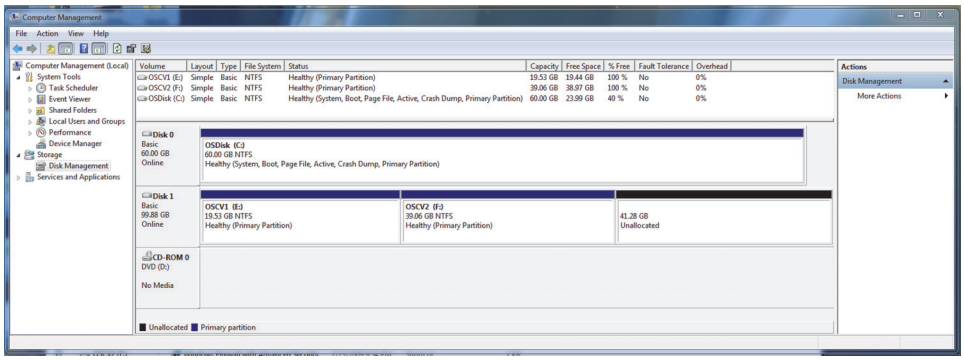


Figure 11.9 Windows 7 `Disk Management` tool showing devices, partitions, volumes, and file systems.

the exact location where each database record is stored. Raw I/O bypasses all the file-system services, such as the buffer cache, file locking, prefetching, space allocation, file names, and directories. We can make certain applications more efficient by allowing them to implement their own special-purpose storage services on a raw partition, but most applications use a provided file system rather than managing data themselves. Note that Linux generally does not support raw I/O but can achieve similar access by using the DIRECT flag to the `open()` system call.

11.5.2 Boot Block

For a computer to start running—for instance, when it is powered up or rebooted—it must have an initial program to run. This initial **bootstrap** loader tends to be simple. For most computers, the bootstrap is stored in NVM flash memory firmware on the system motherboard and mapped to a known memory location. It can be updated by product manufacturers as needed, but also can be written to by viruses, infecting the system. It initializes all aspects of the system, from CPU registers to device controllers and the contents of main memory.

This tiny bootstrap loader program is also smart enough to bring in a full bootstrap program from secondary storage. The full bootstrap program is stored in the "boot blocks" at a fixed location on the device. The default Linux bootstrap loader is grub2 (https://www.gnu.org/software/grub/manual/grub.html/). A device that has a boot partition is called a **boot disk** or **system disk**.

The code in the bootstrap NVM instructs the storage controller to read the boot blocks into memory (no device drivers are loaded at this point) and then starts executing that code. The full bootstrap program is more sophisticated than the bootstrap loader: it is able to load the entire operating system from a non-fixed location on the device and to start the operating system running.

Let's consider as an example the boot process in Windows. First, note that Windows allows a drive to be divided into partitions, and one partition—identified as the **boot partition**—contains the operating system and device drivers. The Windows system places its boot code in the first logical block on the hard disk or first page of the NVM device, which it terms the **master boot**

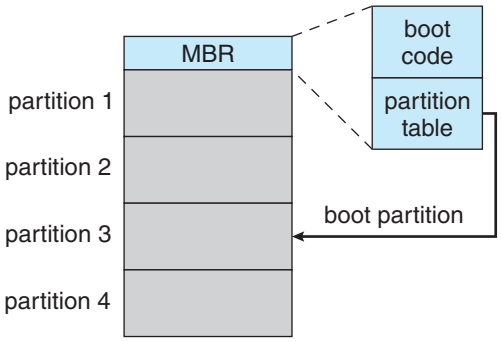


Figure 11.10 Booting from a storage device in Windows.

record, or **MBR**. Booting begins by running code that is resident in the system's firmware. This code directs the system to read the boot code from the MBR, understanding just enough about the storage controller and storage device to load a sector from it. In addition to containing boot code, the MBR contains a table listing the partitions for the drive and a flag indicating which partition the system is to be booted from, as illustrated in Figure 11.10. Once the system identifies the boot partition, it reads the first sector/page from that partition (called the **boot sector**), which directs it to the kernel. It then continues with the remainder of the boot process, which includes loading the various subsystems and system services.

11.5.3 Bad Blocks

Because disks have moving parts and small tolerances (recall that the disk head flies just above the disk surface), they are prone to failure. Sometimes the failure is complete; in this case, the disk needs to be replaced and its contents restored from backup media to the new disk. More frequently, one or more sectors become defective. Most disks even come from the factory with **bad blocks**. Depending on the disk and controller in use, these blocks are handled in a variety of ways.

On older disks, such as some disks with IDE controllers, bad blocks are handled manually. One strategy is to scan the disk to find bad blocks while the disk is being formatted. Any bad blocks that are discovered are flagged as unusable so that the file system does not allocate them. If blocks go bad during normal operation, a special program (such as the Linux `badblocks` command) must be run manually to search for the bad blocks and to lock them away. Data that resided on the bad blocks usually are lost.

More sophisticated disks are smarter about bad-block recovery. The controller maintains a list of bad blocks on the disk. The list is initialized during the low-level formatting at the factory and is updated over the life of the disk. Low-level formatting also sets aside spare sectors not visible to the operating system. The controller can be told to replace each bad sector logically with one of the spare sectors. This scheme is known as **sector sparing** or **forwarding**.

A typical bad-sector transaction might be as follows:

- The operating system tries to read logical block 87.

- The controller calculates the ECC and finds that the sector is bad. It reports this finding to the operating system as an I/O error.
- The device controller replaces the bad sector with a spare.
- After that, whenever the system requests logical block 87, the request is translated into the replacement sector's address by the controller.

Note that such a redirection by the controller could invalidate any optimization by the operating system's disk-scheduling algorithm! For this reason, most disks are formatted to provide a few spare sectors in each cylinder and a spare cylinder as well. When a bad block is remapped, the controller uses a spare sector from the same cylinder, if possible.

As an alternative to sector sparing, some controllers can be instructed to replace a bad block by **sector slipping**. Here is an example: Suppose that logical block 17 becomes defective and the first available spare follows sector 202. Sector slipping then remaps all the sectors from 17 to 202, moving them all down one spot. That is, sector 202 is copied into the spare, then sector 201 into 202, then 200 into 201, and so on, until sector 18 is copied into sector 19. Slipping the sectors in this way frees up the space of sector 18 so that sector 17 can be mapped to it.

Recoverable soft errors may trigger a device activity in which a copy of the block data is made and the block is spared or slipped. An unrecoverable **hard error**, however, results in lost data. Whatever file was using that block must be repaired (for instance, by restoration from a backup tape), and that requires manual intervention.

NVM devices also have bits, bytes, and even pages that either are nonfunctional at manufacturing time or go bad over time. Management of those faulty areas is simpler than for HDDs because there is no seek time performance loss to be avoided. Either multiple pages can be set aside and used as replacement locations, or space from the over-provisioning area can be used (decreasing the usable capacity of the over-provisioning area). Either way, the controller maintains a table of bad pages and never sets those pages as available to write to, so they are never accessed.

11.6 Swap-Space Management

Swapping was first presented in Section 9.5, where we discussed moving entire processes between secondary storage and main memory. Swapping in that setting occurs when the amount of physical memory reaches a critically low point and processes are moved from memory to swap space to free available memory. In practice, very few modern operating systems implement swapping in this fashion. Rather, systems now combine swapping with virtual memory techniques (Chapter 10) and swap pages, not necessarily entire processes. In fact, some systems now use the terms "swapping" and "paging" interchangeably, reflecting the merging of these two concepts.

Swap-space management is another low-level task of the operating system. Virtual memory uses secondary storage space as an extension of main memory. Since drive access is much slower than memory access, using swap

space significantly decreases system performance. The main goal for the design and implementation of swap space is to provide the best throughput for the virtual memory system. In this section, we discuss how swap space is used, where swap space is located on storage devices, and how swap space is managed.

11.6.1 Swap-Space Use

Swap space is used in various ways by different operating systems, depending on the memory-management algorithms in use. For instance, systems that implement swapping may use swap space to hold an entire process image, including the code and data segments. Paging systems may simply store pages that have been pushed out of main memory. The amount of swap space needed on a system can therefore vary from a few megabytes of disk space to gigabytes, depending on the amount of physical memory, the amount of virtual memory it is backing, and the way in which the virtual memory is used.

Note that it may be safer to overestimate than to underestimate the amount of swap space required, because if a system runs out of swap space it may be forced to abort processes or may crash entirely. Overestimation wastes secondary storage space that could otherwise be used for files, but it does no other harm. Some systems recommend the amount to be set aside for swap space. Solaris, for example, suggests setting swap space equal to the amount by which virtual memory exceeds pageable physical memory. In the past, Linux has suggested setting swap space to double the amount of physical memory. Today, the paging algorithms have changed, and most Linux systems use considerably less swap space.

Some operating systems—including Linux—allow the use of multiple swap spaces, including both files and dedicated swap partitions. These swap spaces are usually placed on separate storage devices so that the load placed on the I/O system by paging and swapping can be spread over the system's I/O bandwidth.

11.6.2 Swap-Space Location

A swap space can reside in one of two places: it can be carved out of the normal file system, or it can be in a separate partition. If the swap space is simply a large file within the file system, normal file-system routines can be used to create it, name it, and allocate its space.

Alternatively, swap space can be created in a separate **raw partition**. No file system or directory structure is placed in this space. Rather, a separate swap-space storage manager is used to allocate and deallocate the blocks from the raw partition. This manager uses algorithms optimized for speed rather than for storage efficiency, because swap space is accessed much more frequently than file systems, when it is used (recall that swap space is used for swapping and paging). Internal fragmentation may increase, but this trade-off is acceptable because the life of data in the swap space generally is much shorter than that of files in the file system. Since swap space is reinitialized at boot time, any fragmentation is short-lived. The raw-partition approach creates a fixed amount of swap space during disk partitioning. Adding more swap space requires either repartitioning the device (which involves moving

the other file-system partitions or destroying them and restoring them from backup) or adding another swap space elsewhere.

Some operating systems are flexible and can swap both in raw partitions and in file-system space. Linux is an example: the policy and implementation are separate, allowing the machine's administrator to decide which type of swapping to use. The trade-off is between the convenience of allocation and management in the file system and the performance of swapping in raw partitions.

11.6.3 Swap-Space Management: An Example

We can illustrate how swap space is used by following the evolution of swapping and paging in various UNIX systems. The traditional UNIX kernel started with an implementation of swapping that copied entire processes between contiguous disk regions and memory. UNIX later evolved to a combination of swapping and paging as paging hardware became available.

In Solaris 1 (SunOS), the designers changed standard UNIX methods to improve efficiency and reflect technological developments. When a process executes, text-segment pages containing code are brought in from the file system, accessed in main memory, and thrown away if selected for pageout. It is more efficient to reread a page from the file system than to write it to swap space and then reread it from there. Swap space is only used as a backing store for pages of **anonymous** memory (memory not backed by any file), which includes memory allocated for the stack, heap, and uninitialized data of a process.

More changes were made in later versions of Solaris. The biggest change is that Solaris now allocates swap space only when a page is forced out of physical memory, rather than when the virtual memory page is first created. This scheme gives better performance on modern computers, which have more physical memory than older systems and tend to page less.

Linux is similar to Solaris in that swap space is now used only for anonymous memory. Linux allows one or more swap areas to be established. A swap area may be in either a swap file on a regular file system or a dedicated swap partition. Each swap area consists of a series of 4-KB **page slots**, which are used to hold swapped pages. Associated with each swap area is a **swap map**—an array of integer counters, each corresponding to a page slot in the swap area. If the value of a counter is 0, the corresponding page slot is available. Values greater than 0 indicate that the page slot is occupied by a swapped page. The value of the counter indicates the number of mappings to the swapped page. For example, a value of 3 indicates that the swapped page is mapped to three different processes (which can occur if the swapped page is storing a region of memory shared by three processes). The data structures for swapping on Linux systems are shown in Figure 11.11.

11.7 Storage Attachment

Computers access secondary storage in three ways: via host-attached storage, network-attached storage, and cloud storage.

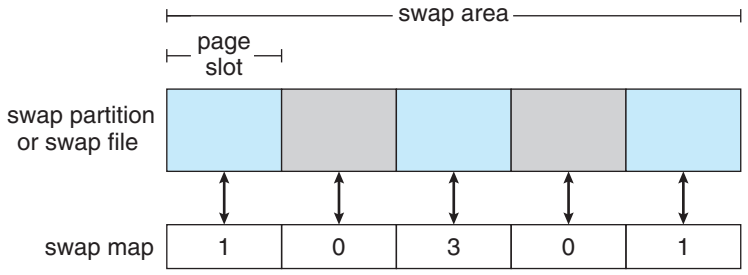


Figure 11.11 The data structures for swapping on Linux systems.

11.7.1 Host-Attached Storage

Host-attached storage is storage accessed through local I/O ports. These ports use several technologies, the most common being SATA, as mentioned earlier. A typical system has one or a few SATA ports.

To allow a system to gain access to more storage, either an individual storage device, a device in a chassis, or multiple drives in a chassis can be connected via USB FireWire or Thunderbolt ports and cables.

High-end workstations and servers generally need more storage or need to share storage, so use more sophisticated I/O architectures, such as **fibr channel** (FC), a high-speed serial architecture that can operate over optical fiber or over a four-conductor copper cable. Because of the large address space and the switched nature of the communication, multiple hosts and storage devices can attach to the fabric, allowing great flexibility in I/O communication.

A wide variety of storage devices are suitable for use as host-attached storage. Among these are HDDs; NVM devices; CD, DVD, Blu-ray, and tape drives; and storage-area networks (**SANs**) (discussed in Section 11.7.4). The I/O commands that initiate data transfers to a host-attached storage device are reads and writes of logical data blocks directed to specifically identified storage units (such as bus ID or target logical unit).

11.7.2 Network-Attached Storage

Network-attached storage (**NAS**) (Figure 11.12) provides access to storage across a network. An NAS device can be either a special-purpose storage system or a general computer system that provides its storage to other hosts across the network. Clients access network-attached storage via a remote-procedure-call interface such as NFS for UNIX and Linux systems or CIFS for Windows machines. The remote procedure calls (RPCs) are carried via TCP or UDP over an IP network—usually the same local-area network (LAN) that carries all data traffic to the clients. The network-attached storage unit is usually implemented as a storage array with software that implements the RPC interface.

CIFS and NFS provide various locking features, allowing the sharing of files between hosts accessing a NAS with those protocols. For example, a user logged in to multiple NAS clients can access her home directory from all of those clients, simultaneously.

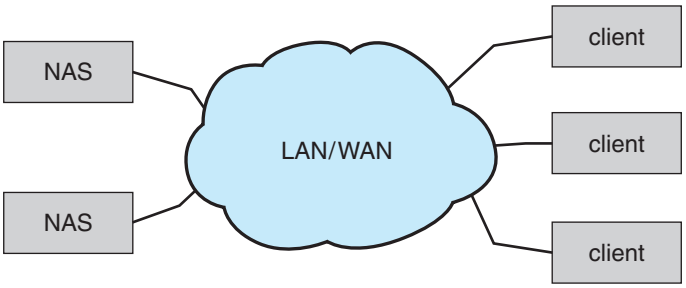


Figure 11.12 Network-attached storage.

Network-attached storage provides a convenient way for all the computers on a LAN to share a pool of storage with the same ease of naming and access enjoyed with local host-attached storage. However, it tends to be less efficient and have lower performance than some direct-attached storage options.

iSCSI is the latest network-attached storage protocol. In essence, it uses the IP network protocol to carry the SCSI protocol. Thus, networks—rather than SCSI cables—can be used as the interconnects between hosts and their storage. As a result, hosts can treat their storage as if it were directly attached, even if the storage is distant from the host. Whereas NFS and CIFS present a file system and send parts of files across the network, iSCSI sends logical blocks across the network and leaves it to the client to use the blocks directly or create a file system with them.

11.7.3 Cloud Storage

Section 1.10.5 discussed cloud computing. One offering from cloud providers is **cloud storage**. Similar to network-attached storage, cloud storage provides access to storage across a network. Unlike NAS, the storage is accessed over the Internet or another WAN to a remote data center that provides storage for a fee (or even for free).

Another difference between NAS and cloud storage is how the storage is accessed and presented to users. NAS is accessed as just another file system if the CIFS or NFS protocols are used, or as a raw block device if the iSCSI protocol is used. Most operating systems have these protocols integrated and present NAS storage in the same way as other storage. In contrast, cloud storage is API based, and programs use the APIs to access the storage. Amazon S3 is a leading cloud storage offering. Dropbox is an example of a company that provides apps to connect to the cloud storage that it provides. Other examples include Microsoft OneDrive and Apple iCloud.

One reason that APIs are used instead of existing protocols is the latency and failure scenarios of a WAN. NAS protocols were designed for use in LANs, which have lower latency than WANs and are much less likely to lose connectivity between the storage user and the storage device. If a LAN connection fails, a system using NFS or CIFS might hang until it recovers. With cloud storage, failures like that are more likely, so an application simply pauses access until connectivity is restored.

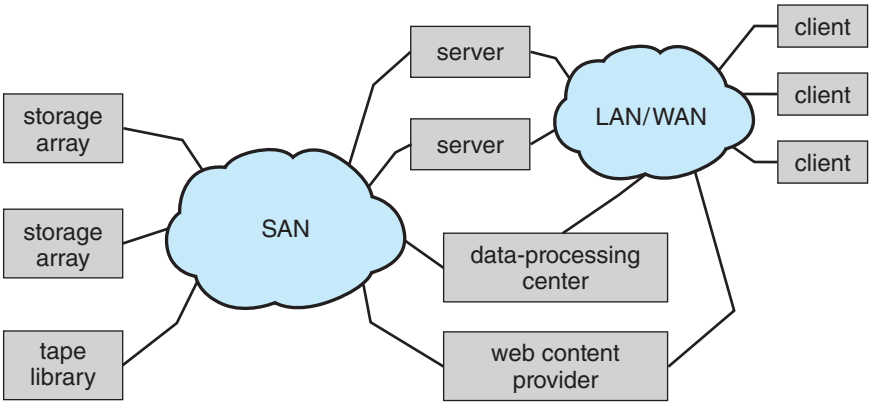


Figure 11.13 Storage-area network.

11.7.4 Storage-Area Networks and Storage Arrays

One drawback of network-attached storage systems is that the storage I/O operations consume bandwidth on the data network, thereby increasing the latency of network communication. This problem can be particularly acute in large client–server installations—the communication between servers and clients competes for bandwidth with the communication among servers and storage devices.

A **storage-area network** (**SAN**) is a private network (using storage protocols rather than networking protocols) connecting servers and storage units, as shown in Figure 11.13. The power of a SAN lies in its flexibility. Multiple hosts and multiple storage arrays can attach to the same SAN, and storage can be dynamically allocated to hosts. The storage arrays can be RAID protected or unprotected drives (**Just a Bunch of Disks** (**JBOD**)). A SAN switch allows or prohibits access between the hosts and the storage. As one example, if a host is running low on disk space, the SAN can be configured to allocate more storage to that host. SANs make it possible for clusters of servers to share the same storage and for storage arrays to include multiple direct host connections. SANs typically have more ports—and cost more—than storage arrays. SAN connectivity is over short distances and typically has no routing, so a NAS can have many more connected hosts than a SAN.

A storage array is a purpose-built device (see Figure 11.14) that includes SAN ports, network ports, or both. It also contains drives to store data and a controller (or redundant set of controllers) to manage the storage and allow access to the storage across the networks. The controllers are composed of CPUs, memory, and software that implement the features of the array, which can include network protocols, user interfaces, RAID protection, snapshots, replication, compression, deduplication, and encryption. Some of those functions are discussed in Chapter 14.

Some storage arrays include SSDs. An array may contain only SSDs, resulting in maximum performance but smaller capacity, or may include a mix of SSDs and HDDs, with the array software (or the administrator) selecting the best medium for a given use or using the SSDs as a cache and HDDs as bulk storage.

Figure 11.14 A storage array.

FC is the most common SAN interconnect, although the simplicity of iSCSI is increasing its use. Another SAN interconnect is **InfiniBan** (IB)—a special-purpose bus architecture that provides hardware and software support for high-speed interconnection networks for servers and storage units.

11.8 RAID Structure

Storage devices have continued to get smaller and cheaper, so it is now economically feasible to attach many drives to a computer system. Having a large number of drives in a system presents opportunities for improving the rate at which data can be read or written, if the drives are operated in parallel. Furthermore, this setup offers the potential for improving the reliability of data storage, because redundant information can be stored on multiple drives. Thus, failure of one drive does not lead to loss of data. A variety of disk-organization techniques, collectively called **redundant arrays of independent disks** (RAIDs), are commonly used to address the performance and reliability issues.

In the past, RAIDs composed of small, cheap disks were viewed as a cost-effective alternative to large, expensive disks. Today, RAIDs are used for their higher reliability and higher data-transfer rate rather than for economic reasons. Hence, the *I* in *RAID*, which once stood for "inexpensive," now stands for "independent."

11.8.1 Improvement of Reliability via Redundancy

Let's first consider the reliability of a RAID of HDDs. The chance that some disk out of a set of N disks will fail is much greater than the chance that a specific single disk will fail. Suppose that the **mean time between failures** (MTBF) of a single disk is 100,000 hours. Then the MTBF of some disk in an array of 100

STRUCTURING RAID

RAID storage can be structured in a variety of ways. For example, a system can have drives directly attached to its buses. In this case, the operating system or system software can implement RAID functionality. Alternatively, an intelligent host controller can control multiple attached devices and can implement RAID on those devices in hardware. Finally, a storage array can be used. A storage array, as just discussed, is a standalone unit with its own controller, cache, and drives. It is attached to the host via one or more standard controllers (for example, FC). This common setup allows an operating system or software without RAID functionality to have RAID-protected storage.

disks will be 100,000/100 = 1,000 hours, or 41.66 days, which is not long at all! If we store only one copy of the data, then each disk failure will result in loss of a significant amount of data—and such a high rate of data loss is unacceptable.

The solution to the problem of reliability is to introduce **redundancy**; we store extra information that is not normally needed but can be used in the event of disk failure to rebuild the lost information. Thus, even if a disk fails, data are not lost. RAID can be applied to NVM devices as well, although NVM devices have no moving parts and therefore are less likely to fail than HDDs.

The simplest (but most expensive) approach to introducing redundancy is to duplicate every drive. This technique is called **mirroring**. With mirroring, a logical disk consists of two physical drives, and every write is carried out on both drives. The result is called a **mirrored volume**. If one of the drives in the volume fails, the data can be read from the other. Data will be lost only if the second drive fails before the first failed drive is replaced.

The MTBF of a mirrored volume—where failure is the loss of data—depends on two factors. One is the MTBF of the individual drives. The other is the **mean time to repair**, which is the time it takes (on average) to replace a failed drive and to restore the data on it. Suppose that the failures of the two drives are independent; that is, the failure of one is not connected to the failure of the other. Then, if the MTBF of a single drive is 100,000 hours and the mean time to repair is 10 hours, the **mean time to data loss** of a mirrored drive system is $100,000^2/(2 * 10) = 500 * 10^6$ hours, or 57,000 years!

You should be aware that we cannot really assume that drive failures will be independent. Power failures and natural disasters, such as earthquakes, fires, and floods, may result in damage to both drives at the same time. Also, manufacturing defects in a batch of drives can cause correlated failures. As drives age, the probability of failure grows, increasing the chance that a second drive will fail while the first is being repaired. In spite of all these considerations, however, mirrored-drive systems offer much higher reliability than do single-drive systems.

Power failures are a particular source of concern, since they occur far more frequently than do natural disasters. Even with mirroring of drives, if writes are in progress to the same block in both drives, and power fails before both blocks are fully written, the two blocks can be in an inconsistent state. One solution to this problem is to write one copy first, then the next. Another is to add a solid-state nonvolatile cache to the RAID array. This write-back cache is

protected from data loss during power failures, so the write can be considered complete at that point, assuming the cache has some kind of error protection and correction, such as ECC or mirroring.

11.8.2 Improvement in Performance via Parallelism

Now let's consider how parallel access to multiple drives improves performance. With mirroring, the rate at which read requests can be handled is doubled, since read requests can be sent to either drive (as long as both in a pair are functional, as is almost always the case). The transfer rate of each read is the same as in a single-drive system, but the number of reads per unit time has doubled.

With multiple drives, we can improve the transfer rate as well (or instead) by striping data across the drives. In its simplest form, **data striping** consists of splitting the bits of each byte across multiple drives; such striping is called **bit-level striping**. For example, if we have an array of eight drives, we write bit i of each byte to drive i. The array of eight drives can be treated as a single drive with sectors that are eight times the normal size and, more important, have eight times the access rate. Every drive participates in every access (read or write); so the number of accesses that can be processed per second is about the same as on a single drive, but each access can read eight times as many data in the same time as on a single drive.

Bit-level striping can be generalized to include a number of drives that either is a multiple of 8 or divides 8. For example, if we use an array of four drives, bits i and $4+i$ of each byte go to drive i. Further, striping need not occur at the bit level. In **block-level striping**, for instance, blocks of a file are striped across multiple drives; with n drives, block i of a file goes to drive $(i \bmod n)+1$. Other levels of striping, such as bytes of a sector or sectors of a block, also are possible. Block-level striping is the only commonly available striping.

Parallelism in a storage system, as achieved through striping, has two main goals:

1. Increase the throughput of multiple small accesses (that is, page accesses) by load balancing.
2. Reduce the response time of large accesses.

11.8.3 RAID Levels

Mirroring provides high reliability, but it is expensive. Striping provides high data-transfer rates, but it does not improve reliability. Numerous schemes to provide redundancy at lower cost by using disk striping combined with "parity" bits (which we describe shortly) have been proposed. These schemes have different cost–performance trade-offs and are classified according to levels called **RAID levels**. We describe only the most common levels here; Figure 11.15 shows them pictorially (in the figure, P indicates error-correcting bits and C indicates a second copy of the data). In all cases depicted in the figure, four drives' worth of data are stored, and the extra drives are used to store redundant information for failure recovery.

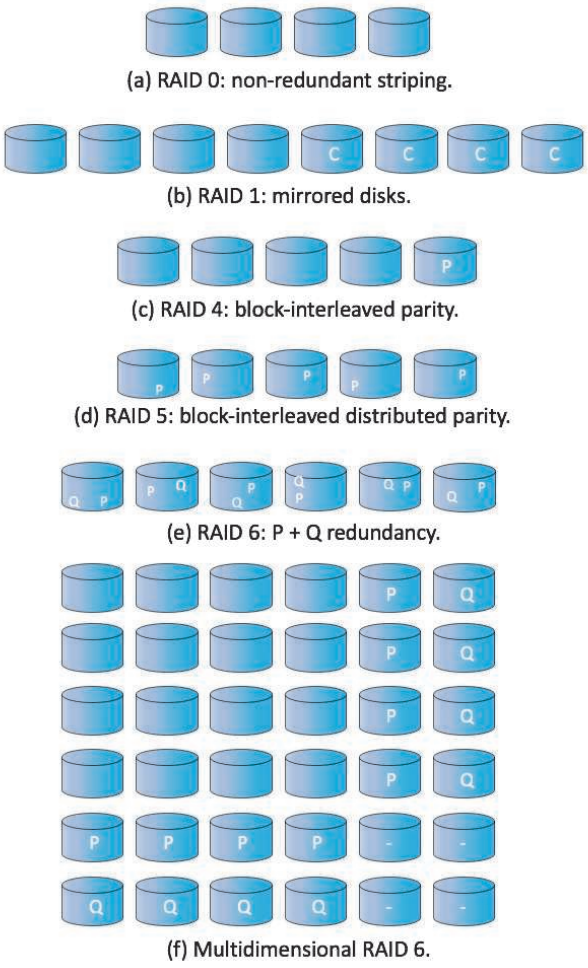


Figure 11.15 RAID levels.

- **RAID level 0**. RAID level 0 refers to drive arrays with striping at the level of blocks but without any redundancy (such as mirroring or parity bits), as shown in Figure 11.15(a).
- **RAID level 1**. RAID level 1 refers to drive mirroring. Figure 11.15(b) shows a mirrored organization.
- **RAID level 4**. RAID level 4 is also known as memory-style error-correcting-code (ECC) organization. ECC is also used in RAID 5 and 6.

  The idea of ECC can be used directly in storage arrays via striping of blocks across drives. For example, the first data block of a sequence of writes can be stored in drive 1, the second block in drive 2, and so on until the Nth block is stored in drive N; the error-correction calculation result of those blocks is stored on drive $N + 1$. This scheme is shown in Figure 11.15(c), where the drive labeled P stores the error-correction block. If one of the drives fails, the error-correction code recalculation detects that and

prevents the data from being passed to the requesting process, throwing an error.

RAID 4 can actually correct errors, even though there is only one ECC block. It takes into account the fact that, unlike memory systems, drive controllers can detect whether a sector has been read correctly, so a single parity block can be used for error correction and detection. The idea is as follows: If one of the sectors is damaged, we know exactly which sector it is. We disregard the data in that sector and use the parity data to recalculate the bad data. For every bit in the block, we can determine if it is a 1 or a 0 by computing the parity of the corresponding bits from sectors in the other drives. If the parity of the remaining bits is equal to the stored parity, the missing bit is 0; otherwise, it is 1.

A block read accesses only one drive, allowing other requests to be processed by the other drives. The transfer rates for large reads are high, since all the disks can be read in parallel. Large writes also have high transfer rates, since the data and parity can be written in parallel.

Small independent writes cannot be performed in parallel. An operating-system write of data smaller than a block requires that the block be read, modified with the new data, and written back. The parity block has to be updated as well. This is known as the **read-modify-write cycle**. Thus, a single write requires four drive accesses: two to read the two old blocks and two to write the two new blocks.

WAFL (which we cover in Chapter 14) uses RAID level 4 because this RAID level allows drives to be added to a RAID set seamlessly. If the added drives are initialized with blocks containing only zeros, then the parity value does not change, and the RAID set is still correct.

RAID level 4 has two advantages over level 1 while providing equal data protection. First, the storage overhead is reduced because only one parity drive is needed for several regular drives, whereas one mirror drive is needed for every drive in level 1. Second, since reads and writes of a series of blocks are spread out over multiple drives with N-way striping of data, the transfer rate for reading or writing a set of blocks is N times as fast as with level 1.

A performance problem with RAID 4—and with all parity-based RAID levels—is the expense of computing and writing the XOR parity. This overhead can result in slower writes than with non-parity RAID arrays. Modern general-purpose CPUs are very fast compared with drive I/O, however, so the performance hit can be minimal. Also, many RAID storage arrays or host bus-adapters include a hardware controller with dedicated parity hardware. This controller offloads the parity computation from the CPU to the array. The array has an NVRAM cache as well, to store the blocks while the parity is computed and to buffer the writes from the controller to the drives. Such buffering can avoid most read-modify-write cycles by gathering data to be written into a full stripe and writing to all drives in the stripe concurrently. This combination of hardware acceleration and buffering can make parity RAID almost as fast as non-parity RAID, frequently outperforming a non-caching non-parity RAID.

- **RAID level 5**. RAID level 5, or block-interleaved distributed parity, differs from level 4 in that it spreads data and parity among all $N+1$ drives, rather

than storing data in N drives and parity in one drive. For each set of N blocks, one of the drives stores the parity and the others store data. For example, with an array of five drives, the parity for the nth block is stored in drive (n mod 5) + 1. The nth blocks of the other four drives store actual data for that block. This setup is shown in Figure 11.15(d), where the Ps are distributed across all the drives. A parity block cannot store parity for blocks in the same drive, because a drive failure would result in loss of data as well as of parity, and hence the loss would not be recoverable. By spreading the parity across all the drives in the set, RAID 5 avoids potential overuse of a single parity drive, which can occur with RAID 4. RAID 5 is the most common parity RAID.

- **RAID level 6**. RAID level 6, also called the **P + Q redundancy scheme**, is much like RAID level 5 but stores extra redundant information to guard against multiple drive failures. XOR parity cannot be used on both parity blocks because they would be identical and would not provide more recovery information. Instead of parity, error-correcting codes such as **Galois fiel math** are used to calculate Q. In the scheme shown in Figure 11.15(e), 2 blocks of redundant data are stored for every 4 blocks of data—compared with 1 parity block in level 5—and the system can tolerate two drive failures.

- **Multidimensional RAID level 6**. Some sophisticated storage arrays amplify RAID level 6. Consider an array containing hundreds of drives. Putting those drives in a RAID level 6 stripe would result in many data drives and only two logical parity drives. Multidimensional RAID level 6 logically arranges drives into rows and columns (two or more dimensional arrays) and implements RAID level 6 both horizontally along the rows and vertically down the columns. The system can recover from any failure—or, indeed, multiple failures—by using parity blocks in any of these locations. This RAID level is shown in Figure 11.15(f). For simplicity, the figure shows the RAID parity on dedicated drives, but in reality the RAID blocks are scattered throughout the rows and columns.

- **RAID levels 0 + 1 and 1 + 0**. RAID level 0 + 1 refers to a combination of RAID levels 0 and 1. RAID 0 provides the performance, while RAID 1 provides the reliability. Generally, this level provides better performance than RAID 5. It is common in environments where both performance and reliability are important. Unfortunately, like RAID 1, it doubles the number of drives needed for storage, so it is also relatively expensive. In RAID 0 + 1, a set of drives are striped, and then the stripe is mirrored to another, equivalent stripe.

  Another RAID variation is RAID level 1 + 0, in which drives are mirrored in pairs and then the resulting mirrored pairs are striped. This scheme has some theoretical advantages over RAID 0 + 1. For example, if a single drive fails in RAID 0 + 1, an entire stripe is inaccessible, leaving only the other stripe. With a failure in RAID 1 + 0, a single drive is unavailable, but the drive that mirrors it is still available, as are all the rest of the drives (Figure 11.16).

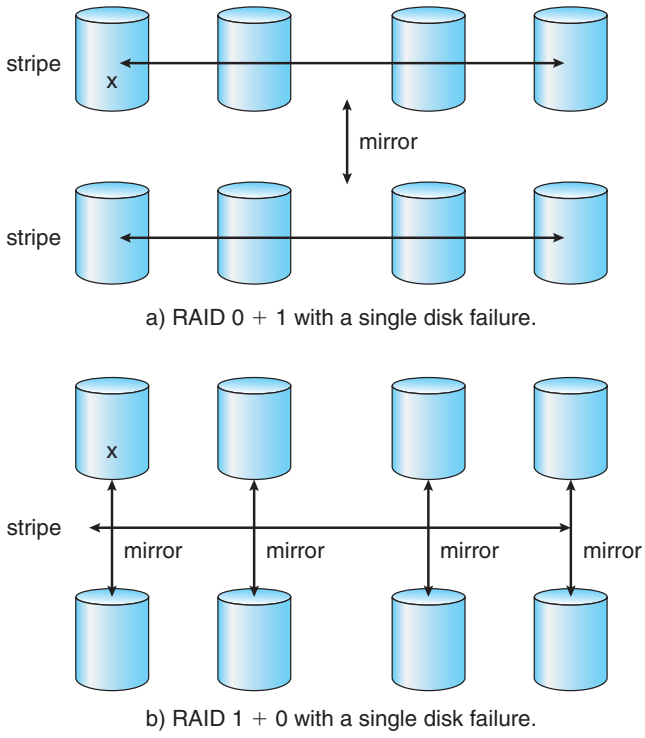


Figure 11.16 RAID 0 + 1 and 1 + 0 with a single disk failure.

Numerous variations have been proposed to the basic RAID schemes described here. As a result, some confusion may exist about the exact definitions of the different RAID levels.

The implementation of RAID is another area of variation. Consider the following layers at which RAID can be implemented.

- Volume-management software can implement RAID within the kernel or at the system software layer. In this case, the storage hardware can provide minimal features and still be part of a full RAID solution.
- RAID can be implemented in the host bus-adapter (HBA) hardware. Only the drives directly connected to the HBA can be part of a given RAID set. This solution is low in cost but not very flexible.
- RAID can be implemented in the hardware of the storage array. The storage array can create RAID sets of various levels and can even slice these sets into smaller volumes, which are then presented to the operating system. The operating system need only implement the file system on each of the volumes. Arrays can have multiple connections available or can be part of a SAN, allowing multiple hosts to take advantage of the array's features.
- RAID can be implemented in the SAN interconnect layer by drive virtualization devices. In this case, a device sits between the hosts and the storage. It

accepts commands from the servers and manages access to the storage. It could provide mirroring, for example, by writing each block to two separate storage devices.

Other features, such as snapshots and replication, can be implemented at each of these levels as well. A **snapshot** is a view of the file system before the last update took place. (Snapshots are covered more fully in Chapter 14.) **Replication** involves the automatic duplication of writes between separate sites for redundancy and disaster recovery. Replication can be synchronous or asynchronous. In synchronous replication, each block must be written locally and remotely before the write is considered complete, whereas in asynchronous replication, the writes are grouped together and written periodically. Asynchronous replication can result in data loss if the primary site fails, but it is faster and has no distance limitations. Increasingly, replication is also used within a data center or even within a host. As an alternative to RAID protection, replication protects against data loss and also increases read performance (by allowing reads from each of the replica copies). It does of course use more storage than most types of RAID.

The implementation of these features differs depending on the layer at which RAID is implemented. For example, if RAID is implemented in software, then each host may need to carry out and manage its own replication. If replication is implemented in the storage array or in the SAN interconnect, however, then whatever the host operating system or its features, the host's data can be replicated.

One other aspect of most RAID implementations is a hot spare drive or drives. A **hot spare** is not used for data but is configured to be used as a replacement in case of drive failure. For instance, a hot spare can be used to rebuild a mirrored pair should one of the drives in the pair fail. In this way, the RAID level can be reestablished automatically, without waiting for the failed drive to be replaced. Allocating more than one hot spare allows more than one failure to be repaired without human intervention.

11.8.4 Selecting a RAID Level

Given the many choices they have, how do system designers choose a RAID level? One consideration is rebuild performance. If a drive fails, the time needed to rebuild its data can be significant. This may be an important factor if a continuous supply of data is required, as it is in high-performance or interactive database systems. Furthermore, rebuild performance influences the mean time between failures.

Rebuild performance varies with the RAID level used. Rebuilding is easiest for RAID level 1, since data can be copied from another drive. For the other levels, we need to access all the other drives in the array to rebuild data in a failed drive. Rebuild times can be hours for RAID level 5 rebuilds of large drive sets.

RAID level 0 is used in high-performance applications where data loss is not critical. For example, in scientific computing where a data set is loaded and explored, RAID level 0 works well because any drive failures would just require a repair and reloading of the data from its source. RAID level 1 is popular for applications that require high reliability with fast recovery. RAID

THE InServ STORAGE ARRAY

Innovation, in an effort to provide better, faster, and less expensive solutions, frequently blurs the lines that separated previous technologies. Consider the InServ storage array from HP 3Par. Unlike most other storage arrays, InServ does not require that a set of drives be configured at a specific RAID level. Rather, each drive is broken into 256-MB "chunklets." RAID is then applied at the chunklet level. A drive can thus participate in multiple and various RAID levels as its chunklets are used for multiple volumes.

InServ also provides snapshots similar to those created by the WAFL file system. The format of InServ snapshots can be read–write as well as read-only, allowing multiple hosts to mount copies of a given file system without needing their own copies of the entire file system. Any changes a host makes in its own copy are copy-on-write and so are not reflected in the other copies.

A further innovation is **utility storage**. Some file systems do not expand or shrink. On these systems, the original size is the only size, and any change requires copying data. An administrator can configure InServ to provide a host with a large amount of logical storage that initially occupies only a small amount of physical storage. As the host starts using the storage, unused drives are allocated to the host, up to the original logical level. The host thus can believe that it has a large fixed storage space, create its file systems there, and so on. Drives can be added to or removed from the file system by InServ without the file system's noticing the change. This feature can reduce the number of drives needed by hosts, or at least delay the purchase of drives until they are really needed.

0 + 1 and 1 + 0 are used where both performance and reliability are important —for example, for small databases. Due to RAID 1's high space overhead, RAID 5 is often preferred for storing moderate volumes of data. RAID 6 and multidimensional RAID 6 are the most common formats in storage arrays. They offer good performance and protection without large space overhead.

RAID system designers and administrators of storage have to make several other decisions as well. For example, how many drives should be in a given RAID set? How many bits should be protected by each parity bit? If more drives are in an array, data-transfer rates are higher, but the system is more expensive. If more bits are protected by a parity bit, the space overhead due to parity bits is lower, but the chance that a second drive will fail before the first failed drive is repaired is greater, and that will result in data loss.

11.8.5 Extensions

The concepts of RAID have been generalized to other storage devices, including arrays of tapes, and even to the broadcast of data over wireless systems. When applied to arrays of tapes, RAID structures are able to recover data even if one of the tapes in an array is damaged. When applied to broadcast of data, a block of data is split into short units and is broadcast along with a parity unit. If one of the units is not received for any reason, it can be reconstructed from the other

units. Commonly, tape-drive robots containing multiple tape drives will stripe data across all the drives to increase throughput and decrease backup time.

11.8.6 Problems with RAID

Unfortunately, RAID does not always assure that data are available for the operating system and its users. A pointer to a file could be wrong, for example, or pointers within the file structure could be wrong. Incomplete writes (called "torn writes"), if not properly recovered, could result in corrupt data. Some other process could accidentally write over a file system's structures, too. RAID protects against physical media errors, but not other hardware and software errors. A failure of the hardware RAID controller, or a bug in the software RAID code, could result in total data loss. As large as is the landscape of software and hardware bugs, that is how numerous are the potential perils for data on a system.

The **Solaris ZFS** file system takes an innovative approach to solving these problems through the use of checksums. ZFS maintains internal checksums of all blocks, including data and metadata. These checksums are not kept with the block that is being checksummed. Rather, they are stored with the pointer to that block. (See Figure 11.17.) Consider an **inode**—a data structure for storing file system metadata—with pointers to its data. Within the inode is the checksum of each block of data. If there is a problem with the data, the checksum will be incorrect, and the file system will know about it. If the data are mirrored, and there is a block with a correct checksum and one with an incorrect checksum, ZFS will automatically update the bad block with the good one. Similarly, the directory entry that points to the inode has a checksum for the inode. Any problem in the inode is detected when the directory is accessed. This checksumming takes places throughout all ZFS structures, providing a much higher level of consistency, error detection, and error cor-

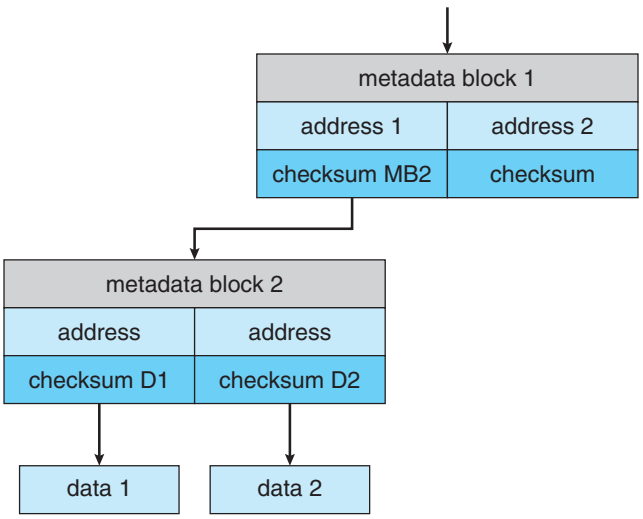


Figure 11.17 ZFS checksums all metadata and data.

rection than is found in RAID drive sets or standard file systems. The extra overhead that is created by the checksum calculation and extra block read-modify-write cycles is not noticeable because the overall performance of ZFS is very fast. (A similar checksum feature is found in the Linux BTRFS file system. See https://btrfs.wiki.kernel.org/index.php/Btrfs_design.)

Another issue with most RAID implementations is lack of flexibility. Consider a storage array with twenty drives divided into four sets of five drives. Each set of five drives is a RAID level 5 set. As a result, there are four separate volumes, each holding a file system. But what if one file system is too large to fit on a five-drive RAID level 5 set? And what if another file system needs very little space? If such factors are known ahead of time, then the drives and volumes can be properly allocated. Very frequently, however, drive use and requirements change over time.

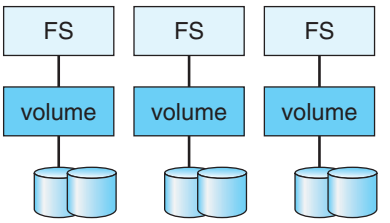
Even if the storage array allowed the entire set of twenty drives to be created as one large RAID set, other issues could arise. Several volumes of various sizes could be built on the set. But some volume managers do not allow us to change a volume's size. In that case, we would be left with the same issue described above—mismatched file-system sizes. Some volume managers allow size changes, but some file systems do not allow for file-system growth or shrinkage. The volumes could change sizes, but the file systems would need to be recreated to take advantage of those changes.

ZFS combines file-system management and volume management into a unit providing greater functionality than the traditional separation of those functions allows. Drives, or partitions of drives, are gathered together via RAID sets into **pools** of storage. A pool can hold one or more ZFS file systems. The entire pool's free space is available to all file systems within that pool. ZFS uses the memory model of `malloc()` and `free()` to allocate and release storage for each file system as blocks are used and freed within the file system. As a result, there are no artificial limits on storage use and no need to relocate file systems between volumes or resize volumes. ZFS provides quotas to limit the size of a file system and reservations to assure that a file system can grow by a specified amount, but those variables can be changed by the file-system owner at any time. Other systems like Linux have volume managers that allow the logical joining of multiple disks to create larger-than-disk volumes to hold large file systems. Figure 11.18(a) depicts traditional volumes and file systems, and Figure 11.18(b) shows the ZFS model.

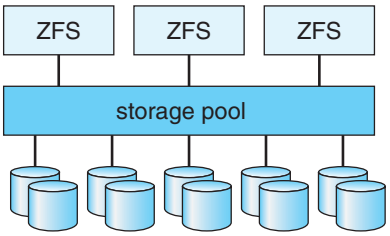
11.8.7 Object Storage

General-purpose computers typically use file systems to store content for users. Another approach to data storage is to start with a storage pool and place objects in that pool. This approach differs from file systems in that there is no way to navigate the pool and find those objects. Thus, rather than being user-oriented, object storage is computer-oriented, designed to be used by programs. A typical sequence is:

1. Create an object within the storage pool, and receive an object ID.
2. Access the object when needed via the object ID.
3. Delete the object via the object ID.



(a) Traditional volumes and file systems.



(b) ZFS and pooled storage.

Figure 11.18 Traditional volumes and file systems compared with the ZFS model.

Object storage management software, such as the **Hadoop fil system** (**HDFS**) and **Ceph**, determines where to store the objects and manages object protection. Typically, this occurs on commodity hardware rather than RAID arrays. For example, HDFS can store *N* copies of an object on *N* different computers. This approach can be lower in cost than storage arrays and can provide fast access to that object (at least on those *N* systems). All systems in a Hadoop cluster can access the object, but only systems that have a copy have fast access via the copy. Computations on the data occur on those systems, with results sent across the network, for example, only to the systems requesting them. Other systems need network connectivity to read and write to the object. Therefore, object storage is usually used for bulk storage, not high-speed random access. Object storage has the advantage of **horizontal scalability**. That is, whereas a storage array has a fixed maximum capacity, to add capacity to an object store, we simply add more computers with internal disks or attached external disks and add them to the pool. Object storage pools can be petabytes in size.

Another key feature of object storage is that each object is self-describing, including description of its contents. In fact, object storage is also known as **content-addressable storage**, because objects can be retrieved based on their contents. There is no set format for the contents, so what the system stores is **unstructured data**.

While object storage is not common on general-purpose computers, huge amounts of data are stored in object stores, including Google's Internet search contents, Dropbox contents, Spotify's songs, and Facebook photos. Cloud computing (such as Amazon AWS) generally uses object stores (in Amazon S3) to hold file systems as well as data objects for customer applications running on cloud computers.

For the history of object stores see http://www.theregister.co.uk/2016/07/15/the_history_boys_cas_and_object_storage_map.

11.9 Summary

- Hard disk drives and nonvolatile memory devices are the major secondary storage I/O units on most computers. Modern secondary storage is structured as large one-dimensional arrays of logical blocks.
- Drives of either type may be attached to a computer system in one of three ways: (1) through the local I/O ports on the host computer, (2) directly connected to motherboards, or (3) through a communications network or storage network connection.
- Requests for secondary storage I/O are generated by the file system and by the virtual memory system. Each request specifies the address on the device to be referenced in the form of a logical block number.
- Disk-scheduling algorithms can improve the effective bandwidth of HDDs, the average response time, and the variance in response time. Algorithms such as SCAN and C-SCAN are designed to make such improvements through strategies for disk-queue ordering. Performance of disk-scheduling algorithms can vary greatly on hard disks. In contrast, because solid-state disks have no moving parts, performance varies little among scheduling algorithms, and quite often a simple FCFS strategy is used.
- Data storage and transmission are complex and frequently result in errors. Error detection attempts to spot such problems to alert the system for corrective action and to avoid error propagation. Error correction can detect and repair problems, depending on the amount of correction data available and the amount of data that was corrupted.
- Storage devices are partitioned into one or more chunks of space. Each partition can hold a volume or be part of a multidevice volume. File systems are created in volumes.
- The operating system manages the storage device's blocks. New devices typically come pre-formatted. The device is partitioned, file systems are created, and boot blocks are allocated to store the system's bootstrap program if the device will contain an operating system. Finally, when a block or page is corrupted, the system must have a way to lock out that block or to replace it logically with a spare.
- An efficient swap space is a key to good performance in some systems. Some systems dedicate a raw partition to swap space, and others use a file within the file system instead. Still other systems allow the user or system administrator to make the decision by providing both options.
- Because of the amount of storage required on large systems, and because storage devices fail in various ways, secondary storage devices are frequently made redundant via RAID algorithms. These algorithms allow more than one drive to be used for a given operation and allow continued

operation and even automatic recovery in the face of a drive failure. RAID algorithms are organized into different levels; each level provides some combination of reliability and high transfer rates.

- Object storage is used for big data problems such as indexing the Internet and cloud photo storage. Objects are self-defining collections of data, addressed by object ID rather than file name. Typically it uses replication for data protection, computes based on the data on systems where a copy of the data exists, and is horizontally scalable for vast capacity and easy expansion.

Practice Exercises

11.1 Is disk scheduling, other than FCFS scheduling, useful in a single-user environment? Explain your answer.

11.2 Explain why SSTF scheduling tends to favor middle cylinders over the innermost and outermost cylinders.

11.3 Why is rotational latency usually not considered in disk scheduling? How would you modify SSTF, SCAN, and C-SCAN to include latency optimization?

11.4 Why is it important to balance file-system I/O among the disks and controllers on a system in a multitasking environment?

11.5 What are the tradeoffs involved in rereading code pages from the file system versus using swap space to store them?

11.6 Is there any way to implement truly stable storage? Explain your answer.

11.7 It is sometimes said that tape is a sequential-access medium, whereas a hard disk is a random-access medium. In fact, the suitability of a storage device for random access depends on the transfer size. The term ***streaming transfer rate*** denotes the rate for a data transfer that is underway, excluding the effect of access latency. In contrast, the ***effective transfer rate*** is the ratio of total bytes to total seconds, including overhead time such as access latency.

Suppose we have a computer with the following characteristics: the level-2 cache has an access latency of 8 nanoseconds and a streaming transfer rate of 800 megabytes per second, the main memory has an access latency of 60 nanoseconds and a streaming transfer rate of 80 megabytes per second, the hard disk has an access latency of 15 milliseconds and a streaming transfer rate of 5 megabytes per second, and a tape drive has an access latency of 60 seconds and a streaming transfer rate of 2 megabytes per second.

a. Random access causes the effective transfer rate of a device to decrease, because no data are transferred during the access time. For the disk described, what is the effective transfer rate if an

average access is followed by a streaming transfer of (1) 512 bytes, (2) 8 kilobytes, (3) 1 megabyte, and (4) 16 megabytes?

b. The utilization of a device is the ratio of effective transfer rate to streaming transfer rate. Calculate the utilization of the disk drive for each of the four transfer sizes given in part a.

c. Suppose that a utilization of 25 percent (or higher) is considered acceptable. Using the performance figures given, compute the smallest transfer size for a disk that gives acceptable utilization.

d. Complete the following sentence: A disk is a random-access device for transfers larger than _____ bytes and is a sequential-access device for smaller transfers.

e. Compute the minimum transfer sizes that give acceptable utilization for cache, memory, and tape.

f. When is a tape a random-access device, and when is it a sequential-access device?

11.8 Could a RAID level 1 organization achieve better performance for read requests than a RAID level 0 organization (with nonredundant striping of data)? If so, how?

11.9 Give three reasons to use HDDs as secondary storage.

11.10 Give three reasons to use NVM devices as secondary storage.

Further Reading

[Services (2012)] provides an overview of data storage in a variety of modern computing environments. Discussions of redundant arrays of independent disks (RAIDs) are presented by [Patterson et al. (1988)]. [Kim et al. (2009)] discuss disk-scheduling algorithms for SSDs. Object-based storage is described by [Mesnier et al. (2003)].

[Russinovich et al. (2017)], [McDougall and Mauro (2007)], and [Love (2010)] discuss file-system details in Windows, Solaris, and Linux, respectively.

Storage devices are continuously evolving, with goals of increasing performance, increasing capacity, or both. For one direction in capacity improvement see http://www.tomsitpro.com/articles/shingled-magnetic-recoding-smr-101-basics,2-933.html).

RedHat (and other) Linux distributions have multiple, selectable disk scheduling algorithms. For details see https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Performance_Tuning_Guide/index.html.

Learn more about the default Linux bootstrap loader at https://www.gnu.org/software/grub/manual/grub.html/.

A relatively new file system, BTRFS, is detailed in https://btrfs.wiki.kernel.org/index.php/Btrfs_design.

For the history of object stores see http://www.theregister.co.uk/2016/07/15/the_history_boys_cas_and_object_storage_map.

Bibliography

[Kim et al. (2009)] J. Kim, Y. Oh, E. Kim, J. C. D. Lee, and S. Noh, "Disk Schedulers for Solid State Drivers", *Proceedings of the seventh ACM international conference on Embedded software* (2009), pages 295–304.

[Love (2010)] R. Love, *Linux Kernel Development*, Third Edition, Developer's Library (2010).

[McDougall and Mauro (2007)] R. McDougall and J. Mauro, *Solaris Internals*, Second Edition, Prentice Hall (2007).

[Mesnier et al. (2003)] M. Mesnier, G. Ganger, and E. Ridel, "Object-based storage", *IEEE Communications Magazine*, Volume 41, Number 8 (2003), pages 84–99.

[Patterson et al. (1988)] D. A. Patterson, G. Gibson, and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", *Proceedings of the ACM SIGMOD International Conference on the Management of Data* (1988), pages 109–116.

[Russinovich et al. (2017)] M. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals–Part 1*, Seventh Edition, Microsoft Press (2017).

[Services (2012)] E. E. Services, *Information Storage and Management: Storing, Managing, and Protecting Digital Information in Classic, Virtualized, and Cloud Environments*, Wiley (2012).

Chapter 11 Exercises

11.11 None of the disk-scheduling disciplines, except FCFS, is truly fair (starvation may occur).

a. Explain why this assertion is true.

b. Describe a way to modify algorithms such as SCAN to ensure fairness.

c. Explain why fairness is an important goal in a multi-user systems.

d. Give three or more examples of circumstances in which it is important that the operating system be unfair in serving I/O requests.

11.12 Explain why NVM devices often use an FCFS disk-scheduling algorithm.

11.13 Suppose that a disk drive has 5,000 cylinders, numbered 0 to 4,999. The drive is currently serving a request at cylinder 2,150, and the previous request was at cylinder 1,805. The queue of pending requests, in FIFO order, is:

2,069; 1,212; 2,296; 2,800; 544; 1,618; 356; 1,523; 4,965; 3,681

Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests for each of the following disk-scheduling algorithms?

a. FCFS

b. SCAN

c. C-SCAN

11.14 Elementary physics states that when an object is subjected to a constant acceleration a, the relationship between distance d and time t is given by $d = \frac{1}{2}at^2$. Suppose that, during a seek, the disk in Exercise 11.14 accelerates the disk arm at a constant rate for the first half of the seek, then decelerates the disk arm at the same rate for the second half of the seek. Assume that the disk can perform a seek to an adjacent cylinder in 1 millisecond and a full-stroke seek over all 5,000 cylinders in 18 milliseconds.

a. The distance of a seek is the number of cylinders over which the head moves. Explain why the seek time is proportional to the square root of the seek distance.

b. Write an equation for the seek time as a function of the seek distance. This equation should be of the form $t = x + y\sqrt{L}$, where t is the time in milliseconds and L is the seek distance in cylinders.

c. Calculate the total seek time for each of the schedules in Exercise 11.14. Determine which schedule is the fastest (has the smallest total seek time).

d. The **percentage speedup** is the time saved divided by the original time. What is the percentage speedup of the fastest schedule over FCFS?

11.15 Suppose that the disk in Exercise 11.15 rotates at 7,200 RPM.

a. What is the average rotational latency of this disk drive?

b. What seek distance can be covered in the time that you found for part a?

11.16 Compare and contrast HDDs and NVM devices. What are the best applications for each type?

11.17 Describe some advantages and disadvantages of using NVM devices as a caching tier and as a disk-drive replacement compared with using only HDDs.

11.18 Compare the performance of C-SCAN and SCAN scheduling, assuming a uniform distribution of requests. Consider the average response time (the time between the arrival of a request and the completion of that request's service), the variation in response time, and the effective bandwidth. How does performance depend on the relative sizes of seek time and rotational latency?

11.19 Requests are not usually uniformly distributed. For example, we can expect a cylinder containing the file-system metadata to be accessed more frequently than a cylinder containing only files. Suppose you know that 50 percent of the requests are for a small, fixed number of cylinders.

a. Would any of the scheduling algorithms discussed in this chapter be particularly good for this case? Explain your answer.

b. Propose a disk-scheduling algorithm that gives even better performance by taking advantage of this "hot spot" on the disk.

11.20 Consider a RAID level 5 organization comprising five disks, with the parity for sets of four blocks on four disks stored on the fifth disk. How many blocks are accessed in order to perform the following?

a. A write of one block of data

b. A write of seven continuous blocks of data

11.21 Compare the throughput achieved by a RAID level 5 organization with that achieved by a RAID level 1 organization for the following:

a. Read operations on single blocks

b. Read operations on multiple contiguous blocks

11.22 Compare the performance of write operations achieved by a RAID level 5 organization with that achieved by a RAID level 1 organization.

11.23 Assume that you have a mixed configuration comprising disks organized as RAID level 1 and RAID level 5 disks. Assume that the system has flexibility in deciding which disk organization to use for storing a

particular file. Which files should be stored in the RAID level 1 disks and which in the RAID level 5 disks in order to optimize performance?

11.24 The reliability of a storage device is typically described in terms of mean time between failures (MTBF). Although this quantity is called a "time," the MTBF actually is measured in drive-hours per failure.

a. If a system contains 1,000 disk drives, each of which has a 750,000-hour MTBF, which of the following best describes how often a drive failure will occur in that disk farm: once per thousand years, once per century, once per decade, once per year, once per month, once per week, once per day, once per hour, once per minute, or once per second?

b. Mortality statistics indicate that, on the average, a U.S. resident has about 1 chance in 1,000 of dying between the ages of 20 and 21. Deduce the MTBF hours for 20-year-olds. Convert this figure from hours to years. What does this MTBF tell you about the expected lifetime of a 20-year-old?

c. The manufacturer guarantees a 1-million-hour MTBF for a certain model of disk drive. What can you conclude about the number of years for which one of these drives is under warranty?

11.25 Discuss the relative advantages and disadvantages of sector sparing and sector slipping.

11.26 Discuss the reasons why the operating system might require accurate information on how blocks are stored on a disk. How could the operating system improve file-system performance with this knowledge?

Programming Problems

11.27 Write a program that implements the following disk-scheduling algorithms:

a. FCFS

b. SCAN

c. C-SCAN

Your program will service a disk with 5,000 cylinders numbered 0 to 4,999. The program will generate a random series of 1,000 cylinder requests and service them according to each of the algorithms listed above. The program will be passed the initial position of the disk head (as a parameter on the command line) and report the total amount of head movement required by each algorithm.