Storing Files

CHAPTER 5

ABOUT THIS CHAPTER

723025

In this chapter, we begin by addressing the aftermath of a security incident and follow this to the problem of file storage. Specifically, we examine the following:

- Incident response and recovering from an attack
- Fundamentals of evidence collection and preservation
- Basics of hard drives and other large-capacity devices
- Hard drive formatting
- File storage on hard drives, flash drives, and other devices
- Features of major file systems used in operating systems and for removable storage

5.1 Incident Response and Attack

After the Trojan horse incident in Section 4.4.3, Bob decided to store working files on small, removable USB flash drives. At first he used the first drives he could find, and gave one copy to Tina. A few days later, he purchased two distinctive-looking drives. He carefully marked them to avoid confusing the confidential USB drives with others in the office.

After he copied all files to the new drives, he deleted the files from the old drives and "emptied the trash."

A few days later, Eve visited Bob's office. She asked Tina to lend her a spare USB drive. Should Tina give Eve one of the erased drives?

When we write information to a file, the file system copies that information onto blocks of storage on our hard drive or flash drive. The system then saves information about the blocks' locations in the file's folder.

When we delete a file, most operating systems move it to a special folder called "trash" or "recycle." If we want to recover the file, we simply move it back to a permanent folder. After emptying the trash, the file system deletes the file from the folder, and

then frees its data blocks to be reused the next time we write to the drive. Once we write new data over the old file's data blocks, it becomes almost impossible to recover the file.

However, drives often have a lot of free space. Recently freed blocks might not be reused for hours or even weeks. An ***undelete*** program tries to reconstruct a file by locating the freed data blocks before they are reused. This act also is called ***file scavenging***. Even though Bob and Tina "emptied the trash" each time they deleted the file, they didn't really remove the information from the hard drive.

Incidents and Damage

Although security problems might feel like "attacks" personally, many don't actually cause damage. Without a loss, some might argue that there really hasn't been an "attack," per se. For that reason, it's often better to refer to these events as *incidents*, not attacks. For example, a "bicycle incident" might unhook a bicycle lock from a fence. The bicycle itself isn't damaged nor is the lock, but now it's vulnerable to theft. We can still ride, but we now have a security problem.

Is it a "security incident" if Tina lends a USB drive to Eve? Yes: We make it a security incident when we ask the question. Bob is the proprietor of Bob's Bookkeeping, and ultimately it's his responsibility if a client's data leaks. He decides whether deleting a file is sufficient security for his customers. If the leak occurs and an investigator traces it back to Bob, he could argue that he made a reasonable effort to protect the data. Most people assume that deletion is as much as they need to do.

If Bob wants to prevent this potential leak, he should never lend, share, or throw away a drive, whether a hard drive or a USB drive. This isn't always practical. Some people try to erase a drive by "formatting" it (see Section 5.3), but this does not necessarily erase the data, either. The most effective solution is to ***overwrite*** the data to be deleted. Apple's OS-X calls this the "Secure Empty Trash" function: the system writes new data over a file's data blocks and then deletes the file. Section 7.3 examines this process further. Overwriting makes data recovery impractical in most cases. A very few high-security applications demand that drives be physically destroyed to protect their contents from leaking (see Chapter 17).

Compromised Systems

Just as a bicycle incident may render a bike vulnerable without damaging it, an incident might render Bob's computer more vulnerable to attack. For example, a suitemate might disable access permissions on system files. Bob's computer now has been *compromised*. In Victorian times, people spoke of "compromising one's reputation." This indicated that an incident had rendered the reputation suspect, even without proof of misbehavior. The "compromise" indicates that the computer is no longer trustworthy, because it *may* have been subverted.

We recover from attacks, incidents, and compromises, by taking steps to recover. The recovery process often is called ***remediation***.

5.1.1 The Aftermath of an Incident

Different incidents demand different forms of recovery. The aftermath may include one or more of the following tasks:

- Identify shortcomings in our risk assessment, security requirements, or implementation to reduce the impact of future incidents.
- Repair any problems caused by the attack. If a Trojan program infests your computer, the repair includes removal of the Trojan.
- If the incident is caused by someone's malicious act and we can hold that person accountable, then we need to collect evidence that clearly ties the person to the incident.
- If someone is using our computer to violate laws, then we need to preserve the evidence so that a prosecutor may use it as evidence in a trial.

Digital Forensics

We apply ***digital forensics*** when we need to collect evidence from computers and other digital storage devices. Forensic techniques recover, preserve, and analyze information from a computer system to show what its users were doing.

When we take a serious action, like firing an employee or pursuing legal measures against an attacker, we must take special care in collecting evidence. If the evidence must support legal action, then it must be admissible in court.

Questions of gathering evidence are fundamental to forensics:

- What data should we try to collect before a security incident that we can use as evidence after one occurs?
- What data are we allowed to collect and use as evidence from an individual's computer?
- What data can we retrieve from persistent computer memories, like hard drives and USB flash drives?

The answers depend heavily on the legal system that applies to the computers, their owners, and the perpetrators of the attack.

Fault and Due Diligence

If harm comes from an incident, there is often a legal or moral obligation to hold someone responsible. The attacker obviously should be held responsible and, if appropriate, should repair the damage or provide restitution. However, if the attack took advantage of carelessness, then the careless parties also may be responsible. This is an established legal concept in many communities.

If someone retrieved their files regardless of the protections they used, are Bob and Tina somehow responsible? The question revolves around whether they exercised ***due diligence***; in other words, they must have taken reasonable steps to protect the files. If

they could have used stronger measures and failed to, then perhaps they bear responsibility for the failure. If, on the other hand, they used the customary security measures and the community accepts those measures as adequate, then they showed due diligence. The instructor could justifiably hold them responsible if they failed to use the same security measures as others.

5.1.2 Legal Disputes

A security incident may be part of a legal dispute. If it is part of a legal dispute, then it is subject to the local legal system. If the dispute is being resolved in the United States or under a similar legal system, then we may need evidence to show what happened and, ideally, identify the people responsible.

When we present the matter to an official, whether a police officer, prosecutor, or judge, our evidence must meet local legal requirements. First, the evidence must be relevant and convincing. Second, those who review the evidence must be confident that it actually illustrates the incident under investigation. The evidence must be unchanged since the incident occurred.

Finally, we may only use information that we have legally obtained. The specific requirements vary under different legal theories and traditions. Here, we will focus on U.S. legal requirements. However, even the U.S. rules are subject to change, because this is a new area of law.

Legal Systems

Worldwide, legal experts classify legal systems into three categories:

1. Civil law—based on legislative enactments. Roman and Napoleonic laws are examples of this.
2. Common law—based on judicial decisions. English Common Law and the U.S. legal system follow this tradition.
3. Religious law—based on religious systems or documents. Jewish, Islamic, and Christian canon law systems are examples of this.

In practice, legal systems often reflect a blend of these systems. Traditionally, people speak of the U.S. and English legal systems as arising from common law, but today both are heavily influenced by new laws passed by legislatures.

Resolving a Legal Dispute

Not all incidents rise to the level of legal dispute. In the United States and in countries with similar systems, problems arising from an incident may be resolved in several ways:

- ***Private action***, in which one party acts against another, based on a shared relationship. For example, an employer might discipline an employee, or a school might discipline a student, based on informal evidence that might not be admissible in court.

- *Mediation*, in which the parties rely on a third party, a mediator, to help negotiate a settlement. The mediator is not bound by particular rules of evidence and may consider evidence that is not admissible by a court.
- *Civil complaint*, in which one party files a lawsuit against another. Such matters still may be resolved privately, possibly through negotiation. If the parties go to court, then legal requirements for digital evidence must be followed precisely.
- *Criminal complaint*, in which a person is charged with breaking particular laws. The complaint sets out the facts of the matter and presents probable cause for accusing a particular person for the crime. A criminal complaint may be made by the police, a district attorney, or any interested party. If there is no plea bargain, the trial goes to court, at which point the digital evidence must fulfill all legal requirements.

Although a dispute may begin as a private action, it could escalate to mediation or to a civil case. In some cases, the incident could become a criminal complaint. The safest strategy, if legal action seems likely, is to collect evidence in a manner that preserves its admissibility in a civil or criminal court action.

A typical forensics investigator does not focus on computer or information evidence alone. The investigator will look for all kinds of evidence related to the incident under investigation: related equipment, papers, articles of clothing, latent fingerprints, and DNA, that may associate suspects with the incident. A typical investigator will follow rules outlined later to collect and secure computer data for later analysis. Investigators do not usually try to perform detailed investigations on site. Such an investigation poses the real risk of disturbing the evidence and making it impossible to distinguish between the suspect's actions and the investigator's actions.

5.2 Digital Evidence

We must collect evidence before we can use it in any dispute. If we want to use this evidence in a legal proceeding, the evidence must be ***admissible***; in other words, it must meet the legal rules and standards for evidence.

We may collect evidence through surveillance or seizure. In surveillance, we watch the behavior of the threat and keep a log of activities. In seizure, we take possession of equipment involved in the dispute. The requirements for surveillance and seizure vary according to whether we act as members of law enforcement or as a private party involved in the incident.

The Fourth Amendment

Under U.S. law, surveillance and seizure are restricted by the Fourth Amendment of the Bill of Rights:

> The right of the people to be secure in their persons, houses, papers, and effects, against unreasonable searches and seizures, shall not be violated, and no Warrants

> shall issue, but upon probable cause, supported by Oath or affirmation, and particularly describing the place to be searched, and the persons or things to be seized.

The Fourth Amendment is, in turn, based on English common law, which treats a person's home as a "castle" intended to ensure personal safety and, if desired, seclusion. Government agents, including police officers, may intrude on one's home, but only after presenting a justifiable reason to a different official, such as a justice of the peace.

Legal Concepts

These legal principles have evolved over the centuries and even over recent decades. They promise to change further as they are applied to electronic devices. While the specific rules may change as new cases are tried, the rules will probably preserve certain concepts:

- Who performs the search. There are different rules for private citizens, for police officers, and for others performing an investigation on behalf of the government.
- Private searches. Private individuals may perform searches without warrants on equipment in their possession. Evidence they find may be admissible in court. There are some restrictions when someone else owns or routinely uses the equipment.
- Reasonable expectation of privacy. We can't arbitrarily search areas where the users expect their privacy to be protected. For example, neither police nor employers can arbitrarily wiretap telephone calls.
- Consent to search. We can eliminate the expectation of privacy in some cases, especially in the workplace. For example, we don't expect privacy when we call a service number and hear: "This call may be monitored for quality assurance purposes." Likewise, a computer's owner can display a warning that computer use may be monitored and recorded.
- Business records. If we collect data in activity logs as a routine part of doing business, then those records often are admissible in court. If we set up special logs to track an intruder, those are not business records and are not necessarily admissible as evidence.

The legal issues can be quite subtle. If we need to collect information for use in court, we must consult an expert first. Depending on the laws and recent court decisions, we may need a warrant, a court order, or a subpoena to collect information—or we might not need any court document at all, especially if we are private individuals investigating our own equipment.

Despite the 4th Amendment, government organizations may collect a lot of information that is inadmissible in a court case. In June 2013, newspapers reported on vast NSA surveillance programs that targeted all telephone and Internet traffic worldwide, including messages by residents and citizens of the United States. These revelations were based on a cache of secret documents provided by Edward Snowden, a former employee of an NSA contractor.

While the 4th Amendment wouldn't allow NSA surveillance data to be used in court, government investigators and prosecutors have developed strategies to employ such

surveillance. One technique, called "parallel construction," uses inadmissible data to identify lines of investigation that yield admissible data. For example, inadmissible surveillance data might identify a particular truck involved in illegal activity. The surveillance data prompts a suggestion that local police perform a routine traffic stop as a pretext to seek evidence of illegal activity. If the police have probable cause to investigate further, the resulting evidence is admissible as long as the traffic stop itself is considered legal. Moreover, if the surveillance data is based on classified intelligence information (see Chapter 17), a prosecutor may be able to shield its role from the defendant by telling the court that such a disclosure could damage national security.

5.2.1 Collecting Legal Evidence

In a purely private matter, we might be able to log in to the attacked computer, extract information from it, and present our discoveries to the attacker. This might be enough to resolve the matter. If, on the other hand, the matter goes to court, our "evidence" might not stand up. If the defendant's attorney can cast doubt on the accuracy or integrity of our evidence, a jury might not believe it even if we're allowed to present it at a trial.

To serve as evidence, we need to show that the attacked computer reflects the actions of the attacker. Whenever we use a computer, our actions make changes to it. If we use the attacked computer itself to investigate the attack, we can't always distinguish between the results of the attack and the results of the investigation. The attacker's attorney might argue that the actions of our investigation produced the evidence. It may be hard to refute that argument.

Collecting Evidence at the Scene

The digital forensic investigation takes place separately from collecting the evidence. We start by collecting the evidence and ensuring its integrity. The actual analysis is a separate step.

Let us look at the process from the point of view of the investigator who must collect evidence for a civil or criminal court case. The investigator's job is to identify the relevant evidence, collect it, and ensure its integrity. The analysis of the digital evidence must take place in a separate step under controlled conditions.

The investigator follows these steps in a single-computer environment:

- Secure the scene and all relevant digital equipment.
- Document the scene.
- Collect digital evidence.

Here is a closer look at the first two steps. The next section examines the third step.

Securing the Scene

Once we have decided we must collect evidence, we first secure the area. We must remove everyone from the scene except the people responsible for collecting the evidence. The regular occupants and residents should leave and allow the investigators to do their work.

Digital evidence is very, very sensitive to being disturbed. Therefore, we don't want anyone tampering with digital devices in the scene. Refuse all assistance with identifying or collecting digital devices.

It is particularly important to leave digital devices turned off if they already are turned off. Every time we turn on a digital device, changes may take place. We need to preserve the device so that it is exactly as it was found at the scene.

If a computer is turned on and displays a destructive process in progress, like a "Format" or "Delete" command, then you may want to unplug the computer immediately. If, however, the computer's display contains relevant evidence, you may need to photograph the display before unplugging. This is a trade-off. In some cases, it's possible to retrieve deleted files, while it's rarely possible to retrieve the display contents after the computer is turned off.

Documenting the Scene

When we document the scene of the incident, we collect information about what is there, where things are, and what condition they are in. We start by describing the geographical location of the scene; what building, floor, room, and so on. We catalog everything in the scene that's relevant: where it is, what it is, and its condition. This becomes our ***evidence log.***

Photography also plays an essential role in documenting the scene. We photograph items and their relative locations. We take special care to photograph any device displays or other conditions that may disappear when we unplug things and collect them for later analysis.

When documenting digital devices, we identify what they are, what identifying marks they have, any serial numbers, and whether they are powered on or not. Photographs should clearly show the contents of any displays or screens.

As a general rule, do not move disk devices that are powered on. Some hard disk drives are sensitive to motion and may fail if we move them while running. We may need to wait and collect serial numbers later in the process, if they are not visible without closer inspection.

5.2.2 Digital Evidence Procedures

Digital items to be collected will vary with the incident, but could include:

- Computers and laptops
- USB drives and other flash memory
- Other portable or removable hard drives
- Cell phones and smartphones
- GPS units

Before we collect a digital device, we need to decide if it is turned on or not. This is, of course, tricky. The main display may be blank or turned off itself while the device is still on. Each device usually poses one of these alternatives:

- *Device might be turned on but its display is blank.* We need to be sure the display power is on and then *carefully* disable any screen power savers, documenting, and photographing the display's contents at each step, then proceed as if it is turned on.

- *Device is obviously turned on.* We again carefully move the mouse to disable any screen saver, and we document and photograph any changes in the display. In most cases, we then simply unplug the computer (or remove a laptop's battery), and proceed as if it is turned off. It takes special training to collect admissible data from a running computer.
- *Device is obviously turned off.* When the device is off, we document all connections and peripherals, then we seal it in an evidence bag and ensure it is on our inventory.

As noted, the best strategy in most cases is to simply unplug a running computer. The shutdown operation in many operating systems may wipe out evidence that a forensics investigation could otherwise recover. In particular, Microsoft Windows is likely to save more information if we simply "pull the plug" than if we perform a clean "shutdown."

Authenticating a Drive

There are important cases, however, where we may lose access to some evidence if we remove the power. For example, if the computer uses drive encryption, then we lose access to the drive when powered off. However, it takes special training to reliably collect evidence from a running computer.

After collecting a USB drive or hard drive as evidence, the next task is to analyze its contents. However, the first question a defense attorney will ask is "How can you prove that the drive is unchanged since collected as evidence?"

We must be able to show that the drive's contents are authentic with the incident. We must be able to prove that our analysis is based on the original contents, and does not reflect side effects of our investigation. To do this, we authenticate the hard drive and any other storage devices we investigate.

We address authentication in two parts. First, we never, ever make changes to the hard drive, or to any other storage device, that we collect as evidence. Instead, we make a copy of the original drive and analyze the copy. Second, we use a utility program to calculate an *integrity check value* on the drive's data. This type of program is a standard feature in software packages for digital forensics.

The check value is a very large integer (10^{48} or larger) that we retain at full precision. The check value uses a special computation, a "one-way hash," that reliably reflects *any* change we make to the drive. We look more closely at the one-way hash in Chapter 6. We recalculate the integrity check value on our copied hard drive. As long as the recalculated value matches that of the original drive, we can present the evidence we recover from our copy.

5.3 Storing Data on a Hard Drive

While Tina and Bob probably don't need to dust their USB drive for fingerprints or identify their culprit in court, they might want to perform a detailed investigation of the USB drive. To do that, they must analyze the file system. Before we look at the file system, we need to understand how hard drives work.

Even though we rely heavily on solid state drives and flash drives today, hard drive storage remains less expensive and is still widely used. Hard drives have heavily influenced how today's file systems work.

These details help us understand what information a file system provides an investigator and what information it might hide. If we can retrieve a file after deleting it, a more clever program might give someone a way to actually hide data without deleting it.

A clever computer hacker, when arranging a hard drive, might set aside space for secret file storage that isn't visible in the normal navigation windows. A really clever programmer might hide small bits of data, like encryption keys, in the "unused" space at the end of a file, or in "unusable" sections of the file system. To understand these tricks, we must understand hard drives and file systems.

Magnetic Recording and Tapes

Hard drives rely on magnetic recording. The easiest way to understand magnetic recording is to look at magnetic tapes. In both cases, the technique works the same:

When we move a magnet past a coil of wire, the coil produces a current.

The power and direction of the current indicates the strength and direction ("north" versus "south") of the magnet. This allows us to read data from a magnetized surface.

The opposite also is true:

We can magnetize a metal surface if it moves past a coil of wire containing a current.

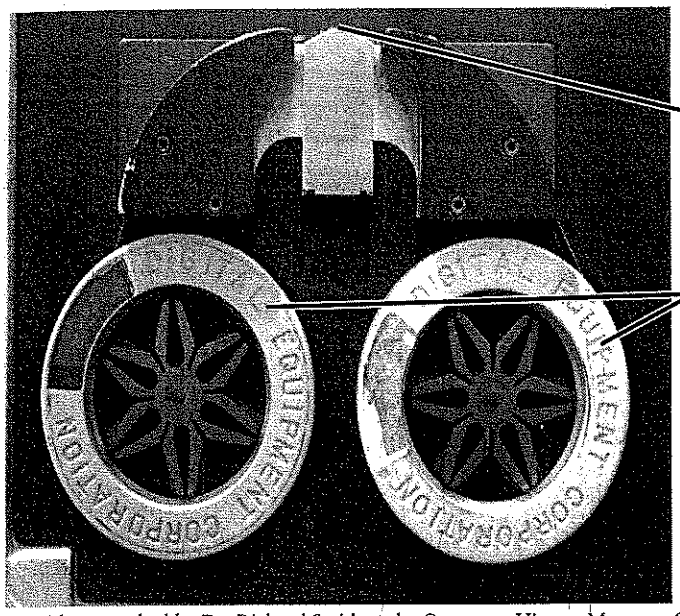
Again, the strength and direction of the magnetized surface reflects the strength and direction ("positive" versus "negative") of the current.

A magnetic tape drive moves tape back and forth between two reels. We read and write data onto the moving tape as a series of magnetized spots. The coil of wire resides in the ***read/write head***, shown at the top of Figure 5.1. As the motor moves the tape, it pulls the tape across the read/write head. When reading, the tape's motion produces a current as data moves across the head.

Magnetic tapes and tape drives were the symbol of computers in the 1950s and 1960s. However, they suffered a fundamental weakness; they were limited to ***sequential access***. In other words, we always encounter the data on the tape "in sequence." We can't get from the start of the tape to the middle without skipping over all of the data in between.

Hard Drive Fundamentals

Hard drives evolved in the 1950s to provide large amounts of low-cost storage for the "giant brains" of that era. Like magnetic tapes, they used magnetic techniques to store and retrieve data. Compared to magnetic tapes, hard drives provided ***random access*** to data, because it could find, retrieve, and rewrite any data on the hard drive in a relatively short time.



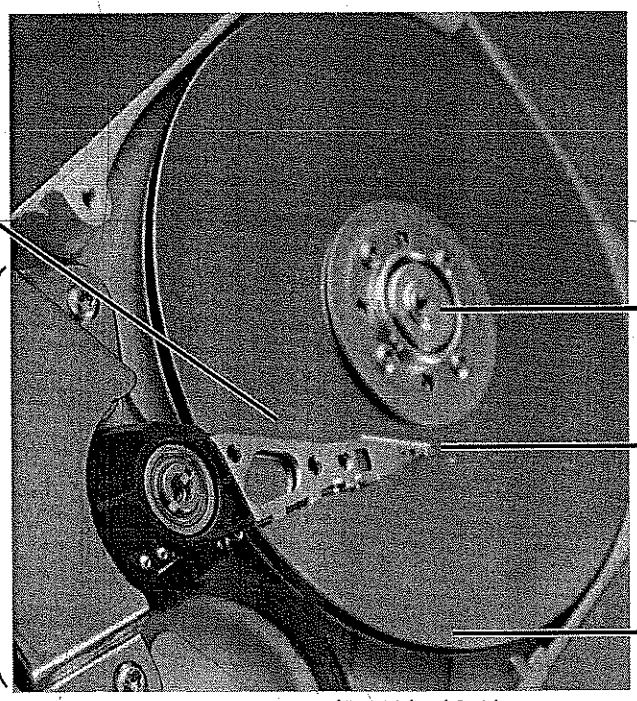
Photographed by Dr. Richard Smith at the Computer History Museum, California

Figure 5.1

A simple magnetic tape drive.

Figure 5.2 shows the inside of a modern hard drive. The read/write head is on the end of a long arm that hovers over the magnetized surface. The drive spins under the head to provide the motion required for reading or writing the magnetic data.

On a hard drive, we record information on the surface of a spinning disk. In fact, *disk drive* is a traditional term for a hard drive. We record the information by magnetizing



Courtesy of Dr. Richard Smith

Figure 5.2

A typical hard drive.

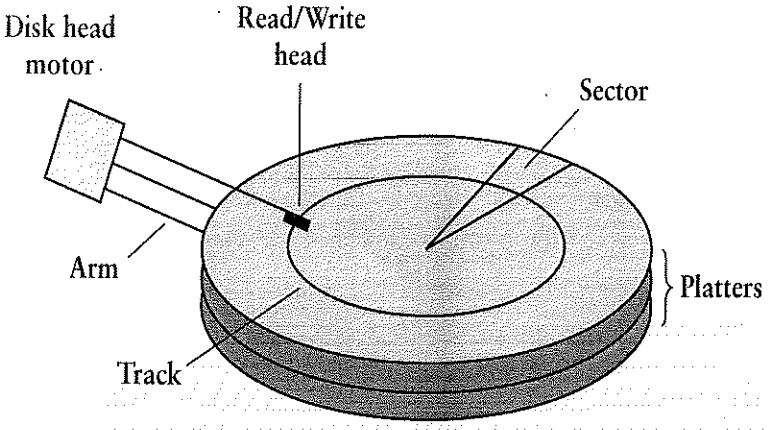


Figure 5.3

Hard drive mechanism.

spots on the surface of the drive's disk *platters*. The platters rotate around the spindle, driven by the moter. A read/write head hovers over each platter's surface, held in place by a mechanical arm.

The hard drive's basic mechanism has not changed since the 1950s, though the speed and capacity has increased dramatically. Like other computing equipment, hard drives have benefited from *Moore's law*. The disk on a typical hard drive today is about $3\frac{3}{4}$ in. (95 mm) in diameter, while some 1960s' hard drives were over 4 feet (1300 mm) in diameter.

Figure 5.3 illustrates the hard drive's basic geometry. We magnetize data in two dimensions on each platter. In one dimension, we move the read/write head inward or outward. When it stops at a particular distance, it inscribes a magnetized ring on the platter. That ring forms a *track*. Each track contains individual blocks of data called *sectors*.

If we wanted to move from the first block to the last block on a magnetic tape, we had to wait while the drive moved the tape past every data block it contained. On a hard drive, we only need to wait for two relatively fast operations. First, the arm containing the read/write head moves from the first track to the last track. Second, we wait while the drive's spin brings the final sector under the read/write head. This was lightning fast compared to magnetic tapes, even in the 1950s.

The simplest hard drives contain a single platter and a single read/write head. We record information on only one side of such platters. Larger drives often contain a second head to record on the other side of the platter. Some drives contain two or more platters, with an additional head for each side. Figure 5.2 shows a hard drive with two platters and four recordable surfaces.

The read/write heads are mounted on a set of arms that move together. When the drive moves the arm to a particular location on the disk surface, each head follows a matching track on its own surface. A very precise motor moves the arm assembly to specific locations on the disk surface; these locations specify the track locations.

A track represents a single ring of sectors recorded on one side of a platter. A *cylinder* represents all tracks recorded by the read/write head when it sits at a particular location.

We get the highest performance out of a hard drive when all of our data resides on a single cylinder. If the data doesn't fit on a single cylinder, we optimize our performance if the data resides on adjacent cylinders.

Dissecting a Hard Drive: First, and most important, *never dissect a working hard drive.* The disk heads must hover slightly above the disk surface, and will "crash" into the surface if it tangles with an obstacle like a dust mote. It is impossible to open a hard drive without making it unusable, unless performed in a clean room by an expert.

Vendors use airtight sealing tape and several small screws to attach the drive cover to the drive body. The cover comes loose once *all* screws are removed and the tape is slit to loosen the drive cover. Note that some screws may be covered by labels.

There will always be at least one label saying "Tampering with this label will void the warranty." This is because hard drives are not designed to be opened. *Never dissect a working hard drive.*

5.3.1 Hard Drive Controller

To read and write data reliably, we must move the head assembly very precisely over the desired track. We also must be sure to read or write the correct sector on the track. Modern hard drives use a special circuit, the ***drive controller***, to operate the head assembly and select the correct sector.

Figure 5.4 illustrates the parts of a controller using a ***hardware block diagram.*** Such diagrams show the major components in a hardware device, and the signals that pass between those components.

A modern controller resides on a circuit board packaged with the rest of the drive hardware. Typically, the hard drive accepts commands from the CPU sent to it across a high-speed bus. For example, a command may direct the drive to read data from one or more sectors. The command identifies the starting sector and the number of bytes to

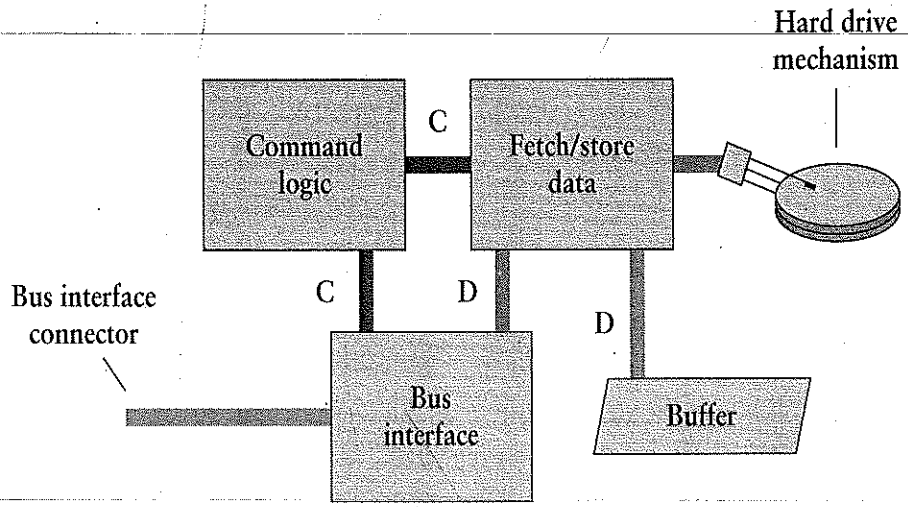


Figure 5.4

Hardware block diagram of a hard drive controller.

read. The controller moves the mechanism to the first sector and retrieves the data from the magnetized surface. As the data arrives from the mechanism, the controller transmits it back across the bus.

A controller typically contains the components shown in Figure 5.4. Connections between components carry control signals (marked "C"), data signals (marked "D"), or both. Here is a description of each component:

- **Bus interface connector**—a socket for connecting the drive to a high-speed bus, like those described in Section 2.1.1. Modern drives have a SATA connector or possibly an ATA or IDE connector.
- **Bus interface**—circuits that convert the commands and data that flow between the bus and the command logic.
- **Command logic**—circuits that convert the commands into a series of operations performed on the hard drive mechanism.
- **Buffer**—a large block of RAM that stores data temporarily on its way to or from the hard drive mechanism. The mechanical operations are time sensitive. The buffer makes it easier to synchronize the bus data transfers with the hard drive's mechanical motions.
- **Fetch/store data**—circuits that directly control the head motor to select tracks, and that retrieve or rewrite individually addressed sectors.

Hard drive speed depends on two variables: rotational speed and head motion. Modern drives in desktop computers may rotate at speeds from 5400 to 10,000 revolutions per minute (RPM), with faster drives being more expensive. Track-to-track head motion is significantly slower. While it may take microseconds to move data to or from a track without moving the head, it may take *milliseconds* to move data if the drive has to move the head between tracks. The data transfer speed is even slower if the heads must move between innermost tracks and outermost tracks.

Modern drive controllers try to automatically correct for bad sectors. The controller detects a possibly bad sector if the sector's cyclic redundancy check (CRC) occasionally fails. When it detects a bad sector, it redirects the sector's address to a different, reliable sector elsewhere on the drive. This takes place automatically and the operating system might never detect the change.

5.3.2 Hard Drive Formatting

If we look at a hard drive before we install an operating system, the drive contains nothing except the numbered sectors. Each sector stores a separate block of data. The hard drive controller uses the sector number as the sector's address.

When we perform a ***low-level format*** on a hard drive, we tell the controller to initialize the raw sectors themselves. Most hard drives use a sector size of 512 bytes, though some devices may have sectors as large as 2048 (2K) bytes. Each sector consists of three parts as shown in Figure 5.5: the ***header***, the data, and the ***check value***.

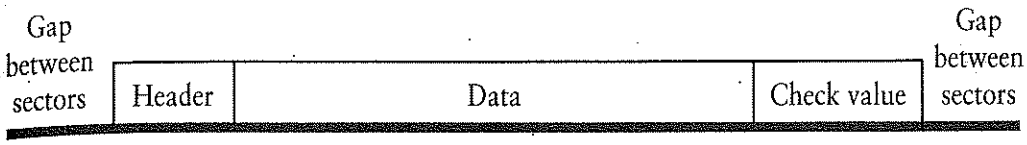


Figure 5.5
Low level format of a hard drive sector.

The header contains the sector's numerical address and other information for the controller. Most drives number sectors consecutively, though older drives used to number them by track. The controller always checks the sector address when reading or writing. Although today's hard drives usually format all sectors identically, some special drives allow variations in the sector format.

The data area, naturally, contains the sector's data. The sector's check value helps the controller detect errors in the sector's data. Magnetic storage is very reliable, but the magnetic properties fail once in a while. We look closer at error detection in Section 5.4.1.

As shown in Figure 5.6, a hard drive is like a huge post office with millions of numbered boxes (the sectors). We find each box by number. We can store or retrieve a fixed amount of data via each box. When we store or retrieve data from one box, we don't affect the contents of other boxes. Unlike post office boxes, though, individual hard drive sectors rarely are large enough to store an entire file. When a file is larger than a sector, it resides in a series of boxes. To read the information, we must retrieve the data from every box in the series.

High-Level Format

The *high-level format* of a hard drive refers to the layout of its file system. Different file systems store particular information in particular places, as described later in this

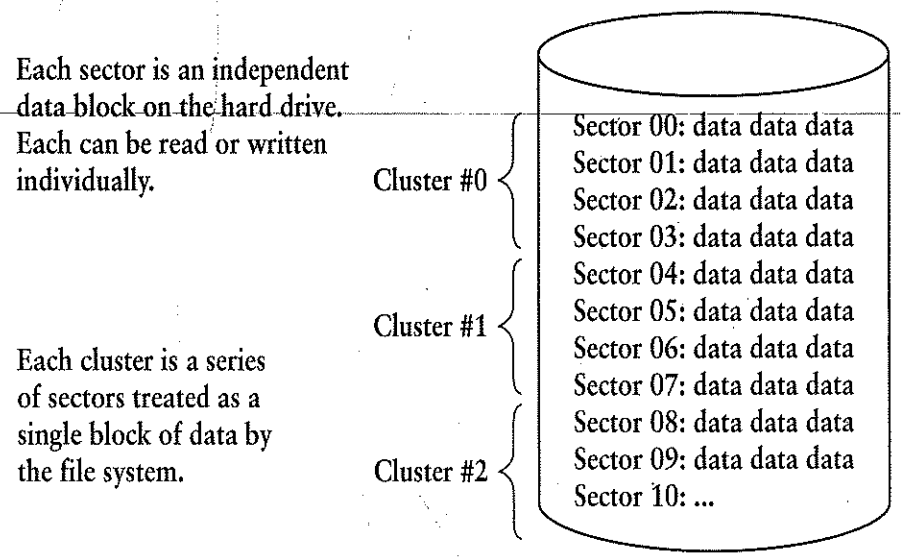


Figure 5.6
Sectors and clusters on a hard drive.

chapter. Typically, the high-level format divides the drive into groups of sectors called *clusters* (Figure 5.6). Each file contains one or more clusters. The file system keeps track of the clusters. When we create a new file or add more data to an existing file, the system locates an unused cluster and adds it to the file.

Fragmentation

A file may consist of several clusters in a row or its clusters may be scattered across the hard drive. The scattering often is called ***fragmentation***. It can slow the system down when we work with those files. This is why people occasionally run a "defragment" utility on their hard drive. The utility moves the clusters in each file so that they are contiguous; this makes it faster to access the files.

Fragmentation also refers to wasted space on the hard drive. Most files contain a "fragment" of wasted space at the very end. If a file ends *exactly* at the end of a cluster, then there is no wasted space. More often, the file ends short of the end of the cluster. The remaining space can't be used in another file, so the space is wasted.

Quick Format

When we perform a "quick format" on a hard drive, we initialize the file system information. This doesn't disturb the existing sector addresses, and it ignores most of the data on the drive. Instead, it recreates all of the basic information for creating files, and gives the drive a single, empty "root" directory. If we had any files on the hard drive before reformatting, the process discards those files.

5.4 Common Drive Concepts

The previous section focused on hard drive technology. Some of the concepts and technology, notably the section on high-level formatting, also applies to flash drives. This section discusses several technologies applied to both hard drives and flash drives. Most of these technologies were developed for hard drives and are now applied to flash drives.

Flash Drives

Mechanically, flash drives and other solid state drives are completely different from hard drives. Flash drives are purely electronic and do not rely on magnetized surfaces or head motions. Instead, electronic circuits store data by trapping charges in a stable form for long periods of time. The name "flash" refers to an early technique of erasing previously stored data. Flash memory is structured in blocks, like hard drives are structured into sectors.

Flash drives use the same high-level formats as hard drives. Most commercial flash drives use the file allocation table (FAT) file format described in Section 5.5. However, drive owners also may apply other formats if allowed by their operating system.

Flash drives also have special control circuitry for converting the read and write operations into reliable changes to the flash storage. Flash controllers don't provide

motor or drive head controls, since there are no moving parts. Otherwise, the controller makes the flash drive operate in the same way as a hard drive.

5.4.1 Error Detection and Correction

Whenever a drive writes data to a sector, the controller calculates a numerical check value on the data. It then writes the check value at the end of the sector. When we read a sector later, the controller repeats the calculation and checks the result against the saved check value. If the results don't match, the controller reports a data error on that sector. This prevents our software from reading incorrect data from the hard drive without realizing it.

Parity Checking

Errors have plagued computer memories since their invention. Magnetic tapes often used a simple technique called *parity checking* to detect errors. The tape drives handled data on a byte-by-byte basis, storing each byte across the width of the tape. The parity check stored an extra "parity bit" with each byte. We calculate parity by looking at the individual bits in a byte. Each bit has a value of either 0 or 1. We count the number of bits containing the value 1.

For example, Figure 5.7 shows the data bits stored in a single byte of a "nine-track tape." The first track contained the parity bit and the remaining contained the eight data bits, one per track. On the left of the figure, the tape contains the numeric value "3" in printable text; its 9-bit code is shown as "1 0011 0011."

The nine-track tape stored data with "odd parity," which means that the parity bit is chosen to yield an odd number of 1 bits. If the remaining eight data bits had already contained an odd number of 1 bits, then the correct parity bit would be 0.

To detect an error in odd parity, we count the 1 bits, including the parity bit. The right side of Figure 5.7 shows the result of a 1-bit tape error that has changed the "3" character to a "7" character.

We detect the error by counting the bits on the tape, including the parity bit. These appear in the lower right of the figure. The character code for "7" contains an odd

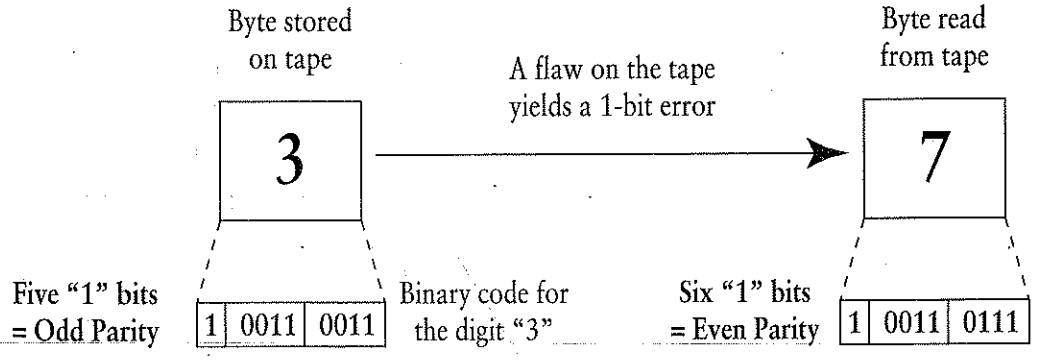


Figure 5.7

Detecting an error using odd parity on a nine-track tape.

number of 1 bits. When we combine this with the parity bit, we have an even number of bits, yielding "even parity." The tape drive detects this as an error when reading the tape, because the byte no longer contains odd parity.

The choice of "odd is correct" and "even is wrong" is arbitrary. Some devices and systems require even parity. It is a design choice.

Unfortunately, errors aren't always limited to individual bits. Some errors may change a series of bits to all 1s or to all 0s, or otherwise affect several bits. We can rely on parity only if the most likely errors affect a single bit; it fails completely if the error changes an even number of bits.

Another problem with parity is that it uses a lot of storage: Over 11 percent of a nine-track tape's storage was used for parity checking. If we take more data into account, we can detect more errors and use less storage for error detection.

Checksums

Hard drives store blocks of data in sectors, so it makes sense to check for errors on a per-sector basis. The simplest approach is to perform a ***checksum***; we use a simple rule to calculate the check value from the sector's data. In Figure 5.8, for example, we have a 4-byte sector of data containing the text "$109."

To calculate the check value, we add together the character codes stored in the sector's data bytes, and discard extra bits that don't fit in an 8-bit byte. We then store the checksum in the final byte of the sector.

If we are working with hard drive sectors, then the hard drive controller calculates the checksum. When writing, it computes the value on the newly written data and writes the

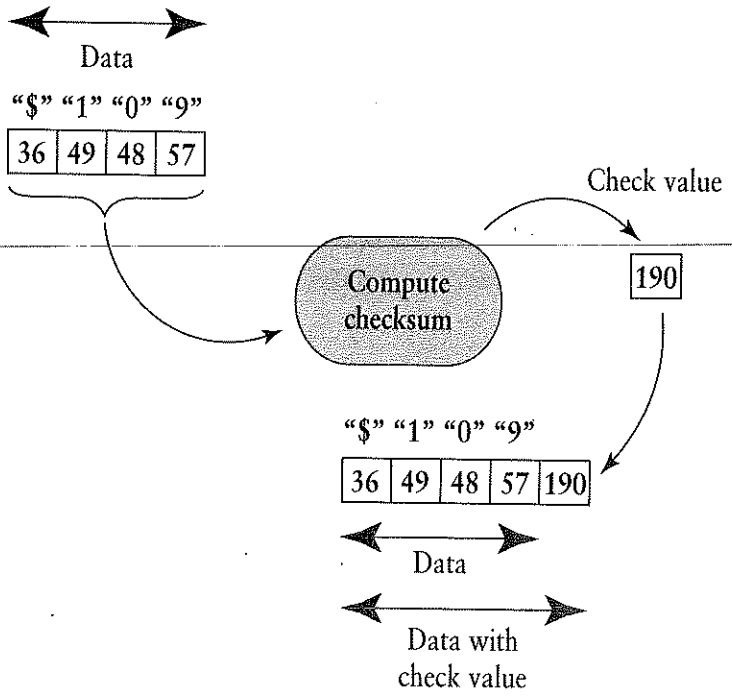


Figure 5.8

Calculating a simple checksum.

result at the end of the sector. When reading, it recalculates the value while reading the sector's data. The result should match the previously stored checksum. Otherwise, one or more bits in the sector have failed.

A checksum field is not limited to 8 bits; we could comfortably collect and store a 16-bit check value with a hard drive sector. Even if we store a 16-bit result, the checksum uses storage far more efficiently than parity. A hard drive sector traditionally contains 512 bytes of storage; a 16-bit checksum uses less than 1 percent of our storage for error detection.

While some older drives used simple checksums, the technique is too simple to detect many errors. For example, a checksum won't detect an error that swaps bytes. The message "$901" has the same check value as the previous correct sector and would not be detected as an error.

Cyclic Redundancy Checks

To improve error detection, more sophisticated drives computed more sophisticated check values using a CRC. A well-designed CRC detects "burst" errors, like a sequence of 0s or 1s, much more reliably than parity checking. CRCs may yield check values of 32 bits or more.

Error Correcting Codes

Parity, checksums, and CRCs are all examples of *error detecting codes* (EDCs), techniques to *detect* errors. There are also techniques that both detect *and* correct errors, called *error correcting codes* (ECCs). These can correct smaller errors in a sector and detect larger errors. An ECC yields one of the few *corrective measures* available to system designers, because it both detects a problem and corrects it in some cases.

Commercial hard drives have traditionally used CRCs to calculate sector check values. Sophisticated techniques like ECCs were used on RAMs and DVDs. As hard drives have increased in size and sophistication, some have adopted ECCs.

5.4.2 Drive Partitions

When an operating system develops a new file system, the designers focus on a particular range of sizes. The oldest formats worked well with drives storing no more than a few megabytes (MB) of data. In 1977, Microsoft developed a system using a *file allocation table* (FAT), which became known as the "FAT File System."

Thanks to Moore's law, hard drive sizes have increased dramatically. They occasionally have exceeded the maximum size of contemporary file systems. The earliest version of FAT, now called "FAT 12," supported a maximum drive size of 15 MB. This was sufficient for the introduction of Microsoft's Disk Operating System (MS-DOS), which premiered with the first IBM PC in 1981. FAT 12 worked with the first PC hard drives, which were limited to 5 or 10 MB, but hard drives soon grew to 20 MB and larger.

The user couldn't simply tell the file system to use the larger size. The FAT 12 format really was designed for diskettes holding less than 700 KB. It had no way to deal with

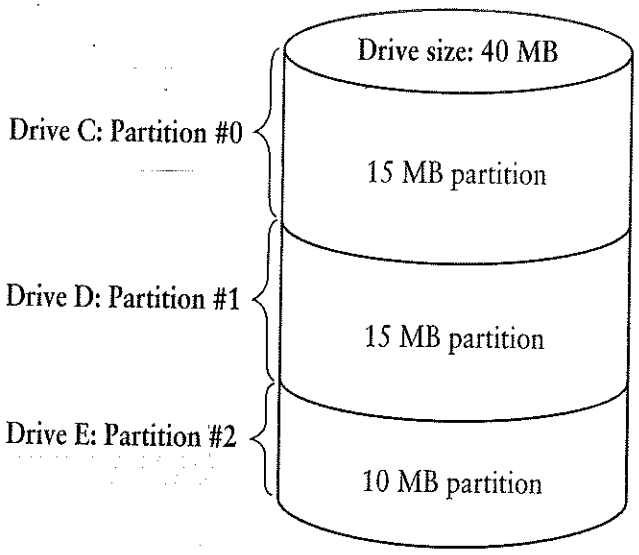


Figure 5.9

Example of MS-DOS drive partitioning.

more than 15 MB of hard drive space. Moreover, it was risky and difficult to try to patch the file system to support larger devices.

Partitioning to Support Older Drive Formats

Instead, vendors introduced drive ***partitions***, another level of drive formatting. If the owner's OS used the FAT 12 file system, but the new hard drive held 40 MB of storage, the owner would divide the drive into three or more separate partitions (Figure 5.9). As long as no partition was larger than 15 MB, the owner could make the partitions any size and use FAT 12.

On the desktop, each partition appears as a separate device. Each partition contains its own, independent file system. We often refer to a large block of storage that contains a single file system as a *volume*. When a hard drive contains two or more partitions, each represents a single volume. In most cases, the hard drive contains only one partition, and we may use the words "volume" and "hard drive" interchangeably.

Some users install different file systems in different partitions. Some even install different operating systems in different partitions. However, different operating systems would require special boot software to allow the user to choose which system to run.

Although not shown in Figure 5.9, a partitioned hard drive sets aside one or more sectors at the beginning to describe the partition arrangement. Microsoft uses the master boot record (MBR) to describe the partition layout. The MBR identifies the starting block of each partition.

The underlying drive hardware isn't aware of partitioning. When a device driver reads or writes sectors on the drive, it must adjust the sector address to match the partitioning. For example, if the driver needs to write sector 0 on drive "D" in Figure 5.9, it must find the partition's starting sector number, stored in the MBR, and add it to the desired sector address. This tells the hard drive to skip over the sectors in the "C" partition.

Partitioning in Modern Systems

As of 2011, modern file systems can easily handle typical desktop hard drives. Even so, some users and system vendors still like to partition their hard drives. Moreover, hard drive capacity continues to grow and they may exceed file system capacities again in the future. Thus, partitioning remains alive and well.

For example, some vendors use a separate partition to store a backup of the main partition. Some users set up separate partitions so that their hard drive can contain two or more different operating systems. For example, a two-partition Macintosh might contain the Windows or Linux operating system in the other partition. The user can switch operating systems by booting the other partition.

Partitioning and Fragmentation

In most cases, the best strategy is to format a hard drive with a single partition containing a single file system. When we divide a drive into separate partitions, we usually are stuck with the result. It is risky, and often impractical, to resize partitions after we install the file system.

If our hard drive has two or more partitions, we may run into a fragmentation problem: One partition may become too full while the other remains partly empty. We can't fix this problem with a built-in "defragment" utility. We must rearrange the files by hand to make more room, and manually move files from one partition to the other.

Hiding Data with Partitions

Normally an operating system tries to make as much of the hard drive available as possible. To do this, it identifies available partitions and tries to mount them as usable file systems. A casual user might not notice that the operating system only uses 100 GB of a 200 GB drive. Some people use this as a way to hide information on a computer.

There are two simple and relatively obvious ways to hide data in a system using partitions:

1. Invisible partition
2. Undersized file system

Operating systems strive to make partitions visible. If the partition contains a recognizable file system, the OS will make it visible to users. A hacker may take steps to make the partition invisible. The exact steps depend on the operating system. In some cases, it is enough to format the partition with an unrecognizable file system. For example, Unix or OS-X file systems might not be recognized by different versions of Windows. In any case, disk utilities can uncover a hidden partition.

A second approach is to create a large partition, but configure its file system to only use part of the partition. If the file system's internal data structures tell it that it mustn't exceed a particular size, then the remaining space in the partition remains free and unused. This requires special utilities to create and manage. Some disk utilities might detect a size mismatch between a file system and its partition, but not necessarily.

In either case, the hacker needs special software to store and retrieve data in this hidden area of the hard drive. The normal OS file system will not be able to access that part of the hard drive.

5.4.3 Memory Sizes and Address Variables

As we look at hard drives and other large (and growing) storage devices, we use particular words and acronyms to identify large numbers. When we talk about storage in particular, we often need to say *where* information resides; we need to express its address. We often need to store that address in a variable; we call such things an ***address variable.***

Address, Index, and Pointer Variables

Most programmers first encounter address variables when working with arrays. We select individual items from an array by using an ***index variable.*** This is the simplest type of address variable. The index value chooses which array element to use, just like an address variable chooses which memory location to use. Other variables may contain the complete address of a memory location; these are called ***pointer variables.***

An important aspect is the *size* of an address variable. For example, if our hard drive contains 1 million sectors, then the address itself will be a number ranging between zero and a million. How much space must we set aside in RAM, or in the hard drive controller, to store a number between zero and a million?

This is the same problem storekeepers face when picking out a cash register for point-of-sale transactions. How large a sale should the register handle? In general, storekeepers measure the size in terms of decimal digits; the four-digit register shown in Figure 5.10 can handle anything up to $99.99.

Many consumer electronic stores have six-digit registers to handle transactions up to $9,999.99. This handles the vast majority of purchases, but poses a problem when someone buys a really expensive home theater.

Computers don't store numbers decimally; they store numbers in binary form. Thus, we speak of variable sizes in terms of bits. To store a number between 0 and n, we need $\log_2(n)$ bits, rounded upward. If we know that $n < 2^k$, then we can store the value n if the variable contains k bits.

Memory Size Names and Acronyms

Table 5.1 summarizes the names and acronyms for large memory sizes. It also notes the number of bits required for an address variable that works with a memory of that size.

Estimating the Number of Bits

The fifth column of Table 5.1 illustrates an approximate relationship between decimal and binary exponents:

$$1000 \approx 1024$$

$$10^3 \approx 2^{10}$$

Courtesy of Dr. Richard Smith

Figure 5.10
A cash register that handles four decimal digits.

We use this to estimate the number of bits required by an address variable if we know the memory's size in decimal. We use the decimal exponent of the address size and convert it to the binary exponent to find the number of bits. For example, assume we have a nine-digit decimal number. To estimate the size of the address variable, we:

- Divide the number of decimal digits by 3.
- Multiply the result by 10 to estimate the number of bits.

Table 5.1 Abbreviations for large numbers

Abbreviation	Prefix Name	Decimal Size	Size in Thousands	Binary Approximation	Address Variable Size
K	kilo-	10^3	1000	$1024 = 2^{10}$	10
M	mega-	10^6	1000^2	$1024^2 = 2^{20}$	20
G	giga-	10^9	1000^3	$1024^3 = 2^{30}$	30
T	tera-	10^{12}	1000^4	$1024^4 = 2^{40}$	40
P	peta-	10^{15}	1000^5	$1024^5 = 2^{50}$	50
E	exa-	10^{18}	1000^6	$1024^6 = 2^{60}$	60

Table 5.2 Classic storage sizes

Address Size in Bits	Size of Memory	Description
8	256	Byte-size address
10	1024	Traditionally called "1K"
12	4096	Traditionally called "4K"
15	32,768	Traditionally called "32K"
16	65,536	Called "64K" or "65K"
24	16,777,216	Traditionally called "16 Meg"
32	4,294,967,296	4.29 gigabytes (4.29 GB)
48	281×10^{12}	281 terabytes (281 TB)
64	18.44×10^{18}	18.44 exabytes (18 EB)

This calculation overestimates the number of bits, because the binary result is slightly larger than the decimal one. Traditionally, RAM sizes fit a power of two, so the decimal size often implies the larger binary value. For example, a "1K" memory contains 1024 elements, and a "4K" memory contains 4096 elements. The convention broke down at 2^{16}, which some call "64K," because 64 is a power of two, and others call "65K," which is numerically more accurate.

Certain memory and address sizes crop up again and again, because the address variable fits into a convenient number of bits. Table 5.2 lists these classic address sizes and indicates the associated memory sizes.

On hard drives and mass storage, the technology doesn't need to match powers of two. One vendor of USB flash drives clearly states "1 GB = 1 billion bytes." A hard drive vendor's "1 terabyte" hard drive contains at least 10^{12} bytes, but falls 99 gigabytes short of 2^{40} bytes.

When we need to abbreviate a large power of two *exactly*, the standard is to affix a lowercase "i" after the uppercase abbreviation. For example:

$$2^{20} \text{ bytes} = 1 \text{ MiB}$$
$$2^{30} \text{ bytes} = 1 \text{ GiB}$$

When we talk about storage sizes, sometimes we use bits and other times we use bytes. To abbreviate the size, we use the lowercase "b" to represent bits and the upper-case "B" to represent bytes. Because 1 byte contains 8 bits:

$$64 \text{ b} = 8 \text{ B}$$

5.5 FAT: An Example File System

The essentials of file system formatting are best illustrated with an example: the FAT file system. While the format dates back to the earliest personal computers, it has evolved since then. The FAT 12 format was superseded by FAT 16, which supported gigabytes of storage. Today's FAT 32 supports terabytes of disk space.

Today, we often use FAT formatting on removable hard drives and flash memory. Most commercial digital cameras use FAT format on their storage cards. FAT provides hierarchical directories and flexible file naming. The major shortcoming is that individual FAT files must contain less than 4 gigabytes.

Volume Layout

Figure 5.11 shows the layout of a FAT formatted volume. Sector 0, the first sector of the volume, marks the beginning of the ***boot blocks***. These first sectors may contain a program to bootstrap the operating system, but they also contain variables that describe the volume's layout. Next comes the file allocation table, or FAT. This table keeps track of every cluster on the volume.

Following the FAT is the root directory, which contains entries for files and subdirectories on the volume. Each entry contains the name and the number of the first sector in the file or subdirectory. FAT directories are hierarchical, so any directory may contain entries for subdirectories.

The rest of the volume is divided into clusters to store files and subdirectories. The FAT contains one entry per cluster.

On older FAT 12 and FAT 16 volumes, the root directory had its own, dedicated set of sectors on the volume. On FAT 32, the root directory is the first file stored in the

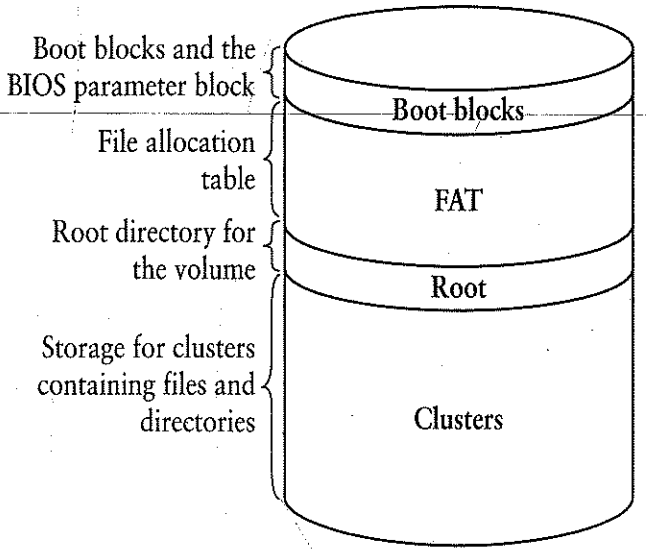


Figure 5.11

Layout of a FAT-formatted volume.

Table 5.3 Contents of the FAT 32 boot block

Offset in Bytes	Size in Bytes	Contents
0	3	A "jump" instruction to skip over the BPB data
3	8	System name, usually "MSWIN 4.1"
11	2	Number of bytes in a hard drive sector, usually 512
13	1	Number of sectors in a cluster; depends on the volume size
14	2	Number of sectors contained in the boot blocks
16	1	Number of FATs on the volume, usually 2
17	15	Variables that aren't used in FAT 32, left as spares, or used by the device driver
32	4	Total number of sectors in the volume
36	4	Total number of sectors in one FAT
40	50	Variables for other purposes, like error recovery, device driver support, and volume identification.

volume's clusters; it appears in roughly the same place, but there is a little more flexibility in its location.

5.5.1 Boot Blocks

The boot block resides on the first sectors of the hard drive; they are the blocks read by the BIOS when we boot from this drive. Every file system stores special information in the boot block, as well as providing room for a bootstrap program.

If we partition a hard drive, the first block contains the MBR, which itself contains a boot program. The MBR's boot program automatically redirects the boot operation to the boot block of the first "bootable" partition.

The first item stored in the FAT boot block is a "jump" instruction. If the BIOS tries to start an operating system stored on the FAT volume, it first reads a boot program from these boot blocks and then jumps to the first instruction read into RAM. This first instruction jumps over the variables FAT provides to describe the volume's format. This block of variables is called the BIOS parameter block (BPB). Table 5.3 describes the contents of this block in the FAT 32 format.

We use the information in the BPB to find the boundaries between the boot blocks, the FAT, and the cluster storage area. To find the start of the FAT, we look at offset 14, which indicates the number of sectors in the boot blocks.

The FAT

The FAT is an array of index variables. It contains one entry for every cluster on the volume. The number associated with different FAT formats (12, 16, and 32) indicates

the size of the address variables in bits. Each array entry corresponds to one cluster on the volume. The array entry's contents indicate one of the following:

- The cluster is part of a file.
- The cluster is free and may be assigned to a file that needs more space.
- The cluster contains a bad disk sector and should not be used.

Volumes often contain two FATs, one following right after the other. This is supposed to increase reliability. If the operating system keeps both FATs up to date and one is damaged by a disk error, then the system still can retrieve files by reading the undamaged FAT.

FAT Format Variations

The file format establishes the rules for finding and storing information on the hard drive, flash drive, or other storage device. We can think of it as a map. First, it leads us to the directories, then to a file in a directory, and then through the clusters of sectors that make up the file. Table 5.4 summarizes the FAT file formats.

The maximum sizes in Table 5.4 represent typical implementations that worked over multiple operating systems. Occasionally, someone would modify one operating system or another to support larger sizes, usually to satisfy a particularly important customer or vendor.

The format names or sizes (12, 16, and 32) indicate the size in bits of the address variables used to track individual clusters on the hard drive. Thus, a FAT 12 drive could only handle a few thousand clusters, because every cluster address had to fit in a 12-bit variable. By the same token, FAT 32 can only handle a quarter billion clusters, because 4 bits of the 32-bit cluster variables are used for other purposes within the file system. In theory, the maximum drive size should be the product of the maximum cluster size and the maximum number of clusters. Operating system and software restrictions may reduce the practical maximum size.

5.5.2 Building Files from Clusters

Files in most file systems, including FAT, consist of a series of clusters. The clusters themselves may be scattered across the volume. When the file system retrieves that file, however, it retrieves the clusters in the order the data appears in the file. In a file

Table 5.4 Microsoft's FAT file system formats

Format	Introduced with OS and Year	Maximum Cluster Size	Maximum Clusters	Maximum Drive Size
FAT 12	Disk Basic, 1977	4 KB	4,077	15 MB
FAT 16	DOS 3.31, 1987	32 KB	65,517	2 GB
FAT 32	Windows 95, 1996	32 KB	268,435,437	2 TB

containing three clusters, the beginning of the file appears in the first cluster, the middle in the second cluster, and the end in the third cluster.

The challenge for the file system is to find a file's clusters and present them in the right order. FAT stores each file as a ***cluster chain***. In other words, each FAT entry provides a "link" to connect one cluster in a file to the next cluster in that file.

Cluster Storage

Cluster storage begins with the first sector following the end of the FAT area. To find the start of cluster storage, we find the size of the FAT area and add it to the area's starting point. We do that by extracting data from the BPB as follows:

- Find the number of FATs: BPB offset 16.
- Find the size of each FAT in sectors: BPB offset 36.
- Multiply the number of FATs and the size per FAT.
- Add the start of the FAT area: BPB offset 14.

For historical reasons, the first cluster in the cluster storage area is numbered 2. This first cluster almost always contains the beginning of the root directory. If the root directory contains more than one cluster, we must look in the cluster's FAT entry to find the root's next cluster.

An Example FAT File

Figure 5.12 displays a set of clusters beginning at cluster 300. The first file begins at 300 and contains the phrase "The quick brown fox jumps over the lazy dog." The text is spread across three separate clusters. A second file, containing Lincoln's Gettysburg Address, begins at cluster 302. The cluster at 305 is not being used in a file.

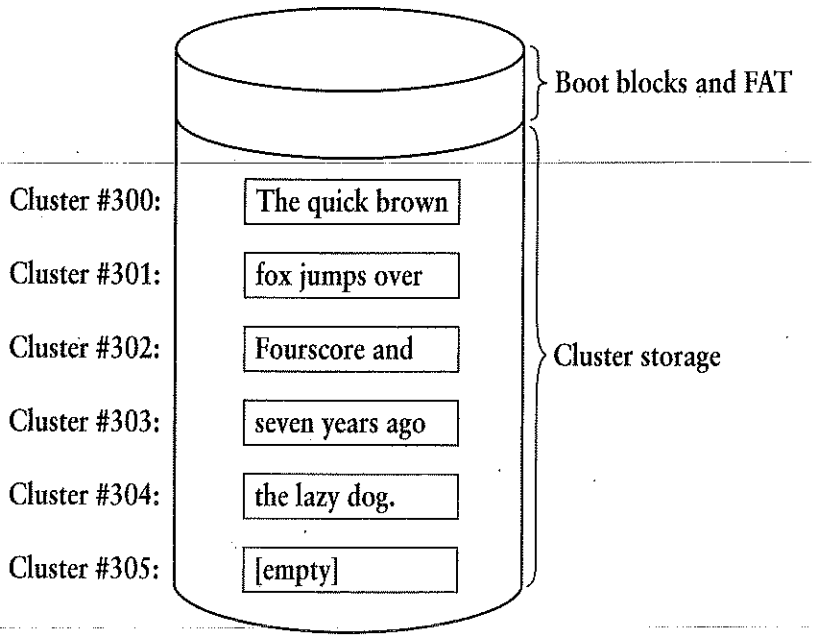


Figure 5.12

Clusters containing parts of files.

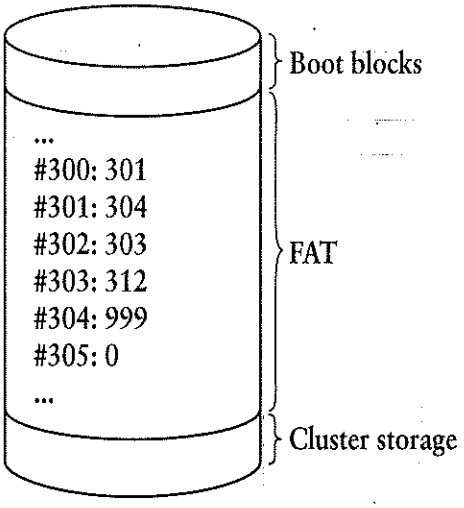


Figure 5.13

The FAT points to the clusters in a file.

Figure 5.13 displays the corresponding FAT entries. The first file contains three clusters (300, 301, and 304), and their FAT entries link them together. The FAT entry for 304, the file's final cluster, is too large to be a legal cluster number; this marks the end of the file. The second file contains at least two clusters (302 and 303) and links to additional clusters outside our example. Cluster 305 is unused and its FAT entry contains a 0.

If a cluster is part of a file, then its FAT entry contains the cluster number of the *next* cluster in that file. If the cluster is the last one in the file, then the FAT entry contains a special value to mark the end of the file. Files or directories may contain two or more clusters, and both use this linking method when their contents don't fit in a single cluster.

FAT and Flash Storage

Successful low-cost flash drives first appeared in the mid-1990s, and they soon became popular components of electronic devices like digital cameras. Early flash drives ranged in size from a few megabytes to 2 gigabytes. By 2005, however, 4 gigabyte flash drives were commercially available, but they would not work in older digital products.

The problem arose from the difference between FAT 16 and FAT 32 formats. The older products used FAT 16, because it was simpler to implement and worked with the smaller memory sizes. The larger flash drives required FAT 32 to handle their full memory size. Unfortunately, the older products did not recognize the FAT 32 file format. The only solution is to use smaller devices, though it may be possible to format a newer device with FAT 16 and simply ignore storage beyond its 2 gigabyte limit.

One type of flash memory product has adopted names to reflect the different formatting standards: the secure digital (SD) card, an offshoot of the multimedia card (MMC), a flash memory in a 24 mm x 32 mm format. The card exists in these formats:

- Secure digital (SD)—up to 2 GB using FAT 16
- SD high capacity (SDHC)—4 GB to 32 GB using FAT 32
- SD extra capacity (SDXC)—32 GB to 2 TB using FAT 32

5.5.3 FAT Directories

The directories, starting with the root directory, tell us what files exist and where to find their starting clusters. Each directory in FAT 32 consists of an array of 32-byte entries. In the simplest case, each entry represents a single file or subdirectory. A typical entry contains the following:

- The name of the file or directory
- Attributes of this directory entry
- Date and time created, last examined, and last modified
- An address variable pointing to the first cluster in the file
- The total number of bytes of data in the file

The FAT directory begins with the root directory, which itself begins at cluster 2. We look in the root directory entries to find files and other directories. The "attributes" indicate whether a directory entry points to a file or to another directory. We follow links in the FAT to find additional clusters belonging to the root directory or to the other files and directories.

Long File Names

FAT directory entries become more complicated as we look at long file names and other advanced features. The traditional FAT directory only supports file names of eight upper case characters or less, with a three-letter extension identifier. Modern FAT devices support much longer names that contain both upper- and lowercase characters.

To do this, the FAT directory sets aside a series of directory entries, one for the actual file, and additional ones to hold segments of the long name. Each "long name" entry may contain up to 13 characters. The "actual" directory entry holds a short, 11-character version of the long name plus the normal contents of the file's directory entry. The short version of the name allows older Windows and DOS systems to handle directories containing long names.

Deleting Files

To delete a FAT file, the system takes two steps. First, it locates the file's directory entry and sets it to "empty." Some files have multiple directory entries, for example, files with long file names, and those entries also are marked as "empty." Next, the system "frees" all clusters used in the file, allowing the clusters to be used in other files.

The system marks a directory entry as empty by storing a special value (decimal 229) in the first character of its file name. The system does *not* make any other changes to the directory entry. When it is time to create a new file and give it a new entry, the system scans *all* entries in the directory to ensure that the new name is unique, and to find an existing empty entry to use. The contents of a deleted directory entry remain unchanged until the entry is reused for a new file.

To free the clusters in the file, the system follows the file's cluster chain in the FAT. For each cluster in the file, the system sets the FAT entry to 0. This marks the cluster as being free for reuse.

Undeleting a File

The most common *undelete* programs are designed to recover FAT files. To undelete a FAT file, the program starts by simply changing the first character of the deleted directory entry to a legal filename character. This retrieves the file's directory entry and its first cluster of data.

To recover the entire file, its clusters must not be fragmented. In other words, they must appear one after another on the drive, in contiguous clusters following the first cluster. In addition, the clusters must not have been allocated to another file in the mean time.

The undelete operation recovers the file's data by using information in its directory entry and in the FAT. The directory entry tells us the starting cluster number and the number of bytes in the file. We calculate the number of clusters by dividing the file's length by the cluster size, then we check the sequence of clusters in the FAT that start with the first cluster in the file. If all clusters are still free (i.e., their FAT entries are 0), then they most likely still contain the file's data.

5.6 Modern File Systems

Because the FAT file system is used on almost all removable storage (especially USB storage), it is probably the most common file system. However, it is not the only file system. FAT lacks many features that people need in a modern file system. For example:

- Files are limited to a size of 4 gigabytes or less.
- Smaller FAT systems (FAT 12, FAT 16) can't recover if drive errors occur in essential locations, like the root directory.
- The simple, linear directory arrangement makes it very slow to search a really large directory (e.g., one with thousands of files).
- FAT files don't identify a file's owner.
- FAT files can't support access restrictions beyond simple flags to indicate "read only," "hidden," or "system" files.

These properties explain why modern operating systems only use FAT file systems for removable storage. Access control poses a special problem. Without access restrictions, the operating system can't trust a FAT-formatted drive to protect critical files. Older versions of Microsoft Windows could be installed on FAT file systems, but those versions couldn't protect the system from tampering by a malicious user.

Three major file systems used today include:

1. Hierarchical file system plus (HFS+)—used with Apple OS-X
2. Unix file system (UFS)—used with most Unix systems
3. Windows NT file system (NTFS)—used with Windows

File systems have evolved as computers and hard drives have become larger and faster. Practical solutions for 20 megabyte hard drives in 1989 could not handle 200 gigabyte drives in 2009. As operating systems have become more complex and sophisticated,

they've required more of their file systems. Most modern file systems address shortcomings encountered in FAT and in other older file systems.

File System Design Goals

File systems focus on a single problem: storing information on a hard drive. This boils down to three basic challenges:

1. How to store files
2. How to find files
3. How to manage the free space on the hard drive

File storage is a deceptively simple problem. The most obvious initial solution is to simply chop the hard drive into contiguous blocks, one per file. Longer files get longer blocks, while shorter files get shorter blocks. This strategy quickly fails due to fragmentation: deleting two 1 MB files doesn't guarantee that we can now create a new 1.5 MB file. The two fragments might be in different places, preventing us from using them in a single file. Modern systems use linking strategies similar to FAT's table of cluster links.

Practically every file system used today supports hierarchical file systems. However, different systems organize their directories differently. Many, but not all, still organize subdirectories as separate files. Different choices provide different blends of simplicity and high performance.

Free-space management is important from three perspectives. First, efficient free-space management makes it fast to update files and add new data to them. Second, effective free-space management minimizes fragmentation. FAT illustrates this; the largest unused fragment in the system should be no larger than a single cluster.

Third, robust free-space management reduces the risk of losing clusters if the system crashes. A robust system may keep information about free space in multiple places, making it harder to lose track of free clusters.

Conflicting File System Objectives

As operating system designers refined their file systems, they were influenced by five objectives:

1. Make the system as simple as possible. Simple systems are the most likely to work correctly.
2. Make every action as fast as possible.
3. Make random access as efficient as possible. The earliest operating systems focused on sequential access, but this is now obsolete.
4. Make the system work effectively with hard drives of a particular size, with room to grow.
5. Make the system robust so that unexpected failures won't lose data or make the system unstable.

These objectives often are contradictory, but all are addressed to some degree. Some objectives take a priority at one point in a file system's evolution, only to be superseded later by different objectives.

The remainder of this section reviews the three file systems noted here.

5.6.1 Unix File System

The Unix file system evolved in the early 1970s. In some ways, it is simpler than FAT, and in some ways, it is more sophisticated. Like FAT, Unix sets aside space among the boot blocks to contain file system layout information. On Unix, this data is called the "superblock." Like FAT, Unix organizes the hard drive into clusters instead of constructing files out of individual sectors. Traditionally, however, Unix refers to clusters as "blocks." Like FAT, Unix also supports hierarchical directories. There are significant differences, too.

Since the introduction of Unix, several modern variations of the file system have appeared. What is today called the Unix file system, or UFS, evolved in the 1980s. Linux systems today often use an "extended" version of the file system called "ext3." Although these systems share fundamental elements of the original Unix systems, they include many changes to improve performance on larger and more demanding systems.

INODES

The centerpiece of the Unix file system is the *inode* (pronounced "eye-node"). The inode is the data structure on the drive that describes each file. All inodes reside in a single data structure, the inode list (Figure 5.14). The inode itself tells us where to find the file's data

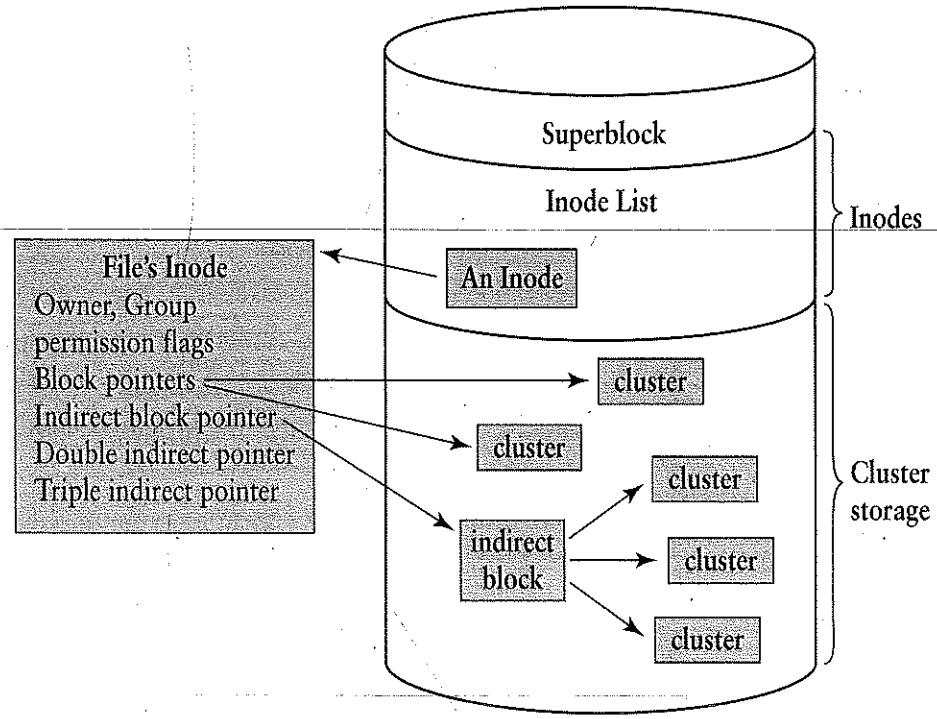


Figure 5.14

Classic Unix volume format.

clusters. Within the Unix operating system itself, some programs can omit the file names and handle files simply by their inode number: the index of the file's entry in the inode table.

To find the clusters belonging to a file, we look at the list of clusters stored in the file's inode. The inode itself points directly to the first several clusters in the file; this makes it very efficient to handle smaller files. If a file contains too many clusters to fit in that list, the inode contains a pointer to a cluster that itself contains a list of clusters. This is called an "indirect block" because each word in the cluster points to another cluster in the file. If that list overflows, too, the inode can point to *another* cluster that serves as a "double" indirect block; each word in that cluster points to an indirect block. There also can be a pointer to a "triple" indirect block whose contents point to more double indirect blocks.

The inode also contains a file's attributes. These include:

- Type of file
- Identity of the file's owner
- Associated group identifier
- File permission flags
- Access dates

Directories

The Unix directory is much simpler than a FAT directory because the file attributes reside in the inode. Each Unix directory entry contains no more than the file's name and its inode index. In older Unix systems, the directory contained fixed-size entries. Newer versions use variable-size entries. This allows the file system to handle a mixture of longer and shorter file names efficiently. Some versions also use a technique called "hash tables" to speed up the search for file names.

Early versions of the Unix file system simply kept lists of free blocks on the hard drive. The system also managed the inode list by marking an entry as being free and reusable for another file. These techniques were not efficient enough for larger, faster systems. More recent systems use ***bitmaps*** to manage these resources.

A bitmap provides a compact and efficient way to keep track of a large collection of numbered items. We assign a 1-bit flag for each item. We use the item's number to locate its individual flag within the bitmap. If the item is in use, we set the flag to 0. If the item is free, we set the flag to 1. We can quickly search for a free item by looking for a word that doesn't equal 0.

Unix uses separate bitmaps to manage inodes and free clusters. For managing inodes, the bitmap contains 1 bit for each inode, indicating which ones currently belong to files. For managing free space on the hard drive, modern systems distribute the data across the hard drive to make operations more efficient. The system creates separate bitmaps for different groups of adjacent cylinders. Each group has its own bitmap to identify free clusters in that part of the drive. When a new file receives all of its clusters from a single group, we minimize the access time required to read the file's clusters.

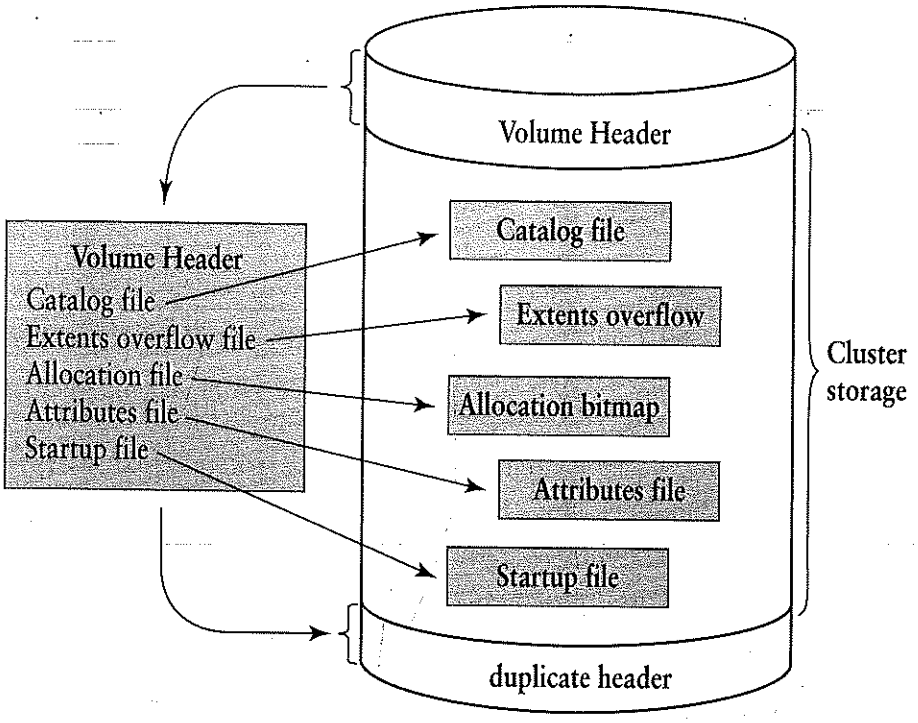


Figure 5.15

Apple's HFS Plus Volume Format.

5.6.2 Apple's HFS Plus

Apple's original hierarchical file system was introduced in 1986. It was enhanced to produce HFS+ in 1998. The HFS+ volume structure is very flexible. Each volume sets aside a few sectors at the beginning for boot blocks and for the "volume header" and a few at the end to hold a spare copy of the volume header (Figure 5.15). The rest of the volume contains clusters that belong to files.

HFS+ organizes the volume's contents around five master files. These may appear anywhere on the volume, but the volume header always contains the location of these five files. Here are the five files:

- Catalog file—contains the hierarchical directory and information about files on the volume
- Extents overflow file—contains additional information for finding clusters belonging to larger files
- Allocation file (bitmap)—indicates which clusters on the hard drive are free
- Attributes file—contains additional information about files that did not fit in the file's directory entry
- Startup file—used to boot non-Macintosh operating systems

HFS+ organizes the clusters within a file into ***extents***. Each extent contains one or more contiguous clusters. The file system describes each extent with two numbers: the

number of the first cluster in the extent, and the number of clusters in the extent. For example, we might have a file that contains 10 clusters arranged in three extents:

- First cluster in the extent: 100, length: three clusters
- First cluster in the extent: 107, length: four clusters
- First cluster in the extent: 115, length: three clusters

The first three clusters in the file are 100, 101, and 102, then we move to the next extent to retrieve the fourth cluster. If the system can find enough empty clusters in a row, it puts the entire file in a single extent.

Unlike earlier file systems, HFS+ directories are not simple lists of names. The directories use "balanced trees" (abbreviated "B-trees"), a data structure that stores the information efficiently and makes it much faster to search. HFS+ also uses this structure to organize the attributes file and extents overflow file.

5.6.3 Microsoft's NTFS

The *NT file system* first appeared as part of the Windows NT operating system, which was first introduced in 1993. In NTFS, the organizing principle is that "everything is a file." The first and most important file is the ***master file table*** or MFT. Every file, directory, or block of data on the hard drive has an entry in the MFT. While HFS+ has five essential files that are located from the boot block, an NTFS boot block points to the single essential file: the MFT. After using that pointer to find the first cluster in the MFT, we can find everything else on the drive volume (Figure 5.16).

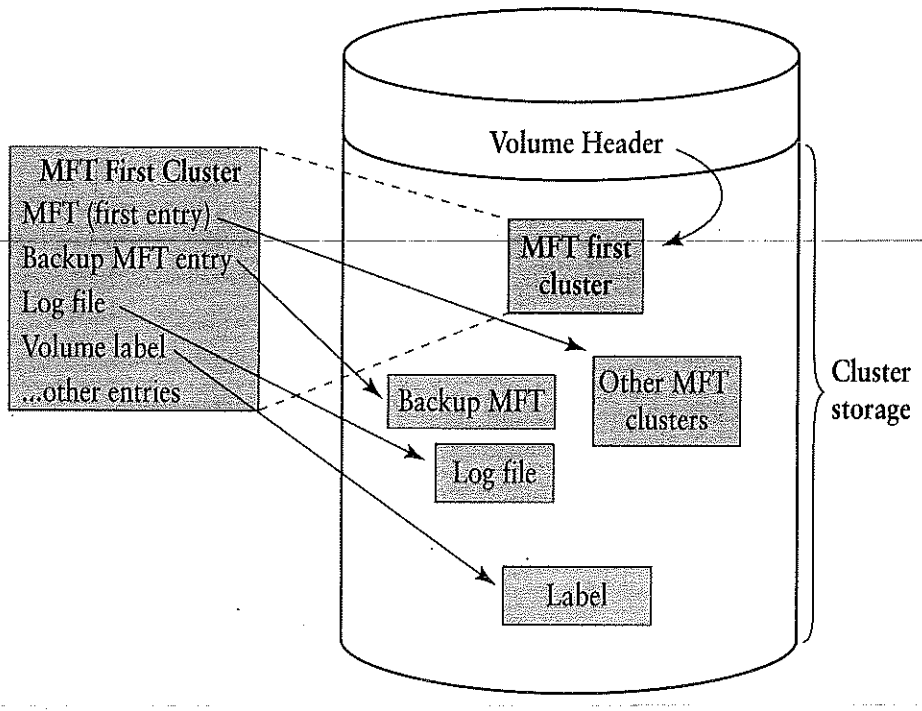


Figure 5.16

NTFS and the Master File Table.

NTFS constructs its files out of clusters and extents, much like HFS+. The clusters in an individual file often are represented by a "run list," which is a special term for a list of extents.

The very first entry in the MFT is for the MFT itself. Because this entry contains the MFT's run list, we can use it to find additional clusters belonging to the MFT. Because the MFT is itself a file, it is easier to write utility programs to examine the MFT.

The first several entries in the MFT are assigned to specific elements of the volume:

1. The MFT itself: This entry fits in the MFT's first cluster so that we can find the subsequent clusters in the MFT.
2. A backup copy of the MFT, in case its original location suffers from a drive error.
3. A built-in log file that keeps track of changes made to the volume.
4. The volume label, including its name, identifier, and version.
5. A file describing the types of data that might appear in MFT entries on this volume. This makes it easier to handle volumes created for different versions of NTFS.
6. The root directory.
7. A bitmap identifying free clusters on the hard drive.
8. An entry for the volume's boot blocks. This lets us treat the boot blocks like any other file.
9. A list of clusters that contain bad sectors. This prevents the clusters from being allocated to new files.

In some ways, the MFT is similar to the inode table in Unix. Both tables keep one entry per file. Both systems use a file's index in the table as another way to refer to the file.

However, the MFT format is significantly different. Although all inodes have the same format, each MFT entry only shares a block of header information in common. The working contents, such as file names or lists of extents, are stored in individually formatted data items called "attribute entries." For example, a typical file contains an attribute entry that lists the file's extents. A really small file, however, may contain a different attribute entry that carries the file's actual data contents. Other attributes indicate the file's ownership, access dates, and encryption information if built-in file encryption is used.

5.7 Input/Output and File System Software

In Section 2.2.2, we looked briefly at the standard features of modern operating systems: processes, memory management, input/output, and files. Here is how the operating system provides the last two:

- The operating system provides a simple and uniform way for programs to use I/O devices or files.

- When a program performs an operation on a file, the file system software transforms it into simple, standard I/O operations performed directly on the hard drive or other storage volume.
- The I/O system converts these standard operations into the specific actions required by individual I/O devices.

As systems have evolved in complexity, I/O has likewise evolved. Originally, systems focused on reading and writing streams of text that programs processed sequentially, 1 byte at a time, or one line at a time. Hard drives, when available at all, were tiny by modern standards.

Today, the I/O system must handle massive hard drives while instantly responding to keystrokes and flicks of the mouse. A practical modern system must be able to integrate a broad range of currently available I/O products, including hundreds of different printers, keyboards, mice, hard drives, USB drives, cell phones, Bluetooth devices, and other things. Moreover, the system also must be flexible enough to be able to integrate newer devices as soon as they appear.

Device Independence

Programming is a difficult task. I/O devices have always posed the biggest challenge. In the early days of computing, every program was custom written to work with the specific hard drives, printers, and other I/O devices on the programmer's computer. The programmer often had to rewrite parts of a program when new I/O devices appeared.

Device independence was an important feature of early operating systems. This provided programmers with a standard, uniform interface for programming I/O devices. The operating system provided specialized programs—the device drivers—that converted standard I/O requests into device-specific requests.

The Hourglass

The I/O system tries to provide maximum flexibility for applications and for I/O devices. This yields the hourglass design shown in Figure 5.17. The broad range of application program requirements are fulfilled by the I/O and file systems. The broad range of I/O device interfaces are mapped to the system's device driver interface.

This flexibility is demanded by the computing industry. Computer systems would be much simpler to build if everyone used the same browser software, or if all hard drive hardware worked exactly the same. The marketplace doesn't work that way in practice. Vendors flourish by providing features that competitors lack. Hardware vendors add new features and operating systems must accommodate these features. Likewise, application developers want to find new ways to use existing computing resources, and operating systems must be flexible enough to meet their needs.

File System Software

The operating system distinguishes between the file system software and the I/O system. The file system software handles the structure and organization of files on hard

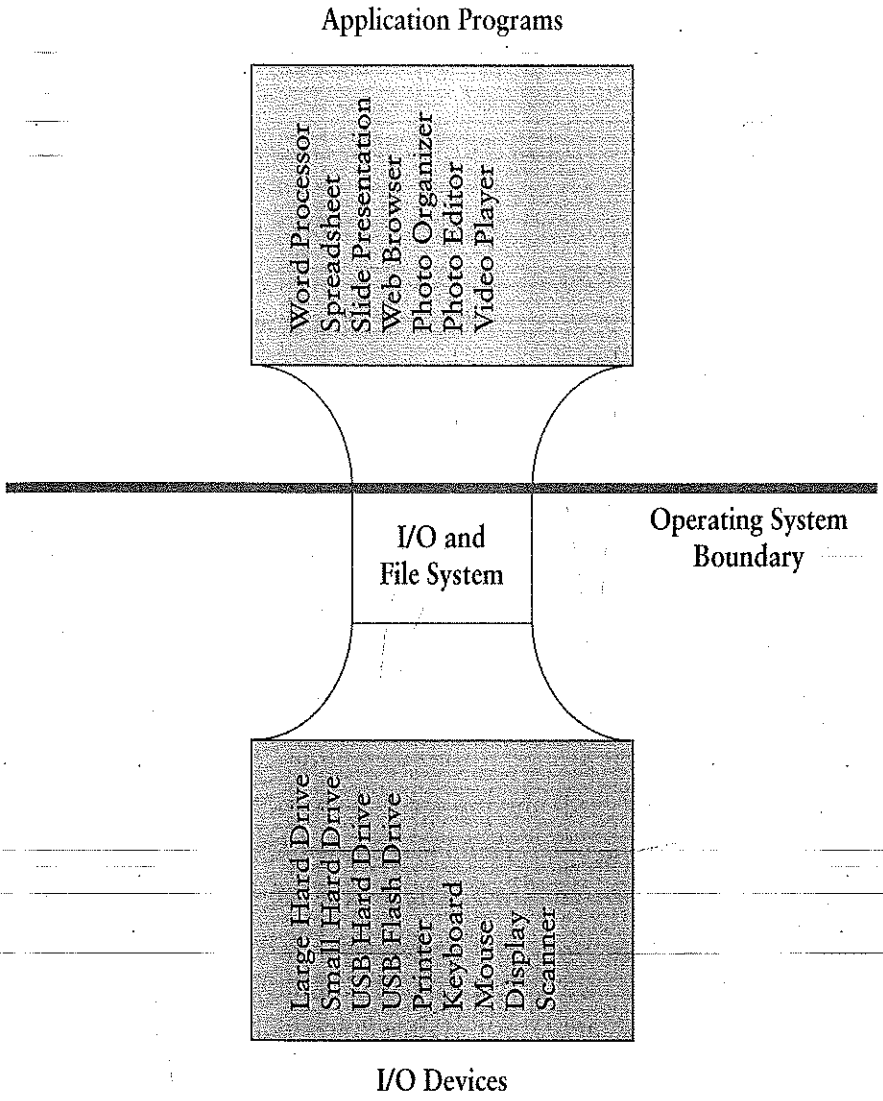


Figure 5.17

"Hourglass" structure of the I/O and file system.

drives. The I/O system provides a uniform interface for performing basic I/O operations. In practice, the phrase *file system* may refer to a set of file storage formats or it may refer to the file system software that manages a particular format.

When we connect a removable drive to a computer, the operating system automatically tries to use the drive for storing and retrieving files. The software reads the boot blocks and volume labels from the drive to try to recognize the file format. If the system doesn't recognize the file format, it may ask for help. Typically, the system asks the user if the drive should be ignored, ejected, or reformatted with a recognizable file system.

If the system recognizes the file format, it tries to ***mount*** the file system. This connects the file system to the operating system's file software. Once mounted, we may use the user interface to navigate through directories and folders stored on the drive. The file system software saves information about the file system in RAM while it is mounted.

When the system mounts a drive, it takes exclusive control of it. Other programs, run either by users or administrators, are supposed to use the file system software for all accesses to that drive.

If a program tries to read or write to the drive directly, instead of using the file system software, there is a risk that the file system might read or write the same data block for a different purpose. This is a ***concurrency problem***. If two separate programs try to use the same resource at the same time, they must coordinate their actions carefully. Otherwise, one program might undo the work of the other, leading to inconsistent data on the drive and confusion among the programs.

To remove a drive from the system, the operating system performs a ***dismount*** operation. This ensures that any information about the drive collected in RAM is properly written to the drive. This is why many operating systems expect users to explicitly perform an "eject" or "safely remove" operation before they physically disconnect a USB drive or other removable device.

Programming Assumptions

Operating systems have always made assumptions about what programmers need, and have organized the I/O requests around those assumptions. Some early systems made all I/O devices look like punched cards and printers: 80-character blocks of input and 132-character blocks of output, always read in sequence. The Multics system, an important early time sharing system, tried to treat hard drives and other mass storage as extensions of RAM. The Unix I/O system had two paradigms; everything was either a sequence of bytes or a mass storage device containing addressable blocks.

Modern graphical-oriented interfaces rely on a much more sophisticated set of functions. These functions construct and manage windows on the desktop and menus of operations to perform. Modern systems organize these functions into an ***application programming interface*** (API). Internally, these operations usually are translated into basic "read" and "write" operations ("raw" I/O).

5.7.1 Software Layering

Modern operating systems are very large and staggeringly complex. Still, they are human artifacts and they follow certain design rules. In particular, most operating systems consist of ***software layers*** (Figure 5.18).

Each layer provides an API to the layer above it. Each layer relies on the API of the layer below it. At the bottom, a layer talks directly to physical resources, like I/O devices.

An Example

In Figure 5.18, the application program opens a file stored on a USB hard drive. The operation travels through four layers of I/O software to perform the operation.

First, the program calls the operating system API to open the file and to read or write the file's data.

Second, the operating system's API forwards these calls to the file system. When the user requests a file on a mounted device, the file system converts those requests into *raw*

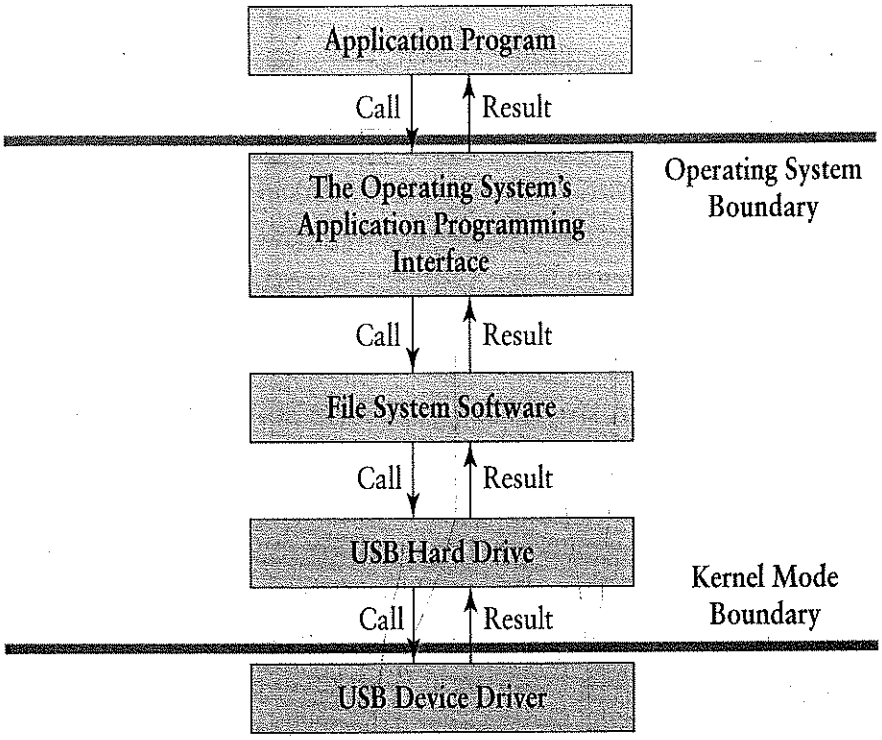


Figure 5.18

Procedure calls between operating system layer.

I/O operations. These are operations performed directly on the device, like read or write operations to specific sectors.

The file system chooses the third API layer depending on the type of device. If the device uses a more traditional mass storage interface, like ATA or SATA, then the file system passes the requests to the actual device driver. These device drivers interact directly with the device's interface circuits. In other words, the driver reads or writes data to storage registers built into the device's controller circuits. This is a privileged operation, so the device drivers run in kernel mode.

In Figure 5.18, though, the file resides on a USB drive. The file system doesn't communicate directly with the USB driver, because this driver supports a broad range of devices. The file system needs to talk to a mass storage device, so there is a separate layer of software to convert the mass storage I/O requests into USB requests. The USB hard drive software converts the file system's requests into USB operations. For the fourth layer, the hard drive software contacts the USB driver.

Layering Logic

Software designers use techniques like layering to organize large software systems. Layering is a special form of ***modularity***, the notion of organizing software into separate modules, each of which contains procedures to implement specific functions.

Some systems are designed around "strict" layering, in which programs may only call procedures in the next-lower layer. All procedures in the system are assigned to layers, and no procedure may bypass one layer to talk to another. While this makes the system

easier to understand, it also may make the program less efficient. For example, if a program needs to perform a specific operation that is implemented several layers lower, the request performs an extra procedure call for each layer it traverses. If the program could perform the low-level call directly, it bypasses all of those extra calls.

Layering often is enforced for security reasons. Potentially untrustworthy software resides at the top layer. It uses a tightly controlled API that checks all requests for proper form and authorization. After validating a request, the next layer passes the request to a lower, more fragile, layer of software that performs the operation efficiently without additional checking.

Within an operating system, strict layering applies in two places. First, all application programs must go through a tightly controlled API to request services from the operating system. Second, the operating system may provide strict layering between the user-mode operations it performs and kernel-mode operations, like those of device drivers.

In some cases, an I/O request may bypass a layer if appropriate. For example, a user program might have direct access to a device, like a USB flash drive. Such I/O requests bypass the file system completely. Likewise, I/O operations on a SATA hard drive won't have a separate layer between the file system and the device driver.

Abstraction

Taken as a whole, computer systems are incomprehensibly complicated. The only way we can understand them is to break them into components and understand the system in terms of those components.

This approach is called ***abstraction***. Instead of thinking of a computer's hardware components together in all of their complexity, we focus on the major behaviors of each one. Instead of considering how the power supply adapts to supply fluctuations while providing several different types of power to different components, we focus on the fact that it plugs in to the wall and simply feeds "the right power" to the computer's components. Instead of thinking of the CPU in terms of layers of silicon masks that exchange electrons, we think of it in terms of executing instructions and moving data around in RAM.

Figure 5.18 portrays an abstraction of the I/O system. It shows some simple features of the system's behavior and how certain components behave. It hides most of the details in order to illustrate a basic feature: that an API call is implemented by calling a series of functions that form layers.

When we use abstraction to simplify a system, we draw boundaries around its components and talk about the interactions that take place across those boundaries. To make matters simpler, we ignore interactions that aren't important for our purposes. This simplification allows us to make any system comprehensible at some level.

It is hard to hide the right amount of detail. If we do it correctly, we can examine our abstract description and use it to predict how the real system behaves. If we do it wrong, then our predictions are wrong. Successful abstraction often takes an incomprehensibly complicated system and turns it into a barely comprehensible one.

Application programming interfaces are abstractions. Instead of learning all about the programs that implement an API, the programmer simply studies the interface

specifications. If the API and its specifications are written correctly, the programmer can predict what the program will do when it calls the API a particular way.

In a sense, few programmers ever see "real" I/O devices. Instead, most see a set of functions provided by the operating system. For example, all operating systems tend to treat mass storage devices as looking more or less the same. All such devices perform "read" or "write" operations with three arguments:

1. Data location in RAM
2. Data location on the device
3. Amount of data in the operation

When performing a "read" operation, the system moves the data from the given location on the device to the given location in RAM. When performing a "write," the data moves in the opposite direction. Most operating systems apply the same concept to file I/O; the data location on the device is calculated relative to the blocks of data in the file.

"Raw" I/O operations bypass the software that provides file system abstractions. Such operations won't recognize partition boundaries, files, or access restrictions.

5.7.2 A Typical I/O Operation

Here we break a typical I/O operation into 16 steps. To make matters a little more simple (more abstract as it were), we break those steps into four parts: A through D.

In this example we read a block of data from a file. We assume the file resides on a SATA hard drive, so the file system communicates directly with the driver when performing I/O. A "write" operation is subtly different in some ways, but the overall flow is similar. Likewise, different systems handle buffers, process scheduling, and other details differently. Despite subtle differences, this is how it happens in almost every modern operating system.

Part A: Call the operating system.

1. The application calls a function to read the next several bytes of data from the file. The function maintains a buffer that contains at least one cluster. If the function requires data from the next cluster, it will need to read that cluster from the file.
2. The function issues an I/O "read" operation to retrieve the next cluster from the file. The operation identifies a buffer in RAM, the buffer's size (the cluster size), and the number of the desired cluster from the file.
3. The operation performs a "trap" operation that starts an OS program running in kernel mode to handle the I/O request. Such requests often block the program from running until the I/O operation is finished. This involves scheduling and dispatching the application program's process, as described in Section 2.7.
4. The trap handler reformats the request into a format the file system can handle. It passes the request to the file system. This request identifies the file being read.

Part B: OS constructs the I/O operation.

5. The file system retrieves data about the file in order to figure out the location of the requested cluster in the file.
6. The file system constructs an I/O request to read the requested cluster from the hard drive. It converts the file's cluster number to an absolute sector address on the hard drive.
7. The file system passes the request to the device driver. The request identifies the specific drive containing the data, because most drivers may handle two or more drives, if all are of the same design.
8. The device driver identifies the physical device associated with the requested "drive." Note that a "drive" may actually be a partition on a larger device. The driver adjusts the hard drive sector address if the sector resides in a partition.

Part C: The driver starts the actual I/O device.

9. The device driver tells the device controller to start the "read" operation. It provides the controller with the sector address on the drive, the RAM address of the buffer, and the number of bytes to transfer. To do this, the driver writes data to registers built into the controller. These registers are connected to the controller circuits and direct the controller's actions.
10. The device controller instructs the drive mechanism to locate the appropriate cylinder and sector.
11. The mechanism starts transmitting data as soon as the sector appears under the read/write head. As the data arrives in the controller, the contoller typically stores the data in an internal buffer.
12. As data arrives in the buffer, the controller starts transferring it to the RAM location specified in the I/O request.

Part D: The I/O operation ends.

13. Once all data has been transferred to the buffer in RAM, the controller generates a special signal called an "interrupt." This signal causes a special "interrupt service routine" to run in the device driver.
14. The driver's interrupt service routine marks the I/O operation as finished. It also changes the application program's process state so that the program may resume.
15. The dispatcher sees that the application is eligible to run and, when its turn arrives, the dispatcher resumes the application program at the point following the I/O request.
16. The function in the application program retrieves the requested data and returns it to the caller within the program.

5.7.3 Security and I/O

Computer systems store all of their permanent resources on I/O devices, typically on hard drives. If application programs can perform whatever I/O they want, the programs can bypass any security measures the operating system tries to enforce. Secure operating systems often begin with restricting access to I/O devices.

Restricting the Devices Themselves

A computer program controls an I/O device by reading or updating data registers in its device controller. In some computers, the registers appear like memory locations in RAM. Each I/O device has a set of assigned locations in RAM, and the device registers appear as storage locations. The program reads or modifies these registers by treating them just like other variables stored in RAM. This is called "memory mapped I/O."

Other computers provide a set of machine instructions to communicate with device registers. The instructions contain a numeric code that selects the device register of interest. Different machine instructions either read or write the device registers. This approach is called "programmed I/O."

Historically, not all operating systems have protected I/O devices, especially on desktop computers. The typical desktop operating system provided very few protections before the late 1990s. Many programs, especially graphics-oriented applications, relied on direct access to the hardware to achieve high performance. CPU improvements, especially in operating speeds and in built-in security mechanisms, helped reduce the impact of tightened security.

The operating system restricts access to I/O devices by blocking access to the device's registers. In memory-mapped I/O, the operating system blocks access to the RAM containing the device registers. The system restricts access to the device drivers. In programmed I/O, the CPU treats I/O operations as privileged instructions. Only kernel mode software may execute privileged instructions, and application programs never run in kernel mode. Because device drivers run in kernel mode, they may execute I/O instructions.

Restricting Parameters in I/O Operations

I/O security extends beyond the problem of directly controlling devices. The I/O system plays an essential role in enforcing security at other levels, too.

For example, imagine the following: Bob and Tina both are running processes on the tower computer. What would happen if Bob's program directed the I/O system to input some data into RAM belonging to Tina's process? Or worse, the input operation might overwrite instructions inside the operating system itself. The I/O system must ensure that the process' I/O request will only affect that particular process. The I/O system has to check the RAM address used in every I/O operation and ensure that the results end up in the process' own RAM.

Likewise, the I/O system needs to check mass storage addresses to ensure they remain inside an approved range. If the process has opened a file, the file system maps the address to clusters within the file. If the address exceeds the file's permitted size, the

operation fails. If the process has permission to open an entire drive partition, the system must ensure that the operation doesn't step over into a different partition.

File Access Restrictions

The file system typically enforces the file access restrictions we studied in Chapters 3 and 4. The file system retrieves permission information when opening a file, and may refuse to open a file if the process lacks the access rights. The system marks each open file to indicate the process' access rights. This allows the file system to reject attempts to write a read-only file or to delete a file the process lacks the right to delete.

5.8 Resources

IMPORTANT TERMS INTRODUCED

ABSTRACTION
ADDRESS VARIABLE
ADMISSIBLE
BITMAP
BOOT BLOCKS
CHECK VALUE
CHECKSUM
CIVIL COMPLAINT
CLUSTER
CLUSTER CHAIN
CONCURRENCY PROBLEM
CRIMINAL COMPLAINT
CYLINDER
DEVICE INDEPENDENCE
DIGITAL FORENSICS
DISK DRIVE
DISMOUNT
DRIVE CONTROLLER
DUE DILIGENCE
EVIDENCE LOG
EXTENT
FILE SCAVENGING
FRAGMENTATION
HARDWARE BLOCK DIAGRAM
HEADER
HIGH-LEVEL FORMAT
INDEX VARIABLE
INODE
LOW-LEVEL FORMAT
MEDIATION
MODULARITY
MOORE'S LAW
MOUNT
PARTITION
PLATTER
POINTER VARIABLE
PRIVATE ACTION
RANDOM ACCESS
RAW I/O
READ/WRITE HEAD
REMEDIATION
SECTOR
SEQUENTIAL ACCESS
SOFTWARE LAYERING
TRACK
UNDELETE

Acronyms Introduced

* Refer to Table 5.1 on page 191 for large number acronyms.
API—Application programming interface
B (uppercase)—Suffix indicating storage in bytes
b (lowercase)—Suffix indicating storage in bits
B-trees—Balanced trees
BPB—BIOS parameter block
CRC—Cyclic redundancy check
ECC—Error correcting code
EDC—Error detecting code
FAT—File allocation table
HFS+ or HFS plus—Hierarchical file system plus
MFT—Master file table
MMC—Multimedia card

NTFS—NT file system
RPM—Revolutions per minute
SD—Secure digital
SDHC—Secure digital high capacity
SDXC—Secure digital extended capacity
UFS—Unix file system

5.8.1 Review Questions

R1. Explain the four general tasks that may play a role in recovering from a security incident.

R2. Describe the basic requirements evidence must meet to be used in a legal proceeding.

R3. List and explain the three general categories of legal systems used in the world. Give an example of each.

R4. List and describe four ways of resolving a security incident that could rise to the level of a legal dispute.

R5. Explain the concept of due diligence.

R6. Does an employer in the United States have an unconditional right to search employee desks or lockers on company premises? Why or why not? Is there a way by which the employer can legally perform such searches?

R7. Describe the three steps an investigator performs when collecting forensic evidence.

R8. Is it better to perform a clean "shutdown" or simply pull the plug when collecting a computer as evidence?

R9. Explain how an investigator can examine a hard drive and still convince a court that the examination is based on the information residing on the drive when the suspect last had possession of it.

R10. Draw a diagram showing the basic components of a hard drive and its controller.

R11. Explain the difference between "high-level" and "low-level" disk formatting. When we perform a "quick format," what formatting do we perform?

R12. Describe two different ways of hiding data on a hard drive using partitions.

R13. What is the difference between 1 GB of storage and 1 GiB of storage? What is the difference between 1 KB of storage and 1 Kb of storage?

R14. Explain how to quickly convert a decimal number to a power of two by converting between decimal and binary exponents.

R15. What is Moore's law?

R16. Describe how to recover a deleted FAT file and its contents.

R17. Summarize shortcomings of the FAT file system compared to other modern file systems.

R18. List the three major hard drive storage problems addressed by file systems.

R19. Outline major similarities and differences between FAT, NTFS, Unix, and HFS+ file systems.

R20. Summarize the three strategies by which the operating system provides input/output services and a file system.

R21. Explain the relationship between device independence and device drivers.

R22. For each step in the example I/O operation described in Section 5.7.2, indicate which layer from Figure 5.18 performs the step.

R23. Indicate which layers from Figure 5.18 enforce which security measures in the I/O and file systems

5.8.2 Exercises

E1. Find the detailed technical specifications for a commercial hard drive. The specifications will identify a precise minimum or total amount of storage provided on the hard drive. Using this information, report the following:

a. The hard drive's advertised size in bytes
b. The exact, or minimum, number of bytes of storage actually provided by the hard drive
c. The number of bytes of the power of two, or small multiple of a power of two, that is closest to the hard drive's advertised size

E2. Search the Internet for a description of a court action whose decision affected how computer equipment and information may be used as evidence. Describe the legal problem and the court's decision.

E3. Unix has a mechanism called a *hard link* by which it creates additional directory entries that all point to the same file. This is easy to manage because most information resides in the file's inode, including a count of the number of links. Bob is trying to create a hard link to a file in a FAT directory by duplicating the file's existing directory entry and giving it a new name. *How well will existing file read, write, and delete operations work?*

a. Which operations work correctly if a FAT file has two directory entries?
b. What operations won't work correctly? How do those operations fail to work correctly?

The following questions involve a forensic examination of a FAT file system. Find a "dump" utility and use it to examine the contents of a FAT file system. First, find an unused removable device, like a USB flash drive. Reformat it. Use online descriptions of the FAT format to locate the FAT and the file directories using the dump utility. Perform these exercises, print the results using the dump utility. Use a marker to highlight the results.

E4. Create a text file. Locate the file's directory entry and print it out. Locate the first cluster in the file and print it out.

E5. Create a subdirectory and place two text files in it. Locate the subdirectory you created. Print out the subdirectory.

E6. Delete a file. Locate the file's directory entry and print it out.

E7. Table 5.5 contains the partition table from the master boot record for the volume in Figure 5.9. The following is a list of sectors stored in different partitions. For each sector and partition, calculate the

Table 5.5 Partition table for the drive in Figure 5.9

Partition ID	Starting Sector Number	Final Sector Number	Number of Sectors
p0	8	30,727	30,719
p1	30,728	61,447	30,719
p2	61,448	81,927	20,479

absolute address of the sector. (Hint: Use a spreadsheet.)

a. Partition 0, sector 1

b. Partition 0, sector 8184

c. Partition 1, sector 2040

d. Partition 1, sector 10,000

e. Partition 2, sector 1

f. Partition 2, sector 4088

The following questions involve Table 5.6, which contains part of a file allocation table. The "Cluster" column contains cluster numbers; the "Pointer" column contains the corresponding FAT entry.

Table 5.6 Part of a file allocation table

Cluster	Pointer	Cluster	Pointer	Cluster	Pointer
100	101	110	111	120	121
101	102	111	112	121	122
102	103	112	113	122	123
103	104	113	9999	123	124
104	105	114	0	124	9999
105	9999	115	0	125	0
106	107	116	117	126	127
107	108	117	118	127	128
108	109	118	119	128	129
109	116	119	9999	129	110

The FAT entry contains one of the following: 0 to indicate a free cluster, 9999 to indicate the end of file, and any value in between indicates the next cluster in the file.

The following directory entries apply to these FAT entries:

- Name: F1, Starting Cluster: 100
- Name: F2, Starting Cluster: 106
- Name: F3, Starting Cluster: 120
- Name: F4, Starting Cluster: 126

E8. For each file named in the directory entries, give the number of clusters in the file.

E9. We want to read individual bytes from these files. Clusters on this volume contain 4096 bytes each. For each of the following file names and byte offsets, identify the cluster that contains that byte.

a. File F1, offset 1000

b. File F2, offset 10,000

c. File F2, offset 20,000

d. File F3, offset 1000

e. File F4, offset 10,000

f. File F4, offset 20,000

E10. We are writing additional data to file F1 and need to add another cluster to the end of the file. Locate a cluster in the FAT to add to the file. List the specific changes to make to the FAT to add the sector to F1.

E11. As in Exercise E10, we are writing data to file F2 and must add another cluster. List the specific changes to make to the FAT to add the cluster.

E12. We are deleting File F3. List the specific changes made to the FAT to delete F3.

E13. The engineering manager has decreed that we must discard the FAT and describe the disk contents in terms of extents. Use the FAT and file entries as they appear in Table 5.6.

a. List the clusters in files F1 and F2 using extents.

b. List the clusters in files F3 and F4 using extents.

c. List all free clusters appearing in Table 5.6 using extents.

The following exercises ask about files of yours that you have not used in a while and that you *may safely modify*. You should use "About" and "Info" commands, and look at folder or directory listings to collect this information. Depending on the file system, you may be able to retrieve a creation date, reference date, and modification date.

E14. Answer the following questions about a file stored on your computer's main hard drive: the hard drive that contains your operating system.

a. What type of device is this: hard drive, solid state, removable, flash?

b. What file system does the drive use? If FAT, try to determine if it is FAT 12, FAT 16, or FAT 32.

c. Get information about the file: What dates can you retrieve and what do the dates say?

d. Open the file with an editor or other program. Look at the file but do **not** change it. Close the file. Now, collect information about the file and report which dates, if any, have changed.

e. Open the file again and make a minor, nondamaging change to it (for example, rotate an image left, then right). Save the file without changing the name. Now, collect the information about the file and report which dates, if any, have changed.

E15. Answer these questions about a file of yours stored on a *removable drive* or *USB flash memory*.

a. What type of device is this: hard drive, solid state, flash?

b. What file system does the drive use? If FAT, try to determine if it is FAT 12, FAT 16, or FAT 32.

c. Get information about the file: What dates can you retrieve and what do the dates say?

d. Open the file with an editor or other program. Look at the file but do **not** change it. Close the file. Now, collect information about the file and report which dates, if any, have changed.

e. Open the file again and make a minor, nondamaging change to it (for example, rotate an image). Save the file without changing the name. Now, collect the information about the file and report which dates, if any, have changed.

E16. Find a discarded hard drive to disassemble. As you disassemble it, keep a list of every part removed or cut. Remove the cover to display the drive mechanism. Identify the major parts.