

## Part Seven

# Security and Protection

Security ensures the authentication of system users to protect the integrity of the information stored in the system (both data and code), as well as the physical resources of the computer system. The security system prevents unauthorized access, malicious destruction or alteration of data, and accidental introduction of inconsistency.

Protection mechanisms control access to a system by limiting the types of file access permitted to users. In addition, protection must ensure that only processes that have gained proper authorization from the operating system can operate on memory segments, the CPU, and other resources.

Protection is provided by a mechanism that controls the access of programs, processes, or users to the resources defined by a computer system. This mechanism must provide a means for specifying the controls to be imposed, together with a means of enforcing them.



# Security



Both protection and security are vital to computer systems. We distinguish between these two concepts in the following way: Security is a measure of confidence that the integrity of a system and its data will be preserved. Protection is the set of mechanisms that control the access of processes and users to the resources defined by a computer system. We focus on security in this chapter and address protection in Chapter 17.

Security involves guarding computer resources against unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency. Computer resources include the information stored in the system (both data and code), as well as the CPU, memory, secondary storage, tertiary storage, and networking that compose the computer facility. In this chapter, we start by examining ways in which resources may be accidentally or purposely misused. We then explore a key security enabler—cryptography. Finally, we look at mechanisms to guard against or detect attacks.

## CHAPTER OBJECTIVES

- Discuss security threats and attacks.
- Explain the fundamentals of encryption, authentication, and hashing.
- Examine the uses of cryptography in computing.
- Describe various countermeasures to security attacks.

### 16.1 The Security Problem

In many applications, ensuring the security of the computer system is worth considerable effort. Large commercial systems containing payroll or other financial data are inviting targets to thieves. Systems that contain data pertaining to corporate operations may be of interest to unscrupulous competitors. Furthermore, loss of such data, whether by accident or fraud, can seriously impair the ability of the corporation to function. Even raw computing resources are attractive to attackers for bitcoin mining, for sending spam, and as a source from which to anonymously attack other systems.

In Chapter 17, we discuss mechanisms that the operating system can provide (with appropriate aid from the hardware) that allow users to protect their resources, including programs and data. These mechanisms work well only as long as the users conform to the intended use of and access to these resources.

We say that a system is **secure** if its resources are used and accessed as intended under all circumstances. Unfortunately, total security cannot be achieved. Nonetheless, we must have mechanisms to make security breaches a rare occurrence, rather than the norm.

Security violations (or misuse) of the system can be categorized as intentional (malicious) or accidental. It is easier to protect against accidental misuse than against malicious misuse. For the most part, protection mechanisms are the core of accident avoidance. The following list includes several forms of accidental and malicious security violations. Note that in our discussion of security, we use the terms **intruder**, **hacker**, and **attacker** for those attempting to breach security. In addition, a **threat** is the potential for a security violation, such as the discovery of a vulnerability, whereas an **attack** is an attempt to break security.

- **Breach of confidentiality**. This type of violation involves unauthorized reading of data (or theft of information). Typically, a breach of confidentiality is the goal of an intruder. Capturing secret data from a system or a data stream, such as credit-card information or identity information for identity theft, or unreleased movies or scripts, can result directly in money for the intruder and embarrassment for the hacked institution.
- **Breach of integrity**. This violation involves unauthorized modification of data. Such attacks can, for example, result in passing of liability to an innocent party or modification of the source code of an important commercial or open-source application.
- **Breach of availability**. This violation involves unauthorized destruction of data. Some attackers would rather wreak havoc and get status or bragging rights than gain financially. Website defacement is a common example of this type of security breach.
- **Theft of service**. This violation involves unauthorized use of resources. For example, an intruder (or intrusion program) may install a daemon on a system that acts as a file server.
- **Denial of service**. This violation involves preventing legitimate use of the system. **Denial-of-service (DOS)** attacks are sometimes accidental. The original Internet worm turned into a DOS attack when a bug failed to delay its rapid spread. We discuss DOS attacks further in Section 16.3.2.

Attackers use several standard methods in their attempts to breach security. The most common is **masquerading**, in which one participant in a communication pretends to be someone else (another host or another person). By masquerading, attackers breach **authentication**, the correctness of identification; they can then gain access that they would not normally be allowed. Another common attack is to replay a captured exchange of data. A **replay attack** consists of the malicious or fraudulent repeat of a valid data transmission. Sometimes the replay comprises the entire attack—for example, in a repeat of a request to transfer money. But frequently it is done along with **message**

**modification**, in which the attacker changes data in a communication without the sender's knowledge. Consider the damage that could be done if a request for authentication had a legitimate user's information replaced with an unauthorized user's. Yet another kind of attack is the **man-in-the-middle attack**, in which an attacker sits in the data flow of a communication, masquerading as the sender to the receiver, and vice versa. In a network communication, a man-in-the-middle attack may be preceded by a **session hijacking**, in which an active communication session is intercepted.

Another broad class of attacks is aimed at **privilege escalation**. Every system assigns privileges to users, even if there is just one user and that user is the administrator. Generally, the system includes several sets of privileges, one for each user account and some for the system. Frequently, privileges are also assigned to nonusers of the system (such as users from across the Internet accessing a web page without logging in or anonymous users of services such as file transfer). Even a sender of email to a remote system can be considered to have privileges—the privilege of sending an email to a receiving user on that system. Privilege escalation gives attackers more privileges than they are supposed to have. For example, an email containing a script or macro that is executed exceeds the email sender's privileges. Masquerading and message modification, mentioned above, are often done to escalate privileges. There are many more examples, as this is a very common type of attack. Indeed, it is difficult to detect and prevent all of the various attacks in this category.

As we have already suggested, absolute protection of the system from malicious abuse is not possible, but the cost to the perpetrator can be made sufficiently high to deter most intruders. In some cases, such as a denial-of-service attack, it is preferable to prevent the attack but sufficient to detect it so that countermeasures can be taken (such as up-stream filtering or adding resources such that the attack is not denying services to legitimate users).

To protect a system, we must take security measures at four levels:

1. **Physical.** The site or sites containing the computer systems must be physically secured against entry by intruders. Both the machine rooms and the terminals or computers that have access to the target machines must be secured, for example by limiting access to the building they reside in, or locking them to the desk on which they sit.
2. **Network.** Most contemporary computer systems—from servers to mobile devices to Internet of Things (IoT) devices—are networked. Networking provides a means for the system to access external resources but also provides a potential vector for unauthorized access to the system itself.

Further, computer data in modern systems frequently travel over private leased lines, shared lines like the Internet, wireless connections, and dial-up lines. Intercepting these data can be just as harmful as breaking into a computer, and interruption of communications can constitute a remote denial-of-service attack, diminishing users' use of and trust in the system.
3. **Operating system.** The operating system and its built-in set of applications and services comprise a huge code base that may harbor many vulnerabilities. Insecure default settings, misconfigurations, and security

bugs are only a few potential problems. Operating systems must thus be kept up to date (via continuous patching) and “hardened”—configured and modified to decrease the attack surface and avoid penetration. The **attack surface** is the set of points at which an attacker can try to break into the system.

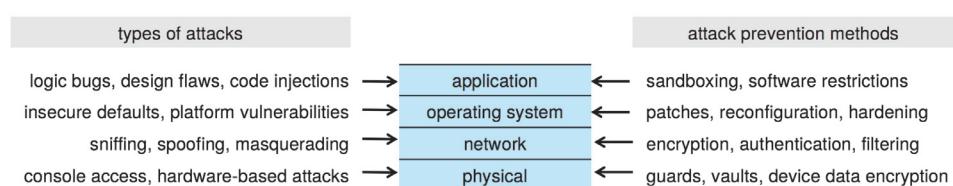
4. **Application.** Third-party applications may also pose risks, especially if they possess significant privileges. Some applications are inherently malicious, but even benign applications may contain security bugs. Due to the vast number of third-party applications and their disparate code bases, it is virtually impossible to ensure that all such applications are secure.

This four-layered security model is shown in Figure 16.1.

The four-layer model of security is like a chain made of links: a vulnerability in any of its layers can lead to full system compromise. In that respect, the old adage that security is only as strong as its weakest link holds true.

Another factor that cannot be overlooked is the human one. Authorization must be performed carefully to ensure that only allowed, trusted users have access to the system. Even authorized users, however, may be malicious or may be “encouraged” to let others use their access—whether willingly or when duped through **social engineering**, which uses deception to persuade people to give up confidential information. One type of social-engineering attack is **phishing**, in which a legitimate-looking e-mail or web page misleads a user into entering confidential information. Sometimes, all it takes is a click of a link on a browser page or in an email to inadvertently download a malicious payload, compromising system security on the user’s computer. Usually that PC is not the end target, but rather some more valuable resource. From that compromised system, attacks on other systems on the LAN or other users ensue.

So far, we’ve seen that all four factors in the four-level model, plus the human factor, must be taken into account if security is to be maintained. Furthermore, the system must provide protection (discussed in great detail in Chapter 17) to allow the implementation of security features. Without the ability to authorize users and processes to control their access, and to log their activities, it would be impossible for an operating system to implement security measures or to run securely. Hardware protection features are needed to support an overall protection scheme. For example, a system without memory



**Figure 16.1** The four-layered model of security.

protection cannot be secure. New hardware features are allowing systems to be made more secure, as we shall discuss.

Unfortunately, little in security is straightforward. As intruders exploit security vulnerabilities, security countermeasures are created and deployed. This causes intruders to become more sophisticated in their attacks. For example, spyware can provide a conduit for spam through innocent systems (we discuss this practice in Section 16.2), which in turn can deliver phishing attacks to other targets. This cat-and-mouse game is likely to continue, with more security tools needed to block the escalating intruder techniques and activities.

In the remainder of this chapter, we address security at the network and operating-system levels. Security at the application, physical and human levels, although important, is for the most part beyond the scope of this text. Security within the operating system and between operating systems is implemented in several ways, ranging from passwords for authentication through guarding against viruses to detecting intrusions. We start with an exploration of security threats.

## 16.2 Program Threats

Processes, along with the kernel, are the only means of accomplishing work on a computer. Therefore, writing a program that creates a breach of security, or causing a normal process to change its behavior and create a breach, is a common goal of attackers. In fact, even most nonprogram security events have as their goal causing a program threat. For example, while it is useful to log in to a system without authorization, it is quite a lot more useful to leave behind a back-door daemon or **Remote Access Tool (RAT)** that provides information or allows easy access even if the original exploit is blocked. In this section, we describe common methods by which programs cause security breaches. Note that there is considerable variation in the naming conventions for security holes and that we use the most common or descriptive terms.

### 16.2.1 Malware

**Malware** is software designed to exploit, disable or damage computer systems. There are many ways to perform such activities, and we explore the major variations in this section.

Many systems have mechanisms for allowing programs written by a user to be executed by other users. If these programs are executed in a domain that provides the access rights of the executing user, the other users may misuse these rights. A program that acts in a clandestine or malicious manner, rather than simply performing its stated function, is called a **Trojan horse**. If the program is executed in another domain, it can escalate privileges. As an example, consider a mobile app that purports to provide some benign functionality—say, a flashlight app—but that meanwhile surreptitiously accesses the user's contacts or messages and smuggles them to some remote server.

A classic variation of the Trojan horse is a “Trojan mule” program that emulates a login program. An unsuspecting user starts to log in at a terminal, computer, or web page and notices that she has apparently mistyped her

password. She tries again and is successful. What has happened is that her authentication key and password have been stolen by the login emulator, which was left running on the computer by the attacker or reached via a bad URL. The emulator stored away the password, printed out a login error message, and exited; the user was then provided with a genuine login prompt. This type of attack can be defeated by having the operating system print a usage message at the end of an interactive session, by requiring a nontrappable key sequence to get to the login prompt, such as the control-alt-delete combination used by all modern Windows operating systems, or by the user ensuring the URL is the right, valid one.

Another variation on the Trojan horse is **spyware**. Spyware sometimes accompanies a program that the user has chosen to install. Most frequently, it comes along with freeware or shareware programs, but sometimes it is included with commercial software. Spyware may download ads to display on the user's system, create pop-up browser windows when certain sites are visited, or capture information from the user's system and return it to a central site. The installation of an innocuous-seeming program on a Windows system could result in the loading of a spyware daemon. The spyware could contact a central site, be given a message and a list of recipient addresses, and deliver a spam message to those users from the Windows machine. This process would continue until the user discovered the spyware. Frequently, the spyware is not discovered. In 2010, it was estimated that 90 percent of spam was being delivered by this method. This theft of service is not even considered a crime in most countries!

A fairly recent and unwelcome development is a class of malware that doesn't steal information. **Ransomware** encrypts some or all of the information on the target computer and renders it inaccessible to the owner. The information itself has little value to the attacker but lots of value to the owner. The idea is to force the owner to pay money (the ransom) to get the decryption key needed to decrypt the data. As with other dealings with criminals, of course, payment of the ransom does not guarantee return of access.

Trojans and other malware especially thrive in cases where there is a violation of the **principle of least privilege**. This commonly occurs when the operating system allows by default more privileges than a normal user needs or when the user runs by default as an administrator (as was true in all Windows operating systems up to Windows 7). In such cases, the operating system's own immune system—permissions and protections of various kinds—cannot "kick in," so the malware can persist and survive across reboot, as well as extend its reach both locally and over the network.

Violating the principle of least privilege is a case of poor operating-system design decision making. An operating system (and, indeed, software in general) should allow fine-grained control of access and security, so that only the privileges needed to perform a task are available during the task's execution. The control feature must also be easy to manage and understand. Inconvenient, inadequate, and misunderstood security measures are bound to be circumvented, causing an overall weakening of the security they were designed to implement.

In yet another form of malware, the designer of a program or system leaves a hole in the software that only she is capable of using. This type of security breach, a **trap door** (or **back door**), was shown in the movie *War Games*. For

### THE PRINCIPLE OF LEAST PRIVILEGE

“The principle of least privilege. Every program and every privileged user of the system should operate using the least amount of privilege necessary to complete the job. The purpose of this principle is to reduce the number of potential interactions among privileged programs to the minimum necessary to operate correctly, so that one may develop confidence that unintentional, unwanted, or improper uses of privilege do not occur.”—Jerome H. Saltzer, describing a design principle of the Multics operating system in 1974: <https://pdfs.semanticscholar.org/1c8d/06510ad449ad24fbdd164f8008cc730cab47.pdf>.

instance, the code might check for a specific user ID or password, and it might circumvent normal security procedures when it receives that ID or password. Programmers have used the trap-door method to embezzle from banks by including rounding errors in their code and having the occasional half-cent credited to their accounts. This account crediting can add up to a large amount of money, considering the number of transactions that a large bank executes.

A trap door may be set to operate only under a specific set of logic conditions, in which case it is referred to as a **logic bomb**. Back doors of this type are especially difficult to detect, as they may remain dormant for a long time, possibly years, before being detected—usually after the damage has been done. For example, one network administrator had a destructive reconfiguration of his company’s network execute when his program detected that he was no longer employed at the company.

A clever trap door could be included in a compiler. The compiler could generate standard object code as well as a trap door, regardless of the source code being compiled. This activity is particularly nefarious, since a search of the source code of the program will not reveal any problems. Only reverse engineering of the code of the compiler itself would reveal this trap door. This type of attack can also be performed by patching the compiler or compile-time libraries after the fact. Indeed, in 2015, malware that targets Apple’s XCode compiler suite (dubbed “XCodeGhost”) affected many software developers who used compromised versions of XCode not downloaded directly from Apple.

Trap doors pose a difficult problem because, to detect them, we have to analyze all the source code for all components of a system. Given that software systems may consist of millions of lines of code, this analysis is not done frequently, and frequently it is not done at all! A software development methodology that can help counter this type of security hole is **code review**. In code review, the developer who wrote the code submits it to the code base, and one or more developers review the code and approve it or provide comments. Once a defined set of reviewers approve the code (sometimes after comments are addressed and the code is resubmitted and re-reviewed), the code is admitted into the code base and then compiled, debugged, and finally released for use. Many good software developers use development version control systems that provide tools for code review—for example, git (<https://github.com/git/>). Note, too, that there are automatic code-review and

```
#include <stdio.h>
#define BUFFER_SIZE 0

int main(int argc, char *argv[])
{
    int j = 0;
    char buffer[BUFFER_SIZE];
    int k = 0;
    if (argc < 2) {return -1;}

    strcpy(buffer, argv[1]);
    printf("K is %d, J is %d, buffer is %s\n", j,k,buffer);
    return 0;
}
```

---

**Figure 16.2** C program with buffer-overflow condition.

code-scanning tools designed to find flaws, including security flaws, but generally good programmers are the best code reviewers.

For those not involved in developing the code, code review is useful for finding and reporting flaws (or for finding and exploiting them). For most software, source code is not available, making code review much harder for nondevelopers.

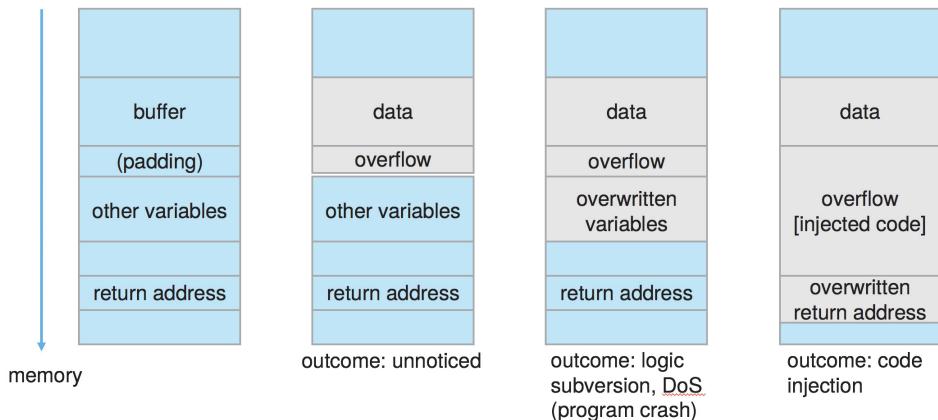
### 16.2.2 Code Injection

Most software is not malicious, but it can nonetheless pose serious threats to security due to a [code-injection attack](#), in which executable code is added or modified. Even otherwise benign software can harbor vulnerabilities that, if exploited, allow an attacker to take over the program code, subverting its existing code flow or entirely reprogramming it by supplying new code.

Code-injection attacks are nearly always the result of poor or insecure programming paradigms, commonly in low-level languages such as C or C++, which allow direct memory access through pointers. This direct memory access, coupled with the need to carefully decide on sizes of memory buffers and take care not to exceed them, can lead to memory corruption when memory buffers are not properly handled.

As an example, consider the simplest code-injection vector—a buffer overflow. The program in Figure 16.2 illustrates such an overflow, which occurs due to an unbounded copy operation, the call to `strcpy()`. The function copies with no regard to the buffer size in question, halting only when a NULL (\0) byte is encountered. If such a byte occurs before the `BUFFER_SIZE` is reached, the program behaves as expected. But the copy could easily exceed the buffer size—what then?

The answer is that the outcome of an overflow depends largely on the length of the overflow and the overflowing contents (Figure 16.3). It also varies greatly with the code generated by the compiler, which may be optimized



**Figure 16.3** The possible outcomes of buffer overflows.

in ways that affect the outcome: optimizations often involve adjustments to memory layout (commonly, repositioning or padding variables).

1. If the overflow is very small (only a little more than BUFFER\_SIZE), there is a good chance it will go entirely unnoticed. This is because the allocation of BUFFER\_SIZE bytes will often be padded to an architecture-specified boundary (commonly 8 or 16 bytes). Padding is unused memory, and therefore an overflow into it, though technically out of bounds, has no ill effect.
2. If the overflow exceeds the padding, the next automatic variable on the stack will be overwritten with the overflowing contents. The outcome here will depend on the exact positioning of the variable and on its semantics (for example, if it is employed in a logical condition that can then be subverted). If uncontrolled, this overflow could lead to a program crash, as an unexpected value in a variable could lead to an uncorrectable error.
3. If the overflow greatly exceeds the padding, all of the current function's stack frame is overwritten. At the very top of the frame is the function's return address, which is accessed when the function returns. The flow of the program is subverted and can be redirected by the attacker to another region of memory, including memory controlled by the attacker (for example, the input buffer itself, or the stack or the heap). The injected code is then executed, allowing the attacker to run arbitrary code as the processes' effective ID.

Note that a careful programmer could have performed bounds checking on the size of `argv[1]` by using the `strncpy()` function rather than `strcpy()`, replacing the line “`strcpy(buffer, argv[1]);`” with “`strncpy(buffer, argv[1], sizeof(buffer)-1);`”. Unfortunately, good bounds checking is the exception rather than the norm. `strcpy()` is one of a known class of vulnerable functions, which include `sprintf()`, `gets()`, and other functions with no

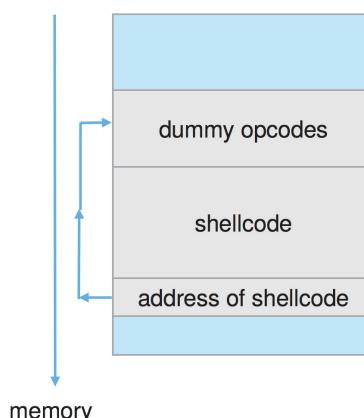
regard to buffer sizes. But even size-aware variants can harbor vulnerabilities when coupled with arithmetic operations over finite-length integers, which may lead to an integer overflow.

At this point, the dangers inherent in a simple oversight in maintaining a buffer should be clearly evident. Brian Kernighan and Dennis Ritchie (in their book *The C Programming Language*) referred to the possible outcome as “undefined behavior,” but perfectly predictable behavior can be coerced by an attacker, as was first demonstrated by the Morris Worm (and documented in RFC1135: <https://tools.ietf.org/html/rfc1135>). It was not until several years later, however, that an article in issue 49 of *Phrack* magazine (“Smashing the Stack for Fun and Profit” <http://phrack.org/issues/49/14.html>) introduced the exploitation technique to the masses, unleashing a deluge of exploits.

To achieve code injection, there must first be injectable code. The attacker first writes a short code segment such as the following:

```
void func (void) {
    execvp("/bin/sh", "/bin/sh", NULL); ;
}
```

Using the `execvp()` system call, this code segment creates a shell process. If the program being attacked runs with root permissions, this newly created shell will gain complete access to the system. Of course, the code segment can do anything allowed by the privileges of the attacked process. The code segment is next compiled into its assembly binary opcode form and then transformed into a binary stream. The compiled form is often referred to as shellcode, due to its classic function of spawning a shell, but the term has grown to encompass any type of code, including more advanced code used to add new users to a system, reboot, or even connect over the network and wait for remote instructions (called a “reverse shell”). A shellcode exploit is shown in Figure 16.4. Code that is briefly used, only to redirect execution to some other location, is much like a trampoline, “bouncing” code flow from one spot to another.



**Figure 16.4** Trampoline to code execution when exploiting a buffer overflow.

There are, in fact, shellcode compilers (the “MetaSploit” project being a notable example), which also take care of such specifics as ensuring that the code is compact and contains no NULL bytes (in case of exploitation via string copy, which would terminate on NULLs). Such a compiler may even mask the shellcode as alphanumeric characters.

If the attacker has managed to overwrite the return address (or any function pointer, such as that of a VTable), then all it takes (in the simple case) is to redirect the address to point to the supplied shellcode, which is commonly loaded as part of the user input, through an environment variable, or over some file or network input. Assuming no mitigations exist (as described later), this is enough for the shellcode to execute and the hacker to succeed in the attack. Alignment considerations are often handled by adding a sequence of NOP instructions before the shellcode. The result is known as a NOP-sled, as it causes execution to “slide” down the NOP instructions until the payload is encountered and executed.

This example of a buffer-overflow attack reveals that considerable knowledge and programming skill are needed to recognize exploitable code and then to exploit it. Unfortunately, it does not take great programmers to launch security attacks. Rather, one hacker can determine the bug and then write an exploit. Anyone with rudimentary computer skills and access to the exploit—a so-called **script kiddie**—can then try to launch the attack at target systems.

The buffer-overflow attack is especially pernicious because it can be run between systems and can travel over allowed communication channels. Such attacks can occur within protocols that are expected to be used to communicate with the target machine, and they can therefore be hard to detect and prevent. They can even bypass the security added by firewalls (Section 16.6.6).

Note that buffer overflows are just one of several vectors which can be manipulated for code injection. Overflows can also be exploited when they occur in the heap. Using memory buffers after freeing them, as well as over-freeing them (calling `free()` twice), can also lead to code injection.

### 16.2.3 Viruses and Worms

Another form of program threat is a **virus**. A virus is a fragment of code embedded in a legitimate program. Viruses are self-replicating and are designed to “infect” other programs. They can wreak havoc in a system by modifying or destroying files and causing system crashes and program malfunctions. As with most penetration attacks (direct attacks on a system), viruses are very specific to architectures, operating systems, and applications. Viruses are a particular problem for users of PCs. UNIX and other multiuser operating systems generally are not susceptible to viruses because the executable programs are protected from writing by the operating system. Even if a virus does infect such a program, its powers usually are limited because other aspects of the system are protected.

Viruses are usually borne via spam e-mail and phishing attacks. They can also spread when users download viral programs from Internet file-sharing services or exchange infected disks. A distinction can be made between viruses, which require human activity, and **worms**, which use a network to replicate without any help from humans.

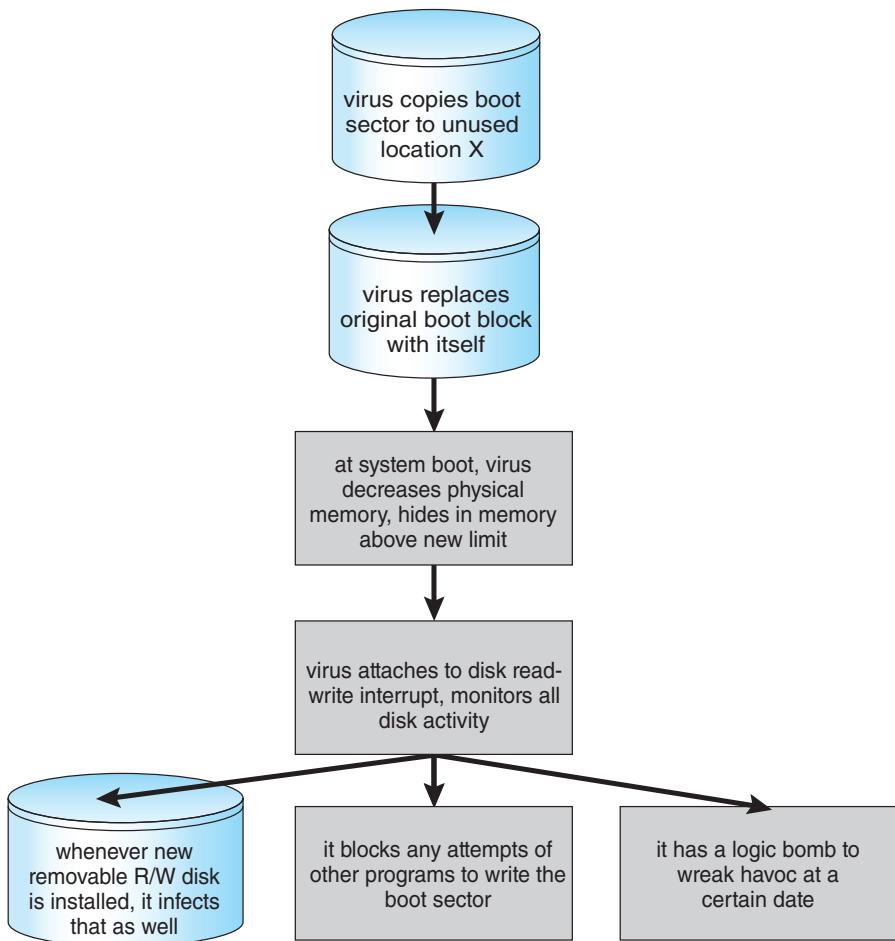
For an example of how a virus “infects” a host, consider Microsoft Office files. These files can contain *macros* (or Visual Basic programs) that programs in the Office suite (Word, PowerPoint, and Excel) will execute automatically. Because these programs run under the user’s own account, the macros can run largely unconstrained (for example, deleting user files at will). The following code sample shows how simple it is to write a Visual Basic macro that a worm could use to format the hard drive of a Windows computer as soon as the file containing the macro was opened:

```
Sub AutoOpen()
Dim oFS
Set oFS = CreateObject("Scripting.FileSystemObject")
vs = Shell("c: command.com /k format c:",vbHide)
End Sub
```

Commonly, the worm will also e-mail itself to others in the user’s contact list.

How do viruses work? Once a virus reaches a target machine, a program known as a **virus dropper** inserts the virus into the system. The virus dropper is usually a Trojan horse, executed for other reasons but installing the virus as its core activity. Once installed, the virus may do any one of a number of things. There are literally thousands of viruses, but they fall into several main categories. Note that many viruses belong to more than one category.

- **File.** A standard file virus infects a system by appending itself to a file. It changes the start of the program so that execution jumps to its code. After it executes, it returns control to the program so that its execution is not noticed. File viruses are sometimes known as parasitic viruses, as they leave no full files behind and leave the host program still functional.
- **Boot.** A boot virus infects the boot sector of the system, executing every time the system is booted and before the operating system is loaded. It watches for other bootable media and infects them. These viruses are also known as memory viruses, because they do not appear in the file system. Figure 16.5 shows how a boot virus works. Boot viruses have also adapted to infect firmware, such as network card PXE and Extensible Firmware Interface (EFI) environments.
- **Macro.** Most viruses are written in a low-level language, such as assembly or C. Macro viruses are written in a high-level language, such as Visual Basic. These viruses are triggered when a program capable of executing the macro is run. For example, a macro virus could be contained in a spreadsheet file.
- **Rootkit.** Originally coined to describe back doors on UNIX systems meant to provide easy root access, the term has since expanded to viruses and malware that infiltrate the operating system itself. The result is complete system compromise; no aspect of the system can be deemed trusted. When malware infects the operating system, it can take over all of the system’s functions, including those functions that would normally facilitate its own detection.



**Figure 16.5** A boot-sector computer virus.

- **Source code.** A source code virus looks for source code and modifies it to include the virus and to help spread the virus.
- **Polymorphic.** A polymorphic virus changes each time it is installed to avoid detection by antivirus software. The changes do not affect the virus's functionality but rather change the virus's signature. A **virus signature** is a pattern that can be used to identify a virus, typically a series of bytes that make up the virus code.
- **Encrypted.** An encrypted virus includes decryption code along with the encrypted virus, again to avoid detection. The virus first decrypts and then executes.
- **Stealth.** This tricky virus attempts to avoid detection by modifying parts of the system that could be used to detect it. For example, it could modify the **read** system call so that if the file it has modified is read, the original form of the code is returned rather than the infected code.

- **Multipartite.** A virus of this type is able to infect multiple parts of a system, including boot sectors, memory, and files. This makes it difficult to detect and contain.
- **Armored.** An armored virus is obfuscated—that is, written so as to be hard for antivirus researchers to unravel and understand. It can also be compressed to avoid detection and disinfection. In addition, virus droppers and other full files that are part of a virus infestation are frequently hidden via file attributes or unviewable file names.

This vast variety of viruses has continued to grow. For example, in 2004 a widespread virus was detected. It exploited three separate bugs for its operation. This virus started by infecting hundreds of Windows servers (including many trusted sites) running Microsoft Internet Information Server (IIS). Any vulnerable Microsoft Explorer web browser visiting those sites received a browser virus with any download. The browser virus installed several back-door programs, including a **keystroke logger**, which records everything entered on the keyboard (including passwords and credit-card numbers). It also installed a daemon to allow unlimited remote access by an intruder and another that allowed an intruder to route spam through the infected desktop computer.

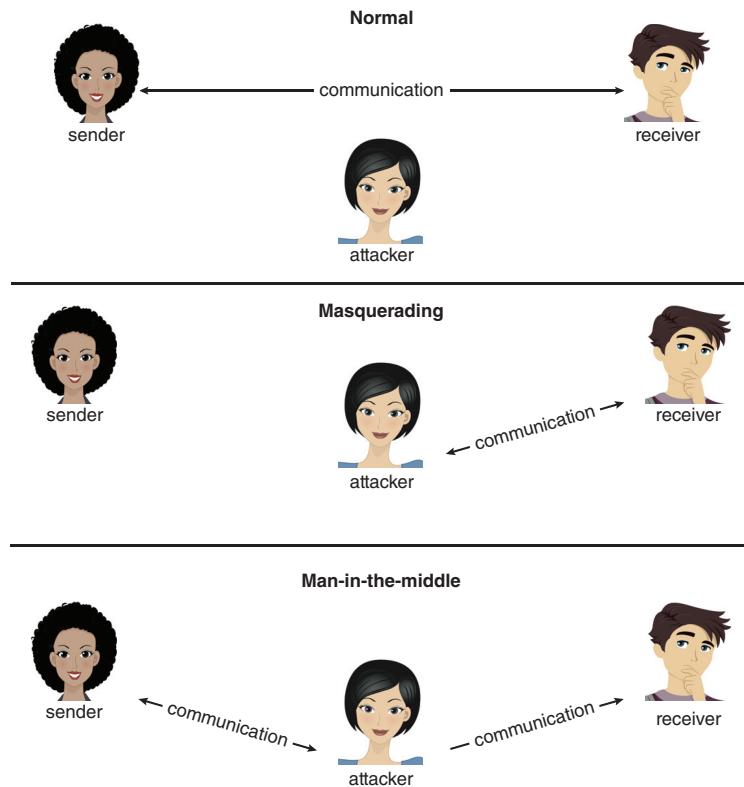
An active security-related debate within the computing community concerns the existence of a **monoculture**, in which many systems run the same hardware, operating system, and application software. This monoculture supposedly consists of Microsoft products. One question is whether such a monoculture even exists today. Another question is whether, if it does, it increases the threat of and damage caused by viruses and other security intrusions. Vulnerability information is bought and sold in places like the **dark web** (World Wide Web systems reachable via unusual client configurations or methods). The more systems an attack can affect, the more valuable the attack.

### 16.3 System and Network Threats

Program threats, by themselves, pose serious security risks. But those risks are compounded by orders of magnitude when a system is connected to a network. Worldwide connectivity makes the system vulnerable to worldwide attacks.

The more *open* an operating system is—the more services it has enabled and the more functions it allows—the more likely it is that a bug is available to exploit it. Increasingly, operating systems strive to be **secure by default**. For example, Solaris 10 moved from a model in which many services (FTP, telnet, and others) were enabled by default when the system was installed to a model in which almost all services are disabled at installation time and must specifically be enabled by system administrators. Such changes reduce the system's attack surface.

All hackers leave tracks behind them—whether via network traffic patterns, unusual packet types, or other means. For that reason, hackers frequently launch attacks from **zombie systems**—independent systems or devices that have been compromised by hackers but that continue to serve their owners while being used without the owners' knowledge for nefarious purposes,



**Figure 16.6** Standard security attacks.<sup>1</sup>

including denial-of-service attacks and spam relay. Zombies make hackers particularly difficult to track because they mask the original source of the attack and the identity of the attacker. This is one of many reasons for securing “inconsequential” systems, not just systems containing “valuable” information or services—lest they be turned into strongholds for hackers.

The widespread use of broadband and WiFi has only exacerbated the difficulty in tracking down attackers: even a simple desktop machine, which can often be easily compromised by malware, can become a valuable machine if used for its bandwidth or network access. Wireless ethernet makes it easy for attackers to launch attacks by joining a public network anonymously or “WarDriving”—locating a private unprotected network to target.

### 16.3.1 Attacking Network Traffic

Networks are common and attractive targets, and hackers have many options for mounting network attacks. As shown in Figure 16.6, an attacker can opt to remain passive and intercept network traffic (an attack commonly referred to as **sniffing**), often obtaining useful information about the types of sessions

<sup>1</sup>Lorelyn Medina/Shutterstock.

conducted between systems or the sessions' content. Alternatively, an attacker can take a more active role, either masquerading as one of the parties (referred to as **spoofin** ), or becoming a fully active man-in-the-middle, intercepting and possibly modifying transactions between two peers.

Next, we describe a common type of network attack, the denial-of-service (DoS) attack. Note that it is possible to guard against attacks through such means as encryption and authentication, which are discussed later in the chapter. Internet protocols do not, however, support either encryption or authentication by default.

### 16.3.2 Denial of Service

As mentioned earlier, denial-of-service attacks are aimed not at gaining information or stealing resources but rather at disrupting legitimate use of a system or facility. Most such attacks involve target systems or facilities that the attacker has not penetrated. Launching an attack that prevents legitimate use is frequently easier than breaking into a system or facility.

Denial-of-service attacks are generally network based. They fall into two categories. Attacks in the first category use so many facility resources that, in essence, no useful work can be done. For example, a website click could download a Java applet that proceeds to use all available CPU time or to pop up windows infinitely. The second category involves disrupting the network of the facility. There have been several successful denial-of-service attacks of this kind against major websites. Such attacks, which can last hours or days, have caused partial or full failure of attempts to use the target facility. The attacks are usually stopped at the network level until the operating systems can be updated to reduce their vulnerability.

Generally, it is impossible to prevent denial-of-service attacks. The attacks use the same mechanisms as normal operation. Even more difficult to prevent and resolve are **Distributed Denial-of-Service (DDoS)** attacks. These attacks are launched from multiple sites at once, toward a common target, typically by zombies. DDoS attacks have become more common and are sometimes associated with blackmail attempts. A site comes under attack, and the attackers offer to halt the attack in exchange for money.

Sometimes a site does not even know it is under attack. It can be difficult to determine whether a system slowdown is an attack or just a surge in system use. Consider that a successful advertising campaign that greatly increases traffic to a site could be considered a DDoS.

There are other interesting aspects of DoS attacks. For example, if an authentication algorithm locks an account for a period of time after several incorrect attempts to access the account, then an attacker could cause all authentication to be blocked by purposely making incorrect attempts to access all accounts. Similarly, a firewall that automatically blocks certain kinds of traffic could be induced to block that traffic when it should not. These examples suggest that programmers and systems managers need to fully understand the algorithms and technologies they are deploying. Finally, computer science classes are notorious sources of accidental system Dos attacks. Consider the first programming exercises in which students learn to create subprocesses or threads. A common bug involves spawning subprocesses infinitely. The system's free memory and CPU resources don't stand a chance.

### 16.3.3 Port Scanning

Port scanning is not itself an attack but is a means for a hacker to detect a system's vulnerabilities to attack. (Security personnel also use port scanning—for example, to detect services that are not needed or are not supposed to be running.) Port scanning typically is automated, involving a tool that attempts to create a TCP/IP connection or send a UDP packet to a specific port or a range of ports.

Port scanning is often part of a reconnaissance technique known as fingerprinting, in which an attacker attempts to deduce the type of operating system in use and its set of services in order to identify known vulnerabilities. Many servers and clients make this easier by disclosing their exact version number as part of network protocol headers (for example, HTTP's "Server:" and "User-Agent:" headers). Detailed analyses of idiosyncratic behaviors by protocol handlers can also help the attacker figure out what operating system the target is using—a necessary step for successful exploitation.

Network vulnerability scanners are sold as commercial products. There are also tools that perform subsets of the functionality of a full scanner. For example, nmap (from <http://www.insecure.org/nmap/>) is a very versatile open-source utility for network exploration and security auditing. When pointed at a target, it will determine what services are running, including application names and versions. It can identify the host operating system. It can also provide information about defenses, such as what firewalls are defending the target. It does not exploit known bugs. Other tools, however (such as Metasploit), pick up where the port scanners leave off and provide payload construction facilities that can be used to test for vulnerabilities—or exploit them by creating a specific payload that triggers the bug.

The seminal work on port-scanning techniques can be found in <http://phrack.org/issues/49/15.html>. Techniques are constantly evolving, as are measures to detect them (which form the basis for network intrusion detection systems, discussed later).

## 16.4 Cryptography as a Security Tool

There are many defenses against computer attacks, running the gamut from methodology to technology. The broadest tool available to system designers and users is cryptography. In this section, we discuss cryptography and its use in computer security. Note that the cryptography discussed here has been simplified for educational purposes; readers are cautioned against using any of the schemes described here in the real world. Good cryptography libraries are widely available and would make a good basis for production applications.

In an isolated computer, the operating system can reliably determine the sender and recipient of all interprocess communication, since it controls all communication channels in the computer. In a network of computers, the situation is quite different. A networked computer receives bits "from the wire" with no immediate and reliable way of determining what machine or application sent those bits. Similarly, the computer sends bits onto the network with no way of knowing who might eventually receive them. Additionally, when either sending or receiving, the system has no way of knowing if an eavesdropper listened to the communication.

Commonly, network addresses are used to infer the potential senders and receivers of network messages. Network packets arrive with a source address, such as an IP address. And when a computer sends a message, it names the intended receiver by specifying a destination address. However, for applications where security matters, we are asking for trouble if we assume that the source or destination address of a packet reliably determines who sent or received that packet. A rogue computer can send a message with a falsified source address, and numerous computers other than the one specified by the destination address can (and typically do) receive a packet. For example, all of the routers on the way to the destination will receive the packet, too. How, then, is an operating system to decide whether to grant a request when it cannot trust the named source of the request? And how is it supposed to provide protection for a request or data when it cannot determine who will receive the response or message contents it sends over the network?

It is generally considered infeasible to build a network of any scale in which the source and destination addresses of packets can be *trusted* in this sense. Therefore, the only alternative is somehow to eliminate the need to trust the network. This is the job of cryptography. Abstractly, **cryptography** is used to constrain the potential senders and/or receivers of a message. Modern cryptography is based on secrets called **keys** that are selectively distributed to computers in a network and used to process messages. Cryptography enables a recipient of a message to verify that the message was created by some computer possessing a certain key. Similarly, a sender can encode its message so that only a computer with a certain key can decode the message. Unlike network addresses, however, keys are designed so that it is not computationally feasible to derive them from the messages they were used to generate or from any other public information. Thus, they provide a much more trustworthy means of constraining senders and receivers of messages.

Cryptography is a powerful tool, and the use of cryptography can cause contention. Some countries ban its use in certain forms or limit how long the keys can be. Others have ongoing debates about whether technology vendors (such as smartphone vendors) must provide a **back door** to the included cryptography, allowing law enforcement to bypass the privacy it provides. Many observers argue, however, that back doors are an intentional security weakness that could be exploited by attackers or even misused by governments.

Finally, note that cryptography is a field of study unto itself, with large and small complexities and subtleties. Here, we explore the most important aspects of the parts of cryptography that pertain to operating systems.

#### 16.4.1 Encryption

Because it solves a wide variety of communication security problems, **encryption** is used frequently in many aspects of modern computing. It is used to send messages securely across a network, as well as to protect database data, files, and even entire disks from having their contents read by unauthorized entities. An encryption algorithm enables the sender of a message to ensure that only a computer possessing a certain key can read the message or to ensure that the writer of data is the only reader of the data. Encryption of messages is an ancient practice, of course, and there have been many encryption algorithms,

dating back to ancient times. In this section, we describe important modern encryption principles and algorithms.

An encryption algorithm consists of the following components:

- A set  $K$  of keys.
- A set  $M$  of messages.
- A set  $C$  of ciphertexts.
- An encrypting function  $E : K \rightarrow (M \rightarrow C)$ . That is, for each  $k \in K$ ,  $E_k$  is a function for generating ciphertexts from messages. Both  $E$  and  $E_k$  for any  $k$  should be efficiently computable functions. Generally,  $E_k$  is a randomized mapping from messages to ciphertexts.
- A decrypting function  $D : K \rightarrow (C \rightarrow M)$ . That is, for each  $k \in K$ ,  $D_k$  is a function for generating messages from ciphertexts. Both  $D$  and  $D_k$  for any  $k$  should be efficiently computable functions.

An encryption algorithm must provide this essential property: given a ciphertext  $c \in C$ , a computer can compute  $m$  such that  $E_k(m) = c$  only if it possesses  $k$ . Thus, a computer holding  $k$  can decrypt ciphertexts to the plaintexts used to produce them, but a computer not holding  $k$  cannot decrypt ciphertexts. Since ciphertexts are generally exposed (for example, sent on a network), it is important that it be infeasible to derive  $k$  from the ciphertexts.

There are two main types of encryption algorithms: symmetric and asymmetric. We discuss both types in the following sections.

#### 16.4.1.1 Symmetric Encryption

In a **symmetric encryption algorithm**, the same key is used to encrypt and to decrypt. Therefore, the secrecy of  $k$  must be protected. Figure 16.7 shows an example of two users communicating securely via symmetric encryption over an insecure channel. Note that the key exchange can take place directly between the two parties or via a trusted third party (that is, a certificate authority), as discussed in Section 16.4.1.4.

For the past several decades, the most commonly used symmetric encryption algorithm in the United States for civilian applications has been the **data-encryption standard (DES)** cipher adopted by the National Institute of Standards and Technology (NIST). DES works by taking a 64-bit value and a 56-bit key and performing a series of transformations that are based on substitution and permutation operations. Because DES works on a block of bits at a time, is known as a **block cipher**, and its transformations are typical of block ciphers. With block ciphers, if the same key is used for encrypting an extended amount of data, it becomes vulnerable to attack.

DES is now considered insecure for many applications because its keys can be exhaustively searched with moderate computing resources. (Note, though, that it is still frequently used.) Rather than giving up on DES, NIST created a modification called **triple DES**, in which the DES algorithm is repeated three times (two encryptions and one decryption) on the same plaintext using two or three keys—for example,  $c = E_{k3}(D_{k2}(E_{k1}(m)))$ . When three keys are used, the effective key length is 168 bits.

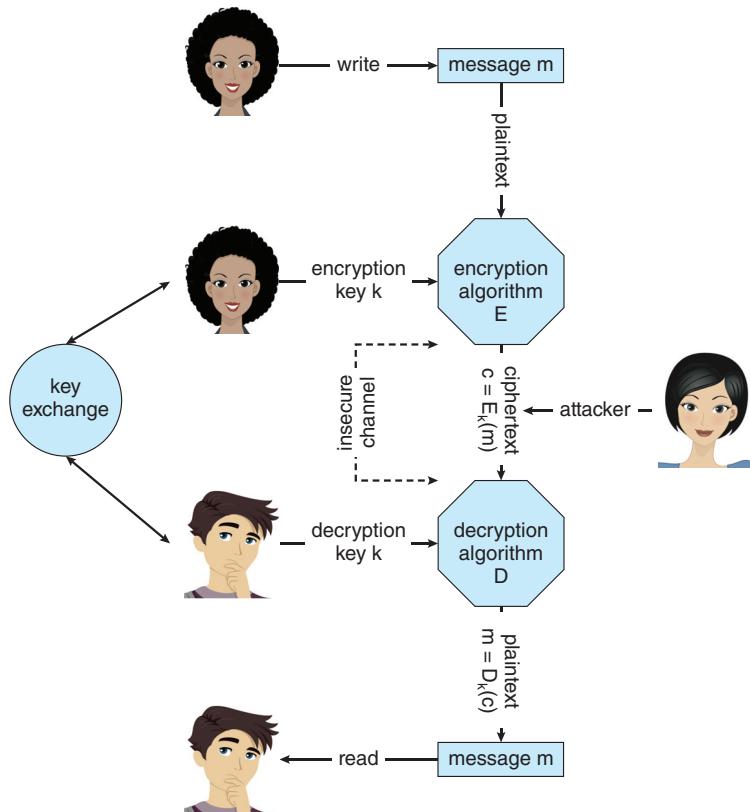


Figure 16.7 A secure communication over an insecure medium.<sup>2</sup>

In 2001, NIST adopted a new block cipher, called the **advanced encryption standard (AES)**, to replace DES. AES (also known as Rijndael) has been standardized in FIPS-197 (<http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf>). It can use key lengths of 128, 192, or 256 bits and works on 128-bit blocks. Generally, the algorithm is compact and efficient.

Block ciphers are not necessarily secure encryption schemes. In particular, they do not directly handle messages longer than their required block sizes. An alternative is stream ciphers, which can be used to securely encrypt longer messages.

A **stream cipher** is designed to encrypt and decrypt a stream of bytes or bits rather than a block. This is useful when the length of a communication would make a block cipher too slow. The key is input into a pseudo-random-bit generator, which is an algorithm that attempts to produce random bits. The output of the generator when fed a key is a keystream. A **keystream** is an infinite set of bits that can be used to encrypt a plaintext stream through an XOR operation. (XOR, for “exclusive OR” is an operation that compares two input bits and generates one output bit. If the bits are the same, the result is 0. If the bits are different, the result is 1.) AES-based cipher suites include stream ciphers and are the most common today.

<sup>2</sup>Lorelyn Medina/Shutterstock.

### 16.4.1.2 Asymmetric Encryption

In an **asymmetric encryption algorithm**, there are different encryption and decryption keys. An entity preparing to receive encrypted communication creates two keys and makes one of them (called the public key) available to anyone who wants it. Any sender can use that key to encrypt a communication, but only the key creator can decrypt the communication. This scheme, known as **public-key encryption**, was a breakthrough in cryptography (first described by Diffie and Hellman in <https://www-ee.stanford.edu/hellman/publications/24.pdf>). No longer must a key be kept secret and delivered securely. Instead, anyone can encrypt a message to the receiving entity, and no matter who else is listening, only that entity can decrypt the message.

As an example of how public-key encryption works, we describe an algorithm known as **RSA**, after its inventors, Rivest, Shamir, and Adleman. RSA is the most widely used asymmetric encryption algorithm. (Asymmetric algorithms based on elliptic curves are gaining ground, however, because the key length of such an algorithm can be shorter for the same amount of cryptographic strength.)

In RSA,  $k_e$  is the **public key**, and  $k_d$  is the **private key**.  $N$  is the product of two large, randomly chosen prime numbers  $p$  and  $q$  (for example,  $p$  and  $q$  are 2048 bits each). It must be computationally infeasible to derive  $k_{d,N}$  from  $k_{e,N}$ , so that  $k_e$  need not be kept secret and can be widely disseminated. The encryption algorithm is  $E_{k_e,N}(m) = m^{k_e} \bmod N$ , where  $k_e$  satisfies  $k_e k_d \bmod (p-1)(q-1) = 1$ . The decryption algorithm is then  $D_{k_d,N}(c) = c^{k_d} \bmod N$ .

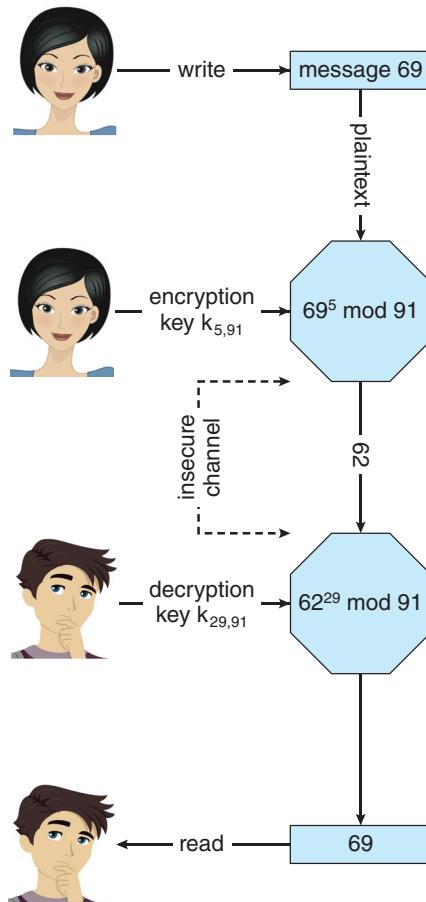
An example using small values is shown in Figure 16.8. In this example, we make  $p = 7$  and  $q = 13$ . We then calculate  $N = 7 * 13 = 91$  and  $(p-1)(q-1) = 72$ . We next select  $k_e$  relatively prime to 72 and < 72, yielding 5. Finally, we calculate  $k_d$  such that  $k_e k_d \bmod 72 = 1$ , yielding 29. We now have our keys: the public key,  $k_{e,N} = 5, 91$ , and the private key,  $k_{d,N} = 29, 91$ . Encrypting the message 69 with the public key results in the message 62, which is then decoded by the receiver via the private key.

The use of asymmetric encryption begins with the publication of the public key of the destination. For bidirectional communication, the source also must publish its public key. “Publication” can be as simple as handing over an electronic copy of the key, or it can be more complex. The private key (or “secret key”) must be zealously guarded, as anyone holding that key can decrypt any message created by the matching public key.

We should note that the seemingly small difference in key use between asymmetric and symmetric cryptography is quite large in practice. Asymmetric cryptography is much more computationally expensive to execute. It is much faster for a computer to encode and decode ciphertext by using the usual symmetric algorithms than by using asymmetric algorithms. Why, then, use an asymmetric algorithm? In truth, these algorithms are not used for general-purpose encryption of large amounts of data. However, they are used not only for encryption of small amounts of data but also for authentication, confidentiality, and key distribution, as we show in the following sections.

### 16.4.1.3 Authentication

We have seen that encryption offers a way of constraining the set of possible receivers of a message. Constraining the set of potential senders of a message



**Figure 16.8** Encryption and decryption using RSA asymmetric cryptography.<sup>3</sup>

is called **authentication**. Authentication is thus complementary to encryption. Authentication is also useful for proving that a message has not been modified. Next, we discuss authentication as a constraint on possible senders of a message. Note that this sort of authentication is similar to but distinct from user authentication, which we discuss in Section 16.5.

An authentication algorithm using symmetric keys consists of the following components:

- A set  $K$  of keys.
- A set  $M$  of messages.
- A set  $A$  of authenticators.
- A function  $S : K \rightarrow (M \rightarrow A)$ . That is, for each  $k \in K$ ,  $S_k$  is a function for generating authenticators from messages. Both  $S$  and  $S_k$  for any  $k$  should be efficiently computable functions.

---

<sup>3</sup>Lorelyn Medina/Shutterstock.

- A function  $V : K \rightarrow (M \times A \rightarrow \{\text{true, false}\})$ . That is, for each  $k \in K$ ,  $V_k$  is a function for verifying authenticators on messages. Both  $V$  and  $V_k$  for any  $k$  should be efficiently computable functions.

The critical property that an authentication algorithm must possess is this: for a message  $m$ , a computer can generate an authenticator  $a \in A$  such that  $V_k(m, a) = \text{true}$  only if it possesses  $k$ . Thus, a computer holding  $k$  can generate authenticators on messages so that any computer possessing  $k$  can verify them. However, a computer not holding  $k$  cannot generate authenticators on messages that can be verified using  $V_k$ . Since authenticators are generally exposed (for example, sent on a network with the messages themselves), it must not be feasible to derive  $k$  from the authenticators. Practically, if  $V_k(m, a) = \text{true}$ , then we know that  $m$  has not been modified and that the sender of the message has  $k$ . If we share  $k$  with only one entity, then we know that the message originated from  $k$ .

Just as there are two types of encryption algorithms, there are two main varieties of authentication algorithms. The first step in understanding these algorithms is to explore hash functions. A **hash function**  $H(m)$  creates a small, fixed-sized block of data, known as a **message digest** or **hash value**, from a message  $m$ . Hash functions work by taking a message, splitting it into blocks, and processing the blocks to produce an  $n$ -bit hash.  $H$  must be collision resistant—that is, it must be infeasible to find an  $m' \neq m$  such that  $H(m) = H(m')$ . Now, if  $H(m) = H(m')$ , we know that  $m = m'$ —that is, we know that the message has not been modified. Common message-digest functions include **MD5** (now considered insecure), which produces a 128-bit hash, and **SHA-1**, which outputs a 160-bit hash. Message digests are useful for detecting changed messages but are not useful as authenticators. For example,  $H(m)$  can be sent along with a message; but if  $H$  is known, then someone could modify  $m$  to  $m'$  and recompute  $H(m')$ , and the message modification would not be detected. Therefore, we must authenticate  $H(m)$ .

The first main type of authentication algorithm uses symmetric encryption. In a **message-authentication code (MAC)**, a cryptographic checksum is generated from the message using a secret key. A MAC provides a way to securely authenticate short values. If we use it to authenticate  $H(m)$  for an  $H$  that is collision resistant, then we obtain a way to securely authenticate long messages by hashing them first. Note that  $k$  is needed to compute both  $S_k$  and  $V_k$ , so anyone able to compute one can compute the other.

The second main type of authentication algorithm is a **digital-signature algorithm**, and the authenticators thus produced are called **digital signatures**. Digital signatures are very useful in that they enable *anyone* to verify the authenticity of the message. In a digital-signature algorithm, it is computationally infeasible to derive  $k_s$  from  $k_v$ . Thus,  $k_v$  is the public key, and  $k_s$  is the private key.

Consider as an example the RSA digital-signature algorithm. It is similar to the RSA encryption algorithm, but the key use is reversed. The digital signature of a message is derived by computing  $S_{k_s}(m) = H(m)^{k_s} \bmod N$ . The key  $k_s$  again is a pair  $\langle d, N \rangle$ , where  $N$  is the product of two large, randomly chosen prime numbers  $p$  and  $q$ . The verification algorithm is then  $V_{k_v}(m, a) = a^{k_v} \bmod N = H(m)$ , where  $k_v$  satisfies  $k_v k_s \bmod (p-1)(q-1) = 1$ . Digital signatures (as is the case with many aspects of cryptography) can be used on other entities

than messages. For example creators of programs can “sign their code” via a digital signature to validate that the code has not been modified between its publication and its installation on a computer. **Code signing** has become a very common security improvement method on many systems.

Note that encryption and authentication may be used together or separately. Sometimes, for instance, we want authentication but not confidentiality. For example, a company could provide a software patch and could “sign” that patch to prove that it came from the company and that it hasn’t been modified.

Authentication is a component of many aspects of security. For example, digital signatures are the core of **nonrepudiation**, which supplies proof that an entity performed an action. A typical example of nonrepudiation involves the filling out of electronic forms as an alternative to the signing of paper contracts. Nonrepudiation assures that a person filling out an electronic form cannot deny that he did so.

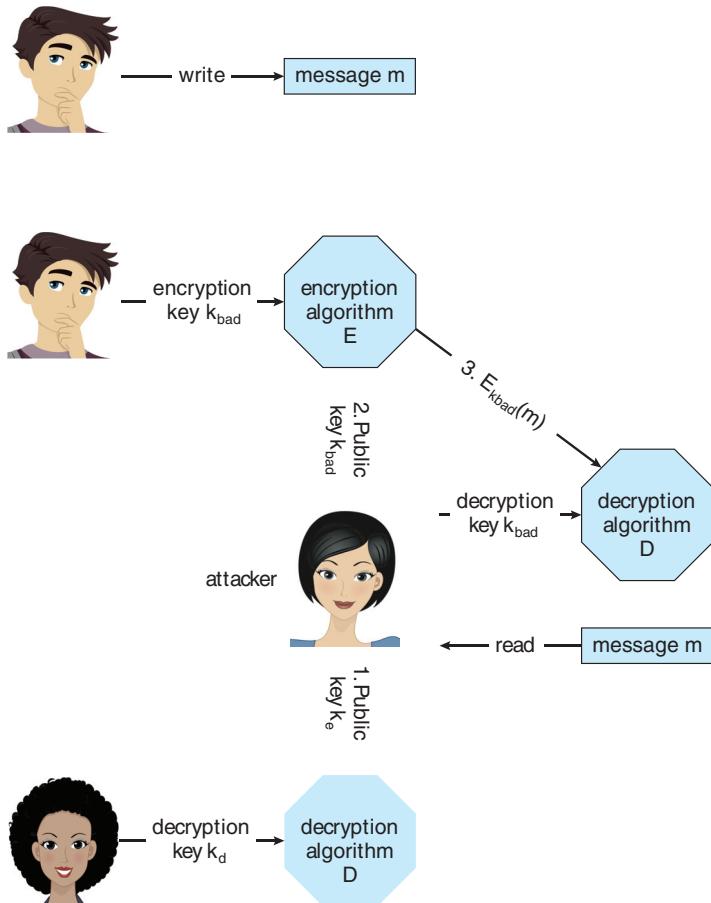
#### 16.4.1.4 Key Distribution

Certainly, a good part of the battle between cryptographers (those inventing ciphers) and cryptanalysts (those trying to break them) involves keys. With symmetric algorithms, both parties need the key, and no one else should have it. The delivery of the symmetric key is a huge challenge. Sometimes it is performed **out-of-band**. For example, if Walter wanted to communicate with Rebecca securely, they could exchange a key via a paper document or a conversation and then have the communication electronically. These methods do not scale well, however. Also consider the key-management challenge. Suppose Lucy wanted to communicate with  $N$  other users privately. Lucy would need  $N$  keys and, for more security, would need to change those keys frequently.

These are the very reasons for efforts to create asymmetric key algorithms. Not only can the keys be exchanged in public, but a given user, say Audra, needs only one private key, no matter how many other people she wants to communicate with. There is still the matter of managing a public key for each recipient of the communication, but since public keys need not be secured, simple storage can be used for that **key ring**.

Unfortunately, even the distribution of public keys requires some care. Consider the man-in-the-middle attack shown in Figure 16.9. Here, the person who wants to receive an encrypted message sends out his public key, but an attacker also sends her “bad” public key (which matches her private key). The person who wants to send the encrypted message knows no better and so uses the bad key to encrypt the message. The attacker then happily decrypts it.

The problem is one of authentication—what we need is proof of who (or what) owns a public key. One way to solve that problem involves the use of digital certificates. A **digital certificate** is a public key digitally signed by a trusted party. The trusted party receives proof of identification from some entity and certifies that the public key belongs to that entity. But how do we know we can trust the certifier? These **certificate authorities** have their public keys included within web browsers (and other consumers of certificates) before they are distributed. The certificate authorities can then vouch for other authorities (digitally signing the public keys of these other authorities), and so on, creating a web of trust. The certificates can be distributed in a standard



**Figure 16.9** A man-in-the-middle attack on asymmetric cryptography.<sup>4</sup>

X.509 digital certificate format that can be parsed by computer. This scheme is used for secure web communication, as we discuss in Section 16.4.3.

### 16.4.2 Implementation of Cryptography

Network protocols are typically organized in *layers*, with each layer acting as a client of the one below it. That is, when one protocol generates a message to send to its protocol peer on another machine, it hands its message to the protocol below it in the network-protocol stack for delivery to its peer on that machine. For example, in an IP network, TCP (a *transport-layer* protocol) acts as a client of IP (a *network-layer* protocol): TCP packets are passed down to IP for delivery to the IP peer at the other end of the connection. IP encapsulates the TCP packet in an IP packet, which it similarly passes down to the *data-link layer* to be transmitted across the network to its peer on the destination computer. This IP peer then delivers the TCP packet up to the TCP peer on that machine. Seven

<sup>4</sup>Lorelyn Medina/Shutterstock.

such layers are included in the OSI model, mentioned earlier and described in detail in Section 19.3.2.

Cryptography can be inserted at almost any layer in network protocol stacks. TLS (Section 16.4.3), for example, provides security at the transport layer. Network-layer security generally has been standardized on **IPSec**, which defines IP packet formats that allow the insertion of authenticators and the encryption of packet contents. IPSec uses symmetric encryption and uses the **Internet Key Exchange (IKE)** protocol for key exchange. IKE is based on public-key encryption. IPSec has widely used as the basis for **virtual private networks (VPNs)**, in which all traffic between two IPSec endpoints is encrypted to make a private network out of one that would otherwise be public. Numerous protocols also have been developed for use by applications, such as PGP for encrypting e-mail; in this type of scheme, the applications themselves must be coded to implement security.

Where is cryptographic protection best placed in a protocol stack? In general, there is no definitive answer. On the one hand, more protocols benefit from protections placed lower in the stack. For example, since IP packets encapsulate TCP packets, encryption of IP packets (using IPSec, for example) also hides the contents of the encapsulated TCP packets. Similarly, authenticators on IP packets detect the modification of contained TCP header information.

On the other hand, protection at lower layers in the protocol stack may give insufficient protection to higher-layer protocols. For example, an application server that accepts connections encrypted with IPSec might be able to authenticate the client computers from which requests are received. However, to authenticate a user at a client computer, the server may need to use an application-level protocol—the user may be required to type a password. Also consider the problem of e-mail. E-mail delivered via the industry-standard SMTP protocol is stored and forwarded, frequently multiple times, before it is delivered. Each of these transmissions could go over a secure or an insecure network. For e-mail to be secure, the e-mail message needs to be encrypted so that its security is independent of the transports that carry it.

Unfortunately, like many tools, encryption can be used not only for “good” but also for “evil.” The ransomware attacks described earlier, for example, are based on encryption. As mentioned, the attackers encrypt information on the target system and render it inaccessible to the owner. The idea is to force the owner to pay a ransom to get the key needed to decrypt the data. Prevention of such attacks takes the form of better system and network security and a well-executed backup plan so that the contents of the files can be restored without the key.

### 16.4.3 An Example: TLS

**Transport Layer Security (TLS)** is a cryptographic protocol that enables two computers to communicate securely—that is, so that each can limit the sender and receiver of messages to the other. It is perhaps the most commonly used cryptographic protocol on the Internet today, since it is the standard protocol by which web browsers communicate securely with web servers. For completeness, we should note that TLS evolved from SSL (Secure Sockets Layer), which was designed by Netscape. It is described in detail in <https://tools.ietf.org/html/rfc5246>.

TLS is a complex protocol with many options. Here, we present only a single variation of it. Even then, we describe it in a very simplified and abstract form, so as to maintain focus on its use of cryptographic primitives. What we are about to see is a complex dance in which asymmetric cryptography is used so that a client and a server can establish a secure **session key** that can be used for symmetric encryption of the session between the two—all of this while avoiding man-in-the-middle and replay attacks. For added cryptographic strength, the session keys are forgotten once a session is completed. Another communication between the two will require generation of new session keys.

The TLS protocol is initiated by a client  $c$  to communicate securely with a server. Prior to the protocol's use, the server  $s$  is assumed to have obtained a certificate, denoted  $\text{cert}_s$ , from certification authority CA. This certificate is a structure containing the following:

- Various attributes (**attrs**) of the server, such as its unique *distinguished* name and its *common* (DNS) name
- The identity of a asymmetric encryption algorithm  $E()$  for the server
- The public key  $k_e$  of this server
- A validity interval (**interval**) during which the certificate should be considered valid
- A digital signature  $a$  on the above information made by the CA—that is,  $a = S_{kCA}(\langle \text{attrs}, E_{ke}, \text{interval} \rangle)$

In addition, prior to the protocol's use, the client is presumed to have obtained the public verification algorithm  $V_{kCA}$  for CA. In the case of the web, the user's browser is shipped from its vendor containing the verification algorithms and public keys of certain certification authorities. The user can delete these or add others.

When  $c$  connects to  $s$ , it sends a 28-byte random value  $n_c$  to the server, which responds with a random value  $n_s$  of its own, plus its certificate  $\text{cert}_s$ . The client verifies that  $V_{kCA}(\langle \text{attrs}, E_{ke}, \text{interval} \rangle, a) = \text{true}$  and that the current time is in the validity interval **interval**. If both of these tests are satisfied, the server has proved its identity. Then the client generates a random 46-byte **premaster secret** pms and sends  $\text{cpms} = E_{ke}(\text{pms})$  to the server. The server recovers  $\text{pms} = D_{kd}(\text{cpms})$ . Now both the client and the server are in possession of  $n_c$ ,  $n_s$ , and  $\text{pms}$ , and each can compute a shared 48-byte **master secret** ms =  $H(n_c, n_s, \text{pms})$ . Only the server and client can compute ms, since only they know pms. Moreover, the dependence of ms on  $n_c$  and  $n_s$  ensures that ms is a *fresh* value—that is, a session key that has not been used in a previous communication. At this point, the client and the server both compute the following keys from the ms:

- A symmetric encryption key  $k_{cs}^{\text{crypt}}$  for encrypting messages from the client to the server
- A symmetric encryption key  $k_{sc}^{\text{crypt}}$  for encrypting messages from the server to the client
- A MAC generation key  $k_{cs}^{\text{mac}}$  for generating authenticators on messages from the client to the server

- A MAC generation key  $k_{sc}^{mac}$  for generating authenticators on messages from the server to the client

To send a message  $m$  to the server, the client sends

$$c = E_{k_{cs}^{crypt}}(\langle m, S_{k_{sc}^{mac}}(m) \rangle).$$

Upon receiving  $c$ , the server recovers

$$\langle m, a \rangle = D_{k_{sc}^{crypt}}(c)$$

and accepts  $m$  if  $V_{k_{sc}^{mac}}(m, a) = \text{true}$ . Similarly, to send a message  $m$  to the client, the server sends

$$c = E_{k_{sc}^{crypt}}(\langle m, S_{k_{sc}^{mac}}(m) \rangle)$$

and the client recovers

$$\langle m, a \rangle = D_{k_{sc}^{crypt}}(c)$$

and accepts  $m$  if  $V_{k_{sc}^{mac}}(m, a) = \text{true}$ .

This protocol enables the server to limit the recipients of its messages to the client that generated pms and to limit the senders of the messages it accepts to that same client. Similarly, the client can limit the recipients of the messages it sends and the senders of the messages it accepts to the party that knows  $k_d$  (that is, the party that can decrypt cpms). In many applications, such as web transactions, the client needs to verify the identity of the party that knows  $k_d$ . This is one purpose of the certificate  $\text{cert}_s$ . In particular, the attrs field contains information that the client can use to determine the identity—for example, the domain name—of the server with which it is communicating. For applications in which the server also needs information about the client, TLS supports an option by which a client can send a certificate to the server.

In addition to its use on the Internet, TLS is being used for a wide variety of tasks. For example, we mentioned earlier that IPSec is widely used as the basis for virtual private networks, or VPNs. IPSec VPNs now have a competitor in TLS VPNs. IPSec is good for point-to-point encryption of traffic—say, between two company offices. TLS VPNs are more flexible but not as efficient, so they might be used between an individual employee working remotely and the corporate office.

## 16.5 User Authentication

Our earlier discussion of authentication involves messages and sessions. But what about users? If a system cannot authenticate a user, then authenticating that a message came from that user is pointless. Thus, a major security problem for operating systems is **user authentication**. The protection system depends on the ability to identify the programs and processes currently executing, which in turn depends on the ability to identify each user of the system. Users normally identify themselves, but how do we determine whether a user's identity is authentic? Generally, user authentication is based on one or more of three things: the user's possession of something (a key or card), the user's knowledge of something (a user identifier and password), or an attribute of the user (fingerprint, retina pattern, or signature).

### 16.5.1 Passwords

The most common approach to authenticating a user identity is the use of **passwords**. When the user identifies herself by user ID or account name, she is asked for a password. If the user-supplied password matches the password stored in the system, the system assumes that the account is being accessed by the owner of that account.

Passwords are often used to protect objects in the computer system, in the absence of more complete protection schemes. They can be considered a special case of either keys or capabilities. For instance, a password may be associated with each resource (such as a file). Whenever a request is made to use the resource, the password must be given. If the password is correct, access is granted. Different passwords may be associated with different access rights. For example, different passwords may be used for reading files, appending files, and updating files.

In practice, most systems require only one password for a user to gain their full rights. Although more passwords theoretically would be more secure, such systems tend not to be implemented due to the classic trade-off between security and convenience. If security makes something inconvenient, then the security is frequently bypassed or otherwise circumvented.

### 16.5.2 Password Vulnerabilities

Passwords are extremely common because they are easy to understand and use. Unfortunately, passwords can often be guessed, accidentally exposed, sniffed (read by an eavesdropper), or illegally transferred from an authorized user to an unauthorized one, as we show next.

There are three common ways to guess a password. One way is for the intruder (either human or program) to know the user or to have information about the user. All too frequently, people use obvious information (such as the names of their cats or spouses) as their passwords. Another way is to use brute force, trying enumeration—or all possible combinations of valid password characters (letters, numbers, and punctuation on some systems)—until the password is found. Short passwords are especially vulnerable to this method. For example, a four-character password provides only 10,000 variations. On average, guessing 5,000 times would produce a correct hit. A program that could try a password every millisecond would take only about 5 seconds to guess a four-character password. Enumeration is less successful where systems allow longer passwords that include both uppercase and lowercase letters, along with numbers and all punctuation characters. Of course, users must take advantage of the large password space and must not, for example, use only lowercase letters. The third, common method is dictionary attacks where all words, word variations, and common passwords are tried.

In addition to being guessed, passwords can be exposed as a result of visual or electronic monitoring. An intruder can look over the shoulder of a user (**shoulder surfing**) when the user is logging in and can learn the password easily by watching the keyboard. Alternatively, anyone with access to the network on which a computer resides can seamlessly add a network monitor, allowing him to **sniff**, or watch, all data being transferred on the network, including user IDs and passwords. Encrypting the data stream containing the password solves this problem. Even such a system could have passwords

stolen, however. For example, if a file is used to contain the passwords, it could be copied for off-system analysis. Or consider a Trojan-horse program installed on the system that captures every keystroke before sending it on to the application. Another common method to grab passwords, specially debit card passcodes, is installing physical devices where the codes are used and recording what the user does, for example a “skimmer” at an ATM machine or a device installed between the keyboard and the computer.

Exposure is a particularly severe problem if the password is written down where it can be read or lost. Some systems force users to select hard-to-remember or long passwords, or to change their password frequently, which may cause a user to record the password or to reuse it. As a result, such systems provide much less security than systems that allow users to select easy passwords!

The final type of password compromise, illegal transfer, is the result of human nature. Most computer installations have a rule that forbids users to share accounts. This rule is sometimes implemented for accounting reasons but is often aimed at improving security. For instance, suppose one user ID is shared by several users, and a security breach occurs from that user ID. It is impossible to know who was using the ID at the time the break occurred or even whether the user was an authorized one. With one user per user ID, any user can be questioned directly about use of the account; in addition, the user might notice something different about the account and detect the break-in. Sometimes, users break account-sharing rules to help friends or to circumvent accounting, and this behavior can result in a system’s being accessed by unauthorized users—possibly harmful ones.

Passwords can be either generated by the system or selected by a user. System-generated passwords may be difficult to remember, and thus users may write them down. As mentioned, however, user-selected passwords are often easy to guess (the user’s name or favorite car, for example). Some systems will check a proposed password for ease of guessing or cracking before accepting it. Some systems also *age* passwords, forcing users to change their passwords at regular intervals (every three months, for instance). This method is not foolproof either, because users can easily toggle between two passwords. The solution, as implemented on some systems, is to record a password history for each user. For instance, the system could record the last  $N$  passwords and not allow their reuse.

Several variants on these simple password schemes can be used. For example, the password can be changed more frequently. At the extreme, the password is changed from session to session. A new password is selected (either by the system or by the user) at the end of each session, and that password must be used for the next session. In such a case, even if a password is used by an unauthorized person, that person can use it only once. When the legitimate user tries to use a now-invalid password at the next session, he discovers the security violation. Steps can then be taken to repair the breached security.

### 16.5.3 Securing Passwords

One problem with all these approaches is the difficulty of keeping the password secret within the computer. How can the system store a password securely yet allow its use for authentication when the user presents her pass-

### STRONG AND EASY TO REMEMBER PASSWORDS

It is extremely important to use strong (hard to guess and hard to shoulder surf) passwords on critical systems like bank accounts. It is also important to not use the same password on lots of systems, as one less important, easily hacked system could reveal the password you use on more important systems. A good technique is to generate your password by using the first letter of each word of an easily remembered phrase using both upper and lower characters with a number or punctuation mark thrown in for good measure. For example, the phrase “My girlfriend’s name is Katherine” might yield the password “Mgn.isK!”. The password is hard to crack but easy for the user to remember. A more secure system would allow more characters in its passwords. Indeed, a system might also allow passwords to include the space character, so that a user could create a **passphrase** which is easy to remember but difficult to break.

word? The UNIX system uses secure hashing to avoid the necessity of keeping its password list secret. Because the password is hashed rather than encrypted, it is impossible for the system to decrypt the stored value and determine the original password.

Hash functions are easy to compute, but hard (if not impossible) to invert. That is, given a value  $x$ , it is easy to compute the hash function value  $f(x)$ . Given a function value  $f(x)$ , however, it is impossible to compute  $x$ . This function is used to encode all passwords. Only encoded passwords are stored. When a user presents a password, it is hashed and compared against the stored encoded password. Even if the stored encoded password is seen, it cannot be decoded, so the password cannot be determined. Thus, the password file does not need to be kept secret.

The drawback to this method is that the system no longer has control over the passwords. Although the passwords are hashed, anyone with a copy of the password file can run fast hash routines against it—hashing each word in a dictionary, for instance, and comparing the results against the passwords. If the user has selected a password that is also a word in the dictionary, the password is cracked. On sufficiently fast computers, or even on clusters of slow computers, such a comparison may take only a few hours. Furthermore, because systems use well-known hashing algorithms, an attacker might keep a cache of passwords that have been cracked previously.

For these reasons, systems include a “salt,” or recorded random number, in the hashing algorithm. The salt value is added to the password to ensure that if two plaintext passwords are the same, they result in different hash values. In addition, the salt value makes hashing a dictionary ineffective, because each dictionary term would need to be combined with each salt value for comparison to the stored passwords. Newer versions of UNIX also store the hashed password entries in a file readable only by the superuser. The programs that compare the hash to the stored value run *setuid* to root, so they can read this file, but other users cannot.

#### 16.5.4 One-Time Passwords

To avoid the problems of password sniffing and shoulder surfing, a system can use a set of **paired passwords**. When a session begins, the system randomly selects and presents one part of a password pair; the user must supply the other part. In this system, the user is *challenged* and must *respond* with the correct answer to that challenge.

This approach can be generalized to the use of an algorithm as a password. In this scheme, the system and the user share a symmetric password. The password  $pw$  is never transmitted over a medium that allows exposure. Rather, the password is used as input to a function, along with a *challenge*  $ch$  presented by the system. The user then computes the function  $H(pw, ch)$ . The result of this function is transmitted as the authenticator to the computer. Because the computer also knows  $pw$  and  $ch$ , it can perform the same computation. If the results match, the user is authenticated. The next time the user needs to be authenticated, another  $ch$  is generated, and the same steps ensue. This time, the authenticator is different. Such algorithmic passwords are not susceptible to reuse. That is, a user can type in a password, and no entity intercepting that password will be able to reuse it. This **one-time password** system is one of only a few ways to prevent improper authentication due to password exposure.

One-time password systems are implemented in various ways. Commercial implementations use hardware calculators with a display or a numeric keypad. These calculators generally take the shape of a credit card, a key-chain dongle, or a USB device. Software running on computers or smartphones provides the user with  $H(pw, ch)$ ;  $pw$  can be input by the user or generated by the calculator in synchronization with the computer. Sometimes,  $pw$  is just a **personal identification number (PIN)**. The output of any of these systems shows the one-time password. A one-time password generator that requires input by the user involves **two-factor authentication**. Two different types of components are needed in this case—for example, a one-time password generator that generates the correct response only if the PIN is valid. Two-factor authentication offers far better authentication protection than single-factor authentication because it requires “something you have” as well as “something you know.”

#### 16.5.5 Biometrics

Yet another variation on the use of passwords for authentication involves the use of biometric measures. Palm- or hand-readers are commonly used to secure physical access—for example, access to a data center. These readers match stored parameters against what is being read from hand-reader pads. The parameters can include a temperature map, as well as finger length, finger width, and line patterns. These devices are currently too large and expensive to be used for normal computer authentication.

Fingerprint readers have become accurate and cost-effective. These devices read finger ridge patterns and convert them into a sequence of numbers. Over time, they can store a set of sequences to adjust for the location of the finger on the reading pad and other factors. Software can then scan a finger on the pad and compare its features with these stored sequences to determine if they match. Of course, multiple users can have profiles stored, and the scanner can differentiate among them. A very accurate two-factor authentication scheme

can result from requiring a password as well as a user name and fingerprint scan. If this information is encrypted in transit, the system can be very resistant to spoofing or replay attack.

**Multifactor authentication** is better still. Consider how strong authentication can be with a USB device that must be plugged into the system, a PIN, and a fingerprint scan. Except for having to place one's finger on a pad and plug the USB into the system, this authentication method is no less convenient than that using normal passwords. Recall, though, that strong authentication by itself is not sufficient to guarantee the ID of the user. An authenticated session can still be hijacked if it is not encrypted.

## 16.6 Implementing Security Defenses

Just as there are myriad threats to system and network security, there are many security solutions. The solutions range from improved user education, through technology, to writing better software. Most security professionals subscribe to the theory of **defense in depth**, which states that more layers of defense are better than fewer layers. Of course, this theory applies to any kind of security. Consider the security of a house without a door lock, with a door lock, and with a lock and an alarm. In this section, we look at the major methods, tools, and techniques that can be used to improve resistance to threats. Note that some security-improving techniques are more properly part of protection than security and are covered in Chapter 17.

### 16.6.1 Security Policy

The first step toward improving the security of any aspect of computing is to have a **security policy**. Policies vary widely but generally include a statement of what is being secured. For example, a policy might state that all outside-accessible applications must have a code review before being deployed, or that users should not share their passwords, or that all connection points between a company and the outside must have port scans run every six months. Without a policy in place, it is impossible for users and administrators to know what is permissible, what is required, and what is not allowed. The policy is a road map to security, and if a site is trying to move from less secure to more secure, it needs a map to know how to get there.

Once the security policy is in place, the people it affects should know it well. It should be their guide. The policy should also be a **living document** that is reviewed and updated periodically to ensure that it is still pertinent and still followed.

### 16.6.2 Vulnerability Assessment

How can we determine whether a security policy has been correctly implemented? The best way is to execute a vulnerability assessment. Such assessments can cover broad ground, from social engineering through risk assessment to port scans. **Risk assessment**, for example, attempts to value the assets of the entity in question (a program, a management team, a system, or a facility) and determine the odds that a security incident will affect the entity and

decrease its value. When the odds of suffering a loss and the amount of the potential loss are known, a value can be placed on trying to secure the entity.

The core activity of most vulnerability assessments is a **penetration test**, in which the entity is scanned for known vulnerabilities. Because this book is concerned with operating systems and the software that runs on them, we concentrate on those aspects of vulnerability assessment.

Vulnerability scans typically are done at times when computer use is relatively low, to minimize their impact. When appropriate, they are done on test systems rather than production systems, because they can induce unhappy behavior from the target systems or network devices.

A scan within an individual system can check a variety of aspects of the system:

- Short or easy-to-guess passwords
- Unauthorized privileged programs, such as setuid programs
- Unauthorized programs in system directories
- Unexpectedly long-running processes
- Improper directory protections on user and system directories
- Improper protections on system data files, such as the password file, device files, or the operating-system kernel itself
- Dangerous entries in the program search path (for example, the Trojan horse discussed in Section 16.2.1), such as the current directory and any easily-written directories such as /tmp
- Changes to system programs detected with checksum values
- Unexpected or hidden network daemons

Any problems found by a security scan can be either fixed automatically or reported to the managers of the system.

Networked computers are much more susceptible to security attacks than are standalone systems. Rather than attacks from a known set of access points, such as directly connected terminals, we face attacks from an unknown and large set of access points—a potentially severe security problem. To a lesser extent, systems connected to telephone lines via modems are also more exposed.

In fact, the U.S. government considers a system to be only as secure as its most far-reaching connection. For instance, a top-secret system may be accessed only from within a building also considered top-secret. The system loses its top-secret rating if any form of communication can occur outside that environment. Some government facilities take extreme security precautions. The connectors that plug a terminal into the secure computer are locked in a safe in the office when the terminal is not in use. A person must have proper ID to gain access to the building and her office, must know a physical lock combination, and must know authentication information for the computer itself to gain access to the computer—an example of multifactor authentication.

Unfortunately for system administrators and computer-security professionals, it is frequently impossible to lock a machine in a room and disallow

all remote access. For instance, the Internet currently connects billions of computers and devices and has become a mission-critical, indispensable resource for many companies and individuals. If you consider the Internet a club, then, as in any club with millions of members, there are many good members and some bad members. The bad members have many tools they can use to attempt to gain access to the interconnected computers.

Vulnerability scans can be applied to networks to address some of the problems with network security. The scans search a network for ports that respond to a request. If services are enabled that should not be, access to them can be blocked, or they can be disabled. The scans then determine the details of the application listening on that port and try to determine if it has any known vulnerabilities. Testing those vulnerabilities can determine if the system is misconfigured or lacks needed patches.

Finally, though, consider the use of port scanners in the hands of an attacker rather than someone trying to improve security. These tools could help attackers find vulnerabilities to attack. (Fortunately, it is possible to detect port scans through anomaly detection, as we discuss next.) It is a general challenge to security that the same tools can be used for good and for harm. In fact, some people advocate **security through obscurity**, stating that no tools should be written to test security, because such tools can be used to find (and exploit) security holes. Others believe that this approach to security is not a valid one, pointing out, for example, that attackers could write their own tools. It seems reasonable that security through obscurity be considered one of the layers of security only so long as it is not the only layer. For example, a company could publish its entire network configuration, but keeping that information secret makes it harder for intruders to know what to attack. Even here, though, a company assuming that such information will remain a secret has a false sense of security.

### 16.6.3 Intrusion Prevention

Securing systems and facilities is intimately linked to intrusion detection and prevention. **Intrusion prevention**, as its name suggests, strives to detect attempted or successful intrusions into computer systems and to initiate appropriate responses to the intrusions. Intrusion prevention encompasses a wide array of techniques that vary on a number of axes, including the following:

- The time at which detection occurs. Detection can occur in real time (while the intrusion is occurring) or after the fact.
- The types of inputs examined to detect intrusive activity. These may include user-shell commands, process system calls, and network packet headers or contents. Some forms of intrusion might be detected only by correlating information from several such sources.
- The range of response capabilities. Simple forms of response include alerting an administrator to the potential intrusion or somehow halting the potentially intrusive activity—for example, killing a process engaged in such activity. In a sophisticated form of response, a system might transparently divert an intruder's activity to a **honeypot**—a false resource exposed

to the attacker. The resource appears real to the attacker and enables the system to monitor and gain information about the attack.

These degrees of freedom in the design space for detecting intrusions have yielded a wide range of solutions, known as **intrusion-prevention systems (IPS)**. IPSS act as self-modifying firewalls, passing traffic unless an intrusion is detected (at which point that traffic is blocked).

But just what constitutes an intrusion? Defining a suitable specification of intrusion turns out to be quite difficult, and thus automatic IPSSs today typically settle for one of two less ambitious approaches. In the first, called **signature-based detection**, system input or network traffic is examined for specific behavior patterns (or **signatures**) known to indicate attacks. A simple example of signature-based detection is scanning network packets for the string “/etc/passwd” targeted for a UNIX system. Another example is virus-detection software, which scans binaries or network packets for known viruses.

The second approach, typically called **anomaly detection**, attempts through various techniques to detect anomalous behavior within computer systems. Of course, not all anomalous system activity indicates an intrusion, but the presumption is that intrusions often induce anomalous behavior. An example of anomaly detection is monitoring system calls of a daemon process to detect whether the system-call behavior deviates from normal patterns, possibly indicating that a buffer overflow has been exploited in the daemon to corrupt its behavior. Another example is monitoring shell commands to detect anomalous commands for a given user or detecting an anomalous login time for a user, either of which may indicate that an attacker has succeeded in gaining access to that user’s account.

Signature-based detection and anomaly detection can be viewed as two sides of the same coin. Signature-based detection attempts to characterize dangerous behaviors and to detect when one of these behaviors occurs, whereas anomaly detection attempts to characterize normal (or nondangerous) behaviors and to detect when something other than these behaviors occurs.

These different approaches yield IPSSs with very different properties, however. In particular, anomaly detection can find previously unknown methods of intrusion (so-called **zero-day attacks**). Signature-based detection, in contrast, will identify only known attacks that can be codified in a recognizable pattern. Thus, new attacks that were not contemplated when the signatures were generated will evade signature-based detection. This problem is well known to vendors of virus-detection software, who must release new signatures with great frequency as new viruses are detected manually.

Anomaly detection is not necessarily superior to signature-based detection, however. Indeed, a significant challenge for systems that attempt anomaly detection is to benchmark “normal” system behavior accurately. If the system has already been penetrated when it is benchmarked, then the intrusive activity may be included in the “normal” benchmark. Even if the system is benchmarked cleanly, without influence from intrusive behavior, the benchmark must give a fairly complete picture of normal behavior. Otherwise, the number of **false positives** (false alarms) or, worse, **false negatives** (missed intrusions) will be excessive.

To illustrate the impact of even a marginally high rate of false alarms, consider an installation consisting of a hundred UNIX workstations from which

security-relevant events are recorded for purposes of intrusion detection. A small installation such as this could easily generate a million audit records per day. Only one or two might be worthy of an administrator's investigation. If we suppose, optimistically, that each actual attack is reflected in ten audit records, we can roughly compute the rate of occurrence of audit records reflecting truly intrusive activity as follows:

$$\frac{\frac{2 \text{ intrusions}}{\text{day}} \cdot 10 \frac{\text{records}}{\text{intrusion}}}{\frac{10^6 \text{ records}}{\text{day}}} = 0.00002.$$

Interpreting this as a “probability of occurrence of intrusive records,” we denote it as  $P(I)$ ; that is, event  $I$  is the occurrence of a record reflecting truly intrusive behavior. Since  $P(I) = 0.00002$ , we also know that  $P(\neg I) = 1 - P(I) = 0.99998$ . Now we let  $A$  denote the raising of an alarm by an IDS. An accurate IDS should maximize both  $P(I|A)$  and  $P(\neg I|\neg A)$ —that is, the probabilities that an alarm indicates an intrusion and that no alarm indicates no intrusion. Focusing on  $P(I|A)$  for the moment, we can compute it using [Bayes' theorem](#):

$$\begin{aligned} P(I|A) &= \frac{P(I) \cdot P(A|I)}{P(I) \cdot P(A|I) + P(\neg I) \cdot P(A|\neg I)} \\ &= \frac{0.00002 \cdot P(A|I)}{0.00002 \cdot P(A|I) + 0.99998 \cdot P(A|\neg I)} \end{aligned}$$

Now consider the impact of the false-alarm rate  $P(A|\neg I)$  on  $P(I|A)$ . Even with a very good true-alarm rate of  $P(A|I) = 0.8$ , a seemingly good false-alarm rate of  $P(A|\neg I) = 0.0001$  yields  $P(I|A) \approx 0.14$ . That is, fewer than one in every seven alarms indicates a real intrusion! In systems where a security administrator investigates each alarm, a high rate of false alarms—called a “Christmas tree effect”—is exceedingly wasteful and will quickly teach the administrator to ignore alarms.

This example illustrates a general principle for IPSs: for usability, they must offer an extremely low false-alarm rate. Achieving a sufficiently low false-alarm rate is an especially serious challenge for anomaly-detection systems, as mentioned, because of the difficulties of adequately benchmarking normal system behavior. However, research continues to improve anomaly-detection techniques. Intrusion-detection software is evolving to implement signatures, anomaly algorithms, and other algorithms and to combine the results to arrive at a more accurate anomaly-detection rate.

#### 16.6.4 Virus Protection

As we have seen, viruses can and do wreak havoc on systems. Protection from viruses thus is an important security concern. Antivirus programs are often used to provide this protection. Some of these programs are effective against only particular known viruses. They work by searching all the programs on a system for the specific pattern of instructions known to make up the virus.

When they find a known pattern, they remove the instructions, **disinfecting** the program. Antivirus programs may have catalogs of thousands of viruses for which they search.

Both viruses and antivirus software continue to become more sophisticated. Some viruses modify themselves as they infect other software to avoid the basic pattern-match approach of antivirus programs. Antivirus programs in turn now look for families of patterns rather than a single pattern to identify a virus. In fact, some antivirus programs implement a variety of detection algorithms. They can decompress compressed viruses before checking for a signature. Some also look for process anomalies. A process opening an executable file for writing is suspicious, for example, unless it is a compiler. Another popular technique is to run a program in a **sandbox** (Section 17.11.3), which is a controlled or emulated section of the system. The antivirus software analyzes the behavior of the code in the sandbox before letting it run unmonitored. Some antivirus programs also put up a complete shield rather than just scanning files within a file system. They search boot sectors, memory, inbound and outbound e-mail, files as they are downloaded, files on removable devices or media, and so on.

The best protection against computer viruses is prevention, or the practice of **safe computing**. Purchasing unopened software from vendors and avoiding free or pirated copies from public sources or disk exchange offer the safest route to preventing infection. However, even new copies of legitimate software applications are not immune to virus infection: in a few cases, disgruntled employees of a software company have infected the master copies of software programs to do economic harm to the company. Likewise, hardware devices can come from the factory pre-infected for your convenience. For macro viruses, one defense is to exchange Microsoft Word documents in an alternative file format called **rich text format (RTF)**. Unlike the native Word format, RTF does not include the capability to attach macros.

Another defense is to avoid opening any e-mail attachments from unknown users. Unfortunately, history has shown that e-mail vulnerabilities appear as fast as they are fixed. For example, in 2000, the *love bug* virus became very widespread by traveling in e-mail messages that pretended to be love notes sent by friends of the receivers. Once a receiver opened the attached Visual Basic script, the virus propagated by sending itself to the first addresses in the receiver's e-mail contact list. Fortunately, except for clogging e-mail systems and users' inboxes, it was relatively harmless. It did, however, effectively negate the defensive strategy of opening attachments only from people known to the receiver. A more effective defense method is to avoid opening any e-mail attachment that contains executable code. Some companies now enforce this as policy by removing all incoming attachments to e-mail messages.

Another safeguard, although it does not prevent infection, does permit early detection. A user must begin by completely reformatting the hard disk, especially the boot sector, which is often targeted for viral attack. Only secure software is uploaded, and a signature of each program is taken via a secure message-digest computation. The resulting file name and associated message-digest list must then be kept free from unauthorized access. Periodically, or each time a program is run, the operating system recomputes the signature

and compares it with the signature on the original list; any differences serve as a warning of possible infection. This technique can be combined with others. For example, a high-overhead antivirus scan, such as a sandbox, can be used; and if a program passes the test, a signature can be created for it. If the signatures match the next time the program is run, it does not need to be virus-scanned again.

### 16.6.5 Auditing, Accounting, and Logging

Auditing, accounting, and logging can decrease system performance, but they are useful in several areas, including security. Logging can be general or specific. All system-call executions can be logged for analysis of program behavior (or misbehavior). More typically, suspicious events are logged. Authentication failures and authorization failures can tell us quite a lot about break-in attempts.

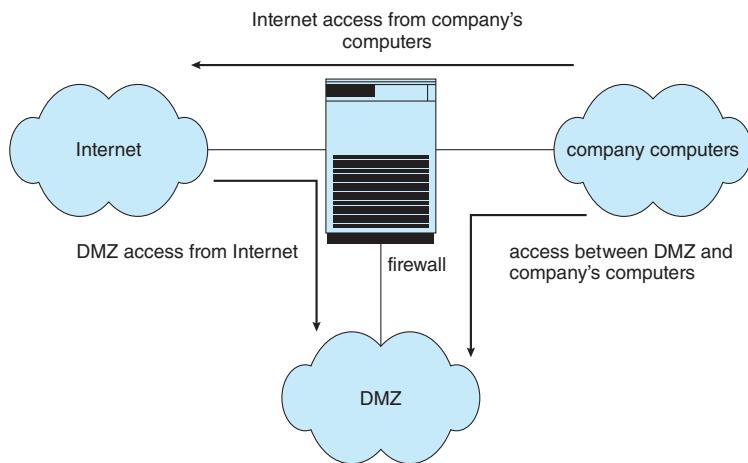
Accounting is another potential tool in a security administrator's kit. It can be used to find performance changes, which in turn can reveal security problems. One of the early UNIX computer break-ins was detected by Cliff Stoll when he was examining accounting logs and spotted an anomaly.

### 16.6.6 Firewalling to Protect Systems and Networks

We turn next to the question of how a trusted computer can be connected safely to an untrustworthy network. One solution is the use of a firewall to separate trusted and untrusted systems. A **firewall** is a computer, appliance, process, or router that sits between the trusted and the untrusted. A network firewall limits network access between the multiple **security domains** and monitors and logs all connections. It can also limit connections based on source or destination address, source or destination port, or direction of the connection. For instance, web servers use HTTP to communicate with web browsers. A firewall therefore may allow only HTTP to pass from all hosts outside the firewall to the web server within the firewall. The first worm, the Morris Internet worm, used the finger protocol to break into computers, so finger would not be allowed to pass, for example.

In fact, a network firewall can separate a network into multiple domains. A common implementation has the Internet as the untrusted domain; a semitrusted and semisecure network, called the **demilitarized zone (DMZ)**, as another domain; and a company's computers as a third domain (Figure 16.10). Connections are allowed from the Internet to the DMZ computers and from the company computers to the Internet but are not allowed from the Internet or DMZ computers to the company computers. Optionally, controlled communications may be allowed between the DMZ and one company computer or more. For instance, a web server on the DMZ may need to query a database server on the corporate network. With a firewall, however, access is contained, and any DMZ systems that are broken into still are unable to access the company computers.

Of course, a firewall itself must be secure and attack-proof. Otherwise, its ability to secure connections can be compromised. Furthermore, firewalls do not prevent attacks that **tunnel**, or travel within protocols or connections



**Figure 16.10** Domain separation via firewall.

that the firewall allows. A buffer-overflow attack to a web server will not be stopped by the firewall, for example, because the HTTP connection is allowed; it is the contents of the HTTP connection that house the attack. Likewise, denial-of-service attacks can affect firewalls as much as any other machines. Another vulnerability of firewalls is spoofing, in which an unauthorized host pretends to be an authorized host by meeting some authorization criterion. For example, if a firewall rule allows a connection from a host and identifies that host by its IP address, then another host could send packets using that same address and be allowed through the firewall.

In addition to the most common network firewalls, there are other, newer kinds of firewalls, each with its pros and cons. A **personal firewall** is a software layer either included with the operating system or added as an application. Rather than limiting communication between security domains, it limits communication to (and possibly from) a given host. A user could add a personal firewall to her PC so that a Trojan horse would be denied access to the network to which the PC is connected, for example. An **application proxy firewall** understands the protocols that applications speak across the network. For example, SMTP is used for mail transfer. An application proxy accepts a connection just as an SMTP server would and then initiates a connection to the original destination SMTP server. It can monitor the traffic as it forwards the message, watching for and disabling illegal commands, attempts to exploit bugs, and so on. Some firewalls are designed for one specific protocol. An **XML firewall**, for example, has the specific purpose of analyzing XML traffic and blocking disallowed or malformed XML. **System-call firewalls** sit between applications and the kernel, monitoring system-call execution. For example, in Solaris 10, the “least privilege” feature implements a list of more than fifty system calls that processes may or may not be allowed to make. A process that does not need to spawn other processes can have that ability taken away, for instance.

### 16.6.7 Other Solutions

In the ongoing battle between CPU designers, operating system implementers, and hackers, one particular technique has been helpful to defend against code injection. To mount a code-injection attack, hackers must be able to deduce the exact address in memory of their target. Normally, this may not be difficult, since memory layout tends to be predictable. An operating system technique called **Address Space Layout Randomization (ASLR)** attempts to solve this problem by randomizing address spaces—that is, putting address spaces, such as the starting locations of the stack and heap, in unpredictable locations. Address randomization, although not foolproof, makes exploitation considerably more difficult. ASLR is a standard feature in many operating systems, including Windows, Linux, and macOS.

In mobile operating systems such as iOS and Android, an approach often adopted is to place the user data and the system files into two separate partitions. The system partition is mounted read-only, whereas the data partition is read-write. This approach has numerous advantages, not the least of which is greater security: the system partition files cannot easily be tampered with, bolstering system integrity. Android takes this a step further by using Linux's dm-verity mechanism to cryptographically hash the system partition and detect any modifications.

### 16.6.8 Security Defenses Summarized

By applying appropriate layers of defense, we can keep systems safe from all but the most persistent attackers. In summary, these layers may include the following:

- Educate users about safe computing—don't attach devices of unknown origin to the computer, don't share passwords, use strong passwords, avoid falling for social engineering appeals, realize that an e-mail is not necessarily a private communication, and so on
- Educate users about how to prevent phishing attacks—don't click on e-mail attachments or links from unknown (or even known) senders; authenticate (for example, via a phone call) that a request is legitimate.
- Use secure communication when possible.
- Physically protect computer hardware.
- Configure the operating system to minimize the attack surface; disable all unused services.
- Configure system daemons, privileges applications, and services to be as secure as possible.
- Use modern hardware and software, as they are likely to have up-to-date security features.
- Keep systems and applications up to date and patched.
- Only run applications from trusted sources (such as those that are code signed).

- Enable logging and auditing; review the logs periodically, or automate alerts.
- Install and use antivirus software on systems susceptible to viruses, and keep the software up to date.
- Use strong passwords and passphrases, and don't record them where they could be found.
- Use intrusion detection, firewalling, and other network-based protection systems as appropriate.
- For important facilities, use periodic vulnerability assessments and other testing methods to test security and response to incidents.
- Encrypt mass-storage devices, and consider encrypting important individual files as well.
- Have a security policy for important systems and facilities, and keep it up to date

## 16.7 An Example: Windows 10

Microsoft Windows 10 is a general-purpose operating system designed to support a variety of security features and methods. In this section, we examine features that Windows 10 uses to perform security functions. For more information and background on Windows, see Appendix B.

The Windows 10 security model is based on the notion of **user accounts**. Windows 10 allows the creation of any number of user accounts, which can be grouped in any manner. Access to system objects can then be permitted or denied as desired. Users are identified to the system by a *unique* security ID. When a user logs on, Windows 10 creates a **security access token** that includes the security ID for the user, security IDs for any groups of which the user is a member, and a list of any special privileges that the user has. Examples of special privileges include backing up files and directories, shutting down the computer, logging on interactively, and changing the system clock. Every process that Windows 10 runs on behalf of a user will receive a copy of the access token. The system uses the security IDs in the access token to permit or deny access to system objects whenever the user, or a process on behalf of the user, attempts to access the object. Authentication of a user account is typically accomplished via a user name and password, although the modular design of Windows 10 allows the development of custom authentication packages. For example, a retinal (or eye) scanner might be used to verify that the user is who she says she is.

Windows 10 uses the idea of a subject to ensure that programs run by a user do not get greater access to the system than the user is authorized to have. A **subject** is used to track and manage permissions for each program that a user runs. It is composed of the user's access token and the program acting on behalf of the user. Since Windows 10 operates with a client–server model, two classes of subjects are used to control access: simple subjects and server subjects. An example of a **simple subject** is the typical application program that a user executes after she logs on. The simple subject is assigned a **security**

**context** based on the security access token of the user. A **server subject** is a process implemented as a protected server that uses the security context of the client when acting on the client's behalf.

As mentioned in Section 16.6.6, auditing is a useful security technique. Windows 10 has built-in auditing that allows many common security threats to be monitored. Examples include failure auditing for login and logoff events to detect random password break-ins, success auditing for login and logoff events to detect login activity at strange hours, success and failure write-access auditing for executable files to track a virus outbreak, and success and failure auditing for file access to detect access to sensitive files.

Windows Vista added mandatory integrity control, which works by assigning an **integrity label** to each securable object and subject. In order for a given subject to have access to an object, it must have the access requested in the discretionary access-control list, and its integrity label must be equal to or higher than that of the secured object (for the given operation). The integrity labels in Windows 7 are: untrusted, low, medium, high, and system. In addition, three access mask bits are permitted for integrity labels: NoReadUp, NoWriteUp, and NoExecuteUp. NoWriteUp is automatically enforced, so a lower-integrity subject cannot perform a write operation on a higher-integrity object. However, unless explicitly blocked by the security descriptor, it can perform read or execute operations.

For securable objects without an explicit integrity label, a default label of medium is assigned. The label for a given subject is assigned during logon. For instance, a nonadministrative user will have an integrity label of medium. In addition to integrity labels, Windows Vista also added User Account Control (UAC), which represents an administrative account (not the built-in Administrators account) with two separate tokens. One, for normal usage, has the built-in Administrators group disabled and has an integrity label of medium. The other, for elevated usage, has the built-in Administrators group enabled and an integrity label of high.

Security attributes of an object in Windows 10 are described by a **security descriptor**. The security descriptor contains the security ID of the owner of the object (who can change the access permissions), a group security ID used only by the POSIX subsystem, a discretionary access-control list that identifies which users or groups are allowed (and which are explicitly denied) access, and a system access-control list that controls which auditing messages the system will generate. Optionally, the system access-control list can set the integrity of the object and identify which operations to block from lower-integrity subjects: read, write (always enforced), or execute. For example, the security descriptor of the file `foo.bar` might have owner `gwen` and this discretionary access-control list:

- owner `gwen`—all access
- group `cs`—read–write access
- user `maddie`—no access

In addition, it might have a system access-control list that tells the system to audit writes by everyone, along with an integrity label of medium that denies read, write, and execute to lower-integrity subjects.

An access-control list is composed of access-control entries that contain the security ID of the individual or group being granted access and an access mask that defines all possible actions on the object, with a value of AccessAllowed or AccessDenied for each action. Files in Windows 10 may have the following access types: ReadData, WriteData, AppendData, Execute, ReadExtendedAttribute, WriteExtendedAttribute, ReadAttributes, and WriteAttributes. We can see how this allows a fine degree of control over access to objects.

Windows 10 classifies objects as either container objects or noncontainer objects. **Container objects**, such as directories, can logically contain other objects. By default, when an object is created within a container object, the new object inherits permissions from the parent object. Similarly, if the user copies a file from one directory to a new directory, the file will inherit the permissions of the destination directory. **Noncontainer objects** inherit no other permissions. Furthermore, if a permission is changed on a directory, the new permissions do not automatically apply to existing files and subdirectories; the user may explicitly apply them if he so desires.

The system administrator can use the Windows 10 Performance Monitor to help her spot approaching problems. In general, Windows 10 does a good job of providing features to help ensure a secure computing environment. Many of these features are not enabled by default, however, which may be one reason for the myriad security breaches on Windows 10 systems. Another reason is the vast number of services Windows 10 starts at system boot time and the number of applications that typically are installed on a Windows 10 system. For a real multiuser environment, the system administrator should formulate a security plan and implement it, using the features that Windows 10 provides and other security tools.

One feature differentiating security in Windows 10 from earlier versions is code signing. Some versions of Windows 10 make it mandatory—applications that are not properly signed by their authors will not execute—while other versions make it optional or leave it to the administrator to determine what to do with unsigned applications.

## 16.8 Summary

- Protection is an internal problem. Security, in contrast, must consider both the computer system and the environment—people, buildings, businesses, valuable objects, and threats—with which the system is used.
- The data stored in the computer system must be protected from unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency. It is easier to protect against accidental loss of data consistency than to protect against malicious access to the data. Absolute protection of the information stored in a computer system from malicious abuse is not possible; but the cost to the perpetrator can be made sufficiently high to deter most, if not all, attempts to access that information without proper authority.
- Several types of attacks can be launched against programs and against individual computers or the masses. Stack- and buffer-overflow tech-

niques allow successful attackers to change their level of system access. Viruses and malware require human interaction, while worms are self-perpetuating, sometimes infecting thousands of computers. Denial-of-service attacks prevent legitimate use of target systems.

- Encryption limits the domain of receivers of data, while authentication limits the domain of senders. Encryption is used to provide confidentiality of data being stored or transferred. Symmetric encryption requires a shared key, while asymmetric encryption provides a public key and a private key. Authentication, when combined with hashing, can prove that data have not been changed.
- User authentication methods are used to identify legitimate users of a system. In addition to standard user-name and password protection, several authentication methods are used. One-time passwords, for example, change from session to session to avoid replay attacks. Two-factor authentication requires two forms of authentication, such as a hardware calculator with an activation PIN, or one that presents a different response based on the time. Multifactor authentication uses three or more forms. These methods greatly decrease the chance of authentication forgery.
- Methods of preventing or detecting security incidents include an up-to-date security policy, intrusion-detection systems, antivirus software, auditing and logging of system events, system-call monitoring, code signing, sandboxing, and firewalls.

## Further Reading

Information about viruses and worms can be found at <http://www.securelist.com>, as well as in [Ludwig (1998)] and [Ludwig (2002)]. Another website containing up-to-date security information is <http://www.eeye.com/resources/security-center/research>. A paper on the dangers of a computer monoculture can be found at <http://cryptome.org/cyberinsecurity.htm>.

The first paper discussing least privilege is a Multics overview: <https://pdfs.semanticscholar.org/1c8d/06510ad449ad24fbdd164f8008cc730cab47.pdf>.

For the original article that explored buffer overflow attacks, see <http://phrack.org/issues/49/14.html>. For the development version control system git, see <https://github.com/git/>.

[C. Kaufman (2002)] and [Stallings and Brown (2011)] explore the use of cryptography in computer systems. Discussions concerning protection of digital signatures are offered by [Akl (1983)], [Davies (1983)], [Denning (1983)], and [Denning (1984)]. Complete cryptography information is presented in [Schneier (1996)] and [Katz and Lindell (2008)].

Asymmetric key encryption is discussed at <https://www-ee.stanford.edu/hellman/publications/24.pdf>). The TLS cryptographic protocol is described in detail at <https://tools.ietf.org/html/rfc5246>. The nmap network scanning tool is from <http://www.insecure.org/nmap/>. For more information on port scans

and how they are hidden, see <http://phrack.org/issues/49/15.html>. Nessus is a commercial vulnerability scanner but can be used for free with limited targets: <https://www.tenable.com/products/nessus-home>.

## Bibliography

- [**Akl (1983)**] S. G. Akl, “Digital Signatures: A Tutorial Survey”, *Computer*, Volume 16, Number 2 (1983), pages 15–24.
- [**C. Kaufman (2002)**] M. S. C. Kaufman, R. Perlman, *Network Security: Private Communication in a Public World*, Second Edition, Prentice Hall (2002).
- [**Davies (1983)**] D. W. Davies, “Applying the RSA Digital Signature to Electronic Mail”, *Computer*, Volume 16, Number 2 (1983), pages 55–62.
- [**Denning (1983)**] D. E. Denning, “Protecting Public Keys and Signature Keys”, *Computer*, Volume 16, Number 2 (1983), pages 27–35.
- [**Denning (1984)**] D. E. Denning, “Digital Signatures with RSA and Other Public-Key Cryptosystems”, *Communications of the ACM*, Volume 27, Number 4 (1984), pages 388–392.
- [**Katz and Lindell (2008)**] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*, Chapman & Hall/CRC Press (2008).
- [**Ludwig (1998)**] M. Ludwig, *The Giant Black Book of Computer Viruses*, Second Edition, American Eagle Publications (1998).
- [**Ludwig (2002)**] M. Ludwig, *The Little Black Book of Email Viruses*, American Eagle Publications (2002).
- [**Schneier (1996)**] B. Schneier, *Applied Cryptography*, Second Edition, John Wiley and Sons (1996).
- [**Stallings and Brown (2011)**] W. Stallings and L. Brown, *Computer Security: Principles and Practice*, Second Edition, Prentice Hall (2011).

## Chapter 16 Exercises

- 16.1 Buffer-overflow attacks can be avoided by adopting a better programming methodology or by using special hardware support. Discuss these solutions.
- 16.2 A password may become known to other users in a variety of ways. Is there a simple method for detecting that such an event has occurred? Explain your answer.
- 16.3 What is the purpose of using a “salt” along with a user-provided password? Where should the salt be stored, and how should it be used?
- 16.4 The list of all passwords is kept in the operating system. Thus, if a user manages to read this list, password protection is no longer provided. Suggest a scheme that will avoid this problem. (Hint: Use different internal and external representations.)
- 16.5 An experimental addition to UNIX allows a user to connect a **watchdog** program to a file. The watchdog is invoked whenever a program requests access to the file. The watchdog then either grants or denies access to the file. Discuss two pros and two cons of using watchdogs for security.
- 16.6 Discuss a means by which managers of systems connected to the Internet could design their systems to limit or eliminate the damage done by worms. What are the drawbacks of making the change that you suggest?
- 16.7 Make a list of six security concerns for a bank’s computer system. For each item on your list, state whether this concern relates to physical, human, or operating-system security.
- 16.8 What are two advantages of encrypting data stored in the computer system?
- 16.9 What commonly used computer programs are prone to man-in-the-middle attacks? Discuss solutions for preventing this form of attack.
- 16.10 Compare symmetric and asymmetric encryption schemes, and discuss the circumstances under which a distributed system would use one or the other.
- 16.11 Why doesn’t  $D_{kd,N}(E_{ke,N}(m))$  provide authentication of the sender? To what uses can such an encryption be put?
- 16.12 Discuss how the asymmetric encryption algorithm can be used to achieve the following goals.
  - a. Authentication: the receiver knows that only the sender could have generated the message.
  - b. Secrecy: only the receiver can decrypt the message.
  - c. Authentication and secrecy: only the receiver can decrypt the message, and the receiver knows that only the sender could have generated the message.

- 16.13** Consider a system that generates 10 million audit records per day. Assume that, on average, there are 10 attacks per day on this system and each attack is reflected in 20 records. If the intrusion-detection system has a true-alarm rate of 0.6 and a false-alarm rate of 0.0005, what percentage of alarms generated by the system corresponds to real intrusions?
- 16.14** Mobile operating systems such as iOS and Android place the user data and the system files into two separate partitions. Aside from security, what is an advantage of that separation?

# Protection



In Chapter 16, we addressed security, which involves guarding computer resources against unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency. In this chapter, we turn to protection, which involves controlling the access of processes and users to the resources defined by a computer system.

The processes in an operating system must be protected from one another's activities. To provide this protection, we can use various mechanisms to ensure that only processes that have gained proper authorization from the operating system can operate on the files, memory segments, CPU, networking, and other resources of a system. These mechanisms must provide a means for specifying the controls to be imposed, together with a means of enforcement.

## CHAPTER OBJECTIVES

- Discuss the goals and principles of protection in a modern computer system.
- Explain how protection domains, combined with an access matrix, are used to specify the resources a process may access.
- Examine capability- and language-based protection systems.
- Describe how protection mechanisms can mitigate system attacks.

### 17.1 Goals of Protection

As computer systems have become more sophisticated and pervasive in their applications, the need to protect their integrity has also grown. Protection was originally conceived as an adjunct to multiprogramming operating systems, so that untrustworthy users might safely share a common logical name space, such as a directory of files, or a common physical name space, such as memory. Modern protection concepts have evolved to increase the reliability of any complex system that makes use of shared resources and is connected to insecure communications platforms such as the Internet.

We need to provide protection for several reasons. The most obvious is the need to prevent the mischievous, intentional violation of an access restriction by a user. Of more general importance, however, is the need to ensure that each process in a system uses system resources only in ways consistent with stated policies. This requirement is an absolute one for a reliable system.

Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by a malfunctioning subsystem. Also, an unprotected resource cannot defend against use (or misuse) by an unauthorized or incompetent user. A protection-oriented system provides means to distinguish between authorized and unauthorized usage.

The role of protection in a computer system is to provide a mechanism for the enforcement of the policies governing resource use. These policies can be established in a variety of ways. Some are fixed in the design of the system, while others are formulated by the management of a system. Still others are defined by individual users to protect resources they “own.” A protection system, then, must have the flexibility to enforce a variety of policies.

Policies for resource use may vary by application, and they may change over time. For these reasons, protection is no longer the concern solely of the designer of an operating system. The application programmer needs to use protection mechanisms as well, to guard resources created and supported by an application subsystem against misuse. In this chapter, we describe the protection mechanisms the operating system should provide, but application designers can use them as well in designing their own protection software.

Note that *mechanisms* are distinct from *policies*. Mechanisms determine *how* something will be done; policies decide *what* will be done. The separation of policy and mechanism is important for flexibility. Policies are likely to change from place to place or time to time. In the worst case, every change in policy would require a change in the underlying mechanism. Using general mechanisms enables us to avoid such a situation.

## 17.2 Principles of Protection

Frequently, a guiding principle can be used throughout a project, such as the design of an operating system. Following this principle simplifies design decisions and keeps the system consistent and easy to understand. A key, time-tested guiding principle for protection is the **principle of least privilege**. As discussed in Chapter 16, this principle dictates that programs, users, and even systems be given just enough privileges to perform their tasks.

Consider one of the tenets of UNIX—that a user should not run as root. (In UNIX, only the root user can execute privileged commands.) Most users innately respect that, fearing an accidental delete operation for which there is no corresponding undelete. Because root is virtually omnipotent, the potential for human error when a user acts as root is grave, and its consequences far reaching.

Now consider that rather than human error, damage may result from malicious attack. A virus launched by an accidental click on an attachment is one example. Another is a buffer overflow or other code-injection attack that is successfully carried out against a root-privileged process (or, in Windows,

a process with administrator privileges). Either case could prove catastrophic for the system.

Observing the principle of least privilege would give the system a chance to mitigate the attack—if malicious code cannot obtain root privileges, there is a chance that adequately defined **permissions** may block all, or at least some, of the damaging operations. In this sense, permissions can act like an immune system at the operating-system level.

The principle of least privilege takes many forms, which we examine in more detail later in the chapter. Another important principle, often seen as a derivative of the principle of least privilege, is **compartmentalization**. Compartmentalization is the process of protecting each individual system component through the use of specific permissions and access restrictions. Then, if a component is subverted, another line of defense will “kick in” and keep the attacker from compromising the system any further. Compartmentalization is implemented in many forms—from network demilitarized zones (DMZs) through virtualization.

The careful use of access restrictions can help make a system more secure and can also be beneficial in producing an **audit trail**, which tracks divergences from allowed accesses. An audit trail is a hard record in the system logs. If monitored closely, it can reveal early warnings of an attack or (if its integrity is maintained despite an attack) provide clues as to which attack vectors were used, as well as accurately assess the damage caused.

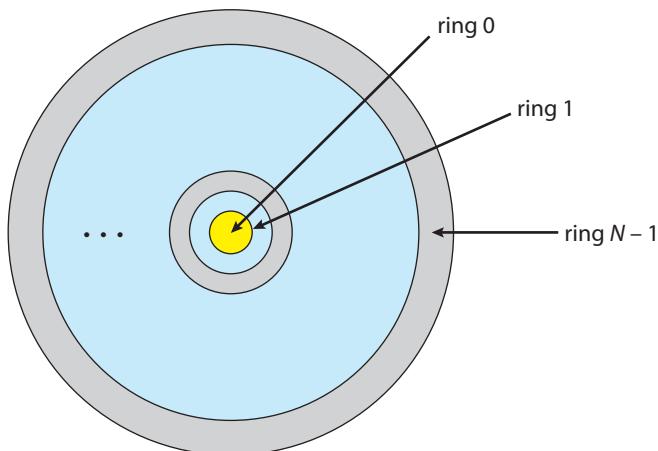
Perhaps most importantly, no single principle is a panacea for security vulnerabilities. **Defense in depth** must be used: multiple layers of protection should be applied one on top of the other (think of a castle with a garrison, a wall, and a moat to protect it). At the same time, of course, attackers use multiple means to bypass defense in depth, resulting in an ever-escalating arms race.

### 17.3 Protection Rings

As we’ve seen, the main component of modern operating systems is the kernel, which manages access to system resources and hardware. The kernel, by definition, is a trusted and privileged component and therefore must run with a higher level of privileges than user processes.

To carry out this **privilege separation**, hardware support is required. Indeed, all modern hardware supports the notion of separate execution levels, though implementations vary somewhat. A popular model of privilege separation is that of protection rings. In this model, fashioned after Bell–LaPadula (<https://www.acsac.org/2005/papers/Bell.pdf>), execution is defined as a set of concentric rings, with ring  $i$  providing a subset of the functionality of ring  $j$  for any  $j < i$ . The innermost ring, ring 0, thus provides the full set of privileges. This pattern is shown in Figure 17.1.

When the system boots, it boots to the highest privilege level. Code at that level performs necessary initialization before dropping to a less privileged level. In order to return to a higher privilege level, code usually calls a special instruction, sometimes referred to as a gate, which provides a portal between rings. The `syscall` instruction (in Intel) is one example. Calling this instruction shifts execution from user to kernel mode. As we have seen, executing a system



**Figure 17.1** Protection-ring structure.

call will always transfer execution to a predefined address, allowing the caller to specify only arguments (including the system call number), and not arbitrary kernel addresses. In this way, the integrity of the more privileged ring can generally be assured.

Another way of ending up in a more privileged ring is on the occurrence of a processor trap or an interrupt. When either occurs, execution is immediately transferred into the higher-privilege ring. Once again, however, the execution in the higher-privilege ring is predefined and restricted to a well-guarded code path.

Intel architectures follow this model, placing user mode code in ring 3 and kernel mode code in ring 0. The distinction is made by two bits in the special EFLAGS register. Access to this register is not allowed in ring 3—thus preventing a malicious process from escalating privileges. With the advent of virtualization, Intel defined an additional ring (-1) to allow for **hypervisors**, or virtual machine managers, which create and run virtual machines. Hypervisors have more capabilities than the kernels of the guest operating systems.

The ARM processor's architecture initially allowed only USR and SVC mode, for user and kernel (supervisor) mode, respectively. In ARMv7 processors, ARM introduced **TrustZone (TZ)**, which provided an additional ring. This most privileged execution environment also has exclusive access to hardware-backed cryptographic features, such as the NFC Secure Element and an on-chip cryptographic key, that make handling passwords and sensitive information more secure. Even the kernel itself has no access to the on-chip key, and it can only request encryption and decryption services from the TrustZone environment (by means of a specialized instruction, **Secure Monitor Call (SMC)**), which is only usable from kernel mode. As with system calls, the kernel has no ability to directly execute to specific addresses in the TrustZone—only to pass arguments via registers. Android uses TrustZone extensively as of Version 5.0, as shown in Figure 17.2.

Correctly employing a trusted execution environment means that, if the kernel is compromised, an attacker can't simply retrieve the key from kernel memory. Moving cryptographic services to a separate, trusted environment

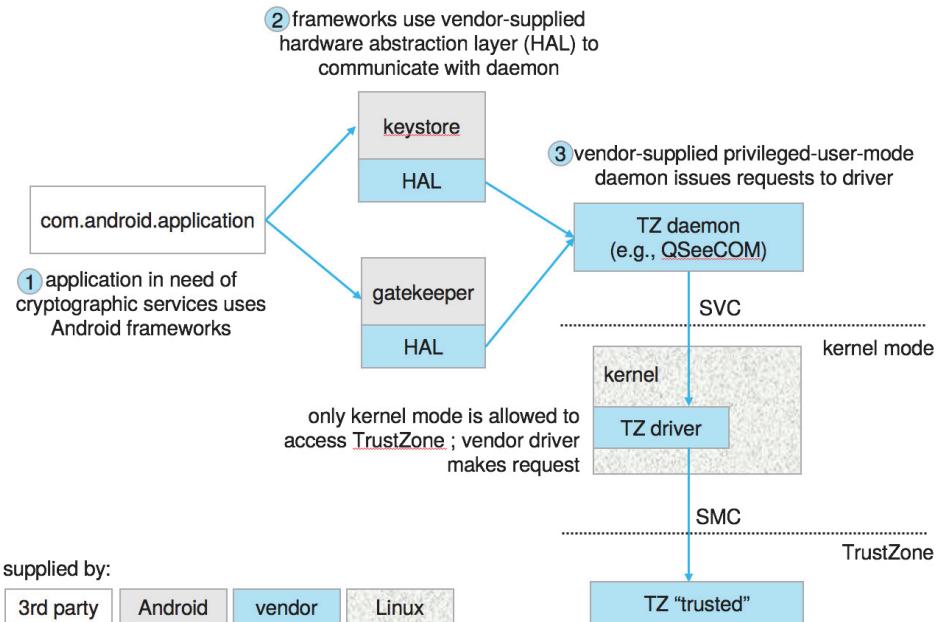


Figure 17.2 Android uses of TrustZone.

also makes brute-force attacks less likely to succeed. (As described in Chapter 16, these attacks involve trying all possible combinations of valid password characters until the password is found.) The various keys used by the system, from the user’s password to the system’s own, are stored in the on-chip key, which is only accessible in a trusted context. When a key—say, a password—is entered, it is verified via a request to the TrustZone environment. If a key is not known and must be guessed, the TrustZone verifier can impose limitations—by capping the number of verification attempts, for example.

In the 64-bit ARMv8 architecture, ARM extended its model to support four levels, called “exception levels,” numbered EL0 through EL3. User mode runs in EL0, and kernel mode in EL1. EL2 is reserved for hypervisors, and EL3 (the most privileged) is reserved for the secure monitor (the TrustZone layer). Any one of the exception levels allows running separate operating systems side by side, as shown in Figure 17.3.

Note that the secure monitor runs at a higher execution level than general-purpose kernels, which makes it the perfect place to deploy code that will check the kernels’ integrity. This functionality is included in Samsung’s Realtime Kernel Protection (RKP) for Android and Apple’s WatchTower (also known as KPP, for Kernel Patch Protection) for iOS.

## 17.4 Domain of Protection

Rings of protection separate functions into domains and order them hierarchically. A generalization of rings is using domains without a hierarchy. A computer system can be treated as a collection of processes and objects. By

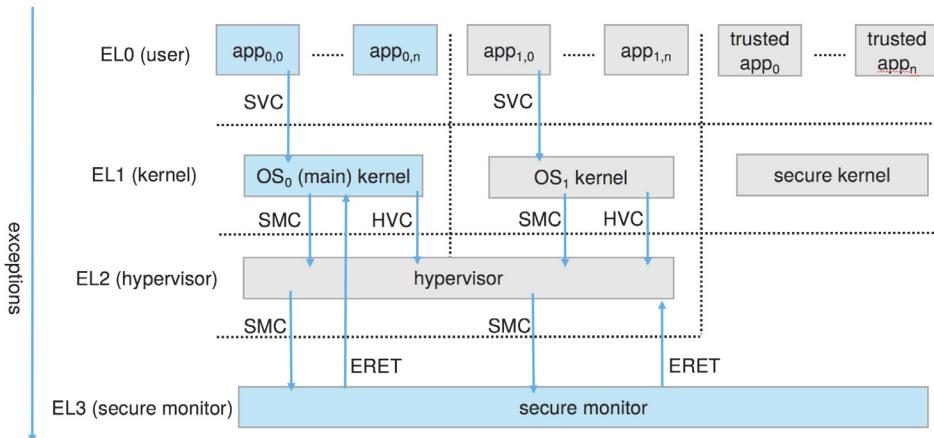


Figure 17.3 ARM architecture.

*objects*, we mean both **hardware objects** (such as the CPU, memory segments, printers, disks, and tape drives) and **software objects** (such as files, programs, and semaphores). Each object has a unique name that differentiates it from all other objects in the system, and each can be accessed only through well-defined and meaningful operations. Objects are essentially abstract data types.

The operations that are possible depend on the object. For example, on a CPU, we can only execute. Memory words can be read and written, whereas a DVD-ROM can only be read. Tape drives can be read, written, and rewound. Data files can be created, opened, read, written, closed, and deleted; program files can be read, written, executed, and deleted.

A process should be allowed to access only those objects for which it has authorization. Furthermore, at any time, a process should be able to access only those objects that it currently requires to complete its task. This second requirement, the **need-to-know principle**, is useful in limiting the amount of damage a faulty process or an attacker can cause in the system. For example, when process  $p$  invokes procedure  $A()$ , the procedure should be allowed to access only its own variables and the formal parameters passed to it; it should not be able to access all the variables of process  $p$ . Similarly, consider the case in which process  $p$  invokes a compiler to compile a particular file. The compiler should not be able to access files arbitrarily but should have access only to a well-defined subset of files (such as the source file, output object file, and so on) related to the file to be compiled. Conversely, the compiler may have private files used for accounting or optimization purposes that process  $p$  should not be able to access.

In comparing need-to-know with least privilege, it may be easiest to think of need-to-know as the policy and least privilege as the mechanism for achieving this policy. For example, in file permissions, need-to-know might dictate that a user have read access but not write or execute access to a file. The principle of least privilege would require that the operating system provide a mechanism to allow read but not write or execute access.

### 17.4.1 Domain Structure

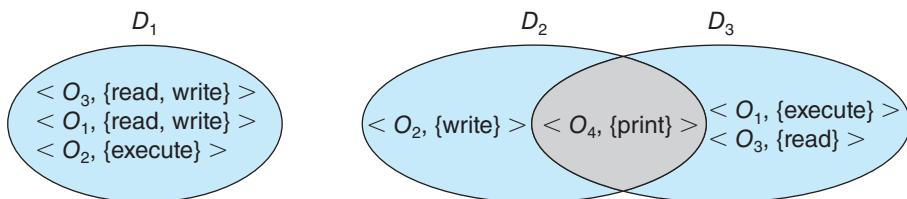
To facilitate the sort of scheme just described, a process may operate within a **protection domain**, which specifies the resources that the process may access. Each domain defines a set of objects and the types of operations that may be invoked on each object. The ability to execute an operation on an object is an **access right**. A domain is a collection of access rights, each of which is an ordered pair  $\langle \text{object-name}, \text{rights-set} \rangle$ . For example, if domain  $D$  has the access right  $\langle \text{file } F, \{\text{read, write}\} \rangle$ , then a process executing in domain  $D$  can both read and write file  $F$ . It cannot, however, perform any other operation on that object.

Domains may share access rights. For example, in Figure 17.4, we have three domains:  $D_1$ ,  $D_2$ , and  $D_3$ . The access right  $\langle O_4, \{\text{print}\} \rangle$  is shared by  $D_2$  and  $D_3$ , implying that a process executing in either of these two domains can print object  $O_4$ . Note that a process must be executing in domain  $D_1$  to read and write object  $O_1$ , while only processes in domain  $D_3$  may execute object  $O_1$ .

The association between a process and a domain may be either **static**, if the set of resources available to the process is fixed throughout the process's lifetime, or **dynamic**. As might be expected, establishing dynamic protection domains is more complicated than establishing static protection domains.

If the association between processes and domains is fixed, and we want to adhere to the need-to-know principle, then a mechanism must be available to change the content of a domain. The reason stems from the fact that a process may execute in two different phases and may, for example, need read access in one phase and write access in another. If a domain is static, we must define the domain to include both read and write access. However, this arrangement provides more rights than are needed in each of the two phases, since we have read access in the phase where we need only write access, and vice versa. Thus, the need-to-know principle is violated. We must allow the contents of a domain to be modified so that the domain always reflects the minimum necessary access rights.

If the association is dynamic, a mechanism is available to allow **domain switching**, enabling the process to switch from one domain to another. We may also want to allow the content of a domain to be changed. If we cannot change the content of a domain, we can provide the same effect by creating a new domain with the changed content and switching to that new domain when we want to change the domain content.



**Figure 17.4** System with three protection domains.

A domain can be realized in a variety of ways:

- Each *user* may be a domain. In this case, the set of objects that can be accessed depends on the identity of the user. Domain switching occurs when the user is changed—generally when one user logs out and another user logs in.
- Each *process* may be a domain. In this case, the set of objects that can be accessed depends on the identity of the process. Domain switching occurs when one process sends a message to another process and then waits for a response.
- Each *procedure* may be a domain. In this case, the set of objects that can be accessed corresponds to the local variables defined within the procedure. Domain switching occurs when a procedure call is made.

We discuss domain switching in greater detail in Section 17.5.

Consider the standard dual-mode (kernel–user mode) model of operating-system execution. When a process is in kernel mode, it can execute privileged instructions and thus gain complete control of the computer system. In contrast, when a process executes in user mode, it can invoke only nonprivileged instructions. Consequently, it can execute only within its predefined memory space. These two modes protect the operating system (executing in kernel domain) from the user processes (executing in user domain). In a multiprogrammed operating system, two protection domains are insufficient, since users also want to be protected from one another. Therefore, a more elaborate scheme is needed. We illustrate such a scheme by examining two influential operating systems—UNIX and Android—to see how they implement these concepts.

#### 17.4.2 Example: UNIX

As noted earlier, in UNIX, the root user can execute privileged commands, while other users cannot. Restricting certain operations to the root user can impair other users in their everyday operations, however. Consider, for example, a user who wants to change his password. Inevitably, this requires access to the password database (commonly, `/etc/shadow`), which can only be accessed by root. A similar challenge is encountered when setting a scheduled job (using the `at` command)—doing so requires access to privileged directories that are beyond the reach of a normal user.

The solution to this problem is the setuid bit. In UNIX, an owner identification and a domain bit, known as the *setuid bit*, are associated with each file. The setuid bit may or may not be enabled. When the bit is enabled on an executable file (through `chmod +s`), whoever executes the file temporarily assumes the identity of the file owner. That means if a user manages to create a file with the user ID “root” and the setuid bit enabled, anyone who gains access to execute the file becomes user “root” for the duration of the process’s lifetime.

If that strikes you as alarming, it is with good reason. Because of their potential power, setuid executable binaries are expected to be both sterile (affecting only necessary files under specific constraints) and hermetic (for example, tamperproof and impossible to subvert). Setuid programs need to

be very carefully written to make these assurances. Returning to the example of changing passwords, the `passwd` command is setuid-root and will indeed modify the password database, but only if first presented with the user's valid password, and it will then restrict itself to editing the password of that user and only that user.

Unfortunately, experience has repeatedly shown that few setuid binaries, if any, fulfill both criteria successfully. Time and again, setuid binaries have been subverted—some through race conditions and others through code injection—yielding instant root access to attackers. Attackers are frequently successful in achieving privilege escalation in this way. Methods of doing so are discussed in Chapter 16. Limiting damage from bugs in setuid programs is discussed in Section 17.8.

#### 17.4.3 Example: Android Application IDs

In Android, distinct user IDs are provided on a per-application basis. When an application is installed, the `installd` daemon assigns it a distinct user ID (UID) and group ID (GID), along with a private data directory (`/data/data/<app-name>`) whose ownership is granted to this UID/GID combination alone. In this way, applications on the device enjoy the same level of protection provided by UNIX systems to separate users. This is a quick and simple way to provide isolation, security, and privacy. The mechanism is extended by modifying the kernel to allow certain operations (such as networking sockets) only to members of a particular GID (for example, AID\_INET, 3003). A further enhancement by Android is to define certain UIDs as “isolated,” which prevents them from initiating RPC requests to any but a bare minimum of services.

## 17.5 Access Matrix

The general model of protection can be viewed abstractly as a matrix, called an **access matrix**. The rows of the access matrix represent domains, and the columns represent objects. Each entry in the matrix consists of a set of access rights. Because the column defines objects explicitly, we can omit the object name from the access right. The entry  $\text{access}(i,j)$  defines the set of operations that a process executing in domain  $D_i$  can invoke on object  $O_j$ .

To illustrate these concepts, we consider the access matrix shown in Figure 17.5. There are four domains and four objects—three files ( $F_1, F_2, F_3$ ) and one laser printer. A process executing in domain  $D_1$  can read files  $F_1$  and  $F_3$ . A process executing in domain  $D_4$  has the same privileges as one executing in domain  $D_1$ ; but in addition, it can also write onto files  $F_1$  and  $F_3$ . The laser printer can be accessed only by a process executing in domain  $D_2$ .

The access-matrix scheme provides us with the mechanism for specifying a variety of policies. The mechanism consists of implementing the access matrix and ensuring that the semantic properties we have outlined hold. More specifically, we must ensure that a process executing in domain  $D_i$  can access only those objects specified in row  $i$ , and then only as allowed by the access-matrix entries.

The access matrix can implement policy decisions concerning protection. The policy decisions involve which rights should be included in the  $(i,j)^{\text{th}}$  entry.

object domain \	$F_1$	$F_2$	$F_3$	printer
$D_1$	read		read	
$D_2$				print
$D_3$		read	execute	
$D_4$	read write		read write	

Figure 17.5 Access matrix.

We must also decide the domain in which each process executes. This last policy is usually decided by the operating system.

The users normally decide the contents of the access-matrix entries. When a user creates a new object  $O_j$ , the column  $O_j$  is added to the access matrix with the appropriate initialization entries, as dictated by the creator. The user may decide to enter some rights in some entries in column  $j$  and other rights in other entries, as needed.

The access matrix provides an appropriate mechanism for defining and implementing strict control for both static and dynamic association between processes and domains. When we switch a process from one domain to another, we are executing an operation (switch) on an object (the domain). We can control domain switching by including domains among the objects of the access matrix. Similarly, when we change the content of the access matrix, we are performing an operation on an object: the access matrix. Again, we can control these changes by including the access matrix itself as an object. Actually, since each entry in the access matrix can be modified individually, we must consider each entry in the access matrix as an object to be protected. Now, we need to consider only the operations possible on these new objects (domains and the access matrix) and decide how we want processes to be able to execute these operations.

Processes should be able to switch from one domain to another. Switching from domain  $D_i$  to domain  $D_j$  is allowed if and only if the access right  $\text{switch} \in \text{access}(i, j)$ . Thus, in Figure 17.6, a process executing in domain  $D_2$  can switch to domain  $D_3$  or to domain  $D_4$ . A process in domain  $D_4$  can switch to  $D_1$ , and one in domain  $D_1$  can switch to  $D_2$ .

Allowing controlled change in the contents of the access-matrix entries requires three additional operations: `copy`, `owner`, and `control`. We examine these operations next.

The ability to copy an access right from one domain (or row) of the access matrix to another is denoted by an asterisk (\*) appended to the access right. The `copy` right allows the access right to be copied only within the column (that is, for the object) for which the right is defined. For example, in Figure 17.7(a), a process executing in domain  $D_2$  can copy the read operation into any entry associated with file  $F_2$ . Hence, the access matrix of Figure 17.7(a) can be modified to the access matrix shown in Figure 17.7(b).

object domain \	$F_1$	$F_2$	$F_3$	laser printer	$D_1$	$D_2$	$D_3$	$D_4$
$D_1$	read		read			switch		
$D_2$				print			switch	switch
$D_3$		read	execute					
$D_4$	read write		read write		switch			

**Figure 17.6** Access matrix of Figure 17.5 with domains as objects.

This scheme has two additional variants:

1. A right is copied from  $\text{access}(i, j)$  to  $\text{access}(k, j)$ ; it is then removed from  $\text{access}(i, j)$ . This action is a transfer of a right, rather than a copy.
2. Propagation of the copy right may be limited. That is, when the right  $R^*$  is copied from  $\text{access}(i, j)$  to  $\text{access}(k, j)$ , only the right  $R$  (not  $R^*$ ) is created. A process executing in domain  $D_k$  cannot further copy the right  $R$ .

A system may select only one of these three copy rights, or it may provide all three by identifying them as separate rights: copy, transfer, and limited copy.

object domain \	$F_1$	$F_2$	$F_3$
$D_1$	execute		write*
$D_2$	execute	read*	execute
$D_3$	execute		

(a)

object domain \	$F_1$	$F_2$	$F_3$
$D_1$	execute		write*
$D_2$	execute	read*	execute
$D_3$	execute	read	

(b)

**Figure 17.7** Access matrix with **copy** rights.

We also need a mechanism to allow addition of new rights and removal of some rights. The **owner** right controls these operations. If  $\text{access}(i, j)$  includes the **owner** right, then a process executing in domain  $D_i$  can add and remove any right in any entry in column  $j$ . For example, in Figure 17.8(a), domain  $D_1$  is the owner of  $F_1$  and thus can add and delete any valid right in column  $F_1$ . Similarly, domain  $D_2$  is the owner of  $F_2$  and  $F_3$  and thus can add and remove any valid right within these two columns. Thus, the access matrix of Figure 17.8(a) can be modified to the access matrix shown in Figure 17.8(b).

The **copy** and **owner** rights allow a process to change the entries in a column. A mechanism is also needed to change the entries in a row. The **control** right is applicable only to domain objects. If  $\text{access}(i, j)$  includes the **control** right, then a process executing in domain  $D_i$  can remove any access right from row  $j$ . For example, suppose that, in Figure 17.6, we include the **control** right in  $\text{access}(D_2, D_4)$ . Then, a process executing in domain  $D_2$  could modify domain  $D_4$ , as shown in Figure 17.9.

The **copy** and **owner** rights provide us with a mechanism to limit the propagation of access rights. However, they do not give us the appropriate tools for preventing the propagation (or disclosure) of information. The problem of guaranteeing that no information initially held in an object can migrate outside of its execution environment is called the **confinement problem**. This problem is in general unsolvable (see the bibliographical notes at the end of the chapter).

object domain \	$F_1$	$F_2$	$F_3$
$D_1$	owner execute		write
$D_2$		read* owner	read* owner write
$D_3$	execute		

(a)

object domain \	$F_1$	$F_2$	$F_3$
$D_1$	owner execute		write
$D_2$		owner read* write*	read* owner write
$D_3$		write	write

(b)

**Figure 17.8** Access matrix with owner rights.

object domain \	$F_1$	$F_2$	$F_3$	laser printer	$D_1$	$D_2$	$D_3$	$D_4$
$D_1$	read		read			switch		
$D_2$				print			switch	switch control
$D_3$		read	execute					
$D_4$	write		write		switch			

**Figure 17.9** Modified access matrix of Figure 17.6.

These operations on the domains and the access matrix are not in themselves important, but they illustrate the ability of the access-matrix model to let us implement and control dynamic protection requirements. New objects and new domains can be created dynamically and included in the access-matrix model. However, we have shown only that the basic mechanism exists. System designers and users must make the policy decisions concerning which domains are to have access to which objects in which ways.

## 17.6 Implementation of the Access Matrix

How can the access matrix be implemented effectively? In general, the matrix will be sparse; that is, most of the entries will be empty. Although data-structure techniques are available for representing sparse matrices, they are not particularly useful for this application, because of the way in which the protection facility is used. Here, we first describe several methods of implementing the access matrix and then compare the methods.

### 17.6.1 Global Table

The simplest implementation of the access matrix is a global table consisting of a set of ordered triples  $\langle \text{domain}, \text{object}, \text{rights-set} \rangle$ . Whenever an operation  $M$  is executed on an object  $O_j$  within domain  $D_i$ , the global table is searched for a triple  $\langle D_i, O_j, R_k \rangle$ , with  $M \in R_k$ . If this triple is found, the operation is allowed to continue; otherwise, an exception (or error) condition is raised.

This implementation suffers from several drawbacks. The table is usually large and thus cannot be kept in main memory, so additional I/O is needed. Virtual memory techniques are often used for managing this table. In addition, it is difficult to take advantage of special groupings of objects or domains. For example, if everyone can read a particular object, this object must have a separate entry in every domain.

### 17.6.2 Access Lists for Objects

Each column in the access matrix can be implemented as an access list for one object, as described in Section 13.4.2. Obviously, the empty entries can be

discarded. The resulting list for each object consists of ordered pairs  $\langle \text{domain}, \text{rights-set} \rangle$ , which define all domains with a nonempty set of access rights for that object.

This approach can be extended easily to define a list plus a *default* set of access rights. When an operation  $M$  on an object  $O_j$  is attempted in domain  $D_i$ , we search the access list for object  $O_j$ , looking for an entry  $\langle D_i, R_k \rangle$  with  $M \in R_k$ . If the entry is found, we allow the operation; if it is not, we check the default set. If  $M$  is in the default set, we allow the access. Otherwise, access is denied, and an exception condition occurs. For efficiency, we may check the default set first and then search the access list.

### 17.6.3 Capability Lists for Domains

Rather than associating the columns of the access matrix with the objects as access lists, we can associate each row with its domain. A **capability list** for a domain is a list of objects together with the operations allowed on those objects. An object is often represented by its physical name or address, called a **capability**. To execute operation  $M$  on object  $O_j$ , the process executes the operation  $M$ , specifying the capability (or pointer) for object  $O_j$  as a parameter. Simple *possession* of the capability means that access is allowed.

The capability list is associated with a domain, but it is never directly accessible to a process executing in that domain. Rather, the capability list is itself a protected object, maintained by the operating system and accessed by the user only indirectly. Capability-based protection relies on the fact that the capabilities are never allowed to migrate into any address space directly accessible by a user process (where they could be modified). If all capabilities are secure, the object they protect is also secure against unauthorized access.

Capabilities were originally proposed as a kind of secure pointer, to meet the need for resource protection that was foreseen as multiprogrammed computer systems came of age. The idea of an inherently protected pointer provides a foundation for protection that can be extended up to the application level.

To provide inherent protection, we must distinguish capabilities from other kinds of objects, and they must be interpreted by an abstract machine on which higher-level programs run. Capabilities are usually distinguished from other data in one of two ways:

- Each object has a **tag** to denote whether it is a capability or accessible data. The tags themselves must not be directly accessible by an application program. Hardware or firmware support may be used to enforce this restriction. Although only one bit is necessary to distinguish between capabilities and other objects, more bits are often used. This extension allows all objects to be tagged with their types by the hardware. Thus, the hardware can distinguish integers, floating-point numbers, pointers, Booleans, characters, instructions, capabilities, and uninitialized values by their tags.
- Alternatively, the address space associated with a program can be split into two parts. One part is accessible to the program and contains the program's normal data and instructions. The other part, containing the capability list, is accessible only by the operating system. A segmented memory space is useful to support this approach.

Several capability-based protection systems have been developed; we describe them briefly in Section 17.10. The Mach operating system also uses a version of capability-based protection; it is described in Appendix D.

#### 17.6.4 A Lock-Key Mechanism

The **lock-key scheme** is a compromise between access lists and capability lists. Each object has a list of unique bit patterns called **locks**. Similarly, each domain has a list of unique bit patterns called **keys**. A process executing in a domain can access an object only if that domain has a key that matches one of the locks of the object.

As with capability lists, the list of keys for a domain must be managed by the operating system on behalf of the domain. Users are not allowed to examine or modify the list of keys (or locks) directly.

#### 17.6.5 Comparison

As you might expect, choosing a technique for implementing an access matrix involves various trade-offs. Using a global table is simple; however, the table can be quite large and often cannot take advantage of special groupings of objects or domains. Access lists correspond directly to the needs of users. When a user creates an object, he can specify which domains can access the object, as well as what operations are allowed. However, because access-right information for a particular domain is not localized, determining the set of access rights for each domain is difficult. In addition, every access to the object must be checked, requiring a search of the access list. In a large system with long access lists, this search can be time consuming.

Capability lists do not correspond directly to the needs of users, but they are useful for localizing information for a given process. The process attempting access must present a capability for that access. Then, the protection system needs only to verify that the capability is valid. Revocation of capabilities, however, may be inefficient (Section 17.7).

The lock-key mechanism, as mentioned, is a compromise between access lists and capability lists. The mechanism can be both effective and flexible, depending on the length of the keys. The keys can be passed freely from domain to domain. In addition, access privileges can be effectively revoked by the simple technique of changing some of the locks associated with the object (Section 17.7).

Most systems use a combination of access lists and capabilities. When a process first tries to access an object, the access list is searched. If access is denied, an exception condition occurs. Otherwise, a capability is created and attached to the process. Additional references use the capability to demonstrate swiftly that access is allowed. After the last access, the capability is destroyed. This strategy was used in the MULTICS system and in the CAL system.

As an example of how such a strategy works, consider a file system in which each file has an associated access list. When a process opens a file, the directory structure is searched to find the file, access permission is checked, and buffers are allocated. All this information is recorded in a new entry in a file table associated with the process. The operation returns an index into this table for the newly opened file. All operations on the file are made by specification of the index into the file table. The entry in the file table then points to the file

and its buffers. When the file is closed, the file-table entry is deleted. Since the file table is maintained by the operating system, the user cannot accidentally corrupt it. Thus, the user can access only those files that have been opened. Since access is checked when the file is opened, protection is ensured. This strategy is used in the UNIX system.

The right to access must still be checked on each access, and the file-table entry has a capability only for the allowed operations. If a file is opened for reading, then a capability for read access is placed in the file-table entry. If an attempt is made to write onto the file, the system identifies this protection violation by comparing the requested operation with the capability in the file-table entry.

## 17.7 Revocation of Access Rights

In a dynamic protection system, we may sometimes need to revoke access rights to objects shared by different users. Various questions about revocation may arise:

- **Immediate versus delayed.** Does revocation occur immediately, or is it delayed? If revocation is delayed, can we find out when it will take place?
- **Selective versus general.** When an access right to an object is revoked, does it affect all the users who have an access right to that object, or can we specify a select group of users whose access rights should be revoked?
- **Partial versus total.** Can a subset of the rights associated with an object be revoked, or must we revoke all access rights for this object?
- **Temporary versus permanent.** Can access be revoked permanently (that is, the revoked access right will never again be available), or can access be revoked and later be obtained again?

With an access-list scheme, revocation is easy. The access list is searched for any access rights to be revoked, and they are deleted from the list. Revocation is immediate and can be general or selective, total or partial, and permanent or temporary.

Capabilities, however, present a much more difficult revocation problem, as mentioned earlier. Since the capabilities are distributed throughout the system, we must find them before we can revoke them. Schemes that implement revocation for capabilities include the following:

- **Reacquisition.** Periodically, capabilities are deleted from each domain. If a process wants to use a capability, it may find that that capability has been deleted. The process may then try to reacquire the capability. If access has been revoked, the process will not be able to reacquire the capability.
- **Back-pointers.** A list of pointers is maintained with each object, pointing to all capabilities associated with that object. When revocation is required, we can follow these pointers, changing the capabilities as necessary. This scheme was adopted in the MULTICS system. It is quite general, but its implementation is costly.

- **Indirection.** The capabilities point indirectly, not directly, to the objects. Each capability points to a unique entry in a global table, which in turn points to the object. We implement revocation by searching the global table for the desired entry and deleting it. Then, when an access is attempted, the capability is found to point to an illegal table entry. Table entries can be reused for other capabilities without difficulty, since both the capability and the table entry contain the unique name of the object. The object for a capability and its table entry must match. This scheme was adopted in the CAL system. It does not allow selective revocation.
- **Keys.** A key is a unique bit pattern that can be associated with a capability. This key is defined when the capability is created, and it can be neither modified nor inspected by the process that owns the capability. A **master key** is associated with each object; it can be defined or replaced with the **set-key** operation. When a capability is created, the current value of the master key is associated with the capability. When the capability is exercised, its key is compared with the master key. If the keys match, the operation is allowed to continue; otherwise, an exception condition is raised. Revocation replaces the master key with a new value via the **set-key** operation, invalidating all previous capabilities for this object.

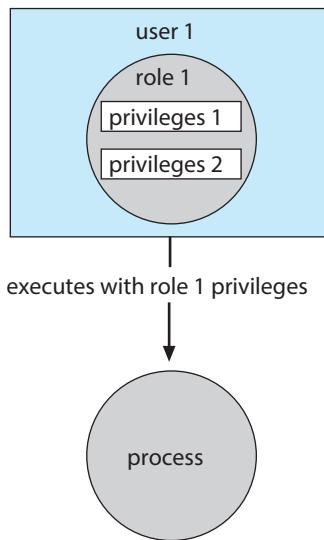
This scheme does not allow selective revocation, since only one master key is associated with each object. If we associate a list of keys with each object, then selective revocation can be implemented. Finally, we can group all keys into one global table of keys. A capability is valid only if its key matches some key in the global table. We implement revocation by removing the matching key from the table. With this scheme, a key can be associated with several objects, and several keys can be associated with each object, providing maximum flexibility.

In key-based schemes, the operations of defining keys, inserting them into lists, and deleting them from lists should not be available to all users. In particular, it would be reasonable to allow only the owner of an object to set the keys for that object. This choice, however, is a policy decision that the protection system can implement but should not define.

## 17.8 Role-Based Access Control

In Section 13.4.2, we described how access controls can be used on files within a file system. Each file and directory is assigned an owner, a group, or possibly a list of users, and for each of those entities, access-control information is assigned. A similar function can be added to other aspects of a computer system. A good example of this is found in Solaris 10 and later versions.

The idea is to advance the protection available in the operating system by explicitly adding the principle of least privilege via **role-based access control (RBAC)**. This facility revolves around privileges. A privilege is the right to execute a system call or to use an option within that system call (such as opening a file with write access). Privileges can be assigned to processes, limiting them to exactly the access they need to perform their work. Privileges and programs can also be assigned to **roles**. Users are assigned roles or can take roles based on passwords assigned to the roles. In this way, a user can take a



**Figure 17.10** Role-based access control in Solaris 10.

role that enables a privilege, allowing the user to run a program to accomplish a specific task, as depicted in Figure 17.10. This implementation of privileges decreases the security risk associated with superusers and setuid programs.

Notice that this facility is similar to the access matrix described in Section 17.5. This relationship is further explored in the exercises at the end of the chapter.

## 17.9 Mandatory Access Control (MAC)

Operating systems have traditionally used **discretionary access control (DAC)** as a means of restricting access to files and other system objects. With DAC, access is controlled based on the identities of individual users or groups. In UNIX-based system, DAC takes the form of file permissions (settable by chmod, chown, and chgrp), whereas Windows (and some UNIX variants) allow finer granularity by means of access-control lists (ACLs).

DACs, however, have proved insufficient over the years. A key weakness lies in their discretionary nature, which allows the owner of a resource to set or modify its permissions. Another weakness is the unlimited access allowed for the administrator or root user. As we have seen, this design can leave the system vulnerable to both accidental and malicious attacks and provides no defense when hackers obtain root privileges.

The need arose, therefore, for a stronger form of protection, which was introduced in the form of **mandatory access control (MAC)**. MAC is enforced as a system policy that even the root user cannot modify (unless the policy explicitly allows modifications or the system is rebooted, usually into an alternate configuration). The restrictions imposed by MAC policy rules are more powerful than the capabilities of the root user and can be used to make resources inaccessible to anyone but their intended owners.

Modern operating systems all provide MAC along with DAC, although implementations differ. Solaris was among the first to introduce MAC, which was part of Trusted Solaris (2.5). FreeBSD made DAC part of its TrustedBSD implementation (FreeBSD 5.0). The FreeBSD implementation was adopted by Apple in macOS 10.5 and has served as the substrate over which most of the security features of MAC and iOS are implemented. Linux's MAC implementation is part of the SELinux project, which was devised by the NSA, and has been integrated into most distributions. Microsoft Windows joined the trend with Windows Vista's Mandatory Integrity Control.

At the heart of MAC is the concept of **labels**. A label is an identifier (usually a string) assigned to an object (files, devices, and the like). Labels may also be applied to subjects (actors, such as processes). When a subject request to perform operations on the objects. When such requests are to be served by the operating system, it first performs checks defined in a policy, which dictates whether or not a given label holding subject is allowed to perform the operation on the labeled object.

As a brief example, consider a simple set of labels, ordered according to level of privilege: “unclassified,” “secret,” and “top secret.” A user with “secret” clearance will be able to create similarly labeled processes, which will then have access to “unclassified” and “secret” files, but not to “top secret” files. Neither the user nor its processes would even be aware of the existence of “top secret” files, since the operating system would filter them out of all file operations (for example, they would not be displayed when listing directory contents). User processes would similarly be protected themselves in this way, so that an “unclassified” process would not be able to see or perform IPC requests to a “secret” (or “top secret”) process. In this way, MAC labels are an implementation of the access matrix described earlier.

## 17.10 Capability-Based Systems

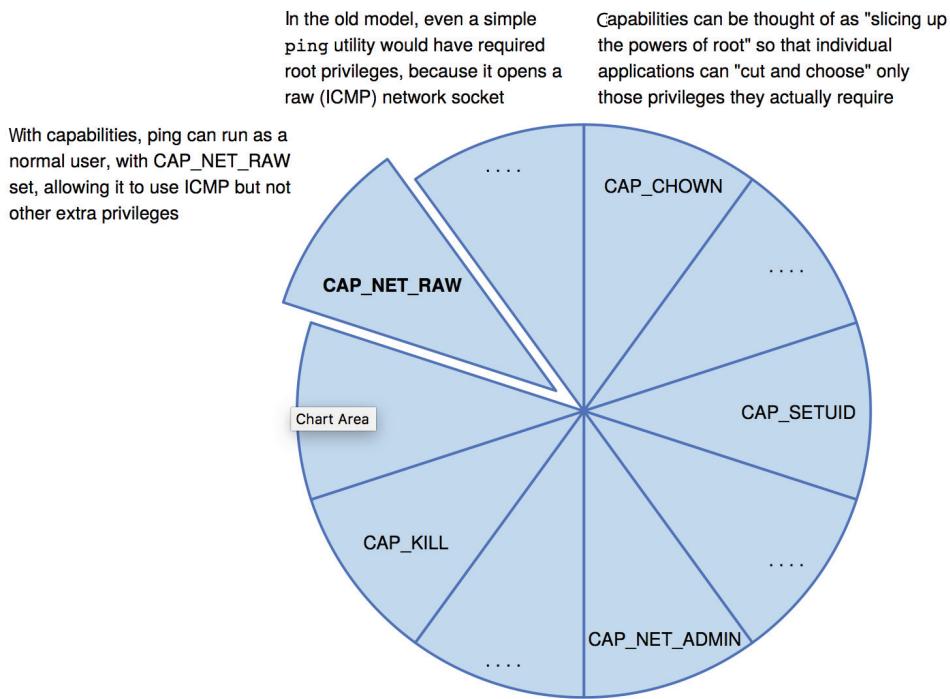
The concept of **capability-based protection** was introduced in the early 1970s. Two early research systems were Hydra and CAP. Neither system was widely used, but both provided interesting proving grounds for protection theories. For more details on these systems, see Section A.14.1 and Section A.14.2. Here, we consider two more contemporary approaches to capabilities.

### 17.10.1 Linux Capabilities

Linux uses capabilities to address the limitations of the UNIX model, which we described earlier. The POSIX standards group introduced capabilities in POSIX 1003.1e. Although POSIX.1e was eventually withdrawn, Linux was quick to adopt capabilities in Version 2.2 and has continued to add new developments.

In essence, Linux's capabilities “slice up” the powers of root into distinct areas, each represented by a bit in a bitmask, as shown in Figure 17.11. Fine-grained control over privileged operations can be achieved by toggling bits in the bitmask.

In practice, three bitmasks are used—denoting the capabilities *permitted*, *effective*, and *inheritable*. Bitmasks can apply on a per-process or a per-thread basis. Furthermore, once revoked, capabilities cannot be reacquired. The usual



**Figure 17.11** Capabilities in POSIX.1e.

sequence of events is that a process or thread starts with the full set of permitted capabilities and voluntarily decreases that set during execution. For example, after opening a network port, a thread might remove that capability so that no further ports can be opened.

You can probably see that capabilities are a direct implementation of the principle of least privilege. As explained earlier, this tenet of security dictates that an application or user must be given only those rights than are required for its normal operation.

Android (which is based on Linux) also utilizes capabilities, which enable system processes (notably, “system server”), to avoid root ownership, instead selectively enabling only those operations required.

The Linux capabilities model is a great improvement over the traditional UNIX model, but it still is inflexible. For one thing, using a bitmap with a bit representing each capability makes it impossible to add capabilities dynamically and requires recompiling the kernel to add more. In addition, the feature applies only to kernel-enforced capabilities.

### 17.10.2 Darwin Entitlements

Apple’s system protection takes the form of entitlements. Entitlements are declaratory permissions—XML property list stating which permissions are claimed as necessary by the program (see Figure 17.12). When the process attempts a privileged operation (in the figure, loading a kernel extension), its

```
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>com.apple.private.kernel.get-kext-info
  <true/>
  <key>com.apple.rootless.kext-management
  <true/>
</dict>
</plist>
```

---

**Figure 17.12** Apple Darwin entitlements

entitlements are checked, and only if the needed entitlements are present is the operation allowed.

To prevent programs from arbitrarily claiming an entitlement, Apple embeds the entitlements in the code signature (explained in Section 17.11.4). Once loaded, a process has no way of accessing its code signature. Other processes (and the kernel) can easily query the signature, and in particular the entitlements. Verifying an entitlement is therefore a simple string-matching operation. In this way, only verifiable, authenticated apps may claim entitlements. All system entitlements (`com.apple.*`) are further restricted to Apple's own binaries.

## 17.11 Other Protection Improvement Methods

As the battle to protect systems from accidental and malicious damage escalates, operating-system designers are implementing more types of protection mechanisms at more levels. This section surveys some important real-world protection improvements.

### 17.11.1 System Integrity Protection

Apple introduced in macOS 10.11 a new protection mechanism called **System Integrity Protection (SIP)**. Darwin-based operating systems use SIP to restrict access to system files and resources in such a way that even the root user cannot tamper with them. SIP uses extended attributes on files to mark them as restricted and further protects system binaries so that they cannot be debugged or scrutinized, much less tampered with. Most importantly, only code-signed kernel extensions are permitted, and SIP can further be configured to allow only code-signed binaries as well.

Under SIP, although root is still the most powerful user in the system, it can do far less than before. The root user can still manage other users' files, as well as install and remove programs, but not in any way that would replace or modify operating-system components. SIP is implemented as a global, inescapable

screen on all processes, with the only exceptions allowed for system binaries (for example, `fsck`, or `kextload`, as shown in Figure 17.12), which are specifically entitled for operations for their designated purpose.

### 17.11.2 System-Call Filtering

Recall from Chapter 2 that monolithic systems place all of the functionality of the kernel into a single file that runs in a single address space. Commonly, general-purpose operating-system kernels are monolithic, and they are therefore implicitly trusted as secure. The trust boundary, therefore, rests between kernel mode and user mode—at the system layer. We can reasonably assume that any attempt to compromise the system’s integrity will be made from user mode by means of a system call. For example, an attacker can try to gain access by exploiting an unprotected system call.

It is therefore imperative to implement some form of **system-call filtering**. To accomplish this, we can add code to the kernel to perform an inspection at the system-call gate, restricting a caller to a subset of system calls deemed safe or required for that caller’s function. Specific system-call profiles can be constructed for individual processes. The Linux mechanism SECCOMP-BPF does just that, harnessing the Berkeley Packet Filter language to load a custom profile through Linux’s proprietary `prctl` system call. This filtering is voluntary but can be effectively enforced if called from within a run-time library when it initializes or from within the loader itself before it transfers control to the program’s entry point.

A second form of system-call filtering goes deeper still and inspects the arguments of each system call. This form of protection is considered much stronger, as even apparently benign system calls can harbor serious vulnerabilities. This was the case with Linux’s fast mutex (`futex`) system call. A race condition in its implementation led to an attacker-controlled kernel memory overwrite and total system compromise. Mutexes are a fundamental component of multitasking, and thus the system call itself could not be filtered out entirely.

A challenge encountered with both approaches is keeping them as flexible as possible while at the same time avoiding the need to rebuild the kernel when changes or new filters are required—a common occurrence due to the differing needs of different processes. Flexibility is especially important given the unpredictable nature of vulnerabilities. New vulnerabilities are discovered every day and may be immediately exploitable by attackers.

One approach to meeting this challenge is to decouple the filter implementation from the kernel itself. The kernel need only contain a set of callouts, which can then be implemented in a specialized driver (Windows), kernel module (Linux), or extension (Darwin). Because an external, modular component provides the filtering logic, it can be updated independently of the kernel. This component commonly makes use of a specialized profiling language by including a built-in interpreter or parser. Thus, the profile itself can be decoupled from the code, providing a human-readable, editable profile and further simplifying updates. It is also possible for the filtering component to call a trusted user-mode daemon process to assist with validation logic.

### 17.11.3 Sandboxing

Sandboxing involves running processes in environments that limit what they can do. In a basic system, a process runs with the credentials of the user that started it and has access to all things that the user can access. If run with system privileges such as root, the process can literally do anything on the system. It is almost always the case that a process does not need full user or system privileges. For example, does a word processor need to accept network connections? Does a network service that provides the time of day need to access files beyond a specific set?

The term **sandboxing** refers to the practice of enforcing strict limitations on a process. Rather than give that process the full set of system calls its privileges would allow, we impose an irremovable set of restrictions on the process in the early stages of its startup—well before the execution of its `main()` function and often as early as its creation with the `fork` system call. The process is then rendered unable to perform any operations outside its allowed set. In this way, it is possible to prevent the process from communicating with any other system component, resulting in tight compartmentalization that mitigates any damage to the system even if the process is compromised.

There are numerous approaches to sandboxing. Java and .net, for example, impose sandbox restrictions at the level of the virtual machine. Other systems enforce sandboxing as part of their mandatory access control (MAC) policy. An example is Android, which draws on an SELinux policy enhanced with specific labels for system properties and service endpoints.

Sandboxing may also be implemented as a combination of multiple mechanisms. Android has found SELinux useful but lacking, because it cannot effectively restrict individual system calls. The latest Android versions (“Nougat” and “O”) use an underlying Linux mechanism called SECCOMP-BPF, mentioned earlier, to apply system-call restrictions through the use of a specialized system call. The C run-time library in Android (“Bionic”) calls this system call to impose restrictions on all Android processes and third-party applications.

Among the major vendors, Apple was the first to implement sandboxing, which appeared in macOS 10.5 (“Tiger”) as “Seatbelt”. Seatbelt was “opt-in” rather than mandatory, allowing but not requiring applications to use it. The Apple sandbox was based on dynamic profiles written in the Scheme language, which provided the ability to control not just which operations were to be allowed or blocked but also their arguments. This capability enabled Apple to create different custom-fit profiles for each binary on the system, a practice that continues to this day. Figure 17.13 depicts a profile example.

Apple’s sandboxing has evolved considerably since its inception. It is now used in the iOS variants, where it serves (along with code signing) as the chief protection against untrusted third-party code. In iOS, and starting with macOS 10.8, the macOS sandbox is mandatory and is automatically enforced for all Mac-store downloaded apps. More recently, as mentioned earlier, Apple adopted the System Integrity Protection (SIP), used in macOS 10.11 and later. SIP is, in effect, a system-wide “platform profile.” Apple enforces it starting at system boot on all processes in the system. Only those processes that are entitled can perform privileged operations, and those are code-signed by Apple and therefore trusted.

```
(version 1)
(deny default)
(allow file-chroot)
(allow file-read-metadata (literal "/var"))
(allow sysctl-read)
(allow mach-per-user-lookup)
(allow mach-lookup)
(global-name "com.apple.system.logger")
```

---

**Figure 17.13** A sandbox profile of a MacOS daemon denying most operations.

#### 17.11.4 Code Signing

At a fundamental level, how can a system “trust” a program or script? Generally, if the item came as part of the operating system, it should be trusted. But what if the item is changed? If it’s changed by a system update, then again it’s trustworthy, but otherwise it should not be executable or should require special permission (from the user or administrator) before it is run. Tools from third parties, commercial or otherwise, are more difficult to judge. How can we be sure the tool wasn’t modified on its way from where it was created to our systems?

Currently, code signing is the best tool in the protection arsenal for solving these problems. **Code signing** is the digital signing of programs and executables to confirm that they have not been changed since the author created them. It uses a cryptographic hash (Section 16.4.1.3) to test for integrity and authenticity. Code signing is used for operating-system distributions, patches, and third-party tools alike. Some operating systems, including iOS, Windows, and macOS, refuse to run programs that fail their code-signing check. It can also enhance system functionality in other ways. For example, Apple can disable all programs written for a now-obsolete version of iOS by stopping its signing of those programs when they are downloaded from the App Store.

### 17.12 Language-Based Protection

To the degree that protection is provided in computer systems, it is usually achieved through an operating-system kernel, which acts as a security agent to inspect and validate each attempt to access a protected resource. Since comprehensive access validation may be a source of considerable overhead, either we must give it hardware support to reduce the cost of each validation, or we must allow the system designer to compromise the goals of protection. Satisfying all these goals is difficult if the flexibility to implement protection policies is restricted by the support mechanisms provided or if protection environments are made larger than necessary to secure greater operational efficiency.

As operating systems have become more complex, and particularly as they have attempted to provide higher-level user interfaces, the goals of protection

have become much more refined. The designers of protection systems have drawn heavily on ideas that originated in programming languages and especially on the concepts of abstract data types and objects. Protection systems are now concerned not only with the identity of a resource to which access is attempted but also with the functional nature of that access. In the newest protection systems, concern for the function to be invoked extends beyond a set of system-defined functions, such as standard file-access methods, to include functions that may be user-defined as well.

Policies for resource use may also vary, depending on the application, and they may be subject to change over time. For these reasons, protection can no longer be considered a matter of concern only to the designer of an operating system. It should also be available as a tool for use by the application designer, so that resources of an application subsystem can be guarded against tampering or the influence of an error.

### 17.12.1 Compiler-Based Enforcement

At this point, programming languages enter the picture. Specifying the desired control of access to a shared resource in a system is making a declarative statement about the resource. This kind of statement can be integrated into a language by an extension of its typing facility. When protection is declared along with data typing, the designer of each subsystem can specify its requirements for protection, as well as its need for use of other resources in a system. Such a specification should be given directly as a program is composed, and in the language in which the program itself is stated. This approach has several significant advantages:

1. Protection needs are simply declared, rather than programmed as a sequence of calls on procedures of an operating system.
2. Protection requirements can be stated independently of the facilities provided by a particular operating system.
3. The means for enforcement need not be provided by the designer of a subsystem.
4. A declarative notation is natural because access privileges are closely related to the linguistic concept of data type.

A variety of techniques can be provided by a programming-language implementation to enforce protection, but any of these must depend on some degree of support from an underlying machine and its operating system. For example, suppose a language is used to generate code to run on the Cambridge CAP system (Section A.14.2). On this system, every storage reference made on the underlying hardware occurs indirectly through a capability. This restriction prevents any process from accessing a resource outside of its protection environment at any time. However, a program may impose arbitrary restrictions on how a resource can be used during execution of a particular code segment. We can implement such restrictions most readily by using the software capabilities provided by CAP. A language implementation might provide standard protected procedures to interpret software capabilities that would realize the protection policies that could be specified in the language. This scheme puts

policy specification at the disposal of the programmers, while freeing them from implementing its enforcement.

Even if a system does not provide a protection kernel as powerful as those of Hydra (Section A.14.1) or CAP, mechanisms are still available for implementing protection specifications given in a programming language. The principal distinction is that the *security* of this protection will not be as great as that supported by a protection kernel, because the mechanism must rely on more assumptions about the operational state of the system. A compiler can separate references for which it can certify that no protection violation could occur from those for which a violation might be possible, and it can treat them differently. The security provided by this form of protection rests on the assumption that the code generated by the compiler will not be modified prior to or during its execution.

What, then, are the relative merits of enforcement based solely on a kernel, as opposed to enforcement provided largely by a compiler?

- **Security.** Enforcement by a kernel provides a greater degree of security of the protection system itself than does the generation of protection-checking code by a compiler. In a compiler-supported scheme, security rests on correctness of the translator, on some underlying mechanism of storage management that protects the segments from which compiled code is executed, and, ultimately, on the security of files from which a program is loaded. Some of these considerations also apply to a software-supported protection kernel, but to a lesser degree, since the kernel may reside in fixed physical storage segments and may be loaded only from a designated file. With a tagged-capability system, in which all address computation is performed either by hardware or by a fixed microprogram, even greater security is possible. Hardware-supported protection is also relatively immune to protection violations that might occur as a result of either hardware or system software malfunction.
- **Flexibility.** There are limits to the flexibility of a protection kernel in implementing a user-defined policy, although it may supply adequate facilities for the system to provide enforcement of its own policies. With a programming language, protection policy can be declared and enforcement provided as needed by an implementation. If a language does not provide sufficient flexibility, it can be extended or replaced with less disturbance than would be caused by the modification of an operating-system kernel.
- **Efficiency.** The greatest efficiency is obtained when enforcement of protection is supported directly by hardware (or microcode). Insofar as software support is required, language-based enforcement has the advantage that static access enforcement can be verified off-line at compile time. Also, since an intelligent compiler can tailor the enforcement mechanism to meet the specified need, the fixed overhead of kernel calls can often be avoided.

In summary, the specification of protection in a programming language allows the high-level description of policies for the allocation and use of resources. A language implementation can provide software for protection enforcement when automatic hardware-supported checking is unavailable. In

addition, it can interpret protection specifications to generate calls on whatever protection system is provided by the hardware and the operating system.

One way of making protection available to the application program is through the use of a software capability that could be used as an object of computation. Inherent in this concept is the idea that certain program components might have the privilege of creating or examining these software capabilities. A capability-creating program would be able to execute a primitive operation that would seal a data structure, rendering the latter's contents inaccessible to any program components that did not hold either the seal or the unseal privilege. Such components might copy the data structure or pass its address to other program components, but they could not gain access to its contents. The reason for introducing such software capabilities is to bring a protection mechanism into the programming language. The only problem with the concept as proposed is that the use of the seal and unseal operations takes a procedural approach to specifying protection. A nonprocedural or declarative notation seems a preferable way to make protection available to the application programmer.

What is needed is a safe, dynamic access-control mechanism for distributing capabilities to system resources among user processes. To contribute to the overall reliability of a system, the access-control mechanism should be safe to use. To be useful in practice, it should also be reasonably efficient. This requirement has led to the development of a number of language constructs that allow the programmer to declare various restrictions on the use of a specific managed resource. (See the bibliographical notes for appropriate references.) These constructs provide mechanisms for three functions:

1. Distributing capabilities safely and efficiently among customer processes. In particular, mechanisms ensure that a user process will use the managed resource only if it was granted a capability to that resource.
2. Specifying the type of operations that a particular process may invoke on an allocated resource (for example, a reader of a file should be allowed only to read the file, whereas a writer should be able both to read and to write). It should not be necessary to grant the same set of rights to every user process, and it should be impossible for a process to enlarge its set of access rights, except with the authorization of the access-control mechanism.
3. Specifying the order in which a particular process may invoke the various operations of a resource (for example, a file must be opened before it can be read). It should be possible to give two processes different restrictions on the order in which they can invoke the operations of the allocated resource.

The incorporation of protection concepts into programming languages, as a practical tool for system design, is in its infancy. Protection will likely become a matter of greater concern to the designers of new systems with distributed architectures and increasingly stringent requirements on data security. Then the importance of suitable language notations in which to express protection requirements will be recognized more widely.

### 17.12.2 Run-Time-Based Enforcement—Protection in Java

Because Java was designed to run in a distributed environment, the Java virtual machine—or JVM—has many built-in protection mechanisms. Java programs are composed of **classes**, each of which is a collection of data fields and functions (called **methods**) that operate on those fields. The JVM loads a class in response to a request to create instances (or objects) of that class. One of the most novel and useful features of Java is its support for dynamically loading untrusted classes over a network and for executing mutually distrusting classes within the same JVM.

Because of these capabilities, protection is a paramount concern. Classes running in the same JVM may be from different sources and may not be equally trusted. As a result, enforcing protection at the granularity of the JVM process is insufficient. Intuitively, whether a request to open a file should be allowed will generally depend on which class has requested the open. The operating system lacks this knowledge.

Thus, such protection decisions are handled within the JVM. When the JVM loads a class, it assigns the class to a protection domain that gives the permissions of that class. The protection domain to which the class is assigned depends on the URL from which the class was loaded and any digital signatures on the class file. (Digital signatures are covered in Section 16.4.1.3.) A configurable policy file determines the permissions granted to the domain (and its classes). For example, classes loaded from a trusted server might be placed in a protection domain that allows them to access files in the user's home directory, whereas classes loaded from an untrusted server might have no file access permissions at all.

It can be complicated for the JVM to determine what class is responsible for a request to access a protected resource. Accesses are often performed indirectly, through system libraries or other classes. For example, consider a class that is not allowed to open network connections. It could call a system library to request the load of the contents of a URL. The JVM must decide whether or not to open a network connection for this request. But which class should be used to determine if the connection should be allowed, the application or the system library?

The philosophy adopted in Java is to require the library class to explicitly permit a network connection. More generally, in order to access a protected resource, some method in the calling sequence that resulted in the request must explicitly assert the privilege to access the resource. By doing so, this method *takes responsibility* for the request. Presumably, it will also perform whatever checks are necessary to ensure the safety of the request. Of course, not every method is allowed to assert a privilege; a method can assert a privilege only if its class is in a protection domain that is itself allowed to exercise the privilege.

This implementation approach is called **stack inspection**. Every thread in the JVM has an associated stack of its ongoing method invocations. When a caller may not be trusted, a method executes an access request within a `doPrivileged` block to perform the access to a protected resource directly or indirectly. `doPrivileged()` is a static method in the `AccessController` class that is passed a class with a `run()` method to invoke. When the `doPrivileged` block is entered, the stack frame for this method is annotated to indicate this fact. Then, the contents of the block are executed. When an access to a protected

protection domain:	untrusted applet	URL loader	networking
socket permission:	none	*.lucent.com:80, connect	any
class:	gui: ... get(url); open(addr); ...	get(URL u): ... doPrivileged { open('proxy.lucent.com:80'); } <request u from proxy> ...	open(Addr a): ... checkPermission(a, connect); connect(a); ...

**Figure 17.14** Stack inspection.

resource is subsequently requested, either by this method or a method it calls, a call to `checkPermissions()` is used to invoke stack inspection to determine if the request should be allowed. The inspection examines stack frames on the calling thread's stack, starting from the most recently added frame and working toward the oldest. If a stack frame is first found that has the `doPrivileged()` annotation, then `checkPermissions()` returns immediately and silently, allowing the access. If a stack frame is first found for which access is disallowed based on the protection domain of the method's class, then `checkPermissions()` throws an `AccessControlException`. If the stack inspection exhausts the stack without finding either type of frame, then whether access is allowed depends on the implementation (some implementations of the JVM may allow access, while other implementations may not).

Stack inspection is illustrated in Figure 17.14. Here, the `gui()` method of a class in the *untrusted applet* protection domain performs two operations, first a `get()` and then an `open()`. The former is an invocation of the `get()` method of a class in the *URL loader* protection domain, which is permitted to `open()` sessions to sites in the `lucent.com` domain, in particular a proxy server `proxy.lucent.com` for retrieving URLs. For this reason, the untrusted applet's `get()` invocation will succeed: the `checkPermissions()` call in the networking library encounters the stack frame of the `get()` method, which performed its `open()` in a `doPrivileged` block. However, the untrusted applet's `open()` invocation will result in an exception, because the `checkPermissions()` call finds no `doPrivileged` annotation before encountering the stack frame of the `gui()` method.

Of course, for stack inspection to work, a program must be unable to modify the annotations on its own stack frame or to otherwise manipulate stack inspection. This is one of the most important differences between Java and many other languages (including C++). A Java program cannot directly access memory; it can manipulate only an object for which it has a reference. References cannot be forged, and manipulations are made only through well-defined interfaces. Compliance is enforced through a sophisticated collection of load-time and run-time checks. As a result, an object cannot manipulate its run-time stack, because it cannot get a reference to the stack or other components of the protection system.

More generally, Java's load-time and run-time checks enforce **type safety** of Java classes. Type safety ensures that classes cannot treat integers as pointers, write past the end of an array, or otherwise access memory in arbitrary ways. Rather, a program can access an object only via the methods defined on that object by its class. This is the foundation of Java protection, since it enables a class to effectively **encapsulate** and protect its data and methods from other classes loaded in the same JVM. For example, a variable can be defined as **private** so that only the class that contains it can access it or **protected** so that it can be accessed only by the class that contains it, subclasses of that class, or classes in the same package. Type safety ensures that these restrictions can be enforced.

### 17.13 Summary

- System protection features are guided by the principle of need-to-know and implement mechanisms to enforce the principle of least privilege.
- Computer systems contain objects that must be protected from misuse. Objects may be hardware (such as memory, CPU time, and I/O devices) or software (such as files, programs, and semaphores).
- An access right is permission to perform an operation on an object. A domain is a set of access rights. Processes execute in domains and may use any of the access rights in the domain to access and manipulate objects. During its lifetime, a process may be either bound to a protection domain or allowed to switch from one domain to another.
- A common method of securing objects is to provide a series of protection rings, each with more privileges than the last. ARM, for example, provides four protection levels. The most privileged, TrustZone, is callable only from kernel mode.
- The access matrix is a general model of protection that provides a mechanism for protection without imposing a particular protection policy on the system or its users. The separation of policy and mechanism is an important design property.
- The access matrix is sparse. It is normally implemented either as access lists associated with each object or as capability lists associated with each domain. We can include dynamic protection in the access-matrix model by considering domains and the access matrix itself as objects. Revocation of access rights in a dynamic protection model is typically easier to implement with an access-list scheme than with a capability list.
- Real systems are much more limited than the general model. Older UNIX distributions are representative, providing discretionary access controls of read, write, and execution protection separately for the owner, group, and general public for each file. More modern systems are closer to the general model, or at least provide a variety of protection features to protect the system and its users.
- Solaris 10 and beyond, among other systems, implement the principle of least privilege via role-based access control, a form of access matrix.

Another protection extension is mandatory access control, a form of system policy enforcement.

- Capability-based systems offer finer-grained protection than older models, providing specific abilities to processes by “slicing up” the powers of root into distinct areas. Other methods of improving protection include System Integrity Protection, system-call filtering, sandboxing, and code signing.
- Language-based protection provides finer-grained arbitration of requests and privileges than the operating system is able to provide. For example, a single Java JVM can run several threads, each in a different protection class. It enforces the resource requests through sophisticated stack inspection and via the type safety of the language.

## Further Reading

The concept of a capability evolved from Iliffe’s and Jodeit’s *codewords*, which were implemented in the Rice University computer ([Iliffe and Jodeit (1962)]). The term *capability* was introduced by [Dennis and Horn (1966)].

The principle of separation of policy and mechanism was advocated by the designer of Hydra ([Levin et al. (1975)]).

The use of minimal operating-system support to enforce protection was advocated by the Exokernel Project ([Ganger et al. (2002)], [Kaashoek et al. (1997)]).

The access-matrix model of protection between domains and objects was developed by [Lampson (1969)] and [Lampson (1971)]. [Popek (1974)] and [Saltzer and Schroeder (1975)] provided excellent surveys on the subject of protection.

The Posix capability standard and the way it was implemented in Linux is described in [https://www.usenix.org/legacy/event/usenix03/tech/freenix03/full\\_papers/gruenbacher/gruenbacher.html/main.html](https://www.usenix.org/legacy/event/usenix03/tech/freenix03/full_papers/gruenbacher/gruenbacher.html/main.html)

Details on POSIX.1e and its Linux implementation are provided in [https://www.usenix.org/legacy/event/usenix03/tech/freenix03/full\\_papers/gruenbacher/gruenbacher.html/main.html](https://www.usenix.org/legacy/event/usenix03/tech/freenix03/full_papers/gruenbacher/gruenbacher.html/main.html).

## Bibliography

**[Dennis and Horn (1966)]** J. B. Dennis and E. C. V. Horn, “Programming Semantics for Multiprogrammed Computations”, *Communications of the ACM*, Volume 9, Number 3 (1966), pages 143–155.

**[Ganger et al. (2002)]** G. R. Ganger, D. R. Engler, M. F. Kaashoek, H. M. Briceno, R. Hunt, and T. Pinckney, “Fast and Flexible Application-Level Networking on Exokernel Systems”, *ACM Transactions on Computer Systems*, Volume 20, Number 1 (2002), pages 49–83.

**[Iliffe and Jodeit (1962)]** J. K. Iliffe and J. G. Jodeit, “A Dynamic Storage Allocation System”, *Computer Journal*, Volume 5, Number 3 (1962), pages 200–209.

- [Kaashoek et al. (1997)] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie, “Application Performance and Flexibility on Exokernel Systems”, *Proceedings of the ACM Symposium on Operating Systems Principles* (1997), pages 52–65.
- [Lampson (1969)] B. W. Lampson, “Dynamic Protection Structures”, *Proceedings of the AFIPS Fall Joint Computer Conference* (1969), pages 27–38.
- [Lampson (1971)] B. W. Lampson, “Protection”, *Proceedings of the Fifth Annual Princeton Conference on Information Systems Science* (1971), pages 437–443.
- [Levin et al. (1975)] R. Levin, E. S. Cohen, W. M. Corwin, F. J. Pollack, and W. A. Wulf, “Policy/Mechanism Separation in Hydra”, *Proceedings of the ACM Symposium on Operating Systems Principles* (1975), pages 132–140.
- [Popek (1974)] G. J. Popek, “Protection Structures”, *Computer*, Volume 7, Number 6 (1974), pages 22–33.
- [Saltzer and Schroeder (1975)] J. H. Saltzer and M. D. Schroeder, “The Protection of Information in Computer Systems”, *Proceedings of the IEEE* (1975), pages 1278–1308.

## Chapter 17 Exercises

- 17.11 The access-control matrix can be used to determine whether a process can switch from, say, domain A to domain B and enjoy the access privileges of domain B. Is this approach equivalent to including the access privileges of domain B in those of domain A?
- 17.12 Consider a computer system in which computer games can be played by students only between 10 P.M. and 6 A.M., by faculty members between 5 P.M. and 8 A.M., and by the computer center staff at all times. Suggest a scheme for implementing this policy efficiently.
- 17.13 What hardware features does a computer system need for efficient capability manipulation? Can these features be used for memory protection?
- 17.14 Discuss the strengths and weaknesses of implementing an access matrix using access lists that are associated with objects.
- 17.15 Discuss the strengths and weaknesses of implementing an access matrix using capabilities that are associated with domains.
- 17.16 Explain why a capability-based system provides greater flexibility than a ring-protection scheme in enforcing protection policies.
- 17.17 What is the need-to-know principle? Why is it important for a protection system to adhere to this principle?
- 17.18 Discuss which of the following systems allow module designers to enforce the need-to-know principle.
  - a. Ring-protection scheme
  - b. JVM's stack-inspection scheme
- 17.19 Describe how the Java protection model would be compromised if a Java program were allowed to directly alter the annotations of its stack frame.
- 17.20 How are the access-matrix facility and the role-based access-control facility similar? How do they differ?
- 17.21 How does the principle of least privilege aid in the creation of protection systems?
- 17.22 How can systems that implement the principle of least privilege still have protection failures that lead to security violations?



## Part Eight

# Advanced Topics

Virtualization permeates all aspects of computing. Virtual machines are one instance of this trend. Generally, with a virtual machine, guest operating systems and applications run in an environment that appears to them to be native hardware. This environment behaves toward them as native hardware would but also protects, manages, and limits them.

A *distributed system* is a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory, and the processors communicate with one another through a local-area or wide-area computer network. Computer networks allow disparate computing devices to communicate by adopting standard communication protocols. Distributed systems offer several benefits: they give users access to more of the resources maintained by the system, boost computation speed, and improve data availability and reliability.



# Virtual Machines



The term *virtualization* has many meanings, and aspects of virtualization permeate all aspects of computing. Virtual machines are one instance of this trend. Generally, with a virtual machine, guest operating systems and applications run in an environment that appears to them to be native hardware and that behaves toward them as native hardware would but that also protects, manages, and limits them.

This chapter delves into the uses, features, and implementation of virtual machines. Virtual machines can be implemented in several ways, and this chapter describes these options. One option is to add virtual machine support to the kernel. Because that implementation method is the most pertinent to this book, we explore it most fully. Additionally, hardware features provided by the CPU and even by I/O devices can support virtual machine implementation, so we discuss how those features are used by the appropriate kernel modules.

## CHAPTER OBJECTIVES

- Explore the history and benefits of virtual machines.
- Discuss the various virtual machine technologies.
- Describe the methods used to implement virtualization.
- Identify the most common hardware features that support virtualization and explain how they are used by operating-system modules.
- Discuss current virtualization research areas.

### 18.1 Overview

The fundamental idea behind a virtual machine is to abstract the hardware of a single computer (the CPU, memory, disk drives, network interface cards, and so forth) into several different execution environments, thereby creating the illusion that each separate environment is running on its own private computer. This concept may seem similar to the layered approach of operating system implementation (see Section 2.8.2), and in some ways it is. In the case of

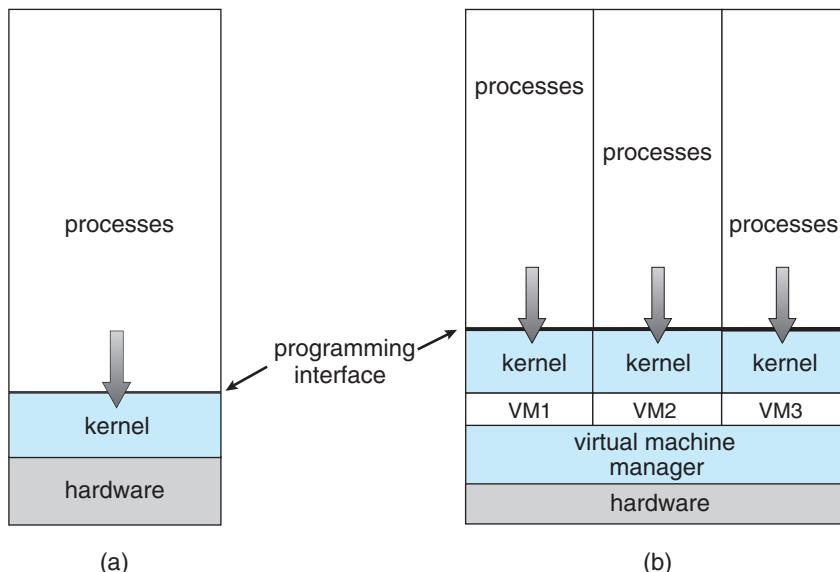
virtualization, there is a layer that creates a virtual system on which operating systems or applications can run.

Virtual machine implementations involve several components. At the base is the **host**, the underlying hardware system that runs the virtual machines. The **virtual machine manager (VMM)** (also known as a **hypervisor**) creates and runs virtual machines by providing an interface that is *identical* to the host (except in the case of paravirtualization, discussed later). Each **guest** process is provided with a virtual copy of the host (Figure 18.1). Usually, the guest process is in fact an operating system. A single physical machine can thus run multiple operating systems concurrently, each in its own virtual machine.

Take a moment to note that with virtualization, the definition of “operating system” once again blurs. For example, consider VMM software such as VMware ESX. This virtualization software is installed on the hardware, runs when the hardware boots, and provides services to applications. The services include traditional ones, such as scheduling and memory management, along with new types, such as migration of applications between systems. Furthermore, the applications are, in fact, guest operating systems. Is the VMware ESX VMM an operating system that, in turn, runs other operating systems? Certainly it acts like an operating system. For clarity, however, we call the component that provides virtual environments a VMM.

The implementation of VMMs varies greatly. Options include the following:

- Hardware-based solutions that provide support for virtual machine creation and management via firmware. These VMMs, which are commonly found in mainframe and large to midsized servers, are generally known as **type 0 hypervisors**. IBM LPARs and Oracle LDOMs are examples.



**Figure 18.1** System models. (a) Nonvirtual machine. (b) Virtual machine.

## INDIRECTION

“All problems in computer science can be solved by another level of indirection”—David Wheeler

“. . . except for the problem of too many layers of indirection.”—Kevlin Henney

- Operating-system-like software built to provide virtualization, including VMware ESX (mentioned above), Joyent SmartOS, and Citrix XenServer. These VMMs are known as **type 1 hypervisors**.
- General-purpose operating systems that provide standard functions as well as VMM functions, including Microsoft Windows Server with HyperV and Red Hat Linux with the KVM feature. Because such systems have a feature set similar to type 1 hypervisors, they are also known as type 1.
- Applications that run on standard operating systems but provide VMM features to guest operating systems. These applications, which include VMware Workstation and Fusion, Parallels Desktop, and Oracle VirtualBox, are **type 2 hypervisors**.
- **Paravirtualization**, a technique in which the guest operating system is modified to work in cooperation with the VMM to optimize performance.
- **Programming-environment virtualization**, in which VMMs do not virtualize real hardware but instead create an optimized virtual system. This technique is used by Oracle Java and Microsoft.Net.
- **Emulators** that allow applications written for one hardware environment to run on a very different hardware environment, such as a different type of CPU.
- **Application containment**, which is not virtualization at all but rather provides virtualization-like features by segregating applications from the operating system. Oracle Solaris Zones, BSD Jails, and IBM AIX WPARs “contain” applications, making them more secure and manageable.

The variety of virtualization techniques in use today is a testament to the breadth, depth, and importance of virtualization in modern computing. Virtualization is invaluable for data-center operations, efficient application development, and software testing, among many other uses.

## 18.2 History

Virtual machines first appeared commercially on IBM mainframes in 1972. Virtualization was provided by the IBM VM operating system. This system has evolved and is still available. In addition, many of its original concepts are found in other systems, making it worth exploring.

IBM VM/370 divided a mainframe into multiple virtual machines, each running its own operating system. A major difficulty with the VM approach involved disk systems. Suppose that the physical machine had three disk drives but wanted to support seven virtual machines. Clearly, it could not allocate a disk drive to each virtual machine. The solution was to provide virtual disks—termed **minidisks** in IBM's VM operating system. The minidisks were identical to the system's hard disks in all respects except size. The system implemented each minidisk by allocating as many tracks on the physical disks as the minidisk needed.

Once the virtual machines were created, users could run any of the operating systems or software packages that were available on the underlying machine. For the IBM VM system, a user normally ran CMS—a single-user interactive operating system.

For many years after IBM introduced this technology, virtualization remained in its domain. Most systems could not support virtualization. However, a formal definition of virtualization helped to establish system requirements and a target for functionality. The virtualization requirements called for:

- **Fidelity.** A VMM provides an environment for programs that is essentially identical to the original machine.
- **Performance.** Programs running within that environment show only minor performance decreases.
- **Safety.** The VMM is in complete control of system resources.

These requirements still guide virtualization efforts today.

By the late 1990s, Intel 80x86 CPUs had become common, fast, and rich in features. Accordingly, developers launched multiple efforts to implement virtualization on that platform. Both **Xen** and **VMware** created technologies, still used today, to allow guest operating systems to run on the 80x86. Since that time, virtualization has expanded to include all common CPUs, many commercial and open-source tools, and many operating systems. For example, the open-source *VirtualBox* project (<http://www.virtualbox.org>) provides a program that runs on Intel x86 and AMD 64 CPUs and on Windows, Linux, macOS, and Solaris host operating systems. Possible guest operating systems include many versions of Windows, Linux, Solaris, and BSD, including even MS-DOS and IBM OS/2.

### 18.3 Benefits and Features

Several advantages make virtualization attractive. Most of them are fundamentally related to the ability to share the same hardware yet run several different execution environments (that is, different operating systems) concurrently.

One important advantage of virtualization is that the host system is protected from the virtual machines, just as the virtual machines are protected from each other. A virus inside a guest operating system might damage that operating system but is unlikely to affect the host or the other guests. Because

each virtual machine is almost completely isolated from all other virtual machines, there are almost no protection problems.

A potential disadvantage of isolation is that it can prevent sharing of resources. Two approaches to providing sharing have been implemented. First, it is possible to share a file-system volume and thus to share files. Second, it is possible to define a network of virtual machines, each of which can send information over the virtual communications network. The network is modeled after physical communication networks but is implemented in software. Of course, the VMM is free to allow any number of its guests to use physical resources, such as a physical network connection (with sharing provided by the VMM), in which case the allowed guests could communicate with each other via the physical network.

One feature common to most virtualization implementations is the ability to freeze, or **suspend**, a running virtual machine. Many operating systems provide that basic feature for processes, but VMMs go one step further and allow copies and **snapshots** to be made of the guest. The copy can be used to create a new VM or to move a VM from one machine to another with its current state intact. The guest can then **resume** where it was, as if on its original machine, creating a **clone**. The snapshot records a point in time, and the guest can be reset to that point if necessary (for example, if a change was made but is no longer wanted). Often, VMMs allow many snapshots to be taken. For example, snapshots might record a guest's state every day for a month, making restoration to any of those snapshot states possible. These abilities are used to good advantage in virtual environments.

A virtual machine system is a perfect vehicle for operating-system research and development. Normally, changing an operating system is a difficult task. Operating systems are large and complex programs, and a change in one part may cause obscure bugs to appear in some other part. The power of the operating system makes changing it particularly dangerous. Because the operating system executes in kernel mode, a wrong change in a pointer could cause an error that would destroy the entire file system. Thus, it is necessary to test all changes to the operating system carefully.

Of course, the operating system runs on and controls the entire machine, so the system must be stopped and taken out of use while changes are made and tested. This period is commonly called **system-development time**. Since it makes the system unavailable to users, system-development time on shared systems is often scheduled late at night or on weekends, when system load is low.

A virtual-machine system can eliminate much of this latter problem. System programmers are given their own virtual machine, and system development is done on the virtual machine instead of on a physical machine. Normal system operation is disrupted only when a completed and tested change is ready to be put into production.

Another advantage of virtual machines for developers is that multiple operating systems can run concurrently on the developer's workstation. This virtualized workstation allows for rapid porting and testing of programs in varying environments. In addition, multiple versions of a program can run, each in its own isolated operating system, within one system. Similarly, quality-assurance engineers can test their applications in multiple environments without buying, powering, and maintaining a computer for each environment.

A major advantage of virtual machines in production data-center use is system **consolidation**, which involves taking two or more separate systems and running them in virtual machines on one system. Such physical-to-virtual conversions result in resource optimization, since many lightly used systems can be combined to create one more heavily used system.

Consider, too, that management tools that are part of the VMM allow system administrators to manage many more systems than they otherwise could. A virtual environment might include 100 physical servers, each running 20 virtual servers. Without virtualization, 2,000 servers would require several system administrators. With virtualization and its tools, the same work can be managed by one or two administrators. One of the tools that make this possible is **templating**, in which one standard virtual machine image, including an installed and configured guest operating system and applications, is saved and used as a source for multiple running VMs. Other features include managing the patching of all guests, backing up and restoring the guests, and monitoring their resource use.

Virtualization can improve not only resource utilization but also resource management. Some VMMs include a **live migration** feature that moves a running guest from one physical server to another without interrupting its operation or active network connections. If a server is overloaded, live migration can thus free resources on the source host while not disrupting the guest. Similarly, when host hardware must be repaired or upgraded, guests can be migrated to other servers, the evacuated host can be maintained, and then the guests can be migrated back. This operation occurs without downtime and without interruption to users.

Think about the possible effects of virtualization on how applications are deployed. If a system can easily add, remove, and move a virtual machine, then why install applications on that system directly? Instead, the application could be preinstalled on a tuned and customized operating system in a virtual machine. This method would offer several benefits for application developers. Application management would become easier, less tuning would be required, and technical support of the application would be more straightforward. System administrators would find the environment easier to manage as well. Installation would be simple, and redeploying the application to another system would be much easier than the usual steps of uninstalling and reinstalling. For widespread adoption of this methodology to occur, though, the format of virtual machines must be standardized so that any virtual machine will run on any virtualization platform. The “Open Virtual Machine Format” is an attempt to provide such standardization, and it could succeed in unifying virtual machine formats.

Virtualization has laid the foundation for many other advances in computer facility implementation, management, and monitoring. **Cloud computing**, for example, is made possible by virtualization in which resources such as CPU, memory, and I/O are provided as services to customers using Internet technologies. By using APIs, a program can tell a cloud computing facility to create thousands of VMs, all running a specific guest operating system and application, that others can access via the Internet. Many multiuser games, photo-sharing sites, and other web services use this functionality.

In the area of desktop computing, virtualization is enabling desktop and laptop computer users to connect remotely to virtual machines located in

remote data centers and access their applications as if they were local. This practice can increase security, because no data are stored on local disks at the user's site. The cost of the user's computing resource may also decrease. The user must have networking, CPU, and some memory, but all that these system components need to do is display an image of the guest as its runs remotely (via a protocol such as [RDP](#)). Thus, they need not be expensive, high-performance components. Other uses of virtualization are sure to follow as it becomes more prevalent and hardware support continues to improve.

## 18.4 Building Blocks

Although the virtual machine concept is useful, it is difficult to implement. Much work is required to provide an *exact* duplicate of the underlying machine. This is especially a challenge on dual-mode systems, where the underlying machine has only user mode and kernel mode. In this section, we examine the building blocks that are needed for efficient virtualization. Note that these building blocks are not required by type 0 hypervisors, as discussed in Section 18.5.2.

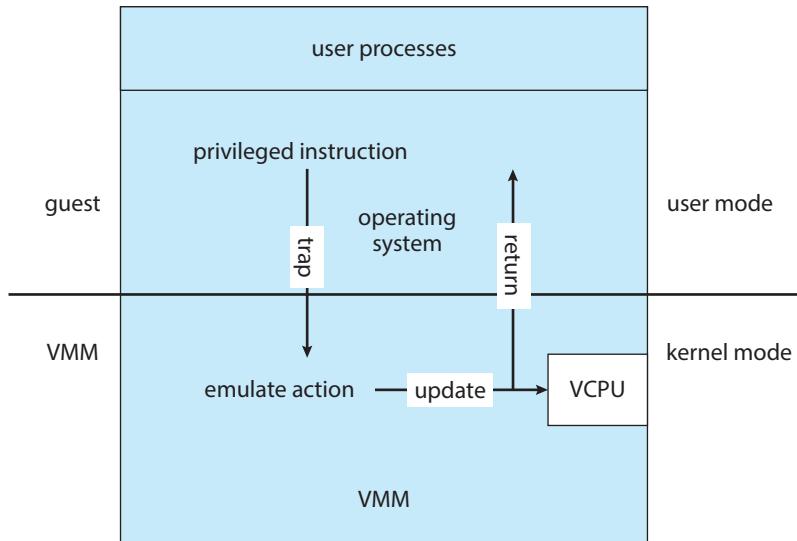
The ability to virtualize depends on the features provided by the CPU. If the features are sufficient, then it is possible to write a VMM that provides a guest environment. Otherwise, virtualization is impossible. VMMs use several techniques to implement virtualization, including trap-and-emulate and binary translation. We discuss each of these techniques in this section, along with the hardware support needed to support virtualization.

As you read the section, keep in mind that an important concept found in most virtualization options is the implementation of a [virtual CPU \(VCPU\)](#). The VCPU does not execute code. Rather, it represents the state of the CPU as the guest machine believes it to be. For each guest, the VMM maintains a VCPU representing that guest's current CPU state. When the guest is context-switched onto a CPU by the VMM, information from the VCPU is used to load the right context, much as a general-purpose operating system would use the PCB.

### 18.4.1 Trap-and-Emulate

On a typical dual-mode system, the virtual machine guest can execute only in user mode (unless extra hardware support is provided). The kernel, of course, runs in kernel mode, and it is not safe to allow user-level code to run in kernel mode. Just as the physical machine has two modes, so must the virtual machine. Consequently, we must have a virtual user mode and a virtual kernel mode, both of which run in physical user mode. Those actions that cause a transfer from user mode to kernel mode on a real machine (such as a system call, an interrupt, or an attempt to execute a privileged instruction) must also cause a transfer from virtual user mode to virtual kernel mode in the virtual machine.

How can such a transfer be accomplished? The procedure is as follows: When the kernel in the guest attempts to execute a privileged instruction, that is an error (because the system is in user mode) and causes a trap to the VMM in the real machine. The VMM gains control and executes (or "emulates") the action that was attempted by the guest kernel on the part of the guest. It



**Figure 18.2** Trap-and-emulate virtualization implementation.

then returns control to the virtual machine. This is called the **trap-and-emulate** method and is shown in Figure 18.2.

With privileged instructions, time becomes an issue. All nonprivileged instructions run natively on the hardware, providing the same performance for guests as native applications. Privileged instructions create extra overhead, however, causing the guest to run more slowly than it would natively. In addition, the CPU is being multiprogrammed among many virtual machines, which can further slow down the virtual machines in unpredictable ways.

This problem has been approached in various ways. IBM VM, for example, allows normal instructions for the virtual machines to execute directly on the hardware. Only the privileged instructions (needed mainly for I/O) must be emulated and hence execute more slowly. In general, with the evolution of hardware, the performance of trap-and-emulate functionality has been improved, and cases in which it is needed have been reduced. For example, many CPUs now have extra modes added to their standard dual-mode operation. The VCPU need not keep track of what mode the guest operating system is in, because the physical CPU performs that function. In fact, some CPUs provide guest CPU state management in hardware, so the VMM need not supply that functionality, removing the extra overhead.

### 18.4.2 Binary Translation

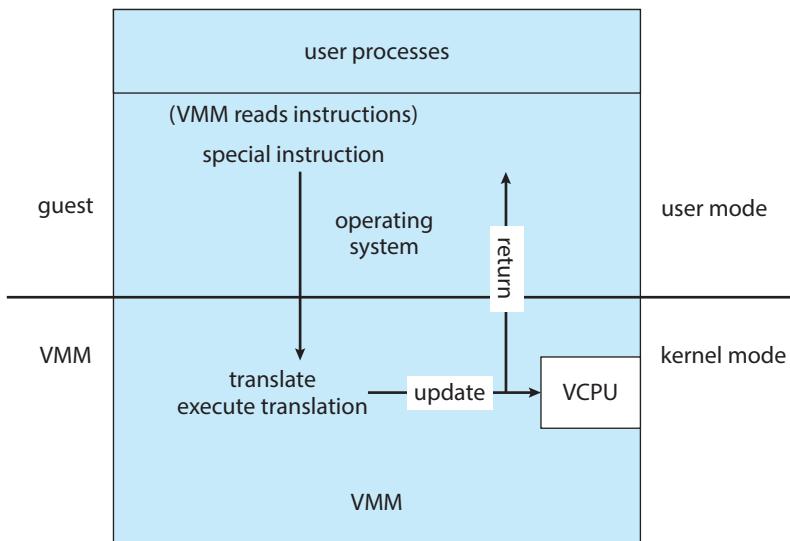
Some CPUs do not have a clean separation of privileged and nonprivileged instructions. Unfortunately for virtualization implementers, the Intel x86 CPU line is one of them. No thought was given to running virtualization on the x86 when it was designed. (In fact, the first CPU in the family—the Intel 4004, released in 1971—was designed to be the core of a calculator.) The chip has maintained backward compatibility throughout its lifetime, preventing changes that would have made virtualization easier through many generations.

Let's consider an example of the problem. The command `popf` loads the flag register from the contents of the stack. If the CPU is in privileged mode, all of the flags are replaced from the stack. If the CPU is in user mode, then only some flags are replaced, and others are ignored. Because no trap is generated if `popf` is executed in user mode, the trap-and-emulate procedure is rendered useless. Other x86 instructions cause similar problems. For the purposes of this discussion, we will call this set of instructions *special instructions*. As recently as 1998, using the trap-and-emulate method to implement virtualization on the x86 was considered impossible because of these special instructions.

This previously insurmountable problem was solved with the implementation of the **binary translation** technique. Binary translation is fairly simple in concept but complex in implementation. The basic steps are as follows:

1. If the guest VCPU is in user mode, the guest can run its instructions natively on a physical CPU.
2. If the guest VCPU is in kernel mode, then the guest believes that it is running in kernel mode. The VMM examines every instruction the guest executes in virtual kernel mode by reading the next few instructions that the guest is going to execute, based on the guest's program counter. Instructions other than special instructions are run natively. Special instructions are translated into a new set of instructions that perform the equivalent task—for example, changing the flags in the VCPU.

Binary translation is shown in Figure 18.3. It is implemented by translation code within the VMM. The code reads native binary instructions dynamically from the guest, on demand, and generates native binary code that executes in place of the original code.



**Figure 18.3** Binary translation virtualization implementation.

The basic method of binary translation just described would execute correctly but perform poorly. Fortunately, the vast majority of instructions would execute natively. But how could performance be improved for the other instructions? We can turn to a specific implementation of binary translation, the VMware method, to see one way of improving performance. Here, caching provides the solution. The replacement code for each instruction that needs to be translated is cached. All later executions of that instruction run from the translation cache and need not be translated again. If the cache is large enough, this method can greatly improve performance.

Let's consider another issue in virtualization: memory management, specifically the page tables. How can the VMM keep page-table state both for guests that believe they are managing the page tables and for the VMM itself? A common method, used with both trap-and-emulate and binary translation, is to use **nested page tables (NPTs)**. Each guest operating system maintains one or more page tables to translate from virtual to physical memory. The VMM maintains NPTs to represent the guest's page-table state, just as it creates a VCPU to represent the guest's CPU state. The VMM knows when the guest tries to change its page table, and it makes the equivalent change in the NPT. When the guest is on the CPU, the VMM puts the pointer to the appropriate NPT into the appropriate CPU register to make that table the active page table. If the guest needs to modify the page table (for example, fulfilling a page fault), then that operation must be intercepted by the VMM and appropriate changes made to the nested and system page tables. Unfortunately, the use of NPTs can cause TLB misses to increase, and many other complexities need to be addressed to achieve reasonable performance.

Although it might seem that the binary translation method creates large amounts of overhead, it performed well enough to launch a new industry aimed at virtualizing Intel x86-based systems. VMware tested the performance impact of binary translation by booting one such system, Windows XP, and immediately shutting it down while monitoring the elapsed time and the number of translations produced by the binary translation method. The result was 950,000 translations, taking 3 microseconds each, for a total increase of 3 seconds (about 5 percent) over native execution of Windows XP. To achieve that result, developers used many performance improvements that we do not discuss here. For more information, consult the bibliographical notes at the end of this chapter.

#### 18.4.3 Hardware Assistance

Without some level of hardware support, virtualization would be impossible. The more hardware support available within a system, the more feature-rich and stable the virtual machines can be and the better they can perform. In the Intel x86 CPU family, Intel added new virtualization support (the **VT-x** instructions) in successive generations beginning in 2005. Now, binary translation is no longer needed.

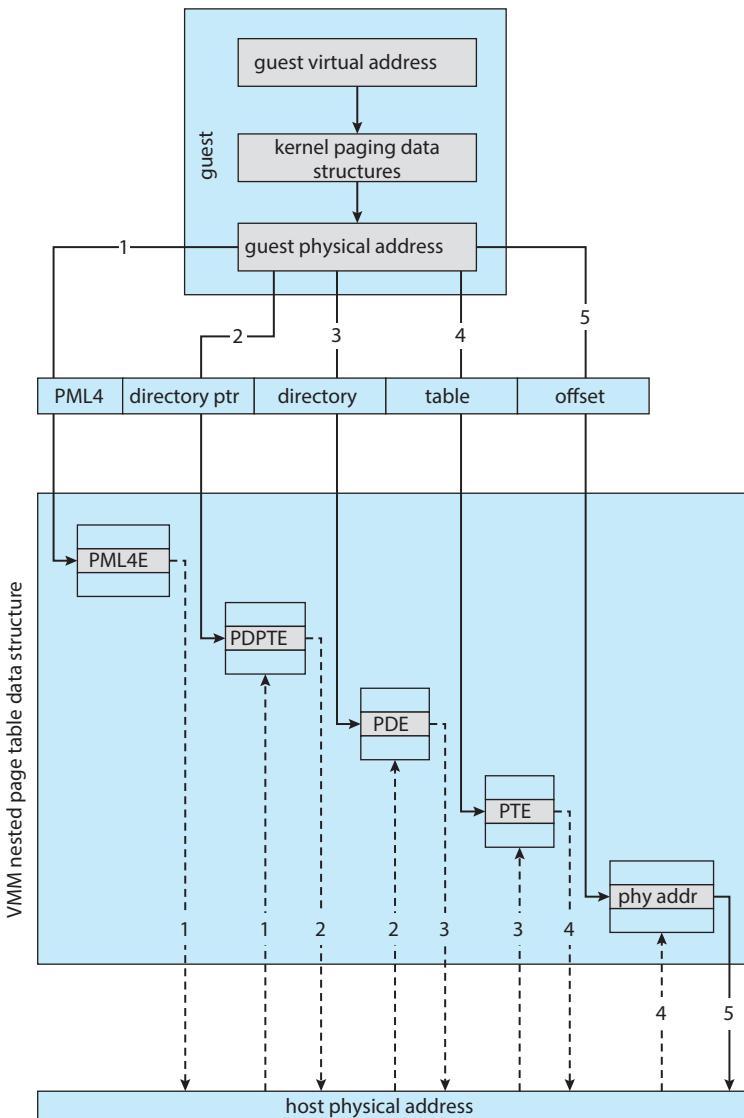
In fact, all major general-purpose CPUs now provide extended hardware support for virtualization. For example, AMD virtualization technology (**AMD-V**) has appeared in several AMD processors starting in 2006. It defines two new modes of operation—host and guest—thus moving from a dual-mode to a

multimode processor. The VMM can enable host mode, define the characteristics of each guest virtual machine, and then switch the system to guest mode, passing control of the system to a guest operating system that is running in the virtual machine. In guest mode, the virtualized operating system thinks it is running on native hardware and sees whatever devices are included in the host's definition of the guest. If the guest tries to access a virtualized resource, then control is passed to the VMM to manage that interaction. The functionality in Intel VT-x is similar, providing root and nonroot modes, equivalent to host and guest modes. Both provide guest VCPU state data structures to load and save guest CPU state automatically during guest context switches. In addition, **virtual machine control structures (VMCSs)** are provided to manage guest and host state, as well as various guest execution controls, exit controls, and information about why guests exit back to the host. In the latter case, for example, a nested page-table violation caused by an attempt to access unavailable memory can result in the guest's exit.

AMD and Intel have also addressed memory management in the virtual environment. With AMD's RVI and Intel's EPT memory-management enhancements, VMMs no longer need to implement software NPTs. In essence, these CPUs implement nested page tables in hardware to allow the VMM to fully control paging while the CPUs accelerate the translation from virtual to physical addresses. The NPTs add a new layer, one representing the guest's view of logical-to-physical address translation. The CPU page-table walking function (traversing the data structure to find the desired data) includes this new layer as necessary, walking through the guest table to the VMM table to find the physical address desired. A TLB miss results in a performance penalty, because more tables (the guest and host page tables) must be traversed to complete the lookup. Figure 18.4 shows the extra translation work performed by the hardware to translate from a guest virtual address to a final physical address.

I/O is another area improved by hardware assistance. Consider that the standard direct-memory-access (DMA) controller accepts a target memory address and a source I/O device and transfers data between the two without operating-system action. Without hardware assistance, a guest might try to set up a DMA transfer that affects the memory of the VMM or other guests. In CPUs that provide hardware-assisted DMA (such as Intel CPUs with VT-d), even DMA has a level of indirection. First, the VMM sets up **protection domains** to tell the CPU which physical memory belongs to each guest. Next, it assigns the I/O devices to the protection domains, allowing them direct access to those memory regions and only those regions. The hardware then transforms the address in a DMA request issued by an I/O device to the host physical memory address associated with the I/O. In this manner, DMA transfers are passed through between a guest and a device without VMM interference.

Similarly, interrupts must be delivered to the appropriate guest and must not be visible to other guests. By providing an interrupt remapping feature, CPUs with virtualization hardware assistance automatically deliver an interrupt destined for a guest to a core that is currently running a thread of that guest. That way, the guest receives interrupts without any need for the VMM to intercede in their delivery. Without interrupt remapping, malicious guests could generate interrupts that could be used to gain control of the host system. (See the bibliographical notes at the end of this chapter for more details.)



**Figure 18.4** Nested page tables.

ARM architectures, specifically ARM v8 (64-bit) take a slightly different approach to hardware support of virtualization. They provide an entire exception level—EL2—which is even more privileged than that of the kernel (EL1). This allows the running of a secluded hypervisor, with its own MMU access and interrupt trapping. To allow for paravirtualization, a special instruction (HVC) is added. It allows the hypervisor to be called from guest kernels. This instruction can only be called from within kernel mode (EL1).

An interesting side effect of hardware-assisted virtualization is that it allows for the creation of thin hypervisors. A good example is macOS's hypervisor framework (“HyperVisor.framework”), which is an operating-system-supplied library that allows the creation of virtual machines in a few lines of

code. The actual work is done via system calls, which have the kernel call the privileged virtualization CPU instructions on behalf of the hypervisor process, allowing management of virtual machines without the hypervisor needing to load a kernel module of its own to execute those calls.

## 18.5 Types of VMs and Their Implementations

We've now looked at some of the techniques used to implement virtualization. Next, we consider the major types of virtual machines, their implementation, their functionality, and how they use the building blocks just described to create a virtual environment. Of course, the hardware on which the virtual machines are running can cause great variation in implementation methods. Here, we discuss the implementations in general, with the understanding that VMMs take advantage of hardware assistance where it is available.

### 18.5.1 The Virtual Machine Life Cycle

Let's begin with the virtual machine life cycle. Whatever the hypervisor type, at the time a virtual machine is created, its creator gives the VMM certain parameters. These parameters usually include the number of CPUs, amount of memory, networking details, and storage details that the VMM will take into account when creating the guest. For example, a user might want to create a new guest with two virtual CPUs, 4 GB of memory, 10 GB of disk space, one network interface that gets its IP address via DHCP, and access to the DVD drive.

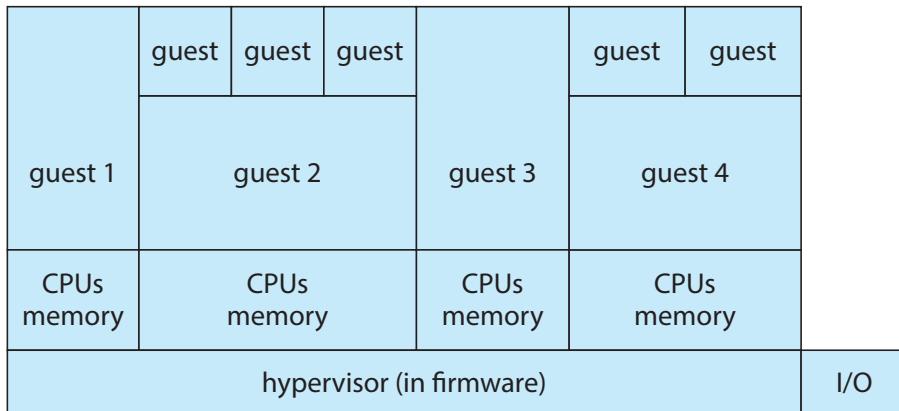
The VMM then creates the virtual machine with those parameters. In the case of a type 0 hypervisor, the resources are usually dedicated. In this situation, if there are not two virtual CPUs available and unallocated, the creation request in our example will fail. For other hypervisor types, the resources are dedicated or virtualized, depending on the type. Certainly, an IP address cannot be shared, but the virtual CPUs are usually multiplexed on the physical CPUs as discussed in Section 18.6.1. Similarly, memory management usually involves allocating more memory to guests than actually exists in physical memory. This is more complicated and is described in Section 18.6.2.

Finally, when the virtual machine is no longer needed, it can be deleted. When this happens, the VMM first frees up any used disk space and then removes the configuration associated with the virtual machine, essentially forgetting the virtual machine.

These steps are quite simple compared with building, configuring, running, and removing physical machines. Creating a virtual machine from an existing one can be as easy as clicking the “clone” button and providing a new name and IP address. This ease of creation can lead to **virtual machine sprawl**, which occurs when there are so many virtual machines on a system that their use, history, and state become confusing and difficult to track.

### 18.5.2 Type 0 Hypervisor

Type 0 hypervisors have existed for many years under many names, including “partitions” and “domains.” They are a hardware feature, and that brings its own positives and negatives. Operating systems need do nothing special to take advantage of their features. The VMM itself is encoded in the firmware and



**Figure 18.5** Type 0 hypervisor.

loaded at boot time. In turn, it loads the guest images to run in each partition. The feature set of a type 0 hypervisor tends to be smaller than those of the other types because it is implemented in hardware. For example, a system might be split into four virtual systems, each with dedicated CPUs, memory, and I/O devices. Each guest believes that it has dedicated hardware because it does, simplifying many implementation details.

I/O presents some difficulty, because it is not easy to dedicate I/O devices to guests if there are not enough. What if a system has two Ethernet ports and more than two guests, for example? Either all guests must get their own I/O devices, or the system must provide I/O device sharing. In these cases, the hypervisor manages shared access or grants all devices to a **control partition**. In the control partition, a guest operating system provides services (such as networking) via daemons to other guests, and the hypervisor routes I/O requests appropriately. Some type 0 hypervisors are even more sophisticated and can move physical CPUs and memory between running guests. In these cases, the guests are paravirtualized, aware of the virtualization and assisting in its execution. For example, a guest must watch for signals from the hardware or VMM that a hardware change has occurred, probe its hardware devices to detect the change, and add or subtract CPUs or memory from its available resources.

Because type 0 virtualization is very close to raw hardware execution, it should be considered separately from the other methods discussed here. A type 0 hypervisor can run multiple guest operating systems (one in each hardware partition). All of those guests, because they are running on raw hardware, can in turn be VMMs. Essentially, each guest operating system in a type 0 hypervisor is a native operating system with a subset of hardware made available to it. Because of that, each can have its own guest operating systems (Figure 18.5). Other types of hypervisors usually cannot provide this virtualization-within-virtualization functionality.

### 18.5.3 Type 1 Hypervisor

Type 1 hypervisors are commonly found in company data centers and are, in a sense, becoming “the data-center operating system.” They are special-purpose operating systems that run natively on the hardware, but rather than providing

system calls and other interfaces for running programs, they create, run, and manage guest operating systems. In addition to running on standard hardware, they can run on type 0 hypervisors, but not on other type 1 hypervisors. Whatever the platform, guests generally do not know they are running on anything but the native hardware.

Type 1 hypervisors run in kernel mode, taking advantage of hardware protection. Where the host CPU allows, they use multiple modes to give guest operating systems their own control and improved performance. They implement device drivers for the hardware they run on, since no other component could do so. Because they are operating systems, they must also provide CPU scheduling, memory management, I/O management, protection, and even security. Frequently, they provide APIs, but those APIs support applications in guests or external applications that supply features like backups, monitoring, and security. Many type 1 hypervisors are closed-source commercial offerings, such as VMware ESX, while some are open source or hybrids of open and closed source, such as Citrix XenServer and its open Xen counterpart.

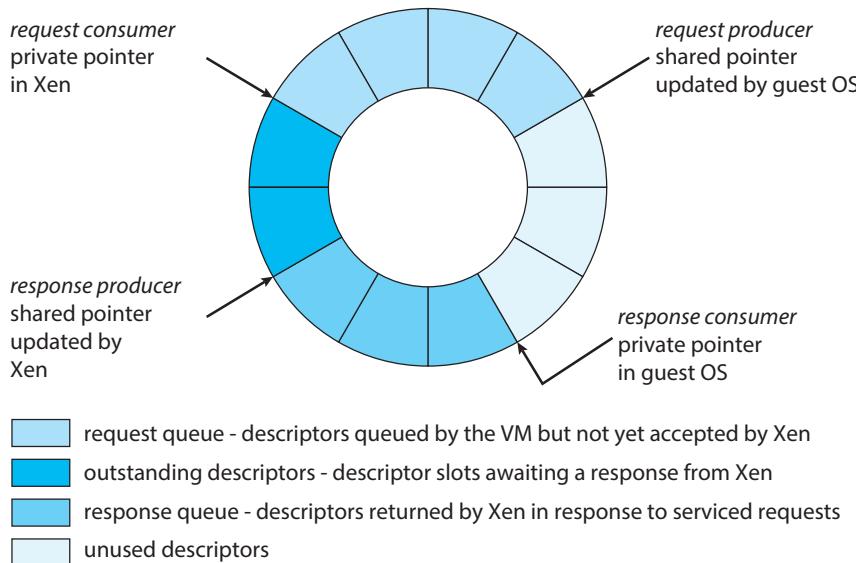
By using type 1 hypervisors, data-center managers can control and manage the operating systems and applications in new and sophisticated ways. An important benefit is the ability to consolidate more operating systems and applications onto fewer systems. For example, rather than having ten systems running at 10 percent utilization each, a data center might have one server manage the entire load. If utilization increases, guests and their applications can be moved to less-loaded systems live, without interruption of service. Using snapshots and cloning, the system can save the states of guests and duplicate those states—a much easier task than restoring from backups or installing manually or via scripts and tools. The price of this increased manageability is the cost of the VMM (if it is a commercial product), the need to learn new management tools and methods, and the increased complexity.

Another type of type 1 hypervisor includes various general-purpose operating systems with VMM functionality. Here, an operating system such as Red-Hat Enterprise Linux, Windows, or Oracle Solaris performs its normal duties as well as providing a VMM allowing other operating systems to run as guests. Because of their extra duties, these hypervisors typically provide fewer virtualization features than other type 1 hypervisors. In many ways, they treat a guest operating system as just another process, but they provide special handling when the guest tries to execute special instructions.

#### 18.5.4 Type 2 Hypervisor

Type 2 hypervisors are less interesting to us as operating-system explorers, because there is very little operating-system involvement in these application-level virtual machine managers. This type of VMM is simply another process run and managed by the host, and even the host does not know that virtualization is happening within the VMM.

Type 2 hypervisors have limits not associated with some of the other types. For example, a user needs administrative privileges to access many of the hardware assistance features of modern CPUs. If the VMM is being run by a standard user without additional privileges, the VMM cannot take advantage of these features. Due to this limitation, as well as the extra overhead of running



**Figure 18.6** Xen I/O via shared circular buffer.<sup>1</sup>

a general-purpose operating system as well as guest operating systems, type 2 hypervisors tend to have poorer overall performance than type 0 or type 1.

As is often the case, the limitations of type 2 hypervisors also provide some benefits. They run on a variety of general-purpose operating systems, and running them requires no changes to the host operating system. A student can use a type 2 hypervisor, for example, to test a non-native operating system without replacing the native operating system. In fact, on an Apple laptop, a student could have versions of Windows, Linux, Unix, and less common operating systems all available for learning and experimentation.

### 18.5.5 Paravirtualization

As we've seen, paravirtualization works differently than the other types of virtualization. Rather than try to trick a guest operating system into believing it has a system to itself, paravirtualization presents the guest with a system that is similar but not identical to the guest's preferred system. The guest must be modified to run on the paravirtualized virtual hardware. The gain for this extra work is more efficient use of resources and a smaller virtualization layer.

The Xen VMM became the leader in paravirtualization by implementing several techniques to optimize the performance of guests as well as of the host system. For example, as mentioned earlier, some VMMs present virtual devices to guests that appear to be real devices. Instead of taking that approach, the Xen VMM presented clean and simple device abstractions that allow efficient I/O as well as good I/O-related communication between the guest and the VMM. For

---

<sup>1</sup>Barham, Paul. "Xen and the Art of Virtualization". SOSP '03 Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, p 164-177. ©2003 Association for Computing Machinery, Inc

each device used by each guest, there was a circular buffer shared by the guest and the VMM via shared memory. Read and write data are placed in this buffer, as shown in Figure 18.6.

For memory management, Xen did not implement nested page tables. Rather, each guest had its own set of page tables, set to read-only. Xen required the guest to use a specific mechanism, a **hypcall** from the guest to the hypervisor VMM, when a page-table change was needed. This meant that the guest operating system's kernel code must have been changed from the default code to these Xen-specific methods. To optimize performance, Xen allowed the guest to queue up multiple page-table changes asynchronously via hypercalls and then checked to ensure that the changes were complete before continuing operation.

Xen allowed virtualization of x86 CPUs without the use of binary translation, instead requiring modifications in the guest operating systems like the one described above. Over time, Xen has taken advantage of hardware features supporting virtualization. As a result, it no longer requires modified guests and essentially does not need the paravirtualization method. Paravirtualization is still used in other solutions, however, such as type 0 hypervisors.

### 18.5.6 Programming-Environment Virtualization

Another kind of virtualization, based on a different execution model, is the virtualization of programming *environments*. Here, a programming language is designed to run within a custom-built virtualized environment. For example, Oracle's Java has many features that depend on its running in the **Java virtual machine (JVM)**, including specific methods for security and memory management.

If we define virtualization as including only duplication of hardware, this is not really virtualization at all. But we need not limit ourselves to that definition. Instead, we can define a virtual environment, based on APIs, that provides a set of features we want to have available for a particular language and programs written in that language. Java programs run within the JVM environment, and the JVM is compiled to be a native program on systems on which it runs. This arrangement means that Java programs are written once and then can run on any system (including all of the major operating systems) on which a JVM is available. The same can be said of **interpreted languages**, which run inside programs that read each instruction and interpret it into native operations.

### 18.5.7 Emulation

Virtualization is probably the most common method for running applications designed for one operating system on a different operating system, but on the same CPU. This method works relatively efficiently because the applications were compiled for the instruction set that the target system uses.

But what if an application or operating system needs to run on a different CPU? Here, it is necessary to translate all of the source CPU's instructions so that they are turned into the equivalent instructions of the target CPU. Such an environment is no longer virtualized but rather is fully emulated.

**Emulation** is useful when the host system has one system architecture and the guest system was compiled for a different architecture. For example,

suppose a company has replaced its outdated computer system with a new system but would like to continue to run certain important programs that were compiled for the old system. The programs could be run in an emulator that translates each of the outdated system's instructions into the native instruction set of the new system. Emulation can increase the life of programs and allow us to explore old architectures without having an actual old machine.

As may be expected, the major challenge of emulation is performance. Instruction-set emulation may run an order of magnitude slower than native instructions, because it may take ten instructions on the new system to read, parse, and simulate an instruction from the old system. Thus, unless the new machine is ten times faster than the old, the program running on the new machine will run more slowly than it did on its native hardware. Another challenge for emulator writers is that it is difficult to create a correct emulator because, in essence, this task involves writing an entire CPU in software.

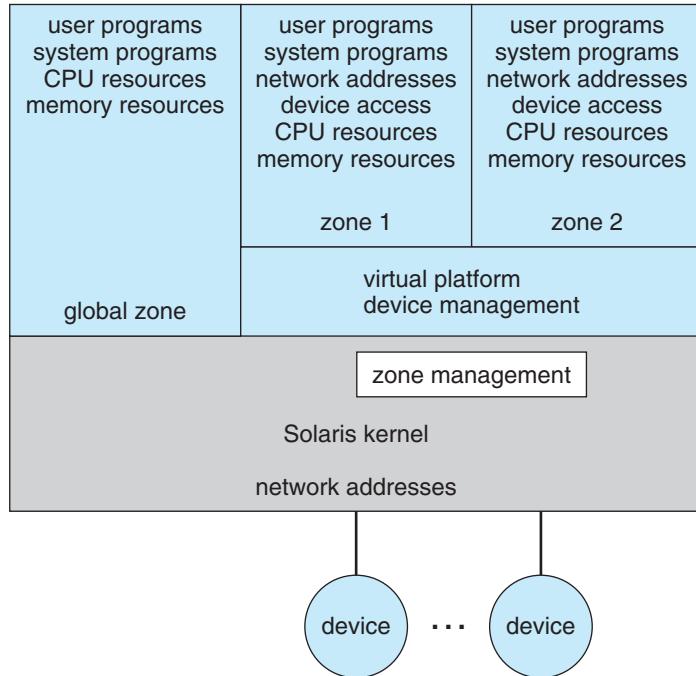
In spite of these challenges, emulation is very popular, particularly in gaming circles. Many popular video games were written for platforms that are no longer in production. Users who want to run those games frequently can find an emulator of such a platform and then run the game unmodified within the emulator. Modern systems are so much faster than old game consoles that even the Apple iPhone has game emulators and games available to run within them.

#### 18.5.8 Application Containment

The goal of virtualization in some instances is to provide a method to segregate applications, manage their performance and resource use, and create an easy way to start, stop, move, and manage them. In such cases, perhaps full-fledged virtualization is not needed. If the applications are all compiled for the same operating system, then we do not need complete virtualization to provide these features. We can instead use application containment.

Consider one example of application containment. Starting with version 10, Oracle Solaris has included **containers**, or **zones**, that create a virtual layer between the operating system and the applications. In this system, only one kernel is installed, and the hardware is not virtualized. Rather, the operating system and its devices are virtualized, providing processes within a zone with the impression that they are the only processes on the system. One or more containers can be created, and each can have its own applications, network stacks, network address and ports, user accounts, and so on. CPU and memory resources can be divided among the zones and the system-wide processes. Each zone, in fact, can run its own scheduler to optimize the performance of its applications on the allotted resources. Figure 18.7 shows a Solaris 10 system with two containers and the standard “global” user space.

Containers are much lighter weight than other virtualization methods. That is, they use fewer system resources and are faster to instantiate and destroy, more similar to processes than virtual machines. For this reason, they are becoming more commonly used, especially in cloud computing. FreeBSD was perhaps the first operating system to include a container-like feature (called “jails”), and AIX has a similar feature. Linux added the **LXC** container feature in 2014. It is now included in the common Linux distributions via



**Figure 18.7** Solaris 10 with two zones.

a flag in the `clone()` system call. (The source code for LXC is available at <https://linuxcontainers.org/lxc/downloads>.)

Containers are also easy to automate and manage, leading to orchestration tools like **Docker** and **Kubernetes**. Orchestration tools are means of automating and coordinating systems and services. Their aim is to make it simple to run entire suites of distributed applications, just as operating systems make it simple to run a single program. These tools offer rapid deployment of full applications, consisting of many processes within containers, and also offer monitoring and other administration features. For more on Docker, see <https://www.docker.com/what-docker>. Information about Kubernetes can be found at <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes>.

## 18.6 Virtualization and Operating-System Components

Thus far, we have explored the building blocks of virtualization and the various types of virtualization. In this section, we take a deeper dive into the operating-system aspects of virtualization, including how the VMM provides core operating-system functions like scheduling, I/O, and memory management. Here, we answer questions such as these: How do VMMs schedule CPU use when guest operating systems believe they have dedicated CPUs? How can memory management work when many guests require large amounts of memory?

### 18.6.1 CPU Scheduling

A system with virtualization, even a single-CPU system, frequently acts like a multiprocessor system. The virtualization software presents one or more virtual CPUs to each of the virtual machines running on the system and then schedules the use of the physical CPUs among the virtual machines.

The significant variations among virtualization technologies make it difficult to summarize the effect of virtualization on scheduling. First, let's consider the general case of VMM scheduling. The VMM has a number of physical CPUs available and a number of threads to run on those CPUs. The threads can be VMM threads or guest threads. Guests are configured with a certain number of virtual CPUs at creation time, and that number can be adjusted throughout the life of the VM. When there are enough CPUs to allocate the requested number to each guest, the VMM can treat the CPUs as dedicated and schedule only a given guest's threads on that guest's CPUs. In this situation, the guests act much like native operating systems running on native CPUs.

Of course, in other situations, there may not be enough CPUs to go around. The VMM itself needs some CPU cycles for guest management and I/O management and can steal cycles from the guests by scheduling its threads across all of the system CPUs, but the impact of this action is relatively minor. More difficult is the case of **overcommitment**, in which the guests are configured for more CPUs than exist in the system. Here, a VMM can use standard scheduling algorithms to make progress on each thread but can also add a fairness aspect to those algorithms. For example, if there are six hardware CPUs and twelve guest-allocated CPUs, the VMM can allocate CPU resources proportionally, giving each guest half of the CPU resources it believes it has. The VMM can still present all twelve virtual CPUs to the guests, but in mapping them onto physical CPUs, the VMM can use its scheduler to distribute them appropriately.

Even given a scheduler that provides fairness, any guest operating-system scheduling algorithm that assumes a certain amount of progress in a given amount of time will most likely be negatively affected by virtualization. Consider a time-sharing operating system that tries to allot 100 milliseconds to each time slice to give users a reasonable response time. Within a virtual machine, this operating system receives only what CPU resources the virtualization system gives it. A 100-millisecond time slice may take much more than 100 milliseconds of virtual CPU time. Depending on how busy the system is, the time slice may take a second or more, resulting in very poor response times for users logged into that virtual machine. The effect on a real-time operating system can be even more serious.

The net outcome of such scheduling is that individual virtualized operating systems receive only a portion of the available CPU cycles, even though they believe they are receiving all of the cycles and indeed are scheduling all of the cycles. Commonly, the time-of-day clocks in virtual machines are incorrect because timers take longer to trigger than they would on dedicated CPUs. Virtualization can thus undo the scheduling-algorithm efforts of the operating systems within virtual machines.

To correct for this, the VMM makes an application available for each type of operating system that the system administrator can install into the guests. This application corrects clock drift and can have other functions, such as virtual device management.

### 18.6.2 Memory Management

Efficient memory use in general-purpose operating systems is a major key to performance. In virtualized environments, there are more users of memory (the guests and their applications, as well as the VMM), leading to more pressure on memory use. Further adding to this pressure is the fact that VMMs typically overcommit memory, so that the total memory allocated to guests exceeds the amount that physically exists in the system. The extra need for efficient memory use is not lost on the implementers of VMMs, who take extensive measures to ensure the optimal use of memory.

For example, VMware ESX uses several methods of memory management. Before memory optimization can occur, the VMM must establish how much real memory each guest should use. To do that, the VMM first evaluates each guest's maximum memory size. General-purpose operating systems do not expect the amount of memory in the system to change, so VMMs must maintain the illusion that the guest has that amount of memory. Next, the VMM computes a target real-memory allocation for each guest based on the configured memory for that guest and other factors, such as overcommitment and system load. It then uses the three low-level mechanisms listed below to reclaim memory from the guests

1. Recall that a guest believes it controls memory allocation via its page-table management, whereas in reality the VMM maintains a nested page table that translates the guest page table to the real page table. The VMM can use this extra level of indirection to optimize the guest's use of memory without the guest's knowledge or help. One approach is to provide double paging. Here, the VMM has its own page-replacement algorithms and loads pages into a backing store that the guest believes is physical memory. Of course, the VMM knows less about the guest's memory access patterns than the guest does, so its paging is less efficient, creating performance problems. VMMs do use this method when other methods are not available or are not providing enough free memory. However, it is not the preferred approach.
2. A common solution is for the VMM to install in each guest a pseudo-device driver or kernel module that the VMM controls. (A **pseudo-device driver** uses device-driver interfaces, appearing to the kernel to be a device driver, but does not actually control a device. Rather, it is an easy way to add kernel-mode code without directly modifying the kernel.) This **balloon memory manager** communicates with the VMM and is told to allocate or deallocate memory. If told to allocate, it allocates memory and tells the operating system to pin the allocated pages into physical memory. Recall that pinning locks a page into physical memory so that it cannot be moved or paged out. To the guest, these pinned pages appear to decrease the amount of physical memory it has available, creating memory pressure. The guest then may free up other physical memory to be sure it has enough free memory. Meanwhile, the VMM, knowing that the pages pinned by the balloon process will never be used, removes those physical pages from the guest and allocates them to another guest. At the same time, the guest is using its own memory-management and paging algorithms to manage the available memory, which is the most

efficient option. If memory pressure within the entire system decreases, the VMM will tell the balloon process within the guest to unpin and free some or all of the memory, allowing the guest more pages for its use.

3. Another common method for reducing memory pressure is for the VMM to determine if the same page has been loaded more than once. If this is the case, the VMM reduces the number of copies of the page to one and maps the other users of the page to that one copy. VMware, for example, randomly samples guest memory and creates a hash for each page sampled. That hash value is a “thumbprint” of the page. The hash of every page examined is compared with other hashes stored in a hash table. If there is a match, the pages are compared byte by byte to see if they really are identical. If they are, one page is freed, and its logical address is mapped to the other’s physical address. This technique might seem at first to be ineffective, but consider that guests run operating systems. If multiple guests run the same operating system, then only one copy of the active operating-system pages need be in memory. Similarly, multiple guests could be running the same set of applications, again a likely source of memory sharing.

The overall effect of these mechanisms is to enable guests to behave and perform as if they had the full amount of memory requested, although in reality they have less.

### 18.6.3 I/O

In the area of I/O, hypervisors have some leeway and can be less concerned with how they represent the underlying hardware to their guests. Because of the wide variation in I/O devices, operating systems are used to dealing with varying and flexible I/O mechanisms. For example, an operating system’s device-driver mechanism provides a uniform interface to the operating system whatever the I/O device. Device-driver interfaces are designed to allow third-party hardware manufacturers to provide device drivers connecting their devices to the operating system. Usually, device drivers can be dynamically loaded and unloaded. Virtualization takes advantage of this built-in flexibility by providing specific virtualized devices to guest operating systems.

As described in Section 18.5, VMMs vary greatly in how they provide I/O to their guests. I/O devices may be dedicated to guests, for example, or the VMM may have device drivers onto which it maps guest I/O. The VMM may also provide idealized device drivers to guests. In this case, the guest sees an easy-to-control device, but in reality that simple device driver communicates to the VMM, which sends the requests to a more complicated real device through a more complex real device driver. I/O in virtual environments is complicated and requires careful VMM design and implementation.

Consider the case of a hypervisor and hardware combination that allows devices to be dedicated to a guest and allows the guest to access those devices directly. Of course, a device dedicated to one guest is not available to any other guests, but this direct access can still be useful in some circumstances. The reason to allow direct access is to improve I/O performance. The less the hypervisor has to do to enable I/O for its guests, the faster the I/O can occur. With type 0 hypervisors that provide direct device access, guests can often

run at the same speed as native operating systems. When type 0 hypervisors instead provide shared devices, performance may suffer.

With direct device access in type 1 and 2 hypervisors, performance can be similar to that of native operating systems if certain hardware support is present. The hardware needs to provide DMA pass-through with facilities like VT-d, as well as direct interrupt delivery (interrupts going directly to the guests). Given how frequently interrupts occur, it should be no surprise that the guests on hardware without these features have worse performance than if they were running natively.

In addition to direct access, VMMs provide shared access to devices. Consider a disk drive to which multiple guests have access. The VMM must provide protection while the device is being shared, assuring that a guest can access only the blocks specified in the guest's configuration. In such instances, the VMM must be part of every I/O, checking it for correctness as well as routing the data to and from the appropriate devices and guests.

In the area of networking, VMMs also have work to do. General-purpose operating systems typically have one Internet protocol (IP) address, although they sometimes have more than one—for example, to connect to a management network, backup network, and production network. With virtualization, each guest needs at least one IP address, because that is the guest's main mode of communication. Therefore, a server running a VMM may have dozens of addresses, and the VMM acts as a virtual switch to route the network packets to the addressed guests.

The guests can be “directly” connected to the network by an IP address that is seen by the broader network (this is known as **bridging**). Alternatively, the VMM can provide a **network address translation (NAT)** address. The NAT address is local to the server on which the guest is running, and the VMM provides routing between the broader network and the guest. The VMM also provides firewalling to guard connections between guests within the system and between guests and external systems.

#### 18.6.4 Storage Management

An important question in determining how virtualization works is this: If multiple operating systems have been installed, what and where is the boot disk? Clearly, virtualized environments need to approach storage management differently than do native operating systems. Even the standard multiboot method of slicing the boot disk into partitions, installing a boot manager in one partition, and installing each other operating system in another partition is not sufficient, because partitioning has limits that would prevent it from working for tens or hundreds of virtual machines.

Once again, the solution to this problem depends on the type of hypervisor. Type 0 hypervisors often allow root disk partitioning, partly because these systems tend to run fewer guests than other systems. Alternatively, a disk manager may be part of the control partition, and that disk manager may provide disk space (including boot disks) to the other partitions.

Type 1 hypervisors store the guest root disk (and configuration information) in one or more files in the file systems provided by the VMM. Type 2 hypervisors store the same information in the host operating system's file systems. In essence, a **disk image**, containing all of the contents of the root disk

of the guest, is contained in one file in the VMM. Aside from the potential performance problems that causes, this is a clever solution, because it simplifies copying and moving guests. If the administrator wants a duplicate of the guest (for testing, for example), she simply copies the associated disk image of the guest and tells the VMM about the new copy. Booting the new virtual machine brings up an identical guest. Moving a virtual machine from one system to another that runs the same VMM is as simple as halting the guest, copying the image to the other system, and starting the guest there.

Guests sometimes need more disk space than is available in their root disk image. For example, a nonvirtualized database server might use several file systems spread across many disks to store various parts of the database. Virtualizing such a database usually involves creating several files and having the VMM present those to the guest as disks. The guest then executes as usual, with the VMM translating the disk I/O requests coming from the guest into file I/O commands to the correct files.

Frequently, VMMs provide a mechanism to capture a physical system as it is currently configured and convert it to a guest that the VMM can manage and run. This **physical-to-virtual (P-to-V)** conversion reads the disk blocks of the physical system's disks and stores them in files on the VMM's system or on shared storage that the VMM can access. VMMs also provide a **virtual-to-physical (V-to-P)** procedure for converting a guest to a physical system. This procedure is sometimes needed for debugging: a problem could be caused by the VMM or associated components, and the administrator could attempt to solve the problem by removing virtualization from the problem variables. V-to-P conversion can take the files containing all of the guest data and generate disk blocks on a physical disk, recreating the guest as a native operating system and applications. Once the testing is concluded, the original system can be reused for other purposes when the virtual machine returns to service, or the virtual machine can be deleted and the original system can continue to run.

### 18.6.5 Live Migration

One feature not found in general-purpose operating systems but found in type 0 and type 1 hypervisors is the live migration of a running guest from one system to another. We mentioned this capability earlier. Here, we explore the details of how live migration works and why VMMs can implement it relatively easily while general-purpose operating systems, in spite of some research attempts, cannot.

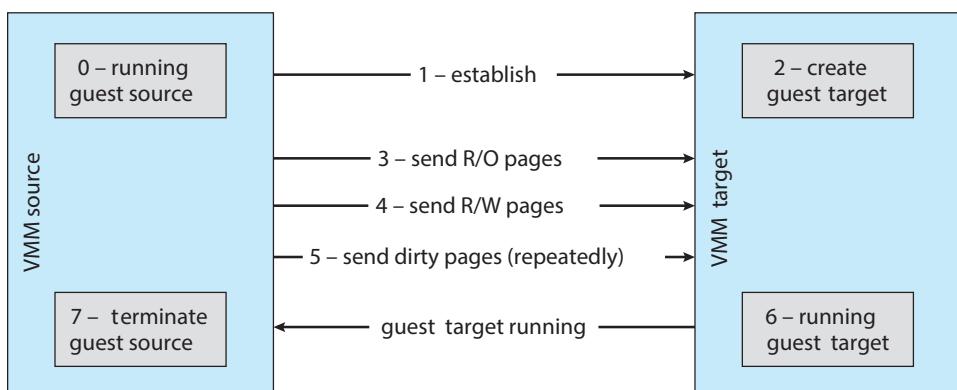
First, let's consider how live migration works. A running guest on one system is copied to another system running the same VMM. The copy occurs with so little interruption of service that users logged in to the guest, as well as network connections to the guest, continue without noticeable impact. This rather astonishing ability is very powerful in resource management and hardware administration. After all, compare it with the steps necessary without virtualization: we must warn users, shut down the processes, possibly move the binaries, and restart the processes on the new system. Only then can users access the services again. With live migration, we can decrease the load on an overloaded system or make hardware or system changes with no discernable disruption for users.

Live migration is made possible by the well-defined interface between each guest and the VMM and the limited state the VMM maintains for the guest. The VMM migrates a guest via the following steps:

1. The source VMM establishes a connection with the target VMM and confirms that it is allowed to send a guest.
2. The target creates a new guest by creating a new VCPU, new nested page table, and other state storage.
3. The source sends all read-only memory pages to the target.
4. The source sends all read-write pages to the target, marking them as clean.
5. The source repeats step 4, because during that step some pages were probably modified by the guest and are now dirty. These pages need to be sent again and marked again as clean.
6. When the cycle of steps 4 and 5 becomes very short, the source VMM freezes the guest, sends the VCPU's final state, other state details, and the final dirty pages, and tells the target to start running the guest. Once the target acknowledges that the guest is running, the source terminates the guest.

This sequence is shown in Figure 18.8.

We conclude this discussion with a few interesting details and limitations concerning live migration. First, for network connections to continue uninterrupted, the network infrastructure needs to understand that a MAC address—the hardware networking address—can move between systems. Before virtualization, this did not happen, as the MAC address was tied to physical hardware. With virtualization, the MAC must be movable for existing networking connections to continue without resetting. Modern network switches understand this and route traffic wherever the MAC address is, even accommodating a move.



**Figure 18.8** Live migration of a guest between two servers.

A limitation of live migration is that no disk state is transferred. One reason live migration is possible is that most of the guest's state is maintained within the guest—for example, open file tables, system-call state, kernel state, and so on. Because disk I/O is much slower than memory access, however, and used disk space is usually much larger than used memory, disks associated with the guest cannot be moved as part of a live migration. Rather, the disk must be remote to the guest, accessed over the network. In that case, disk access state is maintained within the guest, and network connections are all that matter to the VMM. The network connections are maintained during the migration, so remote disk access continues. Typically, NFS, CIFS, or iSCSI is used to store virtual machine images and any other storage a guest needs access to. These network-based storage accesses simply continue when the network connections are continued once the guest has been migrated.

Live migration makes it possible to manage data centers in entirely new ways. For example, virtualization management tools can monitor all the VMMs in an environment and automatically balance resource use by moving guests between the VMMs. These tools can also optimize the use of electricity and cooling by migrating all guests off selected servers if other servers can handle the load and powering down the selected servers entirely. If the load increases, the tools can power up the servers and migrate guests back to them.

## 18.7 Examples

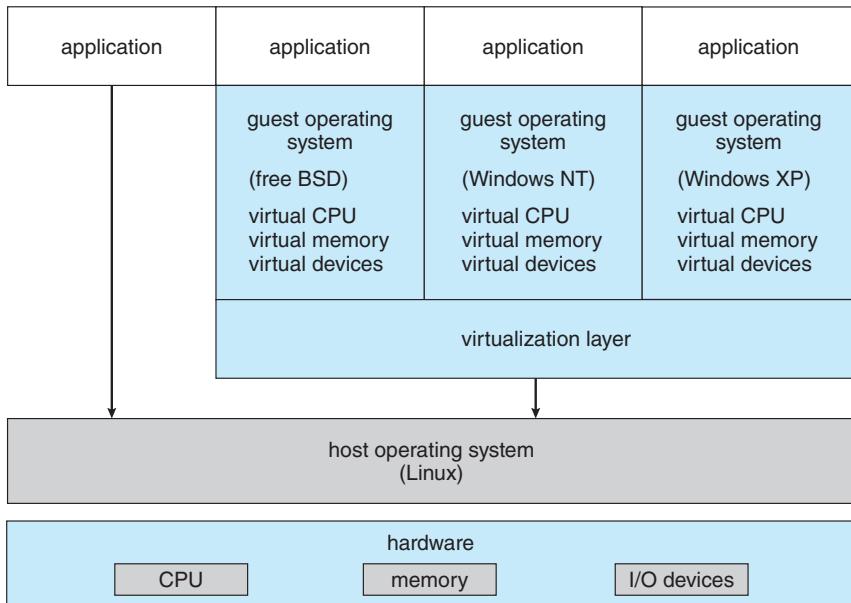
Despite the advantages of virtual machines, they received little attention for a number of years after they were first developed. Today, however, virtual machines are coming into greater use as a means of solving system compatibility problems. In this section, we explore two popular contemporary virtual machines: the VMware Workstation and the Java virtual machine. These virtual machines can typically run on top of operating systems of any of the design types discussed in earlier chapters.

### 18.7.1 VMware

**VMware Workstation** is a popular commercial application that abstracts Intel x86 and compatible hardware into isolated virtual machines. VMware Workstation is a prime example of a Type 2 hypervisor. It runs as an application on a host operating system such as Windows or Linux and allows this host system to run several different guest operating systems concurrently as independent virtual machines.

The architecture of such a system is shown in Figure 18.9. In this scenario, Linux is running as the host operating system, and FreeBSD, Windows NT, and Windows XP are running as guest operating systems. At the heart of VMware is the virtualization layer, which abstracts the physical hardware into isolated virtual machines running as guest operating systems. Each virtual machine has its own virtual CPU, memory, disk drives, network interfaces, and so forth.

The physical disk that the guest owns and manages is really just a file within the file system of the host operating system. To create an identical guest, we can simply copy the file. Copying the file to another location protects the guest against a disaster at the original site. Moving the file to another location



**Figure 18.9** VMware Workstation architecture.

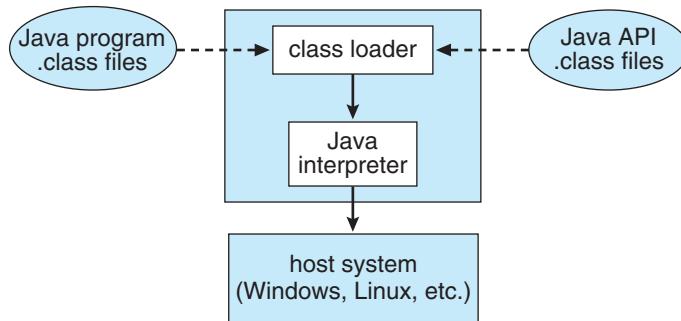
moves the guest system. Such capabilities, as explained earlier, can improve the efficiency of system administration as well as system resource use.

### 18.7.2 The Java Virtual Machine

Java is a popular object-oriented programming language introduced by Sun Microsystems in 1995. In addition to a language specification and a large API library, Java provides a specification for a Java virtual machine, or JVM. Java therefore is an example of programming-environment virtualization, as discussed in Section 18.5.6.

Java objects are specified with the `class` construct; a Java program consists of one or more classes. For each Java class, the compiler produces an architecture-neutral **bytecode** output (`.class`) file that will run on any implementation of the JVM.

The JVM is a specification for an abstract computer. It consists of a **class loader** and a Java interpreter that executes the architecture-neutral bytecodes, as diagrammed in Figure 18.10. The class loader loads the compiled `.class` files from both the Java program and the Java API for execution by the Java interpreter. After a class is loaded, the verifier checks that the `.class` file is valid Java bytecode and that it does not overflow or underflow the stack. It also ensures that the bytecode does not perform pointer arithmetic, which could provide illegal memory access. If the class passes verification, it is run by the Java interpreter. The JVM also automatically manages memory by performing **garbage collection**—the practice of reclaiming memory from objects no longer in use and returning it to the system. Much research focuses on garbage collection algorithms for increasing the performance of Java programs in the virtual machine.



**Figure 18.10** The Java virtual machine.

The JVM may be implemented in software on top of a host operating system, such as Windows, Linux, or macOS, or as part of a web browser. Alternatively, the JVM may be implemented in hardware on a chip specifically designed to run Java programs. If the JVM is implemented in software, the Java interpreter interprets the bytecode operations one at a time. A faster software technique is to use a **just-in-time (JIT)** compiler. Here, the first time a Java method is invoked, the bytecodes for the method are turned into native machine language for the host system. These operations are then cached so that subsequent invocations of a method are performed using the native machine instructions, and the bytecode operations need not be interpreted all over again. Running the JVM in hardware is potentially even faster. Here, a special Java chip executes the Java bytecode operations as native code, thus bypassing the need for either a software interpreter or a just-in-time compiler.

## 18.8 Virtualization Research

As mentioned earlier, machine virtualization has enjoyed growing popularity in recent years as a means of solving system compatibility problems. Research has expanded to cover many other uses of machine virtualization, including support for microservices running on library operating systems and secure partitioning of resources in embedded systems. Consequently, quite a lot of interesting, active research is underway.

Frequently, in the context of cloud computing, the same application is run on thousands of systems. To better manage those deployments, they can be virtualized. But consider the execution stack in that case—the application on top of a service-rich general-purpose operating system within a virtual machine managed by a hypervisor. Projects like **unikernels**, built on **library operating systems**, aim to improve efficiency and security in these environments. Unikernels are specialized machine images, using one address space, that shrink the attack surface and resource footprint of deployed applications. In essence, they compile the application, the system libraries it calls, and the kernel services it uses into a single binary that runs within a virtual environment (or even on bare metal). While research into changing how operating system kernels, hardware, and applications interact is not new (see <https://pdos.csail.mit.edu/6.828/2005/readings/engler95exokernel.pdf>,

for example), cloud computing and virtualization have created renewed interest in the area. See <http://unikernel.org> for more details.

The virtualization instructions in modern CPUs have given rise to a new branch of virtualization research focusing not on more efficient use of hardware but rather on better control of processes. Partitioning hypervisors partition the existing machine physical resources amongst guests, thereby fully committing rather than overcommitting machine resources. Partitioning hypervisors can securely extend the features of an existing operating system via functionality in another operating system (run in a separate guest VM domain), running on a subset of machine physical resources. This avoids the tedium of writing an entire operating system from scratch. For example, a Linux system that lacks real-time capabilities for safety- and security-critical tasks can be extended with a lightweight real-time operating system running in its own virtual machine. Traditional hypervisors have higher overhead than running native tasks, so a new type of hypervisor is needed.

Each task runs within a virtual machine, but the hypervisor only initializes the system and starts the tasks and is not involved with continuing operation. Each virtual machine has its own allocated hardware and is free to manage that hardware without interference from the hypervisor. Because the hypervisor does not interrupt task operations and is not called by the tasks, the tasks can have real-time aspects and can be much more secure.

Within the class of partitioning hypervisors are the [Quest-V](#), [eVMM](#), [Xtratum](#) and [Siemens Jailhouse](#) projects. These are [separation hypervisors](#) (see <http://www.csl.sri.com/users/rushby/papers/sosp81.pdf>) that use virtualization to partition separate system components into a chip-level distributed system. Secure shared memory channels are then implemented using hardware extended page tables so that separate sandboxed guests can communicate with one another. The targets of these projects are areas such as robotics, self-driving cars, and the Internet of Things. See <https://www.cs.bu.edu/richwest/papers/west-toocs16.pdf> for more details.

## 18.9 Summary

- Virtualization is a method for providing a guest with a duplicate of a system's underlying hardware. Multiple guests can run on a given system, each believing that it is the native operating system and is in full control.
- Virtualization started as a method to allow IBM to segregate users and provide them with their own execution environments on IBM mainframes. Since then, thanks to improvements in system and CPU performance and innovative software techniques, virtualization has become a common feature in data centers and even on personal computers. Because of its popularity, CPU designers have added features to support virtualization. This snowball effect is likely to continue, with virtualization and its hardware support increasing over time.
- The virtual machine manager, or hypervisor, creates and runs the virtual machine. Type 0 hypervisors are implemented in the hardware and require modifications to the operating system to ensure proper operation. Some

type 0 hypervisors offer an example of paravirtualization, in which the operating system is aware of virtualization and assists in its execution.

- Type 1 hypervisors provide the environment and features needed to create, run, and manage guest virtual machines. Each guest includes all of the software typically associated with a full native system, including the operating system, device drivers, applications, user accounts, and so on.
- Type 2 hypervisors are simply applications that run on other operating systems, which do not know that virtualization is taking place. These hypervisors do not have hardware or host support so must perform all virtualization activities in the context of a process.
- Programming-environment virtualization is part of the design of a programming language. The language specifies a containing application in which programs run, and this application provides services to the programs.
- Emulation is used when a host system has one architecture and the guest was compiled for a different architecture. Every instruction the guest wants to execute must be translated from its instruction set to that of the native hardware. Although this method involves some performance penalty, it is balanced by the usefulness of being able to run old programs on newer, incompatible hardware or run games designed for old consoles on modern hardware.
- Implementing virtualization is challenging, especially when hardware support is minimal. The more features provided by the system, the easier virtualization is to implement and the better the performance of the guests.
- VMMs take advantage of whatever hardware support is available when optimizing CPU scheduling, memory management, and I/O modules to provide guests with optimum resource use while protecting the VMM from the guests and the guests from one another.
- Current research is extending the uses of virtualization. Unikernels aim to increase efficiency and decrease security attack surface by compiling an application, its libraries, and the kernel resources the application needs into one binary with one address space that runs within a virtual machine. Partitioning hypervisors provide secure execution, real-time operation, and other features traditionally only available to applications running on dedicated hardware.

## Further Reading

The original IBM virtual machine is described in [Meyer and Seawright (1970)]. [Popek and Goldberg (1974)] established the characteristics that help define VMMs. Methods of implementing virtual machines are discussed in [Agesen et al. (2010)].

Intel x86 hardware virtualization support is described in [Neiger et al. (2006)]. AMD hardware virtualization support is described in a white paper available at <http://developer.amd.com/assets/NPT-WP-1%201-final-TM.pdf>.

Memory management in VMware is described in [Waldspurger (2002)]. [Gordon et al. (2012)] propose a solution to the problem of I/O overhead in virtualized environments. Some protection challenges and attacks in virtual environments are discussed in [Wojtczuk and Ruthkowska (2011)].

For early work on alternative kernel designs, see <https://pdos.csail.mit.edu/6.828/2005/readings/engler95exokernel.pdf>. For more on unikernels, see [West et al. (2016)] and <http://unikernel.org>. Partitioning hypervisors are discussed in <http://ethdocs.org/en/latest/introduction/what-is-ethereum.html>, and <https://lwn.net/Articles/578295> and [Madhavapeddy et al. (2013)]. Quest-V, a separation hypervisor, is detailed in <http://www.csl.sri.com/users/rushby/papers/sosp81.pdf> and <https://www.cs.bu.edu/richwest/papers/west-tocs16.pdf>.

The open-source *VirtualBox* project is available from <http://www.virtualbox.org>. The source code for LXC is available at <https://linuxcontainers.org/lxc/downloads>.

For more on docker, see <https://www.docker.com/what-docker>. Information about Kubernetes can be found at <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes>.

## Bibliography

- [Agesen et al. (2010)] O. Agesen, A. Garthwaite, J. Sheldon, and P. Subrahmanyam, “The Evolution of an x86 Virtual Machine Monitor”, *Proceedings of the ACM Symposium on Operating Systems Principles* (2010), pages 3–18.
- [Gordon et al. (2012)] A. Gordon, N. A. N. Har’El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir, “ELI: Bare-metal Performance for I/O Virtualization”, *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (2012), pages 411–422.
- [Madhavapeddy et al. (2013)] A. Madhavapeddy, R. Mirtier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, “Unikernels: Library Operating Systems for the Cloud” (2013).
- [Meyer and Seawright (1970)] R. A. Meyer and L. H. Seawright, “A Virtual Machine Time-Sharing System”, *IBM Systems Journal*, Volume 9, Number 3 (1970), pages 199–218.
- [Neiger et al. (2006)] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig, “Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization”, *Intel Technology Journal*, Volume 10, (2006).
- [Popek and Goldberg (1974)] G. J. Popek and R. P. Goldberg, “Formal Requirements for Virtualizable Third Generation Architectures”, *Communications of the ACM*, Volume 17, Number 7 (1974), pages 412–421.
- [Waldspurger (2002)] C. Waldspurger, “Memory Resource Management in VMware ESX Server”, *Operating Systems Review*, Volume 36, Number 4 (2002), pages 181–194.
- [West et al. (2016)] R. West, Y. Li, E. Missimer, and M. Danish, “A Virtualized Separation Kernel for Mixed Criticality Systems”, Volume 34, (2016).

[**Wojtczuk and Ruthkowska (2011)**] R. Wojtczuk and J. Ruthkowska, “Following the White Rabbit: Software Attacks Against Intel VT-d Technology”, *The Invisible Things Lab’s blog* (2011).

## Chapter 18 Exercises

- 18.1** Describe the three types of traditional hypervisors.
- 18.2** Describe four virtualization-like execution environments, and explain how they differ from “true” virtualization.
- 18.3** Describe four benefits of virtualization.
- 18.4** Why are VMMs unable to implement trap-and-emulate-based virtualization on some CPUs? Lacking the ability to trap and emulate, what method can a VMM use to implement virtualization?
- 18.5** What hardware assistance for virtualization can be provided by modern CPUs?
- 18.6** Why is live migration possible in virtual environments but much less possible for a native operating system?

# *Networks and Distributed Systems*



Updated by Sarah Diesburg

A distributed system is a collection of processors that do not share memory or a clock. Instead, each node has its own local memory. The nodes communicate with one another through various networks, such as high-speed buses. Distributed systems are more relevant than ever, and you have almost certainly used some sort of distributed service. Applications of distributed systems range from providing transparent access to files inside an organization, to large-scale cloud file and photo storage services, to business analysis of trends on large data sets, to parallel processing of scientific data, and more. In fact, the most basic example of a distributed system is one we are all likely very familiar with—the Internet.

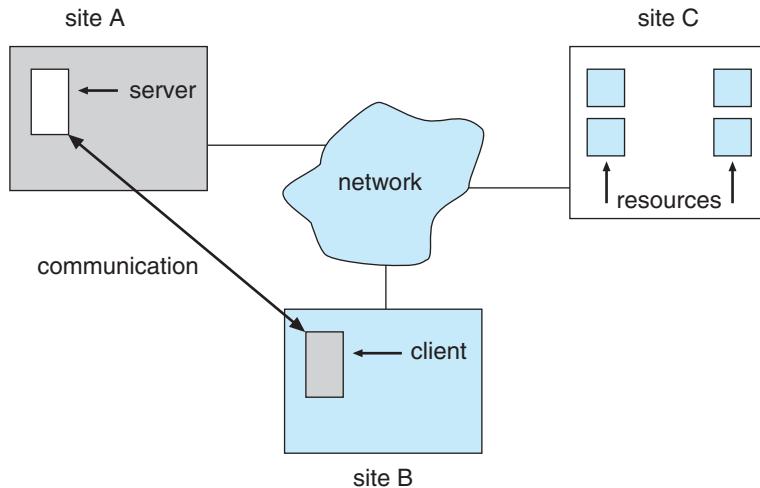
In this chapter, we discuss the general structure of distributed systems and the networks that interconnect them. We also contrast the main differences in the types and roles of current distributed system designs. Finally, we investigate some of the basic designs and design challenges of distributed file systems.

## CHAPTER OBJECTIVES

- Explain the advantages of networked and distributed systems.
- Provide a high-level overview of the networks that interconnect distributed systems.
- Define the roles and types of distributed systems in use today.
- Discuss issues concerning the design of distributed file systems.

### 19.1 Advantages of Distributed Systems

A **distributed system** is a collection of loosely coupled nodes interconnected by a communication network. From the point of view of a specific node in a distributed system, the rest of the nodes and their respective resources are remote, whereas its own resources are local.



**Figure 19.1** A client-server distributed system.

The nodes in a distributed system may vary in size and function. They may include small microprocessors, personal computers, and large general-purpose computer systems. These processors are referred to by a number of names, such as *processors*, *sites*, *machines*, and *hosts*, depending on the context in which they are mentioned. We mainly use *site* to indicate the location of a machine and *node* to refer to a specific system at a site. Nodes can exist in a *client–server* configuration, a *peer-to-peer* configuration, or a hybrid of these. In the common client–server configuration, one node at one site, the *server*, has a resource that another node, the *client* (or user), would like to use. A general structure of a client–server distributed system is shown in Figure 19.1. In a peer-to-peer configuration, there are no servers or clients. Instead, the nodes share equal responsibilities and can act as both clients and servers.

When several sites are connected to one another by a communication network, users at the various sites have the opportunity to exchange information. At a low level, **messages** are passed between systems, much as messages are passed between processes in the single-computer message system discussed in Section 3.4. Given message passing, all the higher-level functionality found in standalone systems can be expanded to encompass the distributed system. Such functions include file storage, execution of applications, and remote procedure calls (RPCs).

There are three major reasons for building distributed systems: resource sharing, computational speedup, and reliability. In this section, we briefly discuss each of them.

### 19.1.1 Resource Sharing

If a number of different sites (with different capabilities) are connected to one another, then a user at one site may be able to use the resources available at another. For example, a user at site A may query a database located at site B. Meanwhile, a user at site B may access a file that resides at site A. In general, **resource sharing** in a distributed system provides mechanisms for

sharing files at remote sites, processing information in a distributed database, printing files at remote sites, using remote specialized hardware devices such as a supercomputer or a **graphics processing unit (GPU)**, and performing other operations.

### 19.1.2 Computation Speedup

If a particular computation can be partitioned into subcomputations that can run concurrently, then a distributed system allows us to distribute the subcomputations among the various sites. The subcomputations can be run concurrently and thus provide **computation speedup**. This is especially relevant when doing large-scale processing of big data sets (such as analyzing large amounts of customer data for trends). In addition, if a particular site is currently overloaded with requests, some of them can be moved or rerouted to other, more lightly loaded sites. This movement of jobs is called **load balancing** and is common among distributed system nodes and other services provided on the Internet.

### 19.1.3 Reliability

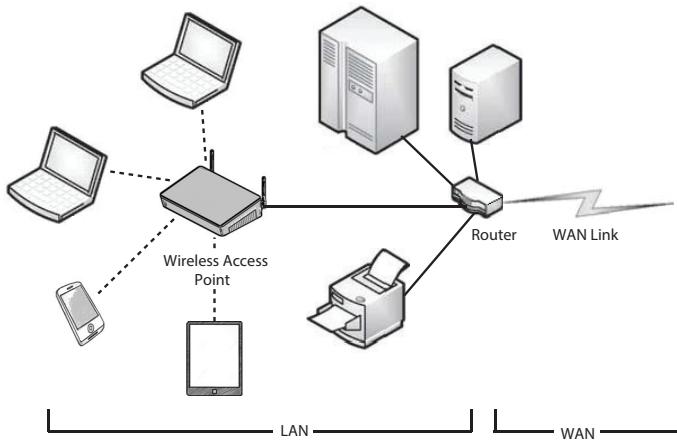
If one site fails in a distributed system, the remaining sites can continue operating, giving the system better reliability. If the system is composed of multiple large autonomous installations (that is, general-purpose computers), the failure of one of them should not affect the rest. If, however, the system is composed of diversified machines, each of which is responsible for some crucial system function (such as the web server or the file system), then a single failure may halt the operation of the whole system. In general, with enough redundancy (in both hardware and data), the system can continue operation even if some of its nodes have failed.

The failure of a node or site must be detected by the system, and appropriate action may be needed to recover from the failure. The system must no longer use the services of that site. In addition, if the function of the failed site can be taken over by another site, the system must ensure that the transfer of function occurs correctly. Finally, when the failed site recovers or is repaired, mechanisms must be available to integrate it back into the system smoothly.

## 19.2 Network Structure

To completely understand the roles and types of distributed systems in use today, we need to understand the networks that interconnect them. This section serves as a network primer to introduce basic networking concepts and challenges as they relate to distributed systems. The rest of the chapter specifically discusses distributed systems.

There are basically two types of networks: **local-area networks (LAN)** and **wide-area networks (WAN)**. The main difference between the two is the way in which they are geographically distributed. Local-area networks are composed of hosts distributed over small areas (such as a single building or a number of adjacent buildings), whereas wide-area networks are composed of systems distributed over a large area (such as the United States). These differences



**Figure 19.2** Local-area network.

imply major variations in the speed and reliability of the communications networks, and they are reflected in the distributed system design.

### 19.2.1 Local-Area Networks

Local-area networks emerged in the early 1970s as a substitute for large mainframe computer systems. For many enterprises, it is more economical to have a number of small computers, each with its own self-contained applications, than to have a single large system. Because each small computer is likely to need a full complement of peripheral devices (such as disks and printers), and because some form of data sharing is likely to occur in a single enterprise, it was a natural step to connect these small systems into a network.

LANs, as mentioned, are usually designed to cover a small geographical area, and they are generally used in an office or home environment. All the sites in such systems are close to one another, so the communication links tend to have a higher speed and lower error rate than their counterparts in wide-area networks.

A typical LAN may consist of a number of different computers (including workstations, servers, laptops, tablets, and smartphones), various shared peripheral devices (such as printers and storage arrays), and one or more **routers** (specialized network communication processors) that provide access to other networks (Figure 19.2). Ethernet and **WiFi** are commonly used to construct LANs. *Wireless access points* connect devices to the LAN wirelessly, and they may or may not be routers themselves.

Ethernet networks are generally found in businesses and organizations in which computers and peripherals tend to be nonmobile. These networks use *coaxial*, *twisted pair*, and/or *fiber optic* cables to send signals. An Ethernet network has no central controller, because it is a multiaccess bus, so new hosts can be added easily to the network. The Ethernet protocol is defined by the IEEE 802.3 standard. Typical Ethernet speeds using common twisted-pair cabling

can vary from 10 Mbps to over 10 Gbps, with other types of cabling reaching speeds of 100 Gbps.

WiFi is now ubiquitous and either supplements traditional Ethernet networks or exists by itself. Specifically, WiFi allows us to construct a network without using physical cables. Each host has a wireless transmitter and receiver that it uses to participate in the network. WiFi is defined by the IEEE 802.11 standard. Wireless networks are popular in homes and businesses, as well as public areas such as libraries, Internet cafes, sports arenas, and even buses and airplanes. WiFi speeds can vary from 11 Mbps to over 400 Mbps.

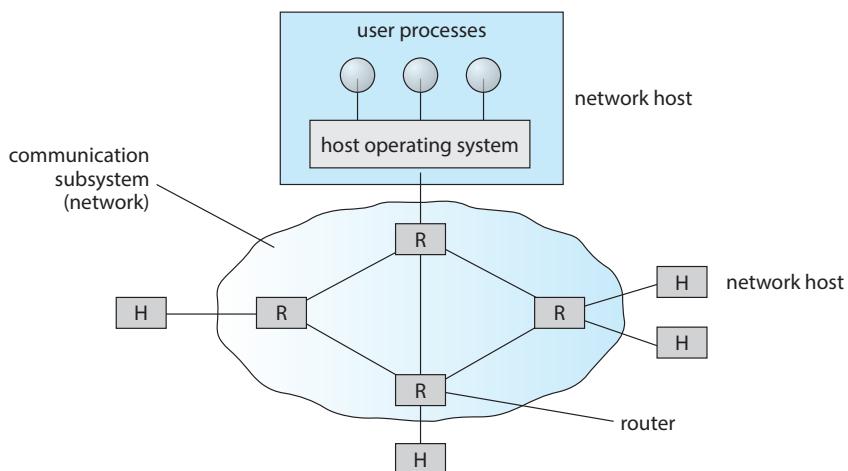
Both the IEEE 802.3 and 802.11 standards are constantly evolving. For the latest information about various standards and speeds, see the references at the end of the chapter.

### 19.2.2 Wide-Area Networks

Wide-area networks emerged in the late 1960s, mainly as an academic research project to provide efficient communication among sites, allowing hardware and software to be shared conveniently and economically by a wide community of users. The first WAN to be designed and developed was the ARPANET. Begun in 1968, the ARPANET has grown from a four-site experimental network to a worldwide network of networks, the [Internet](#) (also known as the [World Wide Web](#)), comprising millions of computer systems.

Sites in a WAN are physically distributed over a large geographical area. Typical links are telephone lines, leased (dedicated data) lines, optical cable, microwave links, radio waves, and satellite channels. These communication links are controlled by routers (Figure 19.3) that are responsible for directing traffic to other routers and networks and transferring information among the various sites.

For example, the Internet WAN enables hosts at geographically separate sites to communicate with one another. The host computers typically differ from one another in speed, CPU type, operating system, and so on. Hosts are



**Figure 19.3** Communication processors in a wide-area network.

generally on LANs, which are, in turn, connected to the Internet via regional networks. The regional networks are interlinked with routers to form the worldwide network. Residences can connect to the Internet by either telephone, cable, or specialized Internet service providers that install routers to connect the residences to central services. Of course, there are other WANs besides the Internet. A company, for example, might create its own private WAN for increased security, performance, or reliability.

WANs are generally slower than LANs, although backbone WAN connections that link major cities may have very fast transfer rates through fiber optic cables. In fact, many backbone providers have fiber optic speeds of 40 Gbps or 100 Gbps. (It is generally the links from local **Internet Service Providers (ISPs)** to homes or businesses that slow things down.) However, WAN links are being constantly updated to faster technologies as the demand for more speed continues to grow.

Frequently, WANs and LANs interconnect, and it is difficult to tell where one ends and the other starts. Consider the cellular phone data network. Cell phones are used for both voice and data communications. Cell phones in a given area connect via radio waves to a cell tower that contains receivers and transmitters. This part of the network is similar to a LAN except that the cell phones do not communicate with each other (unless two people talking or exchanging data happen to be connected to the same tower). Rather, the towers are connected to other towers and to hubs that connect the tower communications to land lines or other communication media and route the packets toward their destinations. This part of the network is more WAN-like. Once the appropriate tower receives the packets, it uses its transmitters to send them to the correct recipient.

## 19.3 Communication Structure

Now that we have discussed the physical aspects of networking, we turn to the internal workings.

### 19.3.1 Naming and Name Resolution

The first issue in network communication involves the naming of the systems in the network. For a process at site A to exchange information with a process at site B, each must be able to specify the other. Within a computer system, each process has a process identifier, and messages may be addressed with the process identifier. Because networked systems share no memory, however, a host within the system initially has no knowledge about the processes on other hosts.

To solve this problem, processes on remote systems are generally identified by the pair <host name, identifier>, where **host name** is a name unique within the network and **identifier** is a process identifier or other unique number within that host. A host name is usually an alphanumeric identifier, rather than a number, to make it easier for users to specify. For instance, site A might have hosts named *program*, *student*, *faculty*, and *cs*. The host name *program* is certainly easier to remember than the numeric host address 128.148.31.100.

Names are convenient for humans to use, but computers prefer numbers for speed and simplicity. For this reason, there must be a mechanism to **resolve** the host name into a **host-id** that describes the destination system to the networking hardware. This mechanism is similar to the name-to-address binding that occurs during program compilation, linking, loading, and execution (Chapter 9). In the case of host names, two possibilities exist. First, every host may have a data file containing the names and numeric addresses of all the other hosts reachable on the network (similar to binding at compile time). The problem with this model is that adding or removing a host from the network requires updating the data files on all the hosts. In fact, in the early days of the ARPANET there was a canonical host file that was copied to every system periodically. As the network grew, however, this method became untenable.

The alternative is to distribute the information among systems on the network. The network must then use a protocol to distribute and retrieve the information. This scheme is like execution-time binding. The Internet uses a **domain-name system (DNS)** for host-name resolution.

DNS specifies the naming structure of the hosts, as well as name-to-address resolution. Hosts on the Internet are logically addressed with multipart names known as IP addresses. The parts of an IP address progress from the most specific to the most general, with periods separating the fields. For instance, *eric.cs.yale.edu* refers to host *eric* in the Department of Computer Science at Yale University within the top-level domain *edu*. (Other top-level domains include *com* for commercial sites and *org* for organizations, as well as a domain for each country connected to the network for systems specified by country rather than organization type.) Generally, the system resolves addresses by examining the host-name components in reverse order. Each component has a **name server**—simply a process on a system—that accepts a name and returns the address of the name server responsible for that name. As the final step, the name server for the host in question is contacted, and a host-id is returned. For example, a request made by a process on system A to communicate with *eric.cs.yale.edu* would result in the following steps:

1. The system library or the kernel on system A issues a request to the name server for the *edu* domain, asking for the address of the name server for *yale.edu*. The name server for the *edu* domain must be at a known address, so that it can be queried.
2. The *edu* name server returns the address of the host on which the *yale.edu* name server resides.
3. System A then queries the name server at this address and asks about *cs.yale.edu*.
4. An address is returned. Now, finally, a request to that address for *eric.cs.yale.edu* returns an Internet address host-id for that host (for example, 128.148.31.100).

This protocol may seem inefficient, but individual hosts cache the IP addresses they have already resolved to speed the process. (Of course, the contents of these caches must be refreshed over time in case the name server is moved

```
/**  
 * Usage: java DNSLookUp <IP name>  
 * i.e. java DNSLookUp www.wiley.com  
 */  
public class DNSLookUp {  
    public static void main(String[] args) {  
        InetAddress hostAddress;  
  
        try {  
            hostAddress = InetAddress.getByName(args[0]);  
            System.out.println(hostAddress.getHostAddress());  
        }  
        catch (UnknownHostException uhe) {  
            System.err.println("Unknown host: " + args[0]);  
        }  
    }  
}
```

---

**Figure 19.4** Java program illustrating a DNS lookup.

or its address changes.) In fact, the protocol is so important that it has been optimized many times and has had many safeguards added. Consider what would happen if the primary *edu* name server crashed. It is possible that no *edu* hosts would be able to have their addresses resolved, making them all unreachable! The solution is to use secondary, backup name servers that duplicate the contents of the primary servers.

Before the domain-name service was introduced, all hosts on the Internet needed to have copies of a file (mentioned above) that contained the names and addresses of each host on the network. All changes to this file had to be registered at one site (host SRI-NIC), and periodically all hosts had to copy the updated file from SRI-NIC to be able to contact new systems or find hosts whose addresses had changed. Under the domain-name service, each name-server site is responsible for updating the host information for that domain. For instance, any host changes at Yale University are the responsibility of the name server for *yale.edu* and need not be reported anywhere else. DNS lookups will automatically retrieve the updated information because they will contact *yale.edu* directly. Domains may contain autonomous subdomains to further distribute the responsibility for host-name and host-id changes.

Java provides the necessary API to design a program that maps IP names to IP addresses. The program shown in Figure 19.4 is passed an IP name (such as *eric.cs.yale.edu*) on the command line and either outputs the IP address of the host or returns a message indicating that the host name could not be resolved. An *InetAddress* is a Java class representing an IP name or address. The static method *getByName()* belonging to the *InetAddress* class is passed a string representation of an IP name, and it returns the corresponding *InetAddress*. The program then invokes the *getHostAddress()* method, which internally uses DNS to look up the IP address of the designated host.

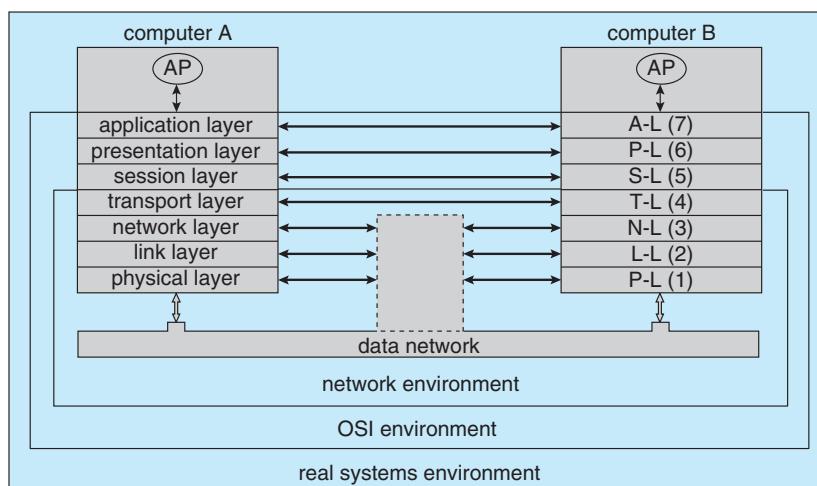
Generally, the operating system is responsible for accepting from its processes a message destined for <host name, identifier> and for transferring that message to the appropriate host. The kernel on the destination host is then responsible for transferring the message to the process named by the identifier. This process is described in Section 19.3.4.

### 19.3.2 Communication Protocols

When we are designing a communication network, we must deal with the inherent complexity of coordinating asynchronous operations communicating in a potentially slow and error-prone environment. In addition, the systems on the network must agree on a protocol or a set of protocols for determining host names, locating hosts on the network, establishing connections, and so on. We can simplify the design problem (and related implementation) by partitioning the problem into multiple layers. Each layer on one system communicates with the equivalent layer on other systems. Typically, each layer has its own protocols, and communication takes place between peer layers using a specific protocol. The protocols may be implemented in hardware or software. For instance, Figure 19.5 shows the logical communications between two computers, with the three lowest-level layers implemented in hardware.

The International Standards Organization created the Open Systems Interconnection (OSI) model for describing the various layers of networking. While these layers are not implemented in practice, they are useful for understanding how networking logically works, and we describe them below:

- **Layer 1: Physical layer.** The physical layer is responsible for handling both the mechanical and the electrical details of the physical transmission of a bit stream. At the physical layer, the communicating systems must agree on the electrical representation of a binary 0 and 1, so that when data are sent as a stream of electrical signals, the receiver is able to interpret the data



**Figure 19.5** Two computers communicating via the OSI network model.

properly as binary data. This layer is implemented in the hardware of the networking device. It is responsible for delivering bits.

- **Layer 2: Data-link layer.** The data-link layer is responsible for handling *frames*, or fixed-length parts of packets, including any error detection and recovery that occur in the physical layer. It sends frames between physical addresses.
- **Layer 3: Network layer.** The network layer is responsible for breaking messages into packets, providing connections between logical addresses, and routing packets in the communication network, including handling the addresses of outgoing packets, decoding the addresses of incoming packets, and maintaining routing information for proper response to changing load levels. Routers work at this layer.
- **Layer 4: Transport layer.** The transport layer is responsible for transfer of messages between nodes, maintaining packet order, and controlling flow to avoid congestion.
- **Layer 5: Session layer.** The session layer is responsible for implementing sessions, or process-to-process communication protocols.
- **Layer 6: Presentation layer.** The presentation layer is responsible for resolving the differences in formats among the various sites in the network, including character conversions and half duplex–full duplex modes (character echoing).
- **Layer 7: Application layer.** The application layer is responsible for interacting directly with users. This layer deals with file transfer, remote-login protocols, and electronic mail, as well as with schemas for distributed databases.

Figure 19.6 summarizes the **OSI protocol stack**—a set of cooperating protocols—showing the physical flow of data. As mentioned, logically each layer of a protocol stack communicates with the equivalent layer on other systems. But physically, a message starts at or above the application layer and is passed through each lower level in turn. Each layer may modify the message and include message-header data for the equivalent layer on the receiving side. Ultimately, the message reaches the data-network layer and is transferred as one or more packets (Figure 19.7). The data-link layer of the target system receives these data, and the message is moved up through the protocol stack. It is analyzed, modified, and stripped of headers as it progresses. It finally reaches the application layer for use by the receiving process.

The OSI model formalizes some of the earlier work done in network protocols but was developed in the late 1970s and is currently not in widespread use. Perhaps the most widely adopted protocol stack is the TCP/IP model (sometimes called the *Internet model*), which has been adopted by virtually all Internet sites. The TCP/IP protocol stack has fewer layers than the OSI model. Theoretically, because it combines several functions in each layer, it is more difficult to implement but more efficient than OSI networking. The relationship between the OSI and TCP/IP models is shown in Figure 19.8.

The TCP/IP application layer identifies several protocols in widespread use in the Internet, including HTTP, FTP, SSH, DNS, and SMTP. The transport layer

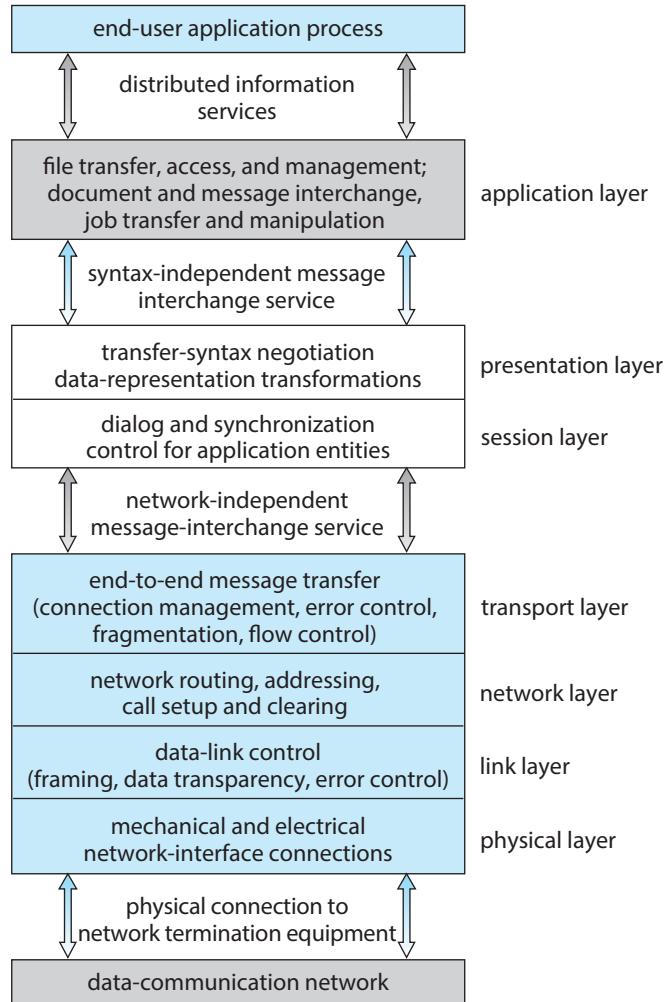
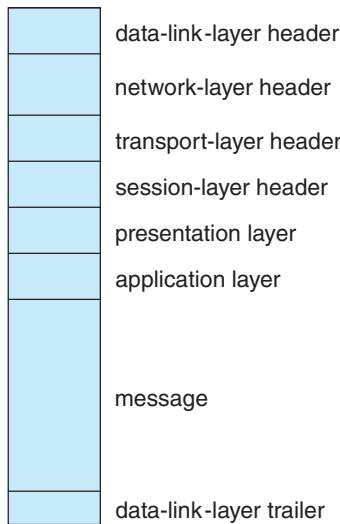


Figure 19.6 The OSI protocol stack.

identifies the unreliable, connectionless **user datagram protocol (UDP)** and the reliable, connection-oriented **transmission control protocol (TCP)**. The **Internet protocol (IP)** is responsible for routing IP **datagrams**, or packets, through the Internet. The TCP/IP model does not formally identify a link or physical layer, allowing TCP/IP traffic to run across any physical network. In Section 19.3.3, we consider the TCP/IP model running over an Ethernet network.

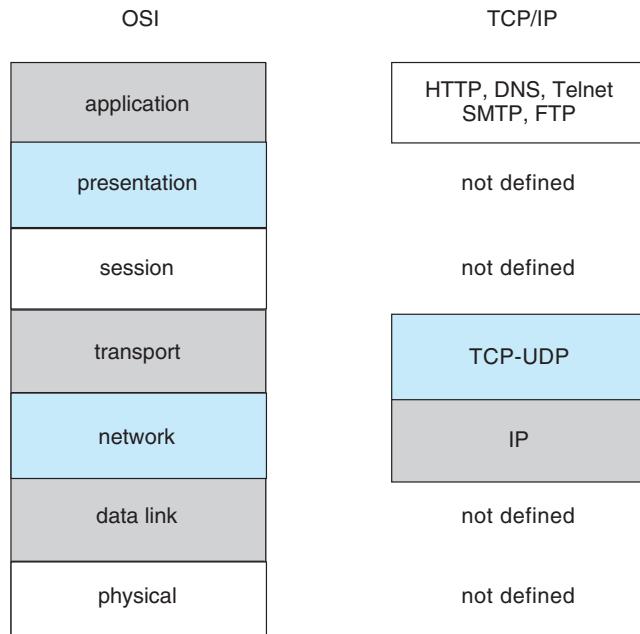
Security should be a concern in the design and implementation of any modern communication protocol. Both strong authentication and encryption are needed for secure communication. Strong authentication ensures that the sender and receiver of a communication are who or what they are supposed to be. Encryption protects the contents of the communication from eavesdropping. Weak authentication and clear-text communication are still very common, however, for a variety of reasons. When most of the common protocols were designed, security was frequently less important than performance, sim-



**Figure 19.7** An OSI network message.

plicity, and efficiency. This legacy is still showing itself today, as adding security to existing infrastructure is proving to be difficult and complex.

Strong authentication requires a multistep handshake protocol or authentication devices, adding complexity to a protocol. As to the encryption requirement, modern CPUs can efficiently perform encryption, frequently including cryptographic acceleration instructions so system performance is not compromised. Long-distance communication can be made secure by authenticating



**Figure 19.8** The OSI and TCP/IP protocol stacks.

the endpoints and encrypting the stream of packets in a virtual private network, as discussed in Section 16.4.2. LAN communication remains unencrypted at most sites, but protocols such as NFS Version 4, which includes strong native authentication and encryption, should help improve even LAN security.

### 19.3.3 TCP/IP Example

Next, we address name resolution and examine its operation with respect to the TCP/IP protocol stack on the Internet. Then we consider the processing needed to transfer a packet between hosts on different Ethernet networks. We base our description on the IPv4 protocols, which are the type most commonly used today.

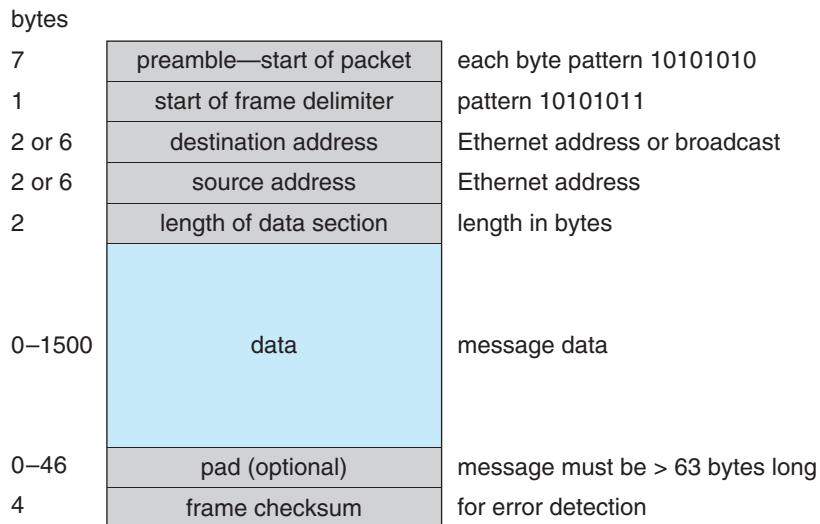
In a TCP/IP network, every host has a name and an associated IP address (or host-id). Both of these strings must be unique; and so that the name space can be managed, they are segmented. As described earlier, the name is hierarchical, describing the host name and then the organization with which the host is associated. The host-id is split into a network number and a host number. The proportion of the split varies, depending on the size of the network. Once the Internet administrators assign a network number, the site with that number is free to assign host-ids.

The sending system checks its routing tables to locate a router to send the frame on its way. This routing table is either configured manually by the system administrator or is populated by one of several routing protocols, such as the **Border Gateway Protocol (BGP)**. The routers use the network part of the host-id to transfer the packet from its source network to the destination network. The destination system then receives the packet. The packet may be a complete message, or it may just be a component of a message, with more packets needed before the message can be reassembled and passed to the TCP/UDP (transport) layer for transmission to the destination process.

Within a network, how does a packet move from sender (host or router) to receiver? Every Ethernet device has a unique byte number, called the **medium access control (MAC) address**, assigned to it for addressing. Two devices on a LAN communicate with each other only with this number. If a system needs to send data to another system, the networking software generates an **address resolution protocol (ARP)** packet containing the IP address of the destination system. This packet is **broadcast** to all other systems on that Ethernet network.

A broadcast uses a special network address (usually, the maximum address) to signal that all hosts should receive and process the packet. The broadcast is not re-sent by routers in between different networks, so only systems on the local network receive it. Only the system whose IP address matches the IP address of the ARP request responds and sends back its MAC address to the system that initiated the query. For efficiency, the host caches the IP–MAC address pair in an internal table. The cache entries are aged, so that an entry is eventually removed from the cache if an access to that system is not required within a given time. In this way, hosts that are removed from a network are eventually forgotten. For added performance, ARP entries for heavily used hosts may be pinned in the ARP cache.

Once an Ethernet device has announced its host-id and address, communication can begin. A process may specify the name of a host with which to communicate. Networking software takes that name and determines the IP address of the target, using a DNS lookup or an entry in a local hosts file

**Figure 19.9** An Ethernet packet.

where translations can be manually stored. The message is passed from the application layer, through the software layers, and to the hardware layer. At the hardware layer, the packet has the Ethernet address at its start; a trailer indicates the end of the packet and contains a **checksum** for detection of packet damage (Figure 19.9). The packet is placed on the network by the Ethernet device. The data section of the packet may contain some or all of the data of the original message, but it may also contain some of the upper-level headers that compose the message. In other words, all parts of the original message must be sent from source to destination, and all headers above the 802.3 layer (data-link layer) are included as data in the Ethernet packets.

If the destination is on the same local network as the source, the system can look in its ARP cache, find the Ethernet address of the host, and place the packet on the wire. The destination Ethernet device then sees its address in the packet and reads in the packet, passing it up the protocol stack.

If the destination system is on a network different from that of the source, the source system finds an appropriate router on its network and sends the packet there. Routers then pass the packet along the WAN until it reaches its destination network. The router that connects the destination network checks its ARP cache, finds the Ethernet number of the destination, and sends the packet to that host. Through all of these transfers, the data-link-layer header may change as the Ethernet address of the next router in the chain is used, but the other headers of the packet remain the same until the packet is received and processed by the protocol stack and finally passed to the receiving process by the kernel.

#### 19.3.4 Transport Protocols UDP and TCP

Once a host with a specific IP address receives a packet, it must somehow pass it to the correct waiting process. The transport protocols TCP and UDP identify the receiving (and sending) processes through the use of a **port number**. Thus,

a host with a single IP address can have multiple server processes running and waiting for packets as long as each server process specifies a different port number. By default, many common services use *well-known* port numbers. Some examples include FTP (21), SSH (22), SMTP (25), and HTTP (80). For example, if you wish to connect to an “http” website through your web browser, your browser will automatically attempt to connect to port 80 on the server by using the number 80 as the port number in the TCP transport header. For an extensive list of well-known ports, log into your favorite Linux or UNIX machine and take a look at the file /etc/services.

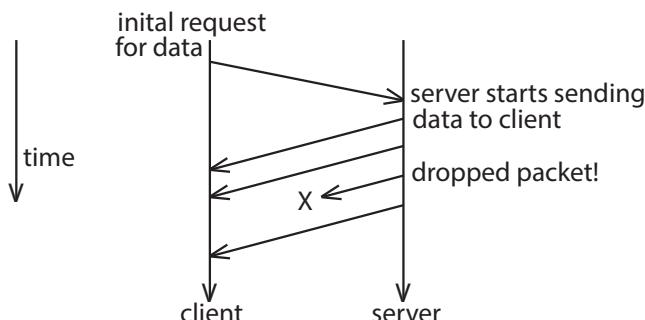
The transport layer can accomplish more than just connecting a network packet to a running process. It can also, if desired, add reliability to a network packet stream. To explain how, we next outline some general behavior of the transport protocols UDP and TCP.

#### 19.3.4.1 User Datagram Protocol

The transport protocol UDP is *unreliable* in that it is a bare-bones extension to IP with the addition of a port number. In fact, the UDP header is very simple and contains only four fields: source port number, destination port number, length, and checksum. Packets may be sent quickly to a destination using UDP. However, since there are no guarantees of delivery in the lower layers of the network stack, packets may become lost. Packets can also arrive at the receiver out of order. It is up to the application to figure out these error cases and to adjust (or not adjust).

Figure 19.10 illustrates a common scenario involving loss of a packet between a client and a server using the UDP protocol. Note that this protocol is known as a *connectionless* protocol because there is no connection setup at the beginning of the transmission to set up state—the client just starts sending data. Similarly, there is no connection teardown.

The client begins by sending some sort of request for information to the server. The server then responds by sending four datagrams, or packets, to the client. Unfortunately, one of the packets is dropped by an overwhelmed router. The client must either make do with only three packets or use logic programmed into the application to request the missing packet. Thus, we



**Figure 19.10** Example of a UDP data transfer with dropped packet.

need to use a different transport protocol if we want any additional reliability guarantees to be handled by the network.

#### 19.3.4.2 Transmission Control Protocol

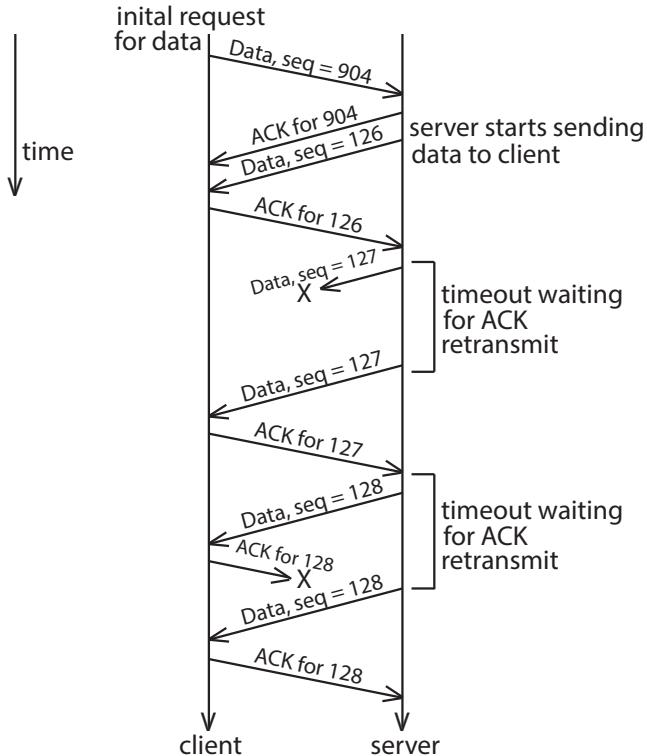
TCP is a transport protocol that is both *reliable* and *connection-oriented*. In addition to specifying port numbers to identify sending and receiving processes on different hosts, TCP provides an abstraction that allows a sending process on one host to send an in-order, uninterrupted *byte stream* across the network to a receiving process on another host. It accomplishes these things through the following mechanisms:

- Whenever a host sends a packet, the receiver must send an **acknowledgment packet**, or ACK, to notify the sender that the packet was received. If the ACK is not received before a timer expires, the sender will send that packet again.
- TCP introduces **sequence numbers** into the TCP header of every packet. These numbers allow the receiver to (1) put packets in order before sending data up to the requesting process and (2) be aware of packets missing from the byte stream.
- TCP connections are initiated with a series of control packets between the sender and the receiver (often called a *three-way handshake*) and closed gracefully with control packets responsible for tearing down the connection. These control packets allow both the sender and the receiver to set up and remove state.

Figure 19.11 demonstrates a possible exchange using TCP (with connection setup and tear-down omitted). After the connection has been established, the client sends a request packet to the server with the sequence number 904. Unlike the server in the UDP example, the server must then send an ACK packet back to the client. Next, the server starts sending its own stream of data packets starting with a different sequence number. The client sends an ACK packet for each data packet it receives. Unfortunately, the data packet with the sequence number 127 is lost, and no ACK packet is sent by the client. The sender times out waiting for the ACK packet, so it must resend data packet 127. Later in the connection, the server sends the data packet with the sequence number 128, but the ACK is lost. Since the server does not receive the ACK it must resend data packet 128. The client then receives a duplicate packet. Because the client knows that it previously received a packet with that sequence number, it throws the duplicate away. However, it must send another ACK back to the server to allow the server to continue.

In the actual TCP specification, an ACK isn't required for each and every packet. Instead, the receiver can send a *cumulative ACK* to ACK a series of packets. The server can also send numerous data packets sequentially before waiting for ACKs, to take advantage of network throughput.

TCP also helps regulate the flow of packets through mechanisms called *flow control* and *congestion control*. **Flow control** involves preventing the sender from overrunning the capacity of the receiver. For example, the receiver may



**Figure 19.11** Example of a TCP data transfer with dropped packets.

have a slower connection or may have slower hardware components (like a slower network card or processor). Flow-control state can be returned in the ACK packets of the receiver to alert the sender to slow down or speed up. **Congestion control** attempts to approximate the state of the networks (and generally the routers) between the sender and the receiver. If a router becomes overwhelmed with packets, it will tend to drop them. Dropping packets results in ACK timeouts, which results in more packets saturating the network. To prevent this condition, the sender monitors the connection for dropped packets by noticing how many packets are not acknowledged. If there are too many dropped packets, the sender will slow down the rate at which it sends them. This helps ensure that the TCP connection is being fair to other connections happening at the same time.

By utilizing a reliable transport protocol like TCP, a distributed system does not need extra logic to deal with lost or out-of-order packets. However, TCP is slower than UDP.

## 19.4 Network and Distributed Operating Systems

In this section, we describe the two general categories of network-oriented operating systems: network operating systems and distributed operating sys-

tems. Network operating systems are simpler to implement but generally more difficult for users to access and use than are distributed operating systems, which provide more features.

### 19.4.1 Network Operating Systems

A **network operating system** provides an environment in which users can access remote resources (implementing resource sharing) by either logging in to the appropriate remote machine or transferring data from the remote machine to their own machines. Currently, all general-purpose operating systems, and even embedded operating systems such as Android and iOS, are network operating systems.

#### 19.4.1.1 Remote Login

An important function of a network operating system is to allow users to log in remotely. The Internet provides the ssh facility for this purpose. To illustrate, suppose that a user at Westminster College wishes to compute on `kristen.cs.yale.edu`, a computer located at Yale University. To do so, the user must have a valid account on that machine. To log in remotely, the user issues the command

```
ssh kristen.cs.yale.edu
```

This command results in the formation of an encrypted socket connection between the local machine at Westminster College and the `kristen.cs.yale.edu` computer. After this connection has been established, the networking software creates a transparent, bidirectional link so that all characters entered by the user are sent to a process on `kristen.cs.yale.edu` and all the output from that process is sent back to the user. The process on the remote machine asks the user for a login name and a password. Once the correct information has been received, the process acts as a proxy for the user, who can compute on the remote machine just as any local user can.

#### 19.4.1.2 Remote File Transfer

Another major function of a network operating system is to provide a mechanism for **remote file transfer** from one machine to another. In such an environment, each computer maintains its own local file system. If a user at one site (say, Kurt at `albion.edu`) wants to access a file owned by Becca located on another computer (say, at `colby.edu`), then the file must be copied explicitly from the computer at Colby in Maine to the computer at Albion in Michigan. The communication is one-directional and individual, such that other users at those sites wishing to transfer a file, say Sean at `colby.edu` to Karen at `albion.edu`, must likewise issue a set of commands.

The Internet provides a mechanism for such a transfer with the file transfer protocol (FTP) and the more private secure file transfer protocol (SFTP). Suppose that user Carla at `wesleyan.edu` wants to copy a file that is owned by Owen at `kzoo.edu`. The user must first invoke the sftp program by executing

```
sftp owen@kzoo.edu
```

The program then asks the user for a login name and a password. Once the correct information has been received, the user can use a series of commands to upload files, download files, and navigate the remote file system structure. Some of these commands are:

- `get`—Transfer a file from the remote machine to the local machine.
- `put`—Transfer a file from the local machine to the remote machine.
- `ls` or `dir`—List files in the current directory on the remote machine.
- `cd`—Change the current directory on the remote machine.

There are also various commands to change transfer modes (for binary or ASCII files) and to determine connection status.

#### 19.4.1.3 Cloud Storage

Basic cloud-based storage applications allow users to transfer files much as with FTP. Users can upload files to a cloud server, download files to the local computer, and share files with other cloud-service users via a web link or other sharing mechanism through a graphical interface. Common examples include Dropbox and Google Drive.

An important point about SSH, FTP, and cloud-based storage applications is that they require the user to change paradigms. FTP, for example, requires the user to know a command set entirely different from the normal operating-system commands. With SSH, the user must know appropriate commands on the remote system. For instance, a user on a Windows machine who connects remotely to a UNIX machine must switch to UNIX commands for the duration of the SSH session. (In networking, a **session** is a complete round of communication, frequently beginning with a login to authenticate and ending with a logoff to terminate the communication.) With cloud-based storage applications, users may have to log into the cloud service (usually through a web browser) or native application and then use a series of graphical commands to upload, download, or share files. Obviously, users would find it more convenient not to be required to use a different set of commands. Distributed operating systems are designed to address this problem.

#### 19.4.2 Distributed Operating Systems

In a distributed operating system, users access remote resources in the same way they access local resources. Data and process migration from one site to another is under the control of the distributed operating system. Depending on the goals of the system, it can implement data migration, computation migration, process migration, or any combination thereof.

##### 19.4.2.1 Data Migration

Suppose a user on site A wants to access data (such as a file) that reside at site B. The system can transfer the data by one of two basic methods. One approach to **data migration** is to transfer the entire file to site A. From that point on, all access to the file is local. When the user no longer needs access to the file, a copy of the file (if it has been modified) is sent back to site B. Even if only a

modest change has been made to a large file, all the data must be transferred. This mechanism can be thought of as an automated FTP system. This approach was used in the Andrew file system, but it was found to be too inefficient.

The other approach is to transfer to site A only those portions of the file that are actually *necessary* for the immediate task. If another portion is required later, another transfer will take place. When the user no longer wants to access the file, any part of it that has been modified must be sent back to site B. (Note the similarity to demand paging.) Most modern distributed systems use this approach.

Whichever method is used, data migration includes more than the mere transfer of data from one site to another. The system must also perform various data translations if the two sites involved are not directly compatible (for instance, if they use different character-code representations or represent integers with a different number or order of bits).

#### 19.4.2.2 Computation Migration

In some circumstances, we may want to transfer the computation, rather than the data, across the system; this process is called **computation migration**. For example, consider a job that needs to access various large files that reside at different sites, to obtain a summary of those files. It would be more efficient to access the files at the sites where they reside and return the desired results to the site that initiated the computation. Generally, if the time to transfer the data is longer than the time to execute the remote command, the remote command should be used.

Such a computation can be carried out in different ways. Suppose that process P wants to access a file at site A. Access to the file is carried out at site A and could be initiated by an RPC. An RPC uses network protocols to execute a routine on a remote system (Section 3.8.2). Process P invokes a predefined procedure at site A. The procedure executes appropriately and then returns the results to P.

Alternatively, process P can send a message to site A. The operating system at site A then creates a new process Q whose function is to carry out the designated task. When process Q completes its execution, it sends the needed result back to P via the message system. In this scheme, process P may execute concurrently with process Q. In fact, it may have several processes running concurrently on several sites.

Either method could be used to access several files (or chunks of files) residing at various sites. One RPC might result in the invocation of another RPC or even in the transfer of messages to another site. Similarly, process Q could, during the course of its execution, send a message to another site, which in turn would create another process. This process might either send a message back to Q or repeat the cycle.

#### 19.4.2.3 Process Migration

A logical extension of computation migration is **process migration**. When a process is submitted for execution, it is not always executed at the site at which it is initiated. The entire process, or parts of it, may be executed at different sites. This scheme may be used for several reasons:

- **Load balancing.** The processes (or subprocesses) may be distributed across the sites to even the workload.
- **Computation speedup.** If a single process can be divided into a number of subprocesses that can run concurrently on different sites or nodes, then the total process turnaround time can be reduced.
- **Hardware preference.** The process may have characteristics that make it more suitable for execution on some specialized processor (such as matrix inversion on a GPU) than on a microprocessor.
- **Software preference.** The process may require software that is available at only a particular site, and either the software cannot be moved, or it is less expensive to move the process.
- **Data access.** Just as in computation migration, if the data being used in the computation are numerous, it may be more efficient to have a process run remotely (say, on a server that hosts a large database) than to transfer all the data and run the process locally.

We use two complementary techniques to move processes in a computer network. In the first, the system can attempt to hide the fact that the process has migrated from the client. The client then need not code her program explicitly to accomplish the migration. This method is usually employed for achieving load balancing and computation speedup among homogeneous systems, as they do not need user input to help them execute programs remotely.

The other approach is to allow (or require) the user to specify explicitly how the process should migrate. This method is usually employed when the process must be moved to satisfy a hardware or software preference.

You have probably realized that the World Wide Web has many aspects of a distributed computing environment. Certainly it provides data migration (between a web server and a web client). It also provides computation migration. For instance, a web client could trigger a database operation on a web server. Finally, with Java, Javascript, and similar languages, it provides a form of process migration: Java applets and Javascript scripts are sent from the server to the client, where they are executed. A network operating system provides most of these features, but a distributed operating system makes them seamless and easily accessible. The result is a powerful and easy-to-use facility —one of the reasons for the huge growth of the World Wide Web.

## 19.5 Design Issues in Distributed Systems

The designers of a distributed system must take a number of design challenges into account. The system should be robust so that it can withstand failures. The system should also be transparent to users in terms of both file location and user mobility. Finally, the system should be scalable to allow the addition of more computation power, more storage, or more users. We briefly introduce these issues here. In the next section, we put them in context when we describe the designs of specific distributed file systems.

### 19.5.1 Robustness

A distributed system may suffer from various types of hardware failure. The failure of a link, a host, or a site and the loss of a message are the most common types. To ensure that the system is robust, we must detect any of these failures, reconfigure the system so that computation can continue, and recover when the failure is repaired.

A system can be **fault tolerant** in that it can tolerate a certain level of failure and continue to function normally. The degree of fault tolerance depends on the design of the distributed system and the specific fault. Obviously, more fault tolerance is better.

We use the term **fault tolerance** in a broad sense. Communication faults, certain machine failures, storage-device crashes, and decays of storage media should all be tolerated to some extent. A **fault-tolerant system** should continue to function, perhaps in a degraded form, when faced with such failures. The degradation can affect performance, functionality, or both. It should be proportional, however, to the failures that caused it. A system that grinds to a halt when only one of its components fails is certainly not fault tolerant.

Unfortunately, fault tolerance can be difficult and expensive to implement. At the network layer, multiple redundant communication paths and network devices such as switches and routers are needed to avoid a communication failure. A storage failure can cause loss of the operating system, applications, or data. Storage units can include redundant hardware components that automatically take over from each other in case of failure. In addition, RAID systems can ensure continued access to the data even in the event of one or more storage device failures (Section 11.8).

#### 19.5.1.1 Failure Detection

In an environment with no shared memory, we generally cannot differentiate among link failure, site failure, host failure, and message loss. We can usually detect only that one of these failures has occurred. Once a failure has been detected, appropriate action must be taken. What action is appropriate depends on the particular application.

To detect link and site failure, we use a **heartbeat** procedure. Suppose that sites A and B have a direct physical link between them. At fixed intervals, the sites send each other an *I-am-up* message. If site A does not receive this message within a predetermined time period, it can assume that site B has failed, that the link between A and B has failed, or that the message from B has been lost. At this point, site A has two choices. It can wait for another time period to receive an *I-am-up* message from B, or it can send an *Are-you-up?* message to B.

If time goes by and site A still has not received an *I-am-up* message, or if site A has sent an *Are-you-up?* message and has not received a reply, the procedure can be repeated. Again, the only conclusion that site A can draw safely is that some type of failure has occurred.

Site A can try to differentiate between link failure and site failure by sending an *Are-you-up?* message to B by another route (if one exists). If and when B receives this message, it immediately replies positively. This positive reply tells A that B is up and that the failure is in the direct link between them. Since we do not know in advance how long it will take the message to travel from A to B and back, we must use a time-out scheme. At the time A sends the *Are-you-up?*

message, it specifies a time interval during which it is willing to wait for the reply from B. If A receives the reply message within that time interval, then it can safely conclude that B is up. If not, however (that is, if a time-out occurs), then A may conclude only that one or more of the following situations has occurred:

- Site B is down.
- The direct link (if one exists) from A to B is down.
- The alternative path from A to B is down.
- The message has been lost. (Although the use of a reliable transport protocol such as TCP should eliminate this concern.)

Site A cannot, however, determine which of these events has occurred.

#### 19.5.1.2 Reconfiguration

Suppose that site A has discovered, through the mechanism just described, that a failure has occurred. It must then initiate a procedure that will allow the system to reconfigure and to continue its normal mode of operation.

- If a direct link from A to B has failed, this information must be broadcast to every site in the system, so that the various routing tables can be updated accordingly.
- If the system believes that a site has failed (because that site can no longer be reached), then all sites in the system must be notified, so that they will no longer attempt to use the services of the failed site. The failure of a site that serves as a central coordinator for some activity (such as deadlock detection) requires the election of a new coordinator. Note that, if the site has not failed (that is, if it is up but cannot be reached), then we may have the undesirable situation in which two sites serve as the coordinator. When the network is partitioned, the two coordinators (each for its own partition) may initiate conflicting actions. For example, if the coordinators are responsible for implementing mutual exclusion, we may have a situation in which two processes are executing simultaneously in their critical sections.

#### 19.5.1.3 Recovery from Failure

When a failed link or site is repaired, it must be integrated into the system gracefully and smoothly.

- Suppose that a link between A and B has failed. When it is repaired, both A and B must be notified. We can accomplish this notification by continuously repeating the heartbeat procedure described in Section 19.5.1.1.
- Suppose that site B has failed. When it recovers, it must notify all other sites that it is up again. Site B then may have to receive information from the other sites to update its local tables. For example, it may need routing-table information, a list of sites that are down, undelivered messages, a

transaction log of unexecuted transactions, and mail. If the site has not failed but simply cannot be reached, then it still needs this information.

### 19.5.2 Transparency

Making the multiple processors and storage devices in a distributed system **transparent** to the users has been a key challenge to many designers. Ideally, a distributed system should look to its users like a conventional, centralized system. The user interface of a transparent distributed system should not distinguish between local and remote resources. That is, users should be able to access remote resources as though these resources were local, and the distributed system should be responsible for locating the resources and for arranging for the appropriate interaction.

Another aspect of transparency is user mobility. It would be convenient to allow users to log into any machine in the system rather than forcing them to use a specific machine. A transparent distributed system facilitates user mobility by bringing over a user's environment (for example, home directory) to wherever he logs in. Protocols like LDAP provide an authentication system for local, remote, and mobile users. Once the authentication is complete, facilities like desktop virtualization allow users to see their desktop sessions at remote facilities.

### 19.5.3 Scalability

Still another issue is **scalability**—the capability of a system to adapt to increased service load. Systems have bounded resources and can become completely saturated under increased load. For example, with respect to a file system, saturation occurs either when a server's CPU runs at a high utilization rate or when disks' I/O requests overwhelm the I/O subsystem. Scalability is a relative property, but it can be measured accurately. A scalable system reacts more gracefully to increased load than does a nonscalable one. First, its performance degrades more moderately; and second, its resources reach a saturated state later. Even perfect design however cannot accommodate an ever-growing load. Adding new resources might solve the problem, but it might generate additional indirect load on other resources (for example, adding machines to a distributed system can clog the network and increase service loads). Even worse, expanding the system can call for expensive design modifications. A scalable system should have the potential to grow without these problems. In a distributed system, the ability to scale up gracefully is of special importance, since expanding a network by adding new machines or interconnecting two networks is commonplace. In short, a scalable design should withstand high service load, accommodate growth of the user community, and allow simple integration of added resources.

Scalability is related to fault tolerance, discussed earlier. A heavily loaded component can become paralyzed and behave like a faulty component. In addition, shifting the load from a faulty component to that component's backup can saturate the latter. Generally, having spare resources is essential for ensuring reliability as well as for handling peak loads gracefully. Thus, the multiple resources in a distributed system represent an inherent advantage, giving the system a greater potential for fault tolerance and scalability. However,

inappropriate design can obscure this potential. Fault-tolerance and scalability considerations call for a design demonstrating distribution of control and data.

Scalability can also be related to efficient storage schemes. For example, many cloud storage providers use **compression** or **deduplication** to cut down on the amount of storage used. *Compression* reduces the size of a file. For example, a zip archive file can be generated out of a file (or files) by executing a zip command, which runs a lossless compression algorithm over the data specified. (*Lossless compression* allows original data to be perfectly reconstructed from compressed data.) The result is a file archive that is smaller than the uncompressed file. To restore the file to its original state, a user runs some sort of unzip command over the zip archive file. *Deduplication* seeks to lower data storage requirements by removing redundant data. With this technology, only one instance of data is stored across an entire system (even across data owned by multiple users). Both compression and deduplication can be performed at the file level or the block level, and they can be used together. These techniques can be automatically built into a distributed system to compress information without users explicitly issuing commands, thereby saving storage space and possibly cutting down on network communication costs without adding user complexity.

## 19.6 Distributed File Systems

Although the World Wide Web is the predominant distributed system in use today, it is not the only one. Another important and popular use of distributed computing is the **distributed file system**, or **DFS**.

To explain the structure of a DFS, we need to define the terms *service*, *server*, and *client* in the DFS context. A **service** is a software entity running on one or more machines and providing a particular type of function to clients. A **server** is the service software running on a single machine. A **client** is a process that can invoke a service using a set of operations that form its **client interface**. Sometimes a lower-level interface is defined for the actual cross-machine interaction; it is the **intermachine interface**.

Using this terminology, we say that a file system provides file services to clients. A client interface for a file service is formed by a set of primitive file operations, such as create a file, delete a file, read from a file, and write to a file. The primary hardware component that a file server controls is a set of local secondary-storage devices (usually, hard disks or solid-state drives) on which files are stored and from which they are retrieved according to the clients' requests.

A DFS is a file system whose clients, servers, and storage devices are dispersed among the machines of a distributed system. Accordingly, service activity has to be carried out across the network. Instead of a single centralized data repository, the system frequently has multiple and independent storage devices. As you will see, the concrete configuration and implementation of a DFS may vary from system to system. In some configurations, servers run on dedicated machines. In others, a machine can be both a server and a client.

The distinctive features of a DFS are the multiplicity and autonomy of clients and servers in the system. Ideally, though, a DFS should appear to its clients to be a conventional, centralized file system. That is, the client interface

of a DFS should not distinguish between local and remote files. It is up to the DFS to locate the files and to arrange for the transport of the data. A *transparent* DFS—like the transparent distributed systems mentioned earlier—facilitates user mobility by bringing a user’s environment (for example, the user’s home directory) to wherever the user logs in.

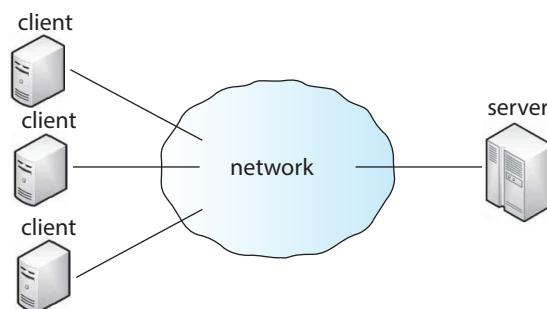
The most important performance measure of a DFS is the amount of time needed to satisfy service requests. In conventional systems, this time consists of storage-access time and a small amount of CPU-processing time. In a DFS, however, a remote access has the additional overhead associated with the distributed structure. This overhead includes the time to deliver the request to a server, as well as the time to get the response across the network back to the client. For each direction, in addition to the transfer of the information, there is the CPU overhead of running the communication protocol software. The performance of a DFS can be viewed as another dimension of the DFS’s transparency. That is, the performance of an ideal DFS would be comparable to that of a conventional file system.

The basic architecture of a DFS depends on its ultimate goals. Two widely used architectural models we discuss here are the **client–server model** and the **cluster-based model**. The main goal of a client–server architecture is to allow transparent file sharing among one or more clients as if the files were stored locally on the individual client machines. The distributed file systems NFS and OpenAFS are prime examples. NFS is the most common UNIX-based DFS. It has several versions, and here we refer to NFS Version 3 unless otherwise noted.

If many applications need to be run in parallel on large data sets with high availability and scalability, the cluster-based model is more appropriate than the client–server model. Two well-known examples are the Google file system and the open-source HDFS, which runs as part of the Hadoop framework.

### 19.6.1 The Client–Server DFS Model

Figure 19.12 illustrates a simple DFS **client–server model**. The server stores both files and metadata on attached storage. In some systems, more than one server can be used to store different files. Clients are connected to the server through a network and can request access to files in the DFS by contacting the server through a well-known protocol such as NFS Version 3. The server



**Figure 19.12** Client–server DFS model.

is responsible for carrying out authentication, checking the requested file permissions, and, if warranted, delivering the file to the requesting client. When a client makes changes to the file, the client must somehow deliver those changes to the server (which holds the master copy of the file). The client's and the server's versions of the file should be kept consistent in a way that minimizes network traffic and the server's workload to the extent possible.

The **network file system (NFS)** protocol was originally developed by Sun Microsystems as an open protocol, which encouraged early adoption across different architectures and systems. From the beginning, the focus of NFS was simple and fast crash recovery in the face of server failure. To implement this goal, the NFS server was designed to be stateless; it does not keep track of which client is accessing which file or of things such as open file descriptors and file pointers. This means that, whenever a client issues a file operation (say, to read a file), that operation has to be idempotent in the face of server crashes. **Idempotent** describes an operation that can be issued more than once yet return the same result. In the case of a read operation, the client keeps track of the state (such as the file pointer) and can simply reissue the operation if the server has crashed and come back online. You can read more about the NFS implementation in Section 15.8.

The **Andrew file system (OpenAFS)** was created at Carnegie Mellon University with a focus on scalability. Specifically, the researchers wanted to design a protocol that would allow the server to support as many clients as possible. This meant minimizing requests and traffic to the server. When a client requests a file, the file's contents are downloaded from the server and stored on the client's local storage. Updates to the file are sent to the server when the file is closed, and new versions of the file are sent to the client when the file is opened. In comparison, NFS is quite chatty and will send block read and write requests to the server as the file is being used by a client.

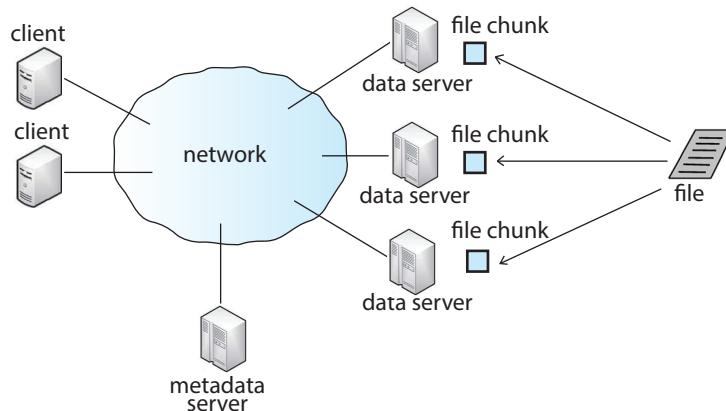
Both OpenAFS and NFS are meant to be used in addition to local file systems. In other words, you would not format a hard drive partition with the NFS file system. Instead, on the server, you would format the partition with a local file system of your choosing, such as ext4, and export the shared directories via the DFS. In the client, you would simply attach the exported directories to your file-system tree. In this way, the DFS can be separated from responsibility for the local file system and can concentrate on distributed tasks.

The DFS client–server model, by design, may suffer from a single point of failure if the server crashes. Computer clustering can help resolve this problem by using redundant components and clustering methods such that failures are detected and failing over to working components continues server operations. In addition, the server presents a bottleneck for all requests for both data and metadata, which results in problems of scalability and bandwidth.

### 19.6.2 The Cluster-Based DFS Model

As the amount of data, I/O workload, and processing expands, so does the need for a DFS to be fault-tolerant and scalable. Large bottlenecks cannot be tolerated, and system component failures must be expected. Cluster-based architecture was developed in part to meet these needs.

Figure 19.13 illustrates a sample cluster-based DFS model. This is the basic model presented by the **Google file system (GFS)** and the **Hadoop distributed**



**Figure 19.13** An example of a cluster-based DFS model

**file system (HDFS)**. One or more clients are connected via a network to a master metadata server and several data servers that house “chunks” (or portions) of files. The metadata server keeps a mapping of which data servers hold chunks of which files, as well as a traditional hierarchical mapping of directories and files. Each file chunk is stored on a data server and is replicated a certain number of times (for example, three times) to protect against component failure and for faster access to the data (servers containing the replicated chunks have fast access to those chunks).

To obtain access to a file, a client must first contact the metadata server. The metadata server then returns to the client the identities of the data servers that hold the requested file chunks. The client can then contact the closest data server (or servers) to receive the file information. Different chunks of the file can be read or written to in parallel if they are stored on different data servers, and the metadata server may need to be contacted only once in the entire process. This makes the metadata server less likely to be a performance bottleneck. The metadata server is also responsible for redistributing and balancing the file chunks among the data servers.

GFS was released in 2003 to support large distributed data-intensive applications. The design of GFS was influenced by four main observations:

- Hardware component failures are the norm rather than the exception and should be routinely expected.
- Files stored on such a system are very large.
- Most files are changed by appending new data to the end of the file rather than overwriting existing data.
- Redesigning the applications and file system API increases the system’s flexibility.

Consistent with the fourth observation, GFS exports its own API and requires applications to be programmed with this API.

Shortly after developing GFS, Google developed a modularized software layer called **MapReduce** to sit on top of GFS. MapReduce allows developers to carry out large-scale parallel computations more easily and utilizes the benefits of the lower-layer file system. Later, HDFS and the Hadoop framework (which includes stackable modules like MapReduce on top of HDFS) were created based on Google's work. Like GFS and MapReduce, Hadoop supports the processing of large data sets in distributed computing environments. As suggested earlier, the drive for such a framework occurred because traditional systems could not scale to the capacity and performance needed by "big data" projects (at least not at reasonable prices). Examples of big data projects include crawling and analyzing social media, customer data, and large amounts of scientific data points for trends.

## 19.7 DFS Naming and Transparency

**Naming** is a mapping between logical and physical objects. For instance, users deal with logical data objects represented by file names, whereas the system manipulates physical blocks of data stored on disk tracks. Usually, a user refers to a file by a textual name. The latter is mapped to a lower-level numerical identifier that in turn is mapped to disk blocks. This multilevel mapping provides users with an abstraction of a file that hides the details of how and where on the disk the file is stored.

In a transparent DFS, a new dimension is added to the abstraction: that of hiding where in the network the file is located. In a conventional file system, the range of the naming mapping is an address within a disk. In a DFS, this range is expanded to include the specific machine on whose disk the file is stored. Going one step further with the concept of treating files as abstractions leads to the possibility of **file replication**. Given a file name, the mapping returns a set of the locations of this file's replicas. In this abstraction, both the existence of multiple copies and their locations are hidden.

### 19.7.1 Naming Structures

We need to differentiate two related notions regarding name mappings in a DFS:

1. **Location transparency**. The name of a file does not reveal any hint of the file's physical storage location.
2. **Location independence**. The name of a file need not be changed when the file's physical storage location changes.

Both definitions relate to the level of naming discussed previously, since files have different names at different levels (that is, user-level textual names and system-level numerical identifiers). A location-independent naming scheme is a dynamic mapping, since it can map the same file name to different locations at two different times. Therefore, location independence is a stronger property than location transparency.

In practice, most of the current DFSs provide a static, location-transparent mapping for user-level names. Some support **file migration**—that is, changing the location of a file automatically, providing location independence. OpenAFS

supports location independence and file mobility, for example. HDFS includes file migration but does so without following POSIX standards, providing more flexibility in implementation and interface. HDFS keeps track of the location of data but hides this information from clients. This dynamic location transparency allows the underlying mechanism to self-tune. In another example, Amazon's S3 cloud storage facility provides blocks of storage on demand via APIs, placing the storage where it sees fit and moving the data as necessary to meet performance, reliability, and capacity requirements.

A few aspects can further differentiate location independence and static location transparency:

- Divorce of data from location, as exhibited by location independence, provides a better abstraction for files. A file name should denote the file's most significant attributes, which are its contents rather than its location. Location-independent files can be viewed as logical data containers that are not attached to a specific storage location. If only static location transparency is supported, the file name still denotes a specific, although hidden, set of physical disk blocks.
- Static location transparency provides users with a convenient way to share data. Users can share remote files by simply naming the files in a location-transparent manner, as though the files were local. Dropbox and other cloud-based storage solutions work this way. Location independence promotes sharing the storage space itself, as well as the data objects. When files can be mobilized, the overall, system-wide storage space looks like a single virtual resource. A possible benefit is the ability to balance the utilization of storage across the system.
- Location independence separates the naming hierarchy from the storage-devices hierarchy and from the intercomputer structure. By contrast, if static location transparency is used (although names are transparent), we can easily expose the correspondence between component units and machines. The machines are configured in a pattern similar to the naming structure. This configuration may restrict the architecture of the system unnecessarily and conflict with other considerations. A server in charge of a root directory is an example of a structure that is dictated by the naming hierarchy and contradicts decentralization guidelines.

Once the separation of name and location has been completed, clients can access files residing on remote server systems. In fact, these clients may be **diskless** and rely on servers to provide all files, including the operating-system kernel. Special protocols are needed for the boot sequence, however. Consider the problem of getting the kernel to a diskless workstation. The diskless workstation has no kernel, so it cannot use the DFS code to retrieve the kernel. Instead, a special boot protocol, stored in read-only memory (ROM) on the client, is invoked. It enables networking and retrieves only one special file (the kernel or boot code) from a fixed location. Once the kernel is copied over the network and loaded, its DFS makes all the other operating-system files available. The advantages of diskless clients are many, including lower cost (because the client machines require no disks) and greater convenience (when an operating-system upgrade occurs, only the server needs to be modified).

The disadvantages are the added complexity of the boot protocols and the performance loss resulting from the use of a network rather than a local disk.

### 19.7.2 Naming Schemes

There are three main approaches to naming schemes in a DFS. In the simplest approach, a file is identified by some combination of its host name and local name, which guarantees a unique system-wide name. In Ibis, for instance, a file is identified uniquely by the name *host:local-name*, where *local-name* is a UNIX-like path. The Internet URL system also uses this approach. This naming scheme is neither location transparent nor location independent. The DFS is structured as a collection of isolated component units, each of which is an entire conventional file system. Component units remain isolated, although means are provided to refer to remote files. We do not consider this scheme any further here.

The second approach was popularized by NFS. NFS provides a means to attach remote directories to local directories, thus giving the appearance of a coherent directory tree. Early NFS versions allowed only previously mounted remote directories to be accessed transparently. The advent of the **automount** feature allowed mounts to be done on demand based on a table of mount points and file-structure names. Components are integrated to support transparent sharing, but this integration is limited and is not uniform, because each machine may attach different remote directories to its tree. The resulting structure is versatile.

We can achieve total integration of the component file systems by using a third approach. Here, a single global name structure spans all the files in the system. OpenAFS provides a single global namespace for the files and directories it exports, allowing a similar user experience across different client machines. Ideally, the composed file-system structure is the same as the structure of a conventional file system. In practice, however, the many special files (for example, UNIX device files and machine-specific binary directories) make this goal difficult to attain.

To evaluate naming structures, we look at their administrative complexity. The most complex and most difficult-to-maintain structure is the NFS structure. Because any remote directory can be attached anywhere on the local directory tree, the resulting hierarchy can be highly unstructured. If a server becomes unavailable, some arbitrary set of directories on different machines becomes unavailable. In addition, a separate accreditation mechanism controls which machine is allowed to attach which directory to its tree. Thus, a user might be able to access a remote directory tree on one client but be denied access on another client.

### 19.7.3 Implementation Techniques

Implementation of transparent naming requires a provision for the mapping of a file name to the associated location. To keep this mapping manageable, we must aggregate sets of files into component units and provide the mapping on a component-unit basis rather than on a single-file basis. This aggregation serves administrative purposes as well. UNIX-like systems use the hierarchical directory tree to provide name-to-location mapping and to aggregate files recursively into directories.

To enhance the availability of the crucial mapping information, we can use replication, local caching, or both. As we noted, location independence means that the mapping changes over time. Hence, replicating the mapping makes a simple yet consistent update of this information impossible. To overcome this obstacle, we can introduce low-level, *location-independent file identifiers*. (OpenAFS uses this approach.) Textual file names are mapped to lower-level file identifiers that indicate to which component unit the file belongs. These identifiers are still location independent. They can be replicated and cached freely without being invalidated by migration of component units. The inevitable price is the need for a second level of mapping, which maps component units to locations and needs a simple yet consistent update mechanism. Implementing UNIX-like directory trees using these low-level, location-independent identifiers makes the whole hierarchy invariant under component-unit migration. The only aspect that does change is the component-unit location mapping.

A common way to implement low-level identifiers is to use structured names. These names are bit strings that usually have two parts. The first part identifies the component unit to which the file belongs; the second part identifies the particular file within the unit. Variants with more parts are possible. The invariant of structured names, however, is that individual parts of the name are unique at all times only within the context of the rest of the parts. We can obtain uniqueness at all times by taking care not to reuse a name that is still in use, by adding sufficiently more bits (this method is used in OpenAFS), or by using a timestamp as one part of the name (as was done in Apollo Domain). Another way to view this process is that we are taking a location-transparent system, such as Ibis, and adding another level of abstraction to produce a location-independent naming scheme.

## 19.8 Remote File Access

Next, let's consider a user who requests access to a remote file. The server storing the file has been located by the naming scheme, and now the actual data transfer must take place.

One way to achieve this transfer is through a **remote-service mechanism**, whereby requests for accesses are delivered to the server, the server machine performs the accesses, and their results are forwarded back to the user. One of the most common ways of implementing remote service is the RPC paradigm, which we discussed in Chapter 3. A direct analogy exists between disk-access methods in conventional file systems and the remote-service method in a DFS: using the remote-service method is analogous to performing a disk access for each access request.

To ensure reasonable performance of a remote-service mechanism, we can use a form of caching. In conventional file systems, the rationale for caching is to reduce disk I/O (thereby increasing performance), whereas in DFSs, the goal is to reduce both network traffic and disk I/O. In the following discussion, we describe the implementation of caching in a DFS and contrast it with the basic remote-service paradigm.

### 19.8.1 Basic Caching Scheme

The concept of caching is simple. If the data needed to satisfy the access request are not already cached, then a copy of the data is brought from the server to

the client system. Accesses are performed on the cached copy. The idea is to retain recently accessed disk blocks in the cache, so that repeated accesses to the same information can be handled locally, without additional network traffic. A replacement policy (for example, the least-recently-used algorithm) keeps the cache size bounded. No direct correspondence exists between accesses and traffic to the server. Files are still identified with one master copy residing at the server machine, but copies (or parts) of the file are scattered in different caches. When a cached copy is modified, the changes need to be reflected on the master copy to preserve the relevant consistency semantics. The problem of keeping the cached copies consistent with the master file is the **cache-consistency problem**, which we discuss in Section 19.8.4. DFS caching could just as easily be called **network virtual memory**. It acts similarly to demand-paged virtual memory, except that the backing store usually is a remote server rather than a local disk. NFS allows the swap space to be mounted remotely, so it actually can implement virtual memory over a network, though with a resulting performance penalty.

The granularity of the cached data in a DFS can vary from blocks of a file to an entire file. Usually, more data are cached than are needed to satisfy a single access, so that many accesses can be served by the cached data. This procedure is much like disk read-ahead (Section 14.6.2). OpenAFS caches files in large chunks (64 KB). The other systems discussed here support caching of individual blocks driven by client demand. Increasing the caching unit increases the hit ratio, but it also increases the miss penalty, because each miss requires more data to be transferred. It increases the potential for consistency problems as well. Selecting the unit of caching involves considering parameters such as the network transfer unit and the RPC protocol service unit (if an RPC protocol is used). The network transfer unit (for Ethernet, a packet) is about 1.5 KB, so larger units of cached data need to be disassembled for delivery and reassembled on reception.

Block size and total cache size are obviously of importance for block-caching schemes. In UNIX-like systems, common block sizes are 4 KB and 8 KB. For large caches (over 1 MB), large block sizes (over 8 KB) are beneficial. For smaller caches, large block sizes are less beneficial because they result in fewer blocks in the cache and a lower hit ratio.

### 19.8.2 Cache Location

Where should the cached data be stored—on disk or in main memory? Disk caches have one clear advantage over main-memory caches: they are reliable. Modifications to cached data are lost in a crash if the cache is kept in volatile memory. Moreover, if the cached data are kept on disk, they are still there during recovery, and there is no need to fetch them again. Main-memory caches have several advantages of their own, however:

- Main-memory caches permit workstations to be diskless.
- Data can be accessed more quickly from a cache in main memory than from one on a disk.
- Technology is moving toward larger and less expensive memory. The resulting performance speedup is predicted to outweigh the advantages of disk caches.

- The server caches (used to speed up disk I/O) will be in main memory regardless of where user caches are located; if we use main-memory caches on the user machine, too, we can build a single caching mechanism for use by both servers and users.

Many remote-access implementations can be thought of as hybrids of caching and remote service. In NFS, for instance, the implementation is based on remote service but is augmented with client- and server-side memory caching for performance. Thus, to evaluate the two methods, we must evaluate the degree to which either method is emphasized. The NFS protocol and most implementations do not provide disk caching (but OpenAFS does).

### 19.8.3 Cache-Update Policy

The policy used to write modified data blocks back to the server's master copy has a critical effect on the system's performance and reliability. The simplest policy is to write data through to disk as soon as they are placed in any cache. The advantage of a **write-through policy** is reliability: little information is lost when a client system crashes. However, this policy requires each write access to wait until the information is sent to the server, so it causes poor write performance. Caching with write-through is equivalent to using remote service for write accesses and exploiting caching only for read accesses.

An alternative is the **delayed-write policy**, also known as **write-back caching**, where we delay updates to the master copy. Modifications are written to the cache and then are written through to the server at a later time. This policy has two advantages over write-through. First, because writes are made to the cache, write accesses complete much more quickly. Second, data may be overwritten before they are written back, in which case only the last update needs to be written at all. Unfortunately, delayed-write schemes introduce reliability problems, since unwritten data are lost whenever a user machine crashes.

Variations of the delayed-write policy differ in when modified data blocks are flushed to the server. One alternative is to flush a block when it is about to be ejected from the client's cache. This option can result in good performance, but some blocks can reside in the client's cache a long time before they are written back to the server. A compromise between this alternative and the write-through policy is to scan the cache at regular intervals and to flush blocks that have been modified since the most recent scan, just as UNIX scans its local cache. NFS uses the policy for file data, but once a write is issued to the server during a cache flush, the write must reach the server's disk before it is considered complete. NFS treats metadata (directory data and file-attribute data) differently. Any metadata changes are issued synchronously to the server. Thus, file-structure loss and directory-structure corruption are avoided when a client or the server crashes.

Yet another variation on delayed write is to write data back to the server when the file is closed. This **write-on-close policy** is used in OpenAFS. In the case of files that are open for short periods or are modified rarely, this policy does not significantly reduce network traffic. In addition, the write-on-close policy requires the closing process to delay while the file is written through,

which reduces the performance advantages of delayed writes. For files that are open for long periods and are modified frequently, however, the performance advantages of this policy over delayed write with more frequent flushing are apparent.

#### 19.8.4 Consistency

A client machine is sometimes faced with the problem of deciding whether a locally cached copy of data is consistent with the master copy (and hence can be used). If the client machine determines that its cached data are out of date, it must cache an up-to-date copy of the data before allowing further accesses. There are two approaches to verifying the validity of cached data:

1. **Client-initiated approach.** The client initiates a validity check in which it contacts the server and checks whether the local data are consistent with the master copy. The frequency of the validity checking is the crux of this approach and determines the resulting consistency semantics. It can range from a check before every access to a check only on first access to a file (on file open, basically). Every access coupled with a validity check is delayed, compared with an access served immediately by the cache. Alternatively, checks can be initiated at fixed time intervals. Depending on its frequency, the validity check can load both the network and the server.
2. **Server-initiated approach.** The server records, for each client, the files (or parts of files) that it caches. When the server detects a potential inconsistency, it must react. A potential for inconsistency occurs when two different clients in conflicting modes cache a file. If UNIX semantics (Section 15.7) is implemented, we can resolve the potential inconsistency by having the server play an active role. The server must be notified whenever a file is opened, and the intended mode (read or write) must be indicated for every open. The server can then act when it detects that a file has been opened simultaneously in conflicting modes by disabling caching for that particular file. Actually, disabling caching results in switching to a remote-service mode of operation.

In a cluster-based DFS, the cache-consistency issue is made more complicated by the presence of a metadata server and several replicated file data chunks across several data servers. Using our earlier examples of HDFS and GFS, we can compare some differences. HDFS allows append-only write operations (no random writes) and a single file writer, while GFS does allow random writes with concurrent writers. This greatly complicates write consistency guarantees for GFS while simplifying them for HDFS.

## 19.9 Final Thoughts on Distributed File Systems

The line between DFS client–server and cluster-based architectures is blurring. The NFS Version 4.1 specification includes a protocol for a parallel version of NFS called pNFS, but as of this writing, adoption is slow.

GFS, HDFS, and other large-scale DFSs export a non-POSIX API, so they cannot transparently map directories to regular user machines as NFS and OpenAFS do. Rather, for systems to access these DFSs, they need client code installed. However, other software layers are rapidly being developed to allow NFS to be mounted on top of such DFSs. This is attractive, as it would take advantage of the scalability and other advantages of cluster-based DFSs while still allowing native operating-system utilities and users to access files directly on the DFS.

As of this writing, the open-source HDFS NFS Gateway supports NFS Version 3 and works as a proxy between HDFS and the NFS server software. Since HDFS currently does not support random writes, the HDFS NFS Gateway also does not support this capability. That means a file must be deleted and recreated from scratch even if only one byte is changed. Commercial organizations and researchers are addressing this problem and building stackable frameworks that allow stacking of a DFS, parallel computing modules (such as MapReduce), distributed databases, and exported file volumes through NFS.

One other type of file system, less complex than a cluster-based DFS but more complex than a client-server DFS, is a **clustered file system (CFS)** or **parallel file system (PFS)**. A CFS typically runs over a LAN. These systems are important and widely used and thus deserve mention here, though we do not cover them in detail. Common CFSs include **Lustre** and **GPFS**, although there are many others. A CFS essentially treats  $N$  systems storing data and  $Y$  systems accessing that data as a single client-server instance. Whereas NFS, for example, has per-server naming, and two separate NFS servers generally provide two different naming schemes, a CFS knits various storage contents on various storage devices on various servers into a uniform, transparent name space. GPFS has its own file-system structure, but Lustre uses existing file systems such as ZFS for file storage and management. To learn more, see <http://lustre.org>.

Distributed file systems are in common use today, providing file sharing within LANs, within cluster environments, and across WANs. The complexity of implementing such a system should not be underestimated, especially considering that the DFS must be operating-system independent for widespread adoption and must provide availability and good performance in the presence of long distances, commodity hardware failures, sometimes frail networking, and ever-increasing users and workloads.

## 19.10 Summary

- A distributed system is a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory, and the processors communicate with one another through various communication lines, such as high-speed buses and the Internet. The processors in a distributed system vary in size and function.
- A distributed system provides the user with access to all system resources. Access to a shared resource can be provided by data migration, computation migration, or process migration. The access can be specified by the user or implicitly supplied by the operating system and applications.

- Protocol stacks, as specified by network layering models, add information to a message to ensure that it reaches its destination.
- A naming system (such as DNS) must be used to translate from a host name to a network address, and another protocol (such as ARP) may be needed to translate the network number to a network device address (an Ethernet address, for instance).
- If systems are located on separate networks, routers are needed to pass packets from source network to destination network.
- The transport protocols UDP and TCP direct packets to waiting processes through the use of unique system-wide port numbers. In addition, the TCP protocol allows the flow of packets to become a reliable, connection-oriented byte stream.
- There are many challenges to overcome for a distributed system to work correctly. Issues include naming of nodes and processes in the system, fault tolerance, error recovery, and scalability. Scalability issues include handling increased load, being fault tolerant, and using efficient storage schemes, including the possibility of compression and/or deduplication.
- A DFS is a file-service system whose clients, servers, and storage devices are dispersed among the sites of a distributed system. Accordingly, service activity has to be carried out across the network; instead of a single centralized data repository, there are multiple independent storage devices.
- There are two main types of DFS models: the client–server model and the cluster-based model. The client-server model allows transparent file sharing among one or more clients. The cluster-based model distributes the files among one or more data servers and is built for large-scale parallel data processing.
- Ideally, a DFS should look to its clients like a conventional, centralized file system (although it may not conform exactly to traditional file-system interfaces such as POSIX). The multiplicity and dispersion of its servers and storage devices should be transparent. A transparent DFS facilitates client mobility by bringing the client's environment to the site where the client logs in.
- There are several approaches to naming schemes in a DFS. In the simplest approach, files are named by some combination of their host name and local name, which guarantees a unique system-wide name. Another approach, popularized by NFS, provides a means to attach remote directories to local directories, thus giving the appearance of a coherent directory tree.
- Requests to access a remote file are usually handled by two complementary methods. With remote service, requests for accesses are delivered to the server. The server machine performs the accesses, and the results are forwarded back to the client. With caching, if the data needed to satisfy the access request are not already cached, then a copy of the data is brought from the server to the client. Accesses are performed on the cached copy. The problem of keeping the cached copies consistent with the master file is the cache-consistency problem.

## Practice Exercises

- 19.1 Why would it be a bad idea for routers to pass broadcast packets between networks? What would be the advantages of doing so?
- 19.2 Discuss the advantages and disadvantages of caching name translations for computers located in remote domains.
- 19.3 What are two formidable problems that designers must solve to implement a network system that has the quality of transparency?
- 19.4 To build a robust distributed system, you must know what kinds of failures can occur.
  - a. List three possible types of failure in a distributed system.
  - b. Specify which of the entries in your list also are applicable to a centralized system.
- 19.5 Is it always crucial to know that the message you have sent has arrived at its destination safely? If your answer is “yes,” explain why. If your answer is “no,” give appropriate examples.
- 19.6 A distributed system has two sites, A and B. Consider whether site A can distinguish among the following:
  - a. B goes down.
  - b. The link between A and B goes down.
  - c. B is extremely overloaded, and its response time is 100 times longer than normal.

What implications does your answer have for recovery in distributed systems?

## Further Reading

[Peterson and Davie (2012)] and [Kurose and Ross (2017)] provide general overviews of computer networks. The Internet and its protocols are described in [Comer (2000)]. Coverage of TCP/IP can be found in [Fall and Stevens (2011)] and [Stevens (1995)]. UNIX network programming is described thoroughly in [Steven et al. (2003)].

Ethernet and WiFi standards and speeds are evolving quickly. Current IEEE 802.3 Ethernet standards can be found at <http://standards.ieee.org/about/get/802/802.3.html>. Current IEEE 802.11 Wireless LAN standards can be found at <http://standards.ieee.org/about/get/802/802.11.html>.

Sun’s network file system (NFS) is described by [Callaghan (2000)]. Information about OpenAFS is available from <http://www.openafs.org>.

Information on the Google file system can be found in [Ghemawat et al. (2003)]. The Google MapReduce method is described in <http://research.google.com/archive/mapreduce.html>. The Hadoop distributed file system is discussed in [K. Shvachko and Chansler (2010)], and the Hadoop framework is discussed in <http://hadoop.apache.org/>.

To learn more about Lustre, see <http://lustre.org>.

## Bibliography

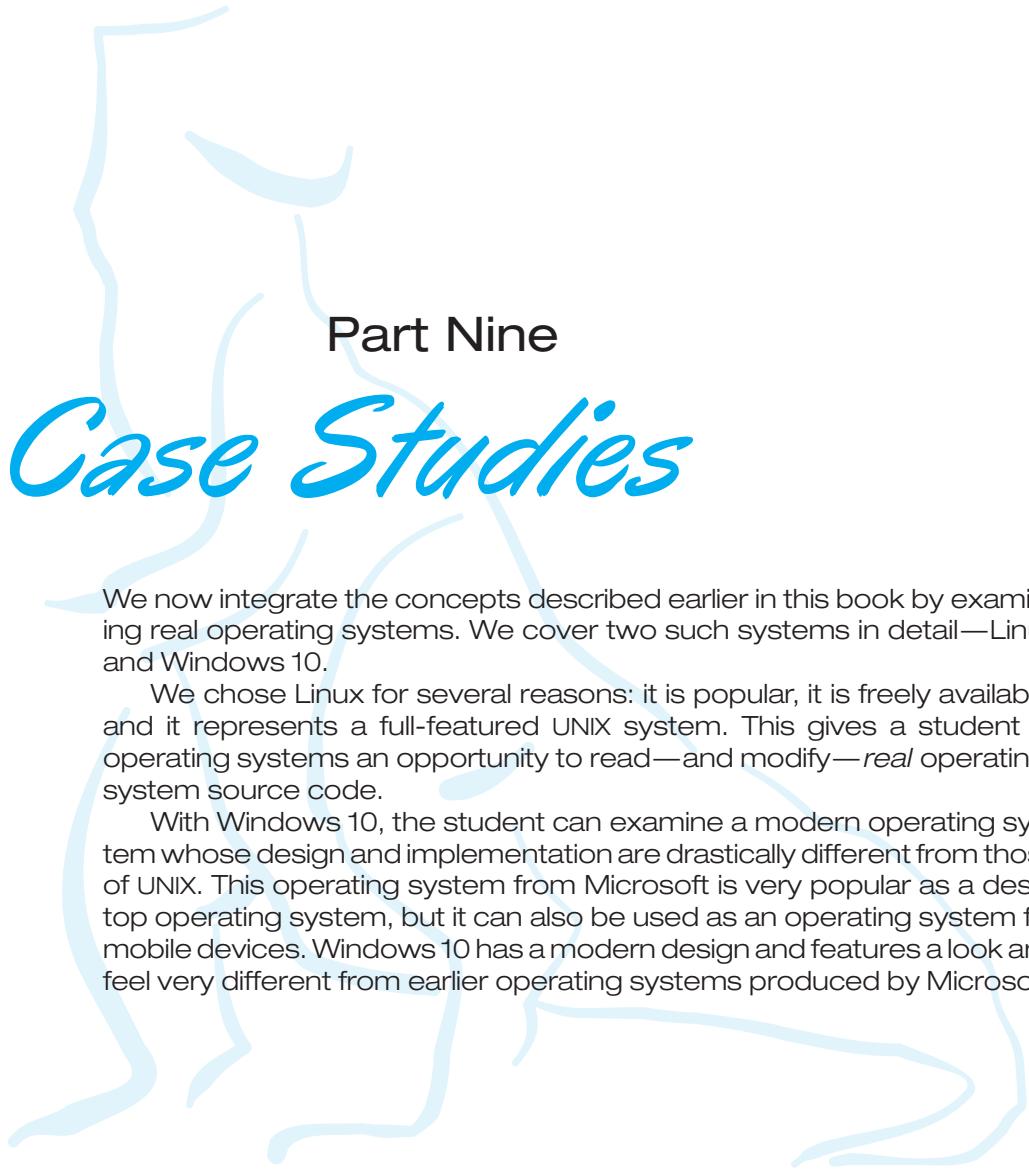
- [Callaghan (2000)] B. Callaghan, *NFS Illustrated*, Addison-Wesley (2000).
- [Comer (2000)] D. Comer, *Internetworking with TCP/IP, Volume I*, Fourth Edition, Prentice Hall (2000).
- [Fall and Stevens (2011)] K. Fall and R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*, Second Edition, John Wiley and Sons (2011).
- [Ghemawat et al. (2003)] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The Google File System”, *Proceedings of the ACM Symposium on Operating Systems Principles* (2003).
- [K. Shvachko and Chansler (2010)] S. R. K. Shvachko, H. Kuang and R. Chansler, “The Hadoop Distributed File System” (2010).
- [Kurose and Ross (2017)] J. Kurose and K. Ross, *Computer Networking—A Top-Down Approach*, Seventh Edition, Addison-Wesley (2017).
- [Peterson and Davie (2012)] L. L. Peterson and B. S. Davie, *Computer Networks: A Systems Approach*, Fifth Edition, Morgan Kaufmann (2012).
- [Steven et al. (2003)] R. Steven, B. Fenner, and A. Rudoff, *Unix Network Programming, Volume 1: The Sockets Networking API*, Third Edition, John Wiley and Sons (2003).
- [Stevens (1995)] R. Stevens, *TCP/IP Illustrated, Volume 2: The Implementation*, Addison-Wesley (1995).

## Chapter 19 Exercises

- 19.7 What is the difference between computation migration and process migration? Which is easier to implement, and why?
- 19.8 Even though the OSI model of networking specifies seven layers of functionality, most computer systems use fewer layers to implement a network. Why do they use fewer layers? What problems could the use of fewer layers cause?
- 19.9 Explain why doubling the speed of the systems on an Ethernet segment may result in decreased network performance when the UDP transport protocol is used. What changes could help solve this problem?
- 19.10 What are the advantages of using dedicated hardware devices for routers? What are the disadvantages of using these devices compared with using general-purpose computers?
- 19.11 In what ways is using a name server better than using static host tables? What problems or complications are associated with name servers? What methods could you use to decrease the amount of traffic name servers generate to satisfy translation requests?
- 19.12 Name servers are organized in a hierarchical manner. What is the purpose of using a hierarchical organization?
- 19.13 The lower layers of the OSI network model provide datagram service, with no delivery guarantees for messages. A transport-layer protocol such as TCP is used to provide reliability. Discuss the advantages and disadvantages of supporting reliable message delivery at the lowest possible layer.
- 19.14 Run the program shown in Figure 19.4 and determine the IP addresses of the following host names:
  - www.wiley.com
  - www.cs.yale.edu
  - www.apple.com
  - www.westminstercollege.edu
  - www.ietf.org
- 19.15 A DNS name can map to multiple servers, such as www.google.com. However, if we run the program shown in Figure 19.4, we get only one IP address. Modify the program to display all the server IP addresses instead of just one.
- 19.16 The original HTTP protocol used TCP/IP as the underlying network protocol. For each page, graphic, or applet, a separate TCP session was constructed, used, and torn down. Because of the overhead of building and destroying TCP/IP connections, performance problems resulted from this implementation method. Would using UDP rather than TCP be a good alternative? What other changes could you make to improve HTTP performance?

- 19.17 What are the advantages and the disadvantages of making the computer network transparent to the user?
- 19.18 What are the benefits of a DFS compared with a file system in a centralized system?
- 19.19 For each of the following workloads, identify whether a cluster-based or a client–server DFS model would handle the workload best. Explain your answers.
  - Hosting student files in a university lab.
  - Processing data sent by the Hubble telescope.
  - Sharing data with multiple devices from a home server.
- 19.20 Discuss whether OpenAFS and NFS provide the following: (a) location transparency and (b) location independence.
- 19.21 Under what circumstances would a client prefer a location-transparent DFS? Under what circumstances would she prefer a location-independent DFS? Discuss the reasons for these preferences.
- 19.22 What aspects of a distributed system would you select for a system running on a totally reliable network?
- 19.23 Compare and contrast the techniques of caching disk blocks locally, on a client system, and remotely, on a server.
- 19.24 Which scheme would likely result in a greater space saving on a multiuser DFS: file-level deduplication or block-level deduplication? Explain your answer.
- 19.25 What types of extra metadata information would need to be stored in a DFS that uses deduplication?





## Part Nine

# Case Studies

We now integrate the concepts described earlier in this book by examining real operating systems. We cover two such systems in detail—Linux and Windows 10.

We chose Linux for several reasons: it is popular, it is freely available, and it represents a full-featured UNIX system. This gives a student of operating systems an opportunity to read—and modify—*real* operating-system source code.

With Windows 10, the student can examine a modern operating system whose design and implementation are drastically different from those of UNIX. This operating system from Microsoft is very popular as a desktop operating system, but it can also be used as an operating system for mobile devices. Windows 10 has a modern design and features a look and feel very different from earlier operating systems produced by Microsoft.





# The Linux System

Updated by Robert Love

This chapter presents an in-depth examination of the Linux operating system. By examining a complete, real system, we can see how the concepts we have discussed relate both to one another and to practice.

Linux is a variant of UNIX that has gained popularity over the last several decades, powering devices as small as mobile phones and as large as room-filling supercomputers. In this chapter, we look at the history and development of Linux and cover the user and programmer interfaces that Linux presents—interfaces that owe a great deal to the UNIX tradition. We also discuss the design and implementation of these interfaces. Linux is a rapidly evolving operating system. This chapter describes developments through the Linux 4.12 kernel, which was released in 2017.

## CHAPTER OBJECTIVES

- Explore the history of the UNIX operating system from which Linux is derived and the principles upon which Linux's design is based.
- Examine the Linux process and thread models and illustrate how Linux schedules threads and provides interprocess communication.
- Look at memory management in Linux.
- Explore how Linux implements file systems and manages I/O devices.

### 20.1 Linux History

Linux looks and feels much like any other UNIX system; indeed, UNIX compatibility has been a major design goal of the Linux project. However, Linux is much younger than most UNIX systems. Its development began in 1991, when a Finnish university student, Linus Torvalds, began creating a small but self-contained kernel for the 80386 processor, the first true 32-bit processor in Intel's range of PC-compatible CPUs.

Early in its development, the Linux source code was made available free—both at no cost and with minimal distributional restrictions—on the Internet. As a result, Linux’s history has been one of collaboration by many developers from all around the world, corresponding almost exclusively over the Internet. From an initial kernel that partially implemented a small subset of the UNIX system services, the Linux system has grown to include all of the functionality expected of a modern UNIX system.

In its early days, Linux development revolved largely around the central operating-system kernel—the core, privileged executive that manages all system resources and interacts directly with the computer hardware. We need much more than this kernel, of course, to produce a full operating system. We thus need to make a distinction between the Linux kernel and a complete Linux system. The **Linux kernel** is an original piece of software developed from scratch by the Linux community. The **Linux system**, as we know it today, includes a multitude of components, some written from scratch, others borrowed from other development projects, and still others created in collaboration with other teams.

The basic Linux system is a standard environment for applications and user programming, but it does not enforce any standard means of managing the available functionality as a whole. As Linux has matured, a need has arisen for another layer of functionality on top of the Linux system. This need has been met by various Linux distributions. A **Linux distribution** includes all the standard components of the Linux system, plus a set of administrative tools to simplify the initial installation and subsequent upgrading of Linux and to manage installation and removal of other packages on the system. A modern distribution also typically includes tools for management of file systems, creation and management of user accounts, administration of networks, web browsers, word processors, and so on.

### 20.1.1 The Linux Kernel

The first Linux kernel released to the public was version 0.01, dated May 14, 1991. It had no networking, ran only on 80386-compatible Intel processors and PC hardware, and had extremely limited device-driver support. The virtual memory subsystem was also fairly basic and included no support for memory-mapped files; however, even this early incarnation supported shared pages with copy-on-write and protected address spaces. The only file system supported was the Minix file system, as the first Linux kernels were cross-developed on a Minix platform.

The next milestone, Linux 1.0, was released on March 14, 1994. This release culminated three years of rapid development of the Linux kernel. Perhaps the single biggest new feature was networking: 1.0 included support for UNIX’s standard TCP/IP networking protocols, as well as a BSD-compatible socket interface for networking programming. Device-driver support was added for running IP over Ethernet or (via the PPP or SLIP protocols) over serial lines or modems.

The 1.0 kernel also included a new, much enhanced file system without the limitations of the original Minix file system, and it supported a range of SCSI controllers for high-performance disk access. The developers extended the vir-

tual memory subsystem to support paging to swap files and memory mapping of arbitrary files (but only read-only memory mapping was implemented in 1.0).

A range of extra hardware support was included in this release. Although still restricted to the Intel PC platform, hardware support had grown to include floppy-disk and CD-ROM devices, as well as sound cards, a range of mice, and international keyboards. Floating-point emulation was provided in the kernel for 80386 users who had no 80387 math coprocessor. System V UNIX-style **interprocess communication (IPC)**, including shared memory, semaphores, and message queues, was implemented.

At this point, development started on the 1.1 kernel stream, but numerous bug-fix patches were released subsequently for 1.0. A pattern was adopted as the standard numbering convention for Linux kernels. Kernels with an odd minor-version number, such as 1.1 or 2.5, are **development kernels**; even-numbered minor-version numbers are stable **production kernels**. Updates for the stable kernels are intended only as remedial versions, whereas the development kernels may include newer and relatively untested functionality. As we will see, this pattern remained in effect until version 3.

In March 1995, the 1.2 kernel was released. This release did not offer nearly the same improvement in functionality as the 1.0 release, but it did support a much wider variety of hardware, including the new PCI hardware bus architecture. Developers added another PC-specific feature—support for the 80386 CPU's virtual 8086 mode—to allow emulation of the DOS operating system for PC computers. They also updated the IP implementation with support for accounting and firewalling. Simple support for dynamically loadable and unloadable kernel modules was supplied as well.

The 1.2 kernel was the final PC-only Linux kernel. The source distribution for Linux 1.2 included partially implemented support for SPARC, Alpha, and MIPS CPUs, but full integration of these other architectures did not begin until after the stable 1.2 kernel was released.

The Linux 1.2 release concentrated on wider hardware support and more complete implementations of existing functionality. Much new functionality was under development at the time, but integration of the new code into the main kernel source code was deferred until after the stable 1.2 kernel was released. As a result, the 1.3 development stream saw a great deal of new functionality added to the kernel.

This work was released in June 1996 as Linux version 2.0. This release was given a major version-number increment because of two major new capabilities: support for multiple architectures, including a 64-bit native Alpha port, and symmetric multiprocessing (SMP) support. Additionally, the memory-management code was substantially improved to provide a unified cache for file-system data independent of the caching of block devices. As a result of this change, the kernel offered greatly increased file-system and virtual-memory performance. For the first time, file-system caching was extended to networked file systems, and writable memory-mapped regions were also supported. Other major improvements included the addition of internal kernel threads, a mechanism exposing dependencies between loadable modules, support for the automatic loading of modules on demand, file-system quotas, and POSIX-compatible real-time process-scheduling classes.

Improvements continued with the release of Linux 2.2 in 1999. A port to UltraSPARC systems was added. Networking was enhanced with more flexible firewalling, improved routing and traffic management, and support for TCP large window and selective acknowledgement. Acorn, Apple, and NT disks could now be read, and NFS was enhanced with a new kernel-mode NFS daemon. Signal handling, interrupts, and some I/O were locked at a finer level than before to improve symmetric multiprocessor (SMP) performance.

Advances in the 2.4 and 2.6 releases of the kernel included increased support for SMP systems, journaling file systems, and enhancements to the memory-management and block I/O systems. The thread scheduler was modified in version 2.6, providing an efficient  $O(1)$  scheduling algorithm. In addition, the 2.6 kernel was preemptive, allowing threads to be preempted even while running in kernel mode.

Linux kernel version 3.0 was released in July 2011. The major version bump from 2 to 3 occurred to commemorate the twentieth anniversary of Linux. New features include improved virtualization support, a new page write-back facility, improvements to the memory-management system, and yet another new thread scheduler—the Completely Fair Scheduler (CFS).

Linux kernel version 4.0 was released in April 2015. This time the major version bump was entirely arbitrary; Linux kernel developers simply grew tired of ever-larger minor versions. Today Linux kernel versions do not signify anything other than release ordering. The 4.0 kernel series provided support for new architectures, improved mobile functionality, and many iterative improvements. We focus on this newest kernel in the remainder of this chapter.

### 20.1.2 The Linux System

As we noted earlier, the Linux kernel forms the core of the Linux project, but other components make up a complete Linux operating system. Whereas the Linux kernel is composed entirely of code written from scratch specifically for the Linux project, much of the supporting software that makes up the Linux system is not exclusive to Linux but is common to a number of UNIX-like operating systems. In particular, Linux uses many tools developed as part of Berkeley’s BSD operating system, MIT’s X Window System, and the Free Software Foundation’s GNU project.

This sharing of tools has worked in both directions. The main system libraries of Linux were originated by the GNU project, but the Linux community greatly improved the libraries by addressing omissions, inefficiencies, and bugs. Other components, such as the [GNU C compiler \(gcc\)](#), were already of sufficiently high quality to be used directly in Linux. The network administration tools under Linux were derived from code first developed for 4.3 BSD, but more recent BSD derivatives, such as FreeBSD, have borrowed code from Linux in return. Examples of this sharing include the Intel floating-point-emulation math library and the PC sound-hardware device drivers.

The Linux system as a whole is maintained by a loose network of developers collaborating over the Internet, with small groups or individuals having responsibility for maintaining the integrity of specific components. A small number of public Internet file-transfer-protocol (FTP) archive sites act as de facto standard repositories for these components. The [File System Hierarchy](#)

**Standard** document is also maintained by the Linux community as a means of ensuring compatibility across the various system components. This standard specifies the overall layout of a standard Linux file system; it determines under which directory names configuration files, libraries, system binaries, and run-time data files should be stored.

### 20.1.3 Linux Distributions

In theory, anybody can install a Linux system by fetching the latest revisions of the necessary system components from the *ftp* sites and compiling them. In Linux's early days, this is precisely what a Linux user had to do. As Linux has matured, however, various individuals and groups have attempted to make this job less painful by providing standard, precompiled sets of packages for easy installation.

These collections, or distributions, include much more than just the basic Linux system. They typically include extra system-installation and management utilities, as well as precompiled and ready-to-install packages of many of the common UNIX tools, such as news servers, web browsers, text-processing and editing tools, and even games.

The first distributions managed these packages by simply providing a means of unpacking all the files into the appropriate places. One of the important contributions of modern distributions, however, is advanced package management. Today's Linux distributions include a package-tracking database that allows packages to be installed, upgraded, or removed painlessly.

The SLS distribution, dating back to the early days of Linux, was the first collection of Linux packages that was recognizable as a complete distribution. Although it could be installed as a single entity, SLS lacked the package-management tools now expected of Linux distributions. The **Slackware** distribution represented a great improvement in overall quality, even though it also had poor package management. In fact, it is still one of the most widely installed distributions in the Linux community.

Since Slackware's release, many commercial and noncommercial Linux distributions have become available. **Red Hat** and **Debian** are particularly popular distributions; the first comes from a commercial Linux support company and the second from the free-software Linux community. Other commercially supported versions of Linux include distributions from **Canonical** and **SuSE**, and many others. There are too many Linux distributions in circulation for us to list all of them here. The variety of distributions does not prevent Linux distributions from being compatible, however. The RPM package file format is used, or at least understood, by the majority of distributions, and commercial applications distributed in this format can be installed and run on any distribution that can accept RPM files.

### 20.1.4 Linux Licensing

The Linux kernel is distributed under version 2.0 of the GNU General Public License (GPL), the terms of which are set out by the Free Software Foundation. Linux is not public-domain software. **Public domain** implies that the authors have waived copyright rights in the software, but copyright rights in Linux code are still held by the code's various authors. Linux is *free* software, how-

ever, in the sense that people can copy it, modify it, use it in any manner they want, and give away (or sell) their own copies.

The main implication of Linux's licensing terms is that nobody using Linux, or creating a derivative of Linux (a legitimate exercise), can distribute the derivative without including the source code. Software released under the GPL cannot be redistributed as a binary-only product. If you release software that includes any components covered by the GPL, then, under the GPL, you must make source code available alongside any binary distributions. (This restriction does not prohibit making—or even selling—binary software distributions, as long as anybody who receives binaries is also given the opportunity to get the originating source code for a reasonable distribution charge.)

## 20.2 Design Principles

In its overall design, Linux resembles other traditional, nonmicrokernel UNIX implementations. It is a multiuser, preemptively multitasking system with a full set of UNIX-compatible tools. Linux's file system adheres to traditional UNIX semantics, and the standard UNIX networking model is fully implemented. The internal details of Linux's design have been influenced heavily by the history of this operating system's development.

Although Linux runs on a wide variety of platforms, it was originally developed exclusively on PC architecture. A great deal of that early development was carried out by individual enthusiasts rather than by well-funded development or research facilities, so from the start Linux attempted to squeeze as much functionality as possible from limited resources. Today, Linux can run happily on a multiprocessor machine with hundreds of gigabytes of main memory and many terabytes of disk space, but it is still capable of operating usefully in under 16-MB of RAM.

As PCs became more powerful and as memory and hard disks became cheaper, the original, minimalist Linux kernels grew to implement more UNIX functionality. Speed and efficiency are still important design goals, but much recent and current work on Linux has concentrated on a third major design goal: standardization. One of the prices paid for the diversity of UNIX implementations currently available is that source code written for one may not necessarily compile or run correctly on another. Even when the same system calls are present on two different UNIX systems, they do not necessarily behave in exactly the same way. The POSIX standards comprise a set of specifications for different aspects of operating-system behavior. There are POSIX documents for common operating-system functionality and for extensions such as process threads and real-time operations. Linux is designed to comply with the relevant POSIX documents, and at least two Linux distributions have achieved official POSIX certification.

Because it gives standard interfaces to both the programmer and the user, Linux presents few surprises to anybody familiar with UNIX. We do not detail these interfaces here. The sections on the programmer interface (Section C.3) and user interface (Section C.4) of BSD apply equally well to Linux. By default, however, the Linux programming interface adheres to SVR4 UNIX semantics, rather than to BSD behavior. A separate set of libraries is available to implement BSD semantics in places where the two behaviors differ significantly.

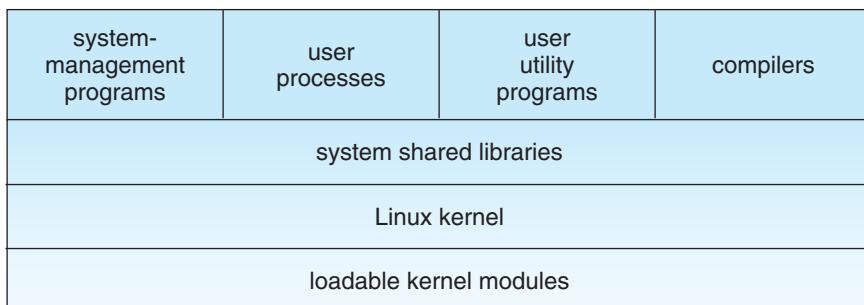
Many other standards exist in the UNIX world, but full certification of Linux with respect to these standards is sometimes slowed because certification is often available only for a fee, and the expense involved in certifying an operating system's compliance with most standards is substantial. However, supporting a wide base of applications is important for any operating system, so implementation of standards is a major goal for Linux development, even without formal certification. In addition to the basic POSIX standard, Linux currently supports the POSIX threading extensions—Pthreads—and a subset of the POSIX extensions for real-time process control.

### 20.2.1 Components of a Linux System

The Linux system is composed of three main bodies of code, in line with most traditional UNIX implementations:

1. **Kernel.** The kernel is responsible for maintaining all the important abstractions of the operating system, including such things as virtual memory and processes.
2. **System libraries.** The system libraries define a standard set of functions through which applications can interact with the kernel. These functions implement much of the operating-system functionality that does not need the full privileges of kernel code. The most important system library is the [C library](#), known as `libc`. In addition to providing the standard C library, `libc` implements the user mode side of the Linux system call interface, as well as other critical system-level interfaces.
3. **System utilities.** The system utilities are programs that perform individual, specialized management tasks. Some system utilities are invoked just once to initialize and configure some aspect of the system. Others—known as [daemons](#) in UNIX terminology—run permanently, handling such tasks as responding to incoming network connections, accepting logon requests from terminals, and updating log files.

Figure 20.1 illustrates the various components that make up a full Linux system. The most important distinction here is between the kernel and everything else. All the kernel code executes in the processor's privileged mode



**Figure 20.1** Components of the Linux system.

with full access to all the physical resources of the computer. Linux refers to this privileged mode as **kernel mode**. Under Linux, no user code is built into the kernel. Any operating-system-support code that does not need to run in kernel mode is placed into the system libraries and runs in **user mode**. Unlike kernel mode, user mode has access only to a controlled subset of the system's resources.

Although various modern operating systems have adopted a message-passing architecture for their kernel internals, Linux retains UNIX's historical model: the kernel is created as a single, monolithic binary. The main reason is performance. Because all kernel code and data structures are kept in a single address space, no context switches are necessary when a thread calls an operating-system function or when a hardware interrupt is delivered. Moreover, the kernel can pass data and make requests between various subsystems using relatively cheap C function invocation and not more complicated inter-process communication (IPC). This single address space contains not only the core scheduling and virtual memory code but all kernel code, including all device drivers, file systems, and networking code.

Even though all the kernel components share this same melting pot, there is still room for modularity. In the same way that user applications can load shared libraries at run time to pull in a needed piece of code, so the Linux kernel can load (and unload) modules dynamically at run time. The kernel does not need to know in advance which modules may be loaded—they are truly independent loadable components.

The Linux kernel forms the core of the Linux operating system. It provides all the functionality necessary to manage processes and run threads, and it provides system services to give arbitrated and protected access to hardware resources. The kernel implements all the features required to qualify as an operating system. On its own, however, the operating system provided by the Linux kernel is not a complete UNIX system. It lacks much of the functionality and behavior of UNIX, and the features that it does provide are not necessarily in the format in which a UNIX application expects them to appear. The operating-system interface visible to running applications is not maintained directly by the kernel. Rather, applications make calls to the system libraries, which in turn call the operating-system services as necessary.

The system libraries provide many types of functionality. At the simplest level, they allow applications to make system calls to the Linux kernel. Making a system call involves transferring control from unprivileged user mode to privileged kernel mode; the details of this transfer vary from architecture to architecture. The libraries take care of collecting the system-call arguments and, if necessary, arranging those arguments in the special form necessary to make the system call.

The libraries may also provide more complex versions of the basic system calls. For example, the C language's buffered file-handling functions are all implemented in the system libraries, providing more advanced control of file I/O than the basic kernel system calls. The libraries also provide routines that do not correspond to system calls at all, such as sorting algorithms, mathematical functions, and string-manipulation routines. All the functions necessary to support the running of UNIX or POSIX applications are implemented in the system libraries.

The Linux system includes a wide variety of user-mode programs—both system utilities and user utilities. The system utilities include all the programs necessary to initialize and then administer the system, such as those to set up networking interfaces and to add and remove users from the system. User utilities are also necessary to the basic operation of the system but do not require elevated privileges to run. They include simple file-management utilities such as those to copy files, create directories, and edit text files. One of the most important user utilities is the **shell**, the standard command-line interface on UNIX systems. Linux supports many shells; the most common is the **bourne-again shell (bash)**.

## 20.3 Kernel Modules

The Linux kernel has the ability to load and unload arbitrary sections of kernel code on demand. These loadable kernel modules run in privileged kernel mode and as a consequence have full access to all the hardware capabilities of the machine on which they run. In theory, there is no restriction on what a kernel module is allowed to do. Among other things, a kernel module can implement a device driver, a file system, or a networking protocol.

Kernel modules are convenient for several reasons. Linux's source code is free, so anybody wanting to write kernel code is able to compile a modified kernel and to reboot into that new functionality. However, recompiling, relinking, and reloading the entire kernel is a cumbersome cycle to undertake when you are developing a new driver. If you use kernel modules, you do not have to make a new kernel to test a new driver—the driver can be compiled on its own and loaded into the already running kernel. Of course, once a new driver is written, it can be distributed as a module so that other users can benefit from it without having to rebuild their kernels.

This latter point has another implication. Because it is covered by the GPL license, the Linux kernel cannot be released with proprietary components added to it unless those new components are also released under the GPL and the source code for them is made available on demand. The kernel's module interface allows third parties to write and distribute, on their own terms, device drivers or file systems that could not be distributed under the GPL.

Kernel modules allow a Linux system to be set up with a standard minimal kernel, without any extra device drivers built in. Any device drivers that the user needs can be either loaded explicitly by the system at startup or loaded automatically by the system on demand and unloaded when not in use. For example, a mouse driver can be loaded when a USB mouse is plugged into the system and unloaded when the mouse is unplugged.

The module support under Linux has four components:

1. The **module-management system** allows modules to be loaded into memory and to communicate with the rest of the kernel.
2. The **module loader and unloader**, which are user-mode utilities, work with the module-management system to load a module into memory.

3. The **driver-registration system** allows modules to tell the rest of the kernel that a new driver has become available.
4. A **conflict-resolution mechanism** allows different device drivers to reserve hardware resources and to protect those resources from accidental use by another driver.

### 20.3.1 Module Management

Loading a module requires more than just loading its binary contents into kernel memory. The system must also make sure that any references the module makes to kernel symbols or entry points are updated to point to the correct locations in the kernel's address space. Linux deals with this reference updating by splitting the job of module loading into two separate sections: the management of sections of module code in kernel memory and the handling of symbols that modules are allowed to reference.

Linux maintains an internal symbol table in the kernel. This symbol table does not contain the full set of symbols defined in the kernel during the latter's compilation; rather, a symbol must be explicitly exported. The set of exported symbols constitutes a well-defined interface by which a module can interact with the kernel.

Although exporting symbols from a kernel function requires an explicit request by the programmer, no special effort is needed to import those symbols into a module. A module writer just uses the standard external linking of the C language. Any external symbols referenced by the module but not declared by it are simply marked as unresolved in the final module binary produced by the compiler. When a module is to be loaded into the kernel, a system utility first scans the module for these unresolved references. All symbols that still need to be resolved are looked up in the kernel's symbol table, and the correct addresses of those symbols in the currently running kernel are substituted into the module's code. Only then is the module passed to the kernel for loading. If the system utility cannot resolve all references in the module by looking them up in the kernel's symbol table, then the module is rejected.

The loading of the module is performed in two stages. First, the module-loader utility asks the kernel to reserve a continuous area of virtual kernel memory for the module. The kernel returns the address of the memory allocated, and the loader utility can use this address to relocate the module's machine code to the correct loading address. A second system call then passes the module, plus any symbol table that the new module wants to export, to the kernel. The module itself is now copied verbatim into the previously allocated space, and the kernel's symbol table is updated with the new symbols for possible use by other modules not yet loaded.

The final module-management component is the module requester. The kernel defines a communication interface to which a module-management program can connect. With this connection established, the kernel will inform the management process whenever a process requests a device driver, file system, or network service that is not currently loaded and will give the manager the opportunity to load that service. The original service request will complete once the module is loaded. The manager process regularly queries the kernel to see whether a dynamically loaded module is still in use and unloads that module when it is no longer actively needed.

### 20.3.2 Driver Registration

Once a module is loaded, it remains no more than an isolated region of memory until it lets the rest of the kernel know what new functionality it provides. The kernel maintains dynamic tables of all known drivers and provides a set of routines to allow drivers to be added to or removed from these tables at any time. The kernel makes sure that it calls a module's startup routine when that module is loaded and calls the module's cleanup routine before that module is unloaded. These routines are responsible for registering the module's functionality.

A module may register many types of functionality; it is not limited to only one type. For example, a device driver might want to register two separate mechanisms for accessing the device. Registration tables include, among others, the following items:

- **Device drivers.** These drivers include character devices (such as printers, terminals, and mice), block devices (including all disk drives), and network interface devices.
- **File systems.** The file system may be anything that implements Linux's virtual file system calling routines. It might implement a format for storing files on a disk, but it might equally well be a network file system, such as NFS, or a virtual file system whose contents are generated on demand, such as Linux's /proc file system.
- **Network protocols.** A module may implement an entire networking protocol, such as TCP, or simply a new set of packet-filtering rules for a network firewall.
- **Binary format.** This format specifies a way of recognizing, loading, and executing a new type of executable file.

In addition, a module can register a new set of entries in the sysctl and /proc tables, to allow that module to be configured dynamically (Section 20.7.4).

### 20.3.3 Conflict Resolution

Commercial UNIX implementations are usually sold to run on a vendor's own hardware. One advantage of a single-supplier solution is that the software vendor has a good idea about what hardware configurations are possible. PC hardware, however, comes in a vast number of configurations, with large numbers of possible drivers for devices such as network cards and video display adapters. The problem of managing the hardware configuration becomes more severe when modular device drivers are supported, since the currently active set of devices becomes dynamically variable.

Linux provides a central conflict-resolution mechanism to help arbitrate access to certain hardware resources. Its aims are as follows:

- To prevent modules from clashing over access to hardware resources
- To prevent **autoprobes**—device-driver probes that auto-detect device configuration—from interfering with existing device drivers

- To resolve conflicts among multiple drivers trying to access the same hardware—as, for example, when both the parallel printer driver and the parallel line IP (PLIP) network driver try to talk to the parallel port

To these ends, the kernel maintains lists of allocated hardware resources. The PC has a limited number of possible I/O ports (addresses in its hardware I/O address space), interrupt lines, and DMA channels. When any device driver wants to access such a resource, it is expected to reserve the resource with the kernel database first. This requirement incidentally allows the system administrator to determine exactly which resources have been allocated by which driver at any given point.

A module is expected to use this mechanism to reserve in advance any hardware resources that it expects to use. If the reservation is rejected because the resource is not present or is already in use, then it is up to the module to decide how to proceed. It may fail in its initialization attempt and request that it be unloaded if it cannot continue, or it may carry on, using alternative hardware resources.

## 20.4 Process Management

A process is the basic context in which all user-requested activity is serviced within the operating system. To be compatible with other UNIX systems, Linux must use a process model similar to those of other versions of UNIX. Linux operates differently from UNIX in a few key places, however. In this section, we review the traditional UNIX process model (Section C.3.2) and introduce Linux's threading model.

### 20.4.1 The fork() and exec() Process Model

The basic principle of UNIX process management is to separate into two steps two operations that are usually combined into one: the creation of a new process and the running of a new program. A new process is created by the `fork()` system call, and a new program is run after a call to `exec()`. These are two distinctly separate functions. We can create a new process with `fork()` without running a new program—the new subprocess simply continues to execute exactly the same program, at exactly the same point, that the first (parent) process was running. In the same way, running a new program does not require that a new process be created first. Any process may call `exec()` at any time. A new binary object is loaded into the process's address space and the new executable starts executing in the context of the existing process.

This model has the advantage of great simplicity. It is not necessary to specify every detail of the environment of a new program in the system call that runs that program. The new program simply runs in its existing environment. If a parent process wishes to modify the environment in which a new program is to be run, it can fork and then, still running the original executable in a child process, make any system calls it requires to modify that child process before finally executing the new program.

Under UNIX, then, a process encompasses all the information that the operating system must maintain to track the context of a single execution of a

single program. Under Linux, we can break down this context into a number of specific sections. Broadly, process properties fall into three groups: the process identity, environment, and context.

#### 20.4.1.1 Process Identity

A process identity consists mainly of the following items:

- **Process ID (PID).** Each process has a unique identifier. The PID is used to specify the process to the operating system when an application makes a system call to signal, modify, or wait for the process. Additional identifiers associate the process with a process group (typically, a tree of processes forked by a single user command) and login session.
- **Credentials.** Each process must have an associated user ID and one or more group IDs (user groups are discussed in Section 13.4.2) that determine the rights of a process to access system resources and files.
- **Personality.** Process personalities are not traditionally found on UNIX systems, but under Linux each process has an associated personality identifier that can slightly modify the semantics of certain system calls. Personalities are primarily used by emulation libraries to request that system calls be compatible with certain varieties of UNIX.
- **Namespace.** Each process is associated with a specific view of the file-system hierarchy, called its **namespace**. Most processes share a common namespace and thus operate on a shared file-system hierarchy. Processes and their children can, however, have different namespaces, each with a unique file-system hierarchy—their own root directory and set of mounted file systems.

Most of these identifiers are under the limited control of the process itself. The process group and session identifiers can be changed if the process wants to start a new group or session. Its credentials can be changed, subject to appropriate security checks. However, the primary PID of a process is unchangeable and uniquely identifies that process until termination.

#### 20.4.1.2 Process Environment

A process's environment is inherited from its parent and is composed of two null-terminated vectors: the argument vector and the environment vector. The **argument vector** simply lists the command-line arguments used to invoke the running program; it conventionally starts with the name of the program itself. The **environment vector** is a list of “NAME=VALUE” pairs that associates named environment variables with arbitrary textual values. The environment is not held in kernel memory but is stored in the process's own user-mode address space as the first datum at the top of the process's stack.

The argument and environment vectors are not altered when a new process is created. The new child process will inherit the environment of its parent. However, a completely new environment is set up when a new program is invoked. On calling `exec()`, a process must supply the environment for the new program. The kernel passes these environment variables to the next

program, replacing the process's current environment. The kernel otherwise leaves the environment and command-line vectors alone—their interpretation is left entirely to the user-mode libraries and applications.

The passing of environment variables from one process to the next and the inheriting of these variables by the children of a process provide flexible ways to pass information to components of the user-mode system software. Various important environment variables have conventional meanings to related parts of the system software. For example, the TERM variable is set up to name the type of terminal connected to a user's login session. Many programs use this variable to determine how to perform operations on the user's display, such as moving the cursor and scrolling a region of text. Programs with multilingual support use the LANG variable to determine the language in which to display system messages for programs that include multilingual support.

The environment-variable mechanism custom-tailors the operating system on a per-process basis. Users can choose their own languages or select their own editors independently of one another.

#### 20.4.1.3 Process Context

The process identity and environment properties are usually set up when a process is created and not changed until that process exits. A process may choose to change some aspects of its identity if it needs to do so, or it may alter its environment. In contrast, process context is the state of the running program at any one time; it changes constantly. Process context includes the following parts:

- **Scheduling context.** The most important part of the process context is its scheduling context—the information that the scheduler needs to suspend and restart the process. This information includes saved copies of all the process's registers. Floating-point registers are stored separately and are restored only when needed. Thus, processes that do not use floating-point arithmetic do not incur the overhead of saving that state. The scheduling context also includes information about scheduling priority and about any outstanding signals waiting to be delivered to the process. A key part of the scheduling context is the process's kernel stack, a separate area of kernel memory reserved for use by kernel-mode code. Both system calls and interrupts that occur while the process is executing will use this stack.
- **Accounting.** The kernel maintains accounting information about the resources currently being consumed by each process and the total resources consumed by the process in its entire lifetime so far.
- **File table.** The file table is an array of pointers to kernel file structures representing open files. When making file-I/O system calls, processes refer to files by an integer, known as a **file descriptor (fd)**, that the kernel uses to index into this table.
- **File-system context.** Whereas the file table lists the existing open files, the file-system context applies to requests to open new files. The file-system context includes the process's root directory, current working directory, and namespace.

- **Signal-handler table.** UNIX systems can deliver asynchronous signals to a process in response to various external events. The signal-handler table defines the action to take in response to a specific signal. Valid actions include ignoring the signal, terminating the process, and invoking a routine in the process's address space.
- **Virtual memory context.** The virtual memory context describes the full contents of a process's private address space; we discuss it in Section 20.6.

### 20.4.2 Processes and Threads

Linux provides the `fork()` system call, which duplicates a process without loading a new executable image. Linux also provides the ability to create threads via the `clone()` system call. Linux does not distinguish between processes and threads, however. In fact, Linux generally uses the term *task*—rather than *process* or *thread*—when referring to a flow of control within a program. The `clone()` system call behaves identically to `fork()`, except that it accepts as arguments a set of flags that dictate what resources are shared between the parent and child (whereas a process created with `fork()` shares no resources with its parent). The flags include:

flag	meaning
<code>CLONE_FS</code>	File-system information is shared.
<code>CLONE_VM</code>	The same memory space is shared.
<code>CLONE_SIGHAND</code>	Signal handlers are shared.
<code>CLONE_FILES</code>	The set of open files is shared.

Thus, if `clone()` is passed the flags `CLONE_FS`, `CLONE_VM`, `CLONE_SIGHAND`, and `CLONE_FILES`, the parent and child tasks will share the same file-system information (such as the current working directory), the same memory space, the same signal handlers, and the same set of open files. Using `clone()` in this fashion is equivalent to creating a thread in other systems, since the parent task shares most of its resources with its child task. If none of these flags is set when `clone()` is invoked, however, the associated resources are not shared, resulting in functionality similar to that of the `fork()` system call.

The lack of distinction between processes and threads is possible because Linux does not hold a process's entire context within the main process data structure. Rather, it holds the context within independent subcontexts. Thus, a process's file-system context, file-descriptor table, signal-handler table, and virtual memory context are held in separate data structures. The process data structure simply contains pointers to these other structures, so any number of processes can easily share a subcontext by pointing to the same subcontext and incrementing a reference count.

The arguments to the `clone()` system call tell it which subcontexts to copy and which to share. The new process is always given a new identity and a new scheduling context—these are the essentials of a Linux process. According to the arguments passed, however, the kernel may either create new subcontext data structures initialized so as to be copies of the parent's or set up the new process to use the same subcontext data structures being used by the parent.

The `fork()` system call is nothing more than a special case of `clone()` that copies all subcontexts, sharing none.

## 20.5 Scheduling

Scheduling is the job of allocating CPU time to different tasks within an operating system. Linux, like all UNIX systems, supports **preemptive multitasking**. In such a system, the process scheduler decides which thread runs and when. Making these decisions in a way that balances fairness and performance across many different workloads is one of the more complicated challenges in modern operating systems.

Normally, we think of scheduling as the running and interrupting of user threads, but another aspect of scheduling is also important in Linux: the running of the various kernel tasks. Kernel tasks encompass both tasks that are requested by a running thread and tasks that execute internally on behalf of the kernel itself, such as tasks spawned by Linux's I/O subsystem.

### 20.5.1 Thread Scheduling

Linux has two separate process-scheduling algorithms. One is a time-sharing algorithm for fair, preemptive scheduling among multiple threads. The other is designed for real-time tasks, where absolute priorities are more important than fairness.

The scheduling algorithm used for routine time-sharing tasks received a major overhaul with version 2.6 of the kernel. Earlier versions ran a variation of the traditional UNIX scheduling algorithm. This algorithm does not provide adequate support for SMP systems, does not scale well as the number of tasks on the system grows, and does not maintain fairness among interactive tasks, particularly on systems such as desktops and mobile devices. The thread scheduler was first overhauled with version 2.5 of the kernel. Version 2.5 implemented a scheduling algorithm that selects which task to run in constant time—known as  $O(1)$ —regardless of the number of tasks or processors in the system. The new scheduler also provided increased support for SMP, including processor affinity and load balancing. These changes, while improving scalability, did not improve interactive performance or fairness—and, in fact, made these problems worse under certain workloads. Consequently, the thread scheduler was overhauled a second time, with Linux kernel version 2.6. This version ushered in the **Completely Fair Scheduler (CFS)**.

The Linux scheduler is a preemptive, priority-based algorithm with two separate priority ranges: a **real-time** range from 0 to 99 and a **nice value** ranging from -20 to 19. Smaller nice values indicate higher priorities. Thus, by increasing the nice value, you are decreasing your priority and being “nice” to the rest of the system.

CFS is a significant departure from the traditional UNIX process scheduler. In the latter, the core variables in the scheduling algorithm are priority and time slice. The **time slice** is the length of time—the *slice* of the processor—that a thread is afforded. Traditional UNIX systems give processes a fixed time slice, perhaps with a boost or penalty for high- or low-priority processes,

respectively. A process may run for the length of its time slice, and higher-priority processes run before lower-priority processes. It is a simple algorithm that many non-UNIX systems employ. Such simplicity worked well for early time-sharing systems but has proved incapable of delivering good interactive performance and fairness on today's modern desktops and mobile devices.

CFS introduced a new scheduling algorithm called **fair scheduling** that eliminates time slices in the traditional sense. Instead of time slices, all threads are allotted a proportion of the processor's time. CFS calculates how long a thread should run as a function of the total number of runnable threads. To start, CFS says that if there are  $N$  runnable threads, then each should be afforded  $1/N$  of the processor's time. CFS then adjusts this allotment by weighting each thread's allotment by its `nice` value. Threads with the default `nice` value have a weight of 1—their priority is unchanged. Threads with a smaller `nice` value (higher priority) receive a higher weight, while threads with a larger `nice` value (lower priority) receive a lower weight. CFS then runs each thread for a “time slice” proportional to the process’s weight divided by the total weight of all runnable processes.

To calculate the actual length of time a thread runs, CFS relies on a configurable variable called **target latency**, which is the interval of time during which every runnable task should run at least once. For example, assume that the target latency is 10 milliseconds. Further assume that we have two runnable threads of the same priority. Each of these threads has the same weight and therefore receives the same proportion of the processor's time. In this case, with a target latency of 10 milliseconds, the first process runs for 5 milliseconds, then the other process runs for 5 milliseconds, then the first process runs for 5 milliseconds again, and so forth. If we have 10 runnable threads, then CFS will run each for a millisecond before repeating.

But what if we had, say, 1,000 threads? Each thread would run for 1 microsecond if we followed the procedure just described. Due to switching costs, scheduling threads for such short lengths of time is inefficient. CFS consequently relies on a second configurable variable, the **minimum granularity**, which is a minimum length of time any thread is allotted the processor. All threads, regardless of the target latency, will run for at least the minimum granularity. In this manner, CFS ensures that switching costs do not grow unacceptably large when the number of runnable threads increases significantly. In doing so, it violates its attempts at fairness. In the usual case, however, the number of runnable threads remains reasonable, and both fairness and switching costs are maximized.

With the switch to fair scheduling, CFS behaves differently from traditional UNIX process schedulers in several ways. Most notably, as we have seen, CFS eliminates the concept of a static time slice. Instead, each thread receives a proportion of the processor's time. How long that allotment is depends on how many other threads are runnable. This approach solves several problems in mapping priorities to time slices inherent in preemptive, priority-based scheduling algorithms. It is possible, of course, to solve these problems in other ways without abandoning the classic UNIX scheduler. CFS, however, solves the problems with a simple algorithm that performs well on interactive workloads such as mobile devices without compromising throughput performance on the largest of servers.

### 20.5.2 Real-Time Scheduling

Linux's real-time scheduling algorithm is significantly simpler than the fair scheduling employed for standard time-sharing threads. Linux implements the two real-time scheduling classes required by POSIX.1b: first-come, first-served (FCFS) and round-robin (Section 5.3.1 and Section 5.3.3, respectively). In both cases, each thread has a priority in addition to its scheduling class. The scheduler always runs the thread with the highest priority. Among threads of equal priority, it runs the thread that has been waiting longest. The only difference between FCFS and round-robin scheduling is that FCFS threads continue to run until they either exit or block, whereas a round-robin thread will be preempted after a while and will be moved to the end of the scheduling queue, so round-robin threads of equal priority will automatically time-share among themselves.

Linux's real-time scheduling is soft—rather than hard—real time. The scheduler offers strict guarantees about the relative priorities of real-time threads, but the kernel does not offer any guarantees about how quickly a real-time thread will be scheduled once that thread becomes runnable. In contrast, a hard real-time system can guarantee a minimum latency between when a thread becomes runnable and when it actually runs.

### 20.5.3 Kernel Synchronization

The way the kernel schedules its own operations is fundamentally different from the way it schedules threads. A request for kernel-mode execution can occur in two ways. A running program may request an operating-system service, either explicitly via a system call or implicitly—for example, when a page fault occurs. Alternatively, a device controller may deliver a hardware interrupt that causes the CPU to start executing a kernel-defined handler for that interrupt.

The problem for the kernel is that all these tasks may try to access the same internal data structures. If one kernel task is in the middle of accessing some data structure when an interrupt service routine executes, then that service routine cannot access or modify the same data without risking data corruption. This fact relates to the idea of critical sections—portions of code that access shared data and thus must not be allowed to execute concurrently. As a result, kernel synchronization involves much more than just thread scheduling. A framework is required that allows kernel tasks to run without violating the integrity of shared data.

Prior to version 2.6, Linux was a nonpreemptive kernel, meaning that a thread running in kernel mode could not be preempted—even if a higher-priority thread became available to run. With version 2.6, the Linux kernel became fully preemptive. Now, a task can be preempted when it is running in the kernel.

The Linux kernel provides spinlocks and semaphores (as well as reader-writer versions of these two locks) for locking in the kernel. On SMP machines, the fundamental locking mechanism is a spinlock, and the kernel is designed so that spinlocks are held for only short durations. On single-processor machines, spinlocks are not appropriate for use and are replaced by enabling and disabling kernel preemption. That is, rather than holding a spinlock, the task dis-

ables kernel preemption. When the task would otherwise release the spinlock, it enables kernel preemption. This pattern is summarized below:

single processor	multiple processors
Disable kernel preemption.	Acquire spin lock.
Enable kernel preemption.	Release spin lock.

Linux uses an interesting approach to disable and enable kernel preemption. It provides two simple kernel interfaces—`preempt_disable()` and `preempt_enable()`. In addition, the kernel is not preemptible if a kernel-mode task is holding a spinlock. To enforce this rule, each task in the system has a `thread_info` structure that includes the field `preempt_count`, which is a counter indicating the number of locks being held by the task. The counter is incremented when a lock is acquired and decremented when a lock is released. If the value of `preempt_count` for the task currently running is greater than zero, it is not safe to preempt the kernel, as this task currently holds a lock. If the count is zero, the kernel can safely be interrupted, assuming there are no outstanding calls to `preempt_disable()`.

Spinlocks—along with the enabling and disabling of kernel preemption—are used in the kernel only when the lock is held for short durations. When a lock must be held for longer periods, semaphores are used.

The second protection technique used by Linux applies to critical sections that occur in interrupt service routines. The basic tool is the processor's interrupt-control hardware. By disabling interrupts (or using spinlocks) during a critical section, the kernel guarantees that it can proceed without the risk of concurrent access to shared data structures.

However, there is a penalty for disabling interrupts. On most hardware architectures, interrupt enable and disable instructions are not cheap. More importantly, as long as interrupts remain disabled, all I/O is suspended, and any device waiting for servicing will have to wait until interrupts are reenabled; thus, performance degrades. To address this problem, the Linux kernel uses a synchronization architecture that allows long critical sections to run for their entire duration without having interrupts disabled. This ability is especially useful in the networking code. An interrupt in a network device driver can signal the arrival of an entire network packet, which may result in a great deal of code being executed to disassemble, route, and forward that packet within the interrupt service routine.

Linux implements this architecture by separating interrupt service routines into two sections: the top half and the bottom half. The *top half* is the standard interrupt service routine that runs with recursive interrupts disabled. Interrupts of the same number (or line) are disabled, but other interrupts may run. The *bottom half* of a service routine is run, with all interrupts enabled, by a miniature scheduler that ensures that bottom halves never interrupt themselves. The bottom-half scheduler is invoked automatically whenever an interrupt service routine exits.

This separation means that the kernel can complete any complex processing that has to be done in response to an interrupt without worrying about being interrupted itself. If another interrupt occurs while a bottom half is exe-

cuting, then that interrupt can request that the same bottom half execute, but the execution will be deferred until the one currently running completes. Each execution of the bottom half can be interrupted by a top half but can never be interrupted by a similar bottom half.

The top-half/bottom-half architecture is completed by a mechanism for disabling selected bottom halves while executing normal, foreground kernel code. The kernel can code critical sections easily using this system. Interrupt handlers can code their critical sections as bottom halves; and when the foreground kernel wants to enter a critical section, it can disable any relevant bottom halves to prevent any other critical sections from interrupting it. At the end of the critical section, the kernel can reenable the bottom halves and run any bottom-half tasks that have been queued by top-half interrupt service routines during the critical section.

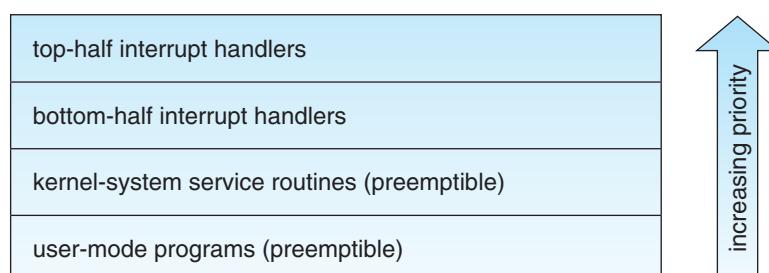
Figure 20.2 summarizes the various levels of interrupt protection within the kernel. Each level may be interrupted by code running at a higher level but will never be interrupted by code running at the same or a lower level. Except for user-mode code, user threads can always be preempted by another thread when a time-sharing scheduling interrupt occurs.

#### 20.5.4 Symmetric Multiprocessing

The Linux 2.0 kernel was the first stable Linux kernel to support **symmetric multiprocessor (SMP)** hardware, allowing separate threads to execute in parallel on separate processors. The original implementation of SMP imposed the restriction that only one processor at a time could be executing kernel code.

In version 2.2 of the kernel, a single kernel spinlock (sometimes termed **BKL** for “big kernel lock”) was created to allow multiple threads (running on different processors) to be active in the kernel concurrently. However, the BKL provided a very coarse level of locking granularity, resulting in poor scalability to machines with many processors and threads. Later releases of the kernel made the SMP implementation more scalable by splitting this single kernel spinlock into multiple locks, each of which protects only a small subset of the kernel’s data structures. Such spinlocks were described in Section 20.5.3.

The 3.0 and 4.0 kernels provided additional SMP enhancements, including ever-finer locking, processor affinity, load-balancing algorithms, and support for hundreds or even thousands of physical processors in a single system.



**Figure 20.2** Interrupt protection levels.

## 20.6 Memory Management

Memory management under Linux has two components. The first deals with allocating and freeing physical memory—pages, groups of pages, and small blocks of RAM. The second handles virtual memory, which is memory-mapped into the address space of running processes. In this section, we describe these two components and then examine the mechanisms by which the loadable components of a new program are brought into a process's virtual memory in response to an `exec()` system call.

### 20.6.1 Management of Physical Memory

Due to specific hardware constraints, Linux separates physical memory into four different **zones**, or regions:

- ZONE\_DMA
- ZONE\_DMA32
- ZONE\_NORMAL
- ZONE\_HIGHMEM

These zones are architecture specific. For example, on the Intel x86-32 architecture, certain ISA (industry standard architecture) devices can only access the lower 16-MB of physical memory using DMA. On these systems, the first 16-MB of physical memory comprise ZONE\_DMA. On other systems, certain devices can only access the first 4-GB of physical memory, despite supporting 64-bit addresses. On such systems, the first 4 GB of physical memory comprise ZONE\_DMA32. ZONE\_HIGHMEM (for “high memory”) refers to physical memory that is not mapped into the kernel address space. For example, on the 32-bit Intel architecture (where  $2^{32}$  provides a 4-GB address space), the kernel is mapped into the first 896 MB of the address space; the remaining memory is referred to as *high memory* and is allocated from ZONE\_HIGHMEM. Finally, ZONE\_NORMAL comprises everything else—the normal, regularly mapped pages. Whether an architecture has a given zone depends on its constraints. A modern, 64-bit architecture such as Intel x86-64 has a small 16-MB ZONE\_DMA (for legacy devices) and all the rest of its memory in ZONE\_NORMAL, with no “high memory”.

The relationship of zones and physical addresses on the Intel x86-32 architecture is shown in Figure 20.3. The kernel maintains a list of free pages for

zone	physical memory
ZONE_DMA	< 16 MB
ZONE_NORMAL	16 .. 896 MB
ZONE_HIGHMEM	> 896 MB

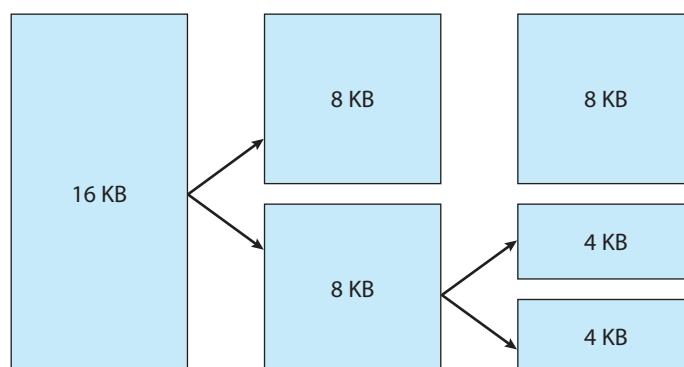
**Figure 20.3** Relationship of zones and physical addresses in Intel x86-32.

each zone. When a request for physical memory arrives, the kernel satisfies the request using the appropriate zone.

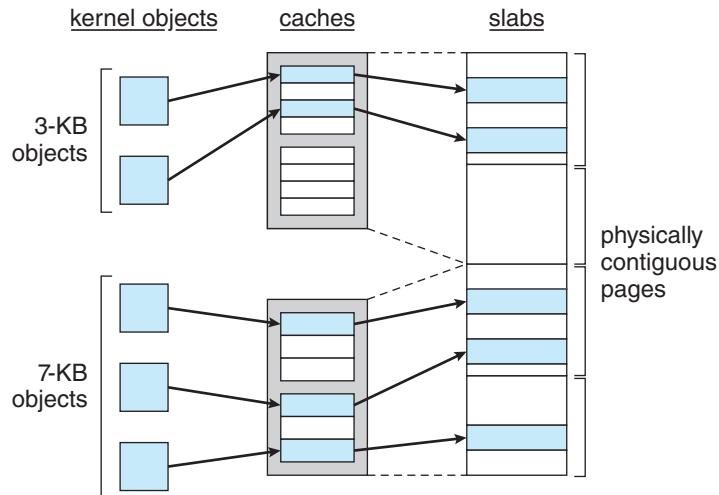
The primary physical-memory manager in the Linux kernel is the [page allocator](#). Each zone has its own allocator, which is responsible for allocating and freeing all physical pages for the zone and is capable of allocating ranges of physically contiguous pages on request. The allocator uses a buddy system (Section 10.8.1) to keep track of available physical pages. In this scheme, adjacent units of allocatable memory are paired together (hence its name). Each allocatable memory region has an adjacent partner or buddy. Whenever two allocated partner regions are freed up, they are combined to form a larger region—a [buddy heap](#). That larger region also has a partner, with which it can combine to form a still larger free region. Conversely, if a small memory request cannot be satisfied by allocation of an existing small free region, then a larger free region will be subdivided into two partners to satisfy the request. Separate linked lists are used to record the free memory regions of each allowable size. Under Linux, the smallest size allocatable under this mechanism is a single physical page. Figure 20.4 shows an example of buddy-heap allocation. A 4-KB region is being allocated, but the smallest available region is 16 KB. The region is broken up recursively until a piece of the desired size is available.

Ultimately, all memory allocations in the Linux kernel are made either statically, by drivers that reserve a contiguous area of memory during system boot time, or dynamically, by the page allocator. However, kernel functions do not have to use the basic allocator to reserve memory. Several specialized memory-management subsystems use the underlying page allocator to manage their own pools of memory. The most important are the virtual memory system, described in Section 20.6.2; the `kmalloc()` variable-length allocator; the slab allocator, used for allocating memory for kernel data structures; and the page cache, used for caching pages belonging to files.

Many components of the Linux operating system need to allocate entire pages on request, but often smaller blocks of memory are required. The kernel provides an additional allocator for arbitrary-sized requests, where the size of a request is not known in advance and may be only a few bytes. Analogous to the C language’s `malloc()` function, this `kmalloc()` service allocates entire physical pages on demand but then splits them into smaller pieces. The



**Figure 20.4** Splitting of memory in the buddy system.



**Figure 20.5** Slab allocator in Linux.

kernel maintains lists of pages in use by the `kmalloc()` service. Allocating memory involves determining the appropriate list and either taking the first free piece available on the list or allocating a new page and splitting it up. Memory regions claimed by the `kmalloc()` system are allocated permanently until they are freed explicitly with a corresponding call to `kfree()`; the `kmalloc()` system cannot reallocate or reclaim these regions in response to memory shortages.

Another strategy adopted by Linux for allocating kernel memory is known as slab allocation. A **slab** is used for allocating memory for kernel data structures and is made up of one or more physically contiguous pages. A **cache** consists of one or more slabs. There is a single cache for each unique kernel data structure—for example, a cache for the data structure representing process descriptors, a cache for file objects, a cache for inodes, and so forth. Each cache is populated with **objects** that are instantiations of the kernel data structure the cache represents. For example, the cache representing inodes stores instances of inode structures, and the cache representing process descriptors stores instances of process descriptor structures. The relationship among slabs, caches, and objects is shown in Figure 20.5. The figure shows two kernel objects 3 KB in size and three objects 7 KB in size. These objects are stored in the respective caches for 3-KB and 7-KB objects.

The slab-allocation algorithm uses caches to store kernel objects. When a cache is created, a number of objects are allocated to the cache. The number of objects in the cache depends on the size of the associated slab. For example, a 12-KB slab (made up of three contiguous 4-KB pages) could store six 2-KB objects. Initially, all the objects in the cache are marked as `free`. When a new object for a kernel data structure is needed, the allocator can assign any free object from the cache to satisfy the request. The object assigned from the cache is marked as `used`.

Let's consider a scenario in which the kernel requests memory from the slab allocator for an object representing a process descriptor. In Linux systems, a process descriptor is of the type `struct task_struct`, which requires

approximately 1.7 KB of memory. When the Linux kernel creates a new task, it requests the necessary memory for the `struct task_struct` object from its cache. The cache will fulfill the request using a `struct task_struct` object that has already been allocated in a slab and is marked as free.

In Linux, a slab may be in one of three possible states:

1. **Full.** All objects in the slab are marked as used.
2. **Empty.** All objects in the slab are marked as free.
3. **Partial.** The slab consists of both used and free objects.

The slab allocator first attempts to satisfy the request with a free object in a partial slab. If none exists, a free object is assigned from an empty slab. If no empty slabs are available, a new slab is allocated from contiguous physical pages and assigned to a cache; memory for the object is allocated from this slab.

Two other main subsystems in Linux do their own management of physical pages: the page cache and the virtual memory system. These systems are closely related to each other. The [page cache](#) is the kernel's main cache for files and is the main mechanism through which I/O to block devices (Section 20.8.1) is performed. File systems of all types, including the native Linux disk-based file systems and the NFS networked file system, perform their I/O through the page cache. The page cache stores entire pages of file contents and is not limited to block devices. It can also cache networked data. The virtual memory system manages the contents of each process's virtual address space. These two systems interact closely with each other because reading a page of data into the page cache requires mapping pages in the page cache using the virtual memory system. In the following section, we look at the virtual memory system in greater detail.

### 20.6.2 Virtual Memory

The Linux virtual memory system is responsible for maintaining the address space accessible to each process. It creates pages of virtual memory on demand and manages loading those pages from disk and swapping them back out to disk as required. Under Linux, the virtual memory manager maintains two separate views of a process's address space: as a set of separate regions and as a set of pages.

The first view of an address space is the logical view, describing instructions that the virtual memory system has received concerning the layout of the address space. In this view, the address space consists of a set of nonoverlapping regions, each region representing a continuous, page-aligned subset of the address space. Each region is described internally by a single `vm_area_struct` structure that defines the properties of the region, including the process's read, write, and execute permissions in the region as well as information about any files associated with the region. The regions for each address space are linked into a balanced binary tree to allow fast lookup of the region corresponding to any virtual address.

The kernel also maintains a second, physical view of each address space. This view is stored in the hardware page tables for the process. The page-table entries identify the exact current location of each page of virtual memory,

whether it is on disk or in physical memory. The physical view is managed by a set of routines, which are invoked from the kernel's software-interrupt handlers whenever a process tries to access a page that is not currently present in the page tables. Each `vm_area_struct` in the address-space description contains a field pointing to a table of functions that implement the key page-management functionality for any given virtual memory region. All requests to read or write an unavailable page are eventually dispatched to the appropriate handler in the function table for the `vm_area_struct`, so that the central memory-management routines do not have to know the details of managing each possible type of memory region.

### 20.6.2.1 Virtual Memory Regions

Linux implements several types of virtual memory regions. One property that characterizes virtual memory is the backing store for the region, which describes where the pages for the region come from. Most memory regions are backed either by a file or by nothing. A region backed by nothing is the simplest type of virtual memory region. Such a region represents **demand-zero memory**: when a process tries to read a page in such a region, it is simply given back a page of memory filled with zeros.

A region backed by a file acts as a viewport onto a section of that file. Whenever the process tries to access a page within that region, the page table is filled with the address of a page within the kernel's page cache corresponding to the appropriate offset in the file. The same page of physical memory is used by both the page cache and the process's page tables, so any changes made to the file by the file system are immediately visible to any processes that have mapped that file into their address space. Any number of processes can map the same region of the same file, and they will all end up using the same page of physical memory for the purpose.

A virtual memory region is also defined by its reaction to writes. The mapping of a region into the process's address space can be either *private* or *shared*. If a process writes to a privately mapped region, then the pager detects that a copy-on-write is necessary to keep the changes local to the process. In contrast, writes to a shared region result in updating of the object mapped into that region, so that the change will be visible immediately to any other process that is mapping that object.

### 20.6.2.2 Lifetime of a Virtual Address Space

The kernel creates a new virtual address space in two situations: when a process runs a new program with the `exec()` system call and when a new process is created by the `fork()` system call. The first case is easy. When a new program is executed, the process is given a new, completely empty virtual address space. It is up to the routines for loading the program to populate the address space with virtual memory regions.

The second case, creating a new process with `fork()`, involves creating a complete copy of the existing process's virtual address space. The kernel copies the parent process's `vm_area_struct` descriptors, then creates a new set of page tables for the child. The parent's page tables are copied directly into the child's, and the reference count of each page covered is incremented. Thus,

after the fork, the parent and child share the same physical pages of memory in their address spaces.

A special case occurs when the copying operation reaches a virtual memory region that is mapped privately. Any pages to which the parent process has written within such a region are private, and subsequent changes to these pages by either the parent or the child must not update the page in the other process's address space. When the page-table entries for such regions are copied, they are set to be read only and are marked for copy-on-write. As long as neither process modifies these pages, the two processes share the same page of physical memory. However, if either process tries to modify a copy-on-write page, the reference count on the page is checked. If the page is still shared, then the process copies the page's contents to a brand-new page of physical memory and uses its copy instead. This mechanism ensures that private data pages are shared between processes whenever possible and copies are made only when absolutely necessary.

### 20.6.2.3 Swapping and Paging

An important task for a virtual memory system is to relocate pages of memory from physical memory out to disk when that memory is needed. Early UNIX systems performed this relocation by swapping out the contents of entire processes at once, but modern versions of UNIX rely more on paging—the movement of individual pages of virtual memory between physical memory and disk. Linux does not implement whole-process swapping; it uses the newer paging mechanism exclusively.

The paging system can be divided into two sections. First, the **policy algorithm** decides which pages to write out to backing store and when to write them. Second, the **paging mechanism** carries out the transfer and pages data back into physical memory when they are needed again.

Linux's **pageout policy** uses a modified version of the standard clock (or second-chance) algorithm described in Section 10.4.5.2. Under Linux, a multiple-pass clock is used, and every page has an *age* that is adjusted on each pass of the clock. The age is more precisely a measure of the page's youthfulness, or how much activity the page has seen recently. Frequently accessed pages will attain a higher age value, but the age of infrequently accessed pages will drop toward zero with each pass. This age valuing allows the pager to select pages to page out based on a least frequently used (LFU) policy.

The paging mechanism supports paging both to dedicated swap devices and partitions and to normal files, although swapping to a file is significantly slower due to the extra overhead incurred by the file system. Blocks are allocated from the swap devices according to a bitmap of used blocks, which is maintained in physical memory at all times. The allocator uses a next-fit algorithm to try to write out pages to continuous runs of secondary storage blocks for improved performance. The allocator records the fact that a page has been paged out to storage by using a feature of the page tables on modern processors: the page-table entry's page-not-present bit is set, allowing the rest of the page-table entry to be filled with an index identifying where the page has been written.

#### 20.6.2.4 Kernel Virtual Memory

Linux reserves for its own internal use a constant, architecture-dependent region of the virtual address space of every process. The page-table entries that map to these kernel pages are marked as protected, so that the pages are not visible or modifiable when the processor is running in user mode. This kernel virtual memory area contains two regions. The first is a static area that contains page-table references to every available physical page of memory in the system, so that a simple translation from physical to virtual addresses occurs when kernel code is run. The core of the kernel, along with all pages allocated by the normal page allocator, resides in this region.

The remainder of the kernel's reserved section of address space is not reserved for any specific purpose. Page-table entries in this address range can be modified by the kernel to point to any other areas of memory. The kernel provides a pair of facilities that allow kernel code to use this virtual memory. The `vmalloc()` function allocates an arbitrary number of physical pages of memory that may not be physically contiguous into a single region of virtually contiguous kernel memory. The `vremap()` function maps a sequence of virtual addresses to point to an area of memory used by a device driver for memory-mapped I/O.

#### 20.6.3 Execution and Loading of User Programs

The Linux kernel's execution of user programs is triggered by a call to the `exec()` system call. This `exec()` call commands the kernel to run a new program within the current process, completely overwriting the current execution context with the initial context of the new program. The first job of this system service is to verify that the calling process has permission rights to the file being executed. Once that matter has been checked, the kernel invokes a loader routine to start running the program. The loader does not necessarily load the contents of the program file into physical memory, but it does at least set up the mapping of the program into virtual memory.

There is no single routine in Linux for loading a new program. Instead, Linux maintains a table of possible loader functions, and it gives each such function the opportunity to try loading the given file when an `exec()` system call is made. The initial reason for this loader table was that, between the releases of the 1.0 and 1.2 kernels, the standard format for Linux's binary files was changed. Older Linux kernels understood the `a.out` format for binary files —a relatively simple format common on older UNIX systems. Newer Linux systems use the more modern `ELF` format, now supported by most current UNIX implementations. ELF has a number of advantages over `a.out`, including flexibility and extendability. New sections can be added to an ELF binary (for example, to add extra debugging information) without causing the loader routines to become confused. By allowing registration of multiple loader routines, Linux can easily support the ELF and `a.out` binary formats in a single running system.

In Section 20.6.3.1 and Section 20.6.3.2, we concentrate exclusively on the loading and running of ELF-format binaries. The procedure for loading `a.out` binaries is simpler but similar in operation.

### 20.6.3.1 Mapping of Programs into Memory

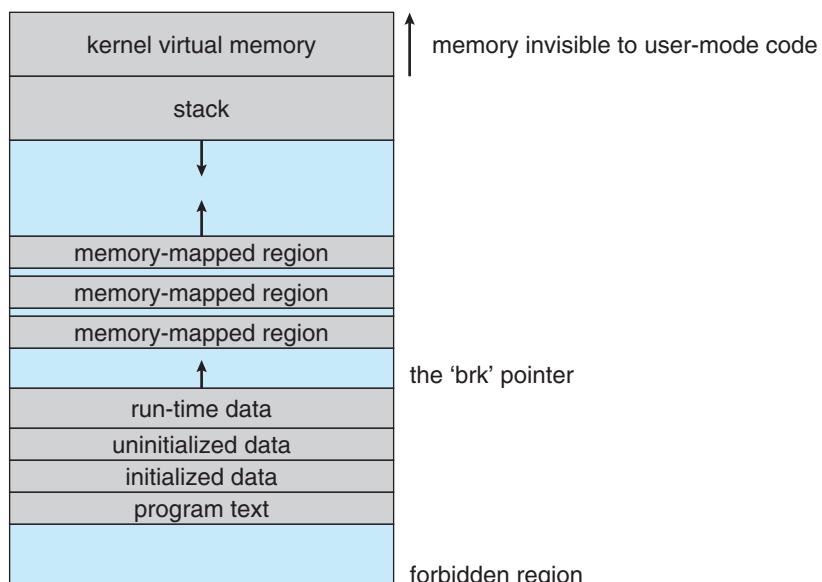
Under Linux, the binary loader does not load a binary file into physical memory. Rather, the pages of the binary file are mapped into regions of virtual memory. Only when the program tries to access a given page will a page fault result in the loading of that page into physical memory using demand paging.

It is the responsibility of the kernel's binary loader to set up the initial memory mapping. An ELF-format binary file consists of a header followed by several page-aligned sections. The ELF loader works by reading the header and mapping the sections of the file into separate regions of virtual memory.

Figure 20.6 shows the typical layout of memory regions set up by the ELF loader. In a reserved region at one end of the address space sits the kernel, in its own privileged region of virtual memory inaccessible to normal user-mode programs. The rest of virtual memory is available to applications, which can use the kernel's memory-mapping functions to create regions that map a portion of a file or that are available for application data.

The loader's job is to set up the initial memory mapping to allow the execution of the program to start. The regions that need to be initialized include the stack and the program's text and data regions.

The stack is created at the top of the user-mode virtual memory; it grows downward toward lower-numbered addresses. It includes copies of the arguments and environment variables given to the program in the `exec()` system call. The other regions are created near the bottom end of virtual memory. The sections of the binary file that contain program text or read-only data are mapped into memory as a write-protected region. Writable initialized data are mapped next; then any uninitialized data are mapped in as a private demand-zero region.



**Figure 20.6** Memory layout for ELF programs.

Directly beyond these fixed-sized regions is a variable-sized region that programs can expand as needed to hold data allocated at run time. Each process has a pointer, `brk`, that points to the current extent of this data region, and processes can extend or contract their `brk` region with a single system call —`sbrk()`.

Once these mappings have been set up, the loader initializes the process's program-counter register with the starting point recorded in the ELF header, and the process can be scheduled.

#### 20.6.3.2 Static and Dynamic Linking

Once the program has been loaded and has started running, all the necessary contents of the binary file have been loaded into the process's virtual address space. However, most programs also need to run functions from the system libraries, and these library functions must also be loaded. In the simplest case, the necessary library functions are embedded directly in the program's executable binary file. Such a program is statically linked to its libraries, and statically linked executables can commence running as soon as they are loaded.

The main disadvantage of static linking is that every program generated must contain copies of exactly the same common system library functions. It is much more efficient, in terms of both physical memory and disk-space usage, to load the system libraries into memory only once. Dynamic linking allows that to happen.

Linux implements dynamic linking in user mode through a special linker library. Every dynamically linked program contains a small, statically linked function that is called when the program starts. This static function just maps the link library into memory and runs the code that the function contains. The link library determines the dynamic libraries required by the program and the names of the variables and functions needed from those libraries by reading the information contained in sections of the ELF binary. It then maps the libraries into the middle of virtual memory and resolves the references to the symbols contained in those libraries. It does not matter exactly where in memory these shared libraries are mapped: they are compiled into **position-independent code (PIC)**, which can run at any address in memory.

## 20.7 File Systems

Linux retains UNIX's standard file-system model. In UNIX, a file does not have to be an object stored on disk or fetched over a network from a remote file server. Rather, UNIX files can be anything capable of handling the input or output of a stream of data. Device drivers can appear as files, and interprocess-communication channels or network connections also look like files to the user.

The Linux kernel handles all these types of files by hiding the implementation details of any single file type behind a layer of software, the virtual file system (VFS). Here, we first cover the virtual file system and then discuss the standard Linux file system—ext3.

### 20.7.1 The Virtual File System

The Linux VFS is designed around object-oriented principles. It has two components: a set of definitions that specify what file-system objects are allowed to look like and a layer of software to manipulate the objects. The VFS defines four main object types:

- An **inode object** represents an individual file.
- A **fil object** represents an open file.
- A **superblock object** represents an entire file system.
- A **dentry object** represents an individual directory entry.

For each of these four object types, the VFS defines a set of operations. Every object of one of these types contains a pointer to a function table. The function table lists the addresses of the actual functions that implement the defined operations for that object. For example, an abbreviated API for some of the file object's operations includes:

- `int open(. . .)` — Open a file.
- `ssize_t read(. . .)` — Read from a file.
- `ssize_t write(. . .)` — Write to a file.
- `int mmap(. . .)` — Memory-map a file.

The complete definition of the file object is specified in the `struct file_operations`, which is located in the file `/usr/include/linux/fs.h`. An implementation of the file object (for a specific file type) is required to implement each function specified in the definition of the file object.

The VFS software layer can perform an operation on one of the file-system objects by calling the appropriate function from the object's function table, without having to know in advance exactly what kind of object it is dealing with. The VFS does not know, or care, whether an inode represents a networked file, a disk file, a network socket, or a directory file. The appropriate function for that file's `read()` operation will always be at the same place in its function table, and the VFS software layer will call that function without caring how the data are actually read.

The inode and file objects are the mechanisms used to access files. An inode object is a data structure containing pointers to the disk blocks that contain the actual file contents, and a file object represents a point of access to the data in an open file. A thread cannot access an inode's contents without first obtaining a file object pointing to the inode. The file object keeps track of where in the file the process is currently reading or writing, to keep track of sequential file I/O. It also remembers the permissions (for example, read or write) requested when the file was opened and tracks the thread's activity if necessary to perform adaptive read-ahead, fetching file data into memory before the thread requests the data, to improve performance.

File objects typically belong to a single process, but inode objects do not. There is one file object for every instance of an open file, but always only a

single inode object. Even when a file is no longer in use by any process, its inode object may still be cached by the VFS to improve performance if the file is used again in the near future. All cached file data are linked onto a list in the file's inode object. The inode also maintains standard information about each file, such as the owner, size, and time most recently modified.

Directory files are dealt with slightly differently from other files. The UNIX programming interface defines a number of operations on directories, such as creating, deleting, and renaming a file in a directory. The system calls for these directory operations do not require that the user open the files concerned, unlike the case for reading or writing data. The VFS therefore defines these directory operations in the inode object, rather than in the file object.

The superblock object represents a connected set of files that form a self-contained file system. The operating-system kernel maintains a single superblock object for each disk device mounted as a file system and for each networked file system currently connected. The main responsibility of the superblock object is to provide access to inodes. The VFS identifies every inode by a unique file-system/inode number pair, and it finds the inode corresponding to a particular inode number by asking the superblock object to return the inode with that number.

Finally, a dentry object represents a directory entry, which may include the name of a directory in the path name of a file (such as /usr) or the actual file (such as stdio.h). For example, the file /usr/include/stdio.h contains the directory entries (1) /, (2) usr, (3) include, and (4) stdio.h. Each of these values is represented by a separate dentry object.

As an example of how dentry objects are used, consider the situation in which a thread wishes to open the file with the pathname /usr/include/stdio.h using an editor. Because Linux treats directory names as files, translating this path requires first obtaining the inode for the root—/. The operating system must then read through this file to obtain the inode for the file include. It must continue this thread until it obtains the inode for the file stdio.h. Because path-name translation can be a time-consuming task, Linux maintains a cache of dentry objects, which is consulted during path-name translation. Obtaining the inode from the dentry cache is considerably faster than having to read the on-disk file.

### 20.7.2 The Linux ext3 File System

The standard on-disk file system used by Linux is called **ext3**, for historical reasons. Linux was originally programmed with a Minix-compatible file system, to ease exchanging data with the Minix development system, but that file system was severely restricted by 14-character file-name limits and a maximum file-system size of 64-MB. The Minix file system was superseded by a new file system, which was christened the **extended file system (extfs)**. A later redesign to improve performance and scalability and to add a few missing features led to the **second extended file system (ext2)**. Further development added journaling capabilities, and the system was renamed the **third extended file system (ext3)**. Linux kernel developers then augmented ext3 with modern file-system features such as extents. This new file system is called the **fourth extended file system (ext4)**. The rest of this section discusses ext3, however, since it remains

the most-deployed Linux file system. Most of the discussion applies equally to ext4.

Linux's ext3 has much in common with the BSD Fast File System (FFS) (Section C.7.7). It uses a similar mechanism for locating the data blocks belonging to a specific file, storing data-block pointers in indirect blocks throughout the file system with up to three levels of indirection. As in FFS, directory files are stored on disk just like normal files, although their contents are interpreted differently. Each block in a directory file consists of a linked list of entries. In turn, each entry contains the length of the entry, the name of a file, and the inode number of the inode to which that entry refers.

The main differences between ext3 and FFS lie in their disk-allocation policies. In FFS, the disk is allocated to files in blocks of 8 KB. These blocks are subdivided into fragments of 1 KB for storage of small files or partially filled blocks at the ends of files. In contrast, ext3 does not use fragments at all but performs all its allocations in smaller units. The default block size on ext3 varies as a function of the total size of the file system. Supported block sizes are 1, 2, 4, and 8 KB.

To maintain high performance, the operating system must try to perform I/O operations in large chunks whenever possible by clustering physically adjacent I/O requests. Clustering reduces the per-request overhead incurred by device drivers, disks, and disk-controller hardware. A block-sized I/O request size is too small to maintain good performance, so ext3 uses allocation policies designed to place logically adjacent blocks of a file into physically adjacent blocks on disk, so that it can submit an I/O request for several disk blocks as a single operation.

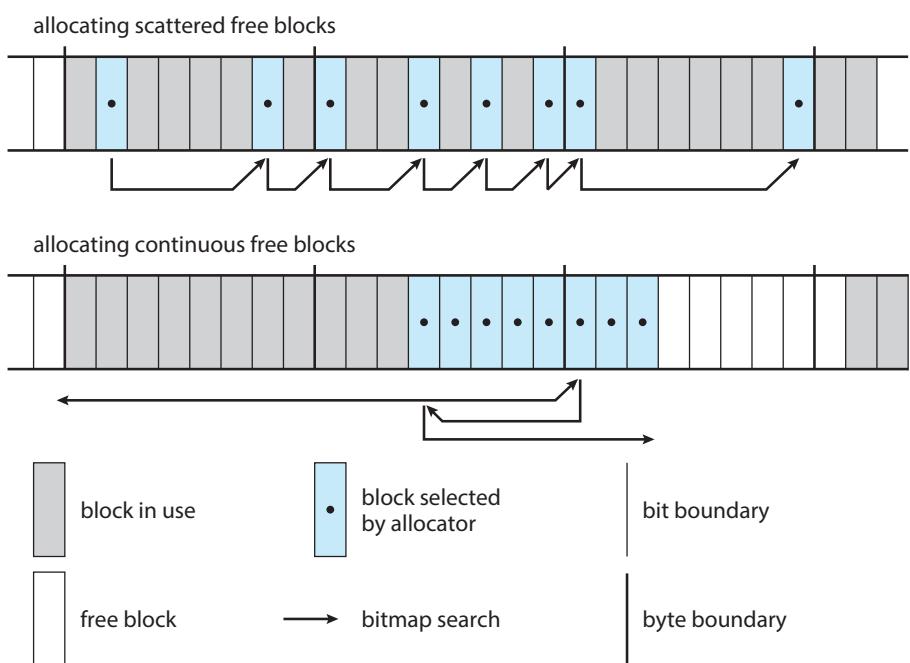
The ext3 allocation policy works as follows: As in FFS, an ext3 file system is partitioned into multiple segments. In ext3, these are called **block groups**. FFS uses the similar concept of **cylinder groups**, where each group corresponds to a single cylinder of a physical disk. (Note that modern disk-drive technology packs sectors onto the disk at different densities, and thus with different cylinder sizes, depending on how far the disk head is from the center of the disk. Therefore, fixed-sized cylinder groups do not necessarily correspond to the disk's geometry.)

When allocating a file, ext3 must first select the block group for that file. For data blocks, it attempts to allocate the file to the block group to which the file's inode has been allocated. For inode allocations, it selects the block group in which the file's parent directory resides for nondirectory files. Directory files are not kept together but rather are dispersed throughout the available block groups. These policies are designed not only to keep related information within the same block group but also to spread out the disk load among the disk's block groups to reduce the fragmentation of any one area of the disk.

Within a block group, ext3 tries to keep allocations physically contiguous if possible, reducing fragmentation if it can. It maintains a bitmap of all free blocks in a block group. When allocating the first blocks for a new file, it starts searching for a free block from the beginning of the block group. When extending a file, it continues the search from the block most recently allocated to the file. The search is performed in two stages. First, ext3 searches for an entire free byte in the bitmap; if it fails to find one, it looks for any free bit. The search for free bytes aims to allocate disk space in chunks of at least eight blocks where possible.

Once a free block has been identified, the search is extended backward until an allocated block is encountered. When a free byte is found in the bitmap, this backward extension prevents ext3 from leaving a hole between the most recently allocated block in the previous nonzero byte and the zero byte found. Once the next block to be allocated has been found by either bit or byte search, ext3 extends the allocation forward for up to eight blocks and preallocates these extra blocks to the file. This preallocation helps to reduce fragmentation during interleaved writes to separate files and also reduces the CPU cost of disk allocation by allocating multiple blocks simultaneously. The preallocated blocks are returned to the free-space bitmap when the file is closed.

Figure 20.7 illustrates the allocation policies. Each row represents a sequence of set and unset bits in an allocation bitmap, indicating used and free blocks on disk. In the first case, if we can find any free blocks sufficiently near the start of the search, then we allocate them no matter how fragmented they may be. The fragmentation is partially compensated for by the fact that the blocks are close together and can probably all be read without any disk seeks. Furthermore, allocating them all to one file is better in the long run than allocating isolated blocks to separate files once large free areas become scarce on disk. In the second case, we have not immediately found a free block close by, so we search forward for an entire free byte in the bitmap. If we allocated that byte as a whole, we would end up creating a fragmented area of free space between it and the allocation preceding it. Thus, before allocating, we back up to make this allocation flush with the allocation preceding it, and then we allocate forward to satisfy the default allocation of eight blocks.



**Figure 20.7** ext3 block-allocation policies.

### 20.7.3 Journaling

The ext3 file system supports a popular feature called **journaling**, whereby modifications to the file system are written sequentially to a journal. A set of operations that performs a specific task is a **transaction**. Once a transaction is written to the journal, it is considered to be committed. Meanwhile, the journal entries relating to the transaction are replayed across the actual file-system structures. As the changes are made, a pointer is updated to indicate which actions have completed and which are still incomplete. When an entire committed transaction is completed, it is removed from the journal. The journal, which is actually a circular buffer, may be in a separate section of the file system, or it may even be on a separate disk spindle. It is more efficient, but more complex, to have it under separate read–write heads, thereby decreasing head contention and seek times.

If the system crashes, some transactions may remain in the journal. Those transactions were never completed to the file system even though they were committed by the operating system, so they must be completed once the system recovers. The transactions can be executed from the pointer until the work is complete, and the file-system structures remain consistent. The only problem occurs when a transaction has been aborted—that is, it was not committed before the system crashed. Any changes from those transactions that were applied to the file system must be undone, again preserving the consistency of the file system. This recovery is all that is needed after a crash, eliminating all problems with consistency checking.

Journaling file systems may perform some operations faster than nonjournaling systems, as updates proceed much faster when they are applied to the in-memory journal rather than directly to the on-disk data structures. The reason for this improvement is found in the performance advantage of sequential I/O over random I/O. Costly synchronous random writes to the file system are turned into much less costly synchronous sequential writes to the file system’s journal. Those changes, in turn, are replayed asynchronously via random writes to the appropriate structures. The overall result is a significant gain in performance of file-system metadata-oriented operations, such as file creation and deletion. Due to this performance improvement, ext3 can be configured to journal only metadata and not file data.

### 20.7.4 The Linux Proc File System

The flexibility of the Linux VFS enables us to implement a file system that does not store data persistently at all but rather provides an interface to some other functionality. The Linux /proc file system is an example of a file system whose contents are not actually stored anywhere but are computed on demand according to user file I/O requests.

A /proc file system is not unique to Linux. UNIX v8 introduced a /proc file system and its use has been adopted and expanded into many other operating systems. It is an efficient interface to the kernel’s process name space and helps with debugging. Each subdirectory of the file system corresponded not to a directory on any disk but rather to an active process on the current system. A listing of the file system reveals one directory per process, with the directory

name being the ASCII decimal representation of the process's unique process identifier (PID).

Linux implements such a /proc file system but extends it greatly by adding a number of extra directories and text files under the file system's root directory. These new entries correspond to various statistics about the kernel and the associated loaded drivers. The /proc file system provides a way for programs to access this information as plain text files; the standard UNIX user environment provides powerful tools to process such files. For example, in the past, the traditional UNIX ps command for listing the states of all running processes has been implemented as a privileged process that reads the process state directly from the kernel's virtual memory. Under Linux, this command is implemented as an entirely unprivileged program that simply parses and formats the information from /proc.

The /proc file system must implement two things: a directory structure and the file contents within. Because a UNIX file system is defined as a set of file and directory inodes identified by their inode numbers, the /proc file system must define a unique and persistent inode number for each directory and the associated files. Once such a mapping exists, the file system can use this inode number to identify just what operation is required when a user tries to read from a particular file inode or to perform a lookup in a particular directory inode. When data are read from one of these files, the /proc file system will collect the appropriate information, format it into textual form, and place it into the requesting process's read buffer.

The mapping from inode number to information type splits the inode number into two fields. In Linux, a PID is 16 bits in size, but an inode number is 32 bits. The top 16 bits of the inode number are interpreted as a PID, and the remaining bits define what type of information is being requested about that process.

A PID of zero is not valid, so a zero PID field in the inode number is taken to mean that this inode contains global—rather than process-specific—information. Separate global files exist in /proc to report information such as the kernel version, free memory, performance statistics, and drivers currently running.

Not all the inode numbers in this range are reserved. The kernel can allocate new /proc inode mappings dynamically, maintaining a bitmap of allocated inode numbers. It also maintains a tree data structure of registered global /proc file-system entries. Each entry contains the file's inode number, file name, and access permissions, along with the special functions used to generate the file's contents. Drivers can register and deregister entries in this tree at any time, and a special section of the tree—appearing under the /proc/sys directory—is reserved for kernel variables. Files under this tree are managed by a set of common handlers that allow both reading and writing of these variables, so a system administrator can tune the value of kernel parameters simply by writing out the new desired values in ASCII decimal to the appropriate file.

To allow efficient access to these variables from within applications, the /proc/sys subtree is made available through a special system call, `sysctl()`, that reads and writes the same variables in binary, rather than in text, without the overhead of the file system. `sysctl()` is not an extra facility; it simply reads the /proc dynamic entry tree to identify the variables to which the application is referring.

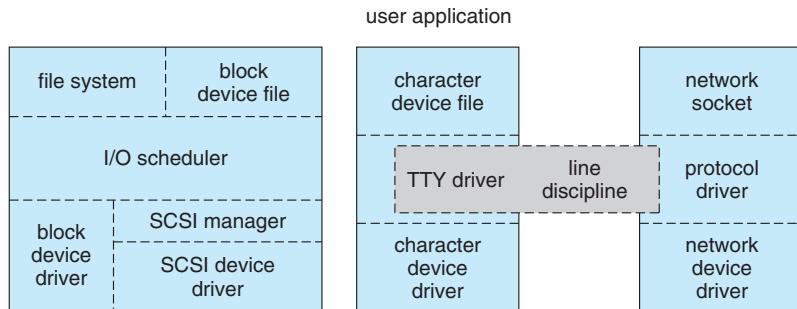


Figure 20.8 Device-driver block structure.

## 20.8 Input and Output

To the user, the I/O system in Linux looks much like that in any UNIX system. That is, to the extent possible, all device drivers appear as normal files. Users can open an access channel to a device in the same way they open any other file—devices can appear as objects within the file system. The system administrator can create special files within a file system that contain references to a specific device driver, and a user opening such a file will be able to read from and write to the device referenced. By using the normal file-protection system, which determines who can access which file, the administrator can set access permissions for each device.

Linux splits all devices into three classes: block devices, character devices, and network devices. Figure 20.8 illustrates the overall structure of the device-driver system.

**Block devices** include all devices that allow random access to completely independent, fixed-sized blocks of data, including hard disks and floppy disks, CD-ROMs and Blu-ray discs, and flash memory. Block devices are typically used to store file systems, but direct access to a block device is also allowed so that programs can create and repair the file system that the device contains. Applications can also access these block devices directly if they wish. For example, a database application may prefer to perform its own fine-tuned layout of data onto a disk rather than using the general-purpose file system.

**Character devices** include most other devices, such as mice and keyboards. The fundamental difference between block and character devices is random access—block devices are accessed randomly, while character devices are accessed serially. For example, seeking to a certain position in a file might be supported for a DVD but makes no sense for a pointing device such as a mouse.

**Network devices** are dealt with differently from block and character devices. Users cannot directly transfer data to network devices. Instead, they must communicate indirectly by opening a connection to the kernel’s networking subsystem. We discuss the interface to network devices separately in Section 20.10.

### 20.8.1 Block Devices

Block devices provide the main interface to all disk devices in a system. Performance is particularly important for disks, and the block-device system must

provide functionality to ensure that disk access is as fast as possible. This functionality is achieved through the scheduling of I/O operations.

In the context of block devices, a block represents the unit with which the kernel performs I/O. When a block is read into memory, it is stored in a buffer. The **request manager** is the layer of software that manages the reading and writing of buffer contents to and from a block-device driver.

A separate list of requests is kept for each block-device driver. Traditionally, these requests have been scheduled according to a unidirectional-elevator (C-SCAN) algorithm that exploits the order in which requests are inserted in and removed from the lists. The request lists are maintained in sorted order of increasing starting-sector number. When a request is accepted for processing by a block-device driver, it is not removed from the list. It is removed only after the I/O is complete, at which point the driver continues with the next request in the list, even if new requests have been inserted in the list before the active request. As new I/O requests are made, the request manager attempts to merge requests in the lists.

Linux kernel version 2.6 introduced a new I/O scheduling algorithm. Although a simple elevator algorithm remains available, the default I/O scheduler is now the **Completely Fair Queueing (CFQ)** scheduler. The CFQ I/O scheduler is fundamentally different from elevator-based algorithms. Instead of sorting requests into a list, CFQ maintains a set of lists—by default, one for each process. Requests originating from a process go in that process's list. For example, if two processes are issuing I/O requests, CFQ will maintain two separate lists of requests, one for each process. The lists are maintained according to the C-SCAN algorithm.

CFQ services the lists differently as well. Where a traditional C-SCAN algorithm is indifferent to a specific process, CFQ services each process's list round-robin. It pulls a configurable number of requests (by default, four) from each list before moving on to the next. This method results in fairness at the process level—each process receives an equal fraction of the disk's bandwidth. The result is beneficial with interactive workloads where I/O latency is important. In practice, however, CFQ performs well with most workloads.

### 20.8.2 Character Devices

A character-device driver can be almost any device driver that does not offer random access to fixed blocks of data. Any character-device drivers registered to the Linux kernel must also register a set of functions that implement the file I/O operations that the driver can handle. The kernel performs almost no preprocessing of a file read or write request to a character device. It simply passes the request to the device in question and lets the device deal with the request.

The main exception to this rule is the special subset of character-device drivers that implement terminal devices. The kernel maintains a standard interface to these drivers by means of a set of `tty_struct` structures. Each of these structures provides buffering and flow control on the data stream from the terminal device and feeds those data to a line discipline.

A **line discipline** is an interpreter for the information from the terminal device. The most common line discipline is the `tty` discipline, which glues the terminal's data stream onto the standard input and output streams of a user's running processes, allowing those processes to communicate directly with the

user's terminal. This job is complicated by the fact that several such processes may be running simultaneously, and the `tty` line discipline is responsible for attaching and detaching the terminal's input and output from the various processes connected to it as those processes are suspended or awakened by the user.

Other line disciplines also are implemented that have nothing to do with I/O to a user process. The PPP and SLIP networking protocols are ways of encoding a networking connection over a terminal device such as a serial line. These protocols are implemented under Linux as drivers that at one end appear to the terminal system as line disciplines and at the other end appear to the networking system as network-device drivers. After one of these line disciplines has been enabled on a terminal device, any data appearing on that terminal will be routed directly to the appropriate network-device driver.

## 20.9 Interprocess Communication

Linux provides a rich environment for processes to communicate with each other. Communication may be just a matter of letting another process know that some event has occurred, or it may involve transferring data from one process to another.

### 20.9.1 Synchronization and Signals

The standard Linux mechanism for informing a process that an event has occurred is the `signal`. Signals can be sent from any process to any other process, with restrictions on signals sent to processes owned by another user. However, a limited number of signals is available, and they cannot carry information. Only the fact that a signal has occurred is available to a process. Signals are not generated only by processes. The kernel also generates signals internally. For example, it can send a signal to a server process when data arrive on a network channel, to a parent process when a child terminates, or to a waiting process when a timer expires.

Internally, the Linux kernel does not use signals to communicate with processes running in kernel mode. If a kernel-mode process is expecting an event to occur, it will not use signals to receive notification of that event. Rather, communication about incoming asynchronous events within the kernel takes place through the use of scheduling states and `wait_queue` structures. These mechanisms allow kernel-mode processes to inform one another about relevant events, and they also allow events to be generated by device drivers or by the networking system. Whenever a process wants to wait for some event to complete, it places itself on a wait queue associated with that event and tells the scheduler that it is no longer eligible for execution. Once the event has completed, every process on the wait queue will be awakened. This procedure allows multiple processes to wait for a single event. For example, if several processes are trying to read a file from a disk, then they will all be awakened once the data have been read into memory successfully.

Although signals have always been the main mechanism for communicating asynchronous events among processes, Linux also implements the semaphore mechanism of System V UNIX. A process can wait on a semaphore as easily as it can wait for a signal, but semaphores have two advantages: large

numbers of semaphores can be shared among multiple independent processes, and operations on multiple semaphores can be performed atomically. Internally, the standard Linux wait queue mechanism synchronizes processes that are communicating with semaphores.

### 20.9.2 Passing of Data among Processes

Linux offers several mechanisms for passing data among processes. The standard UNIX **pipe** mechanism allows a child process to inherit a communication channel from its parent; data written to one end of the pipe can be read at the other. Under Linux, pipes appear as just another type of inode to virtual file system software, and each pipe has a pair of wait queues to synchronize the reader and writer. UNIX also defines a set of networking facilities that can send streams of data to both local and remote processes. Networking is covered in Section 20.10.

Another process communications method, shared memory, offers an extremely fast way to communicate large or small amounts of data. Any data written by one process to a shared memory region can be read immediately by any other process that has mapped that region into its address space. The main disadvantage of shared memory is that, on its own, it offers no synchronization. A process can neither ask the operating system whether a piece of shared memory has been written to nor suspend execution until such a write occurs. Shared memory becomes particularly powerful when used in conjunction with another interprocess-communication mechanism that provides the missing synchronization.

A shared-memory region in Linux is a persistent object that can be created or deleted by processes. Such an object is treated as though it were a small, independent address space. The Linux paging algorithms can elect to page shared-memory pages out to disk, just as they can page out a process's data pages. The shared-memory object acts as a backing store for shared-memory regions, just as a file can act as a backing store for a memory-mapped memory region. When a file is mapped into a virtual address space region, then any page faults that occur cause the appropriate page of the file to be mapped into virtual memory. Similarly, shared-memory mappings direct page faults to map in pages from a persistent shared-memory object. Also just as for files, shared-memory objects remember their contents even if no processes are currently mapping them into virtual memory.

## 20.10 Network Structure

Networking is a key area of functionality for Linux. Not only does Linux support the standard Internet protocols used for most UNIX-to-UNIX communications, but it also implements a number of protocols native to other, non-UNIX operating systems. In particular, since Linux was originally implemented primarily on PCs, rather than on large workstations or on server-class systems, it supports many of the protocols typically used on PC networks, such as AppleTalk and IPX.

Internally, networking in the Linux kernel is implemented by three layers of software:

1. The socket interface
2. Protocol drivers
3. Network-device drivers

User applications perform all networking requests through the socket interface. This interface is designed to look like the 4.3 BSD socket layer, so that any programs designed to make use of Berkeley sockets will run on Linux without any source-code changes. This interface is described in Section C.9.1. The BSD socket interface is sufficiently general to represent network addresses for a wide range of networking protocols. This single interface is used in Linux to access not just those protocols implemented on standard BSD systems but all the protocols supported by the system.

The next layer of software is the protocol stack, which is similar in organization to BSD's own framework. Whenever any networking data arrive at this layer, either from an application's socket or from a network-device driver, the data are expected to have been tagged with an identifier specifying which network protocol they contain. Protocols can communicate with one another if they desire; for example, within the Internet protocol set, separate protocols manage routing, error reporting, and reliable retransmission of lost data.

The protocol layer may rewrite packets, create new packets, split or reassemble packets into fragments, or simply discard incoming data. Ultimately, once the protocol layer has finished processing a set of packets, it passes them on, either upward to the socket interface if the data are destined for a local connection or downward to a device driver if the data need to be transmitted remotely. The protocol layer decides to which socket or device it will send the packet.

All communication between the layers of the networking stack is performed by passing single `skbuff` (socket buffer) structures. Each of these structures contains a set of pointers into a single continuous area of memory, representing a buffer inside which network packets can be constructed. The valid data in a `skbuff` do not need to start at the beginning of the `skbuff`'s buffer, and they do not need to run to the end. The networking code can add data to or trim data from either end of the packet, as long as the result still fits into the `skbuff`. This capacity is especially important on modern microprocessors, where improvements in CPU speed have far outstripped the performance of main memory. The `skbuff` architecture allows flexibility in manipulating packet headers and checksums while avoiding any unnecessary data copying.

The most important set of protocols in the Linux networking system is the TCP/IP protocol suite. This suite comprises a number of separate protocols. The IP protocol implements routing between different hosts anywhere on the network. On top of the routing protocol are the UDP, TCP, and ICMP protocols. The UDP protocol carries arbitrary individual datagrams between hosts. The TCP protocol implements reliable connections between hosts with guaranteed in-order delivery of packets and automatic retransmission of lost data. The ICMP protocol carries various error and status messages between hosts.

Each packet (`skbuff`) arriving at the networking stack's protocol software is expected to be already tagged with an internal identifier indicating the protocol to which the packet is relevant. Different networking-device drivers

encode the protocol type in different ways; thus, the protocol for incoming data must be identified in the device driver. The device driver uses a hash table of known networking-protocol identifiers to look up the appropriate protocol and passes the packet to that protocol. New protocols can be added to the hash table as kernel-loadable modules.

Incoming IP packets are delivered to the IP driver. The job of this layer is to perform routing. After deciding where the packet is to be sent, the IP driver forwards the packet to the appropriate internal protocol driver to be delivered locally or injects it back into a selected network-device-driver queue to be forwarded to another host. It performs the routing decision using two tables: the persistent forwarding information base (FIB) and a cache of recent routing decisions. The FIB holds routing-configuration information and can specify routes based either on a specific destination address or on a wildcard representing multiple destinations. The FIB is organized as a set of hash tables indexed by destination address; the tables representing the most specific routes are always searched first. Successful lookups from this table are added to the route-caching table, which caches routes only by specific destination. No wildcards are stored in the cache, so lookups can be made quickly. An entry in the route cache expires after a fixed period with no hits.

At various stages, the IP software passes packets to a separate section of code for **firewall** management—selective filtering of packets according to arbitrary criteria, usually for security purposes. The firewall manager maintains a number of separate **firewall chains** and allows a `skbuff` to be matched against any chain. Chains are reserved for separate purposes: one is used for forwarded packets, one for packets being input to this host, and one for data generated at this host. Each chain is held as an ordered list of rules, where a rule specifies one of a number of possible firewall-decision functions plus some arbitrary data for matching purposes.

Two other functions performed by the IP driver are disassembly and reassembly of large packets. If an outgoing packet is too large to be queued to a device, it is simply split up into smaller **fragments**, which are all queued to the driver. At the receiving host, these fragments must be reassembled. The IP driver maintains an `ipfrag` object for each fragment awaiting reassembly and an `ipq` for each datagram being assembled. Incoming fragments are matched against each known `ipq`. If a match is found, the fragment is added to it; otherwise, a new `ipq` is created. Once the final fragment has arrived for a `ipq`, a completely new `skbuff` is constructed to hold the new packet, and this packet is passed back into the IP driver.

Packets identified by the IP as destined for this host are passed on to one of the other protocol drivers. The UDP and TCP protocols share a means of associating packets with source and destination sockets: each connected pair of sockets is uniquely identified by its source and destination addresses and by the source and destination port numbers. The socket lists are linked to hash tables keyed on these four address and port values for socket lookup on incoming packets. The TCP protocol has to deal with unreliable connections, so it maintains ordered lists of unacknowledged outgoing packets to retransmit after a timeout and of incoming out-of-order packets to be presented to the socket when the missing data have arrived.

## 20.11 Security

Linux's security model is closely related to typical UNIX security mechanisms. The security concerns can be classified in two groups:

1. **Authentication.** Making sure that nobody can access the system without first proving that she has entry rights
2. **Access control.** Providing a mechanism for checking whether a user has the right to access a certain object and preventing access to objects as required

### 20.11.1 Authentication

Authentication in UNIX has typically been performed through the use of a publicly readable password file. A user's password is combined with a random "salt" value, and the result is encoded with a one-way transformation function and stored in the password file. The use of the one-way function means that the original password cannot be deduced from the password file except by trial and error. When a user presents a password to the system, the password is recombined with the salt value stored in the password file and passed through the same one-way transformation. If the result matches the contents of the password file, then the password is accepted.

Historically, UNIX implementations of this mechanism have had several drawbacks. Passwords were often limited to eight characters, and the number of possible salt values was so low that an attacker could easily combine a dictionary of commonly used passwords with every possible salt value and have a good chance of matching one or more passwords in the password file, gaining unauthorized access to any accounts compromised as a result. Extensions to the password mechanism have been introduced that keep the encrypted password secret in a file that is not publicly readable, that allow longer passwords, or that use more secure methods of encoding the password. Other authentication mechanisms have been introduced that limit the periods during which a user is permitted to connect to the system. Also, mechanisms exist to distribute authentication information to all the related systems in a network.

A new security mechanism has been developed by UNIX vendors to address authentication problems. The **pluggable authentication modules (PAM)** system is based on a shared library that can be used by any system component that needs to authenticate users. An implementation of this system is available under Linux. PAM allows authentication modules to be loaded on demand as specified in a system-wide configuration file. If a new authentication mechanism is added at a later date, it can be added to the configuration file, and all system components will immediately be able to take advantage of it. PAM modules can specify authentication methods, account restrictions, session-setup functions, and password-changing functions (so that, when users change their passwords, all the necessary authentication mechanisms can be updated at once).

### 20.11.2 Access Control

Access control under UNIX systems, including Linux, is performed through the use of unique numeric identifiers. A user identifier (UID) identifies a single user or a single set of access rights. A group identifier (GID) is an extra identifier that can be used to identify rights belonging to more than one user.

Access control is applied to various objects in the system. Every file available in the system is protected by the standard access-control mechanism. In addition, other shared objects, such as shared-memory sections and semaphores, employ the same access system.

Every object in a UNIX system under user and group access control has a single UID and a single GID associated with it. User processes also have a single UID, but they may have more than one GID. If a process's UID matches the UID of an object, then the process has **user rights** or **owner rights** to that object. If the UIDs do not match but any GID of the process matches the object's GID, then **group rights** are conferred; otherwise, the process has **world rights** to the object.

Linux performs access control by assigning objects a **protection mask** that specifies which access modes—read, write, or execute—are to be granted to processes with owner, group, or world access. Thus, the owner of an object might have full read, write, and execute access to a file; other users in a certain group might be given read access but denied write access; and everybody else might be given no access at all.

The only exception is the privileged *root* UID. A process with this special UID is granted automatic access to any object in the system, bypassing normal access checks. Such processes are also granted permission to perform privileged operations, such as reading any physical memory or opening reserved network sockets. This mechanism allows the kernel to prevent normal users from accessing these resources: most of the kernel's key internal resources are implicitly owned by the root UID.

Linux implements the standard UNIX **setuid** mechanism described in Section C.3.2. This mechanism allows a program to run with privileges different from those of the user running the program. For example, the `lpr` program (which submits a job to a print queue) has access to the system's print queues even if the user running that program does not. The UNIX implementation of **setuid** distinguishes between a process's **real** and **effective** UID. The real UID is that of the user running the program; the effective UID is that of the file's owner.

Under Linux, this mechanism is augmented in two ways. First, Linux implements the POSIX specification's **saved user-id** mechanism, which allows a process to drop and reacquire its effective UID repeatedly. For security reasons, a program may want to perform most of its operations in a safe mode, waiving the privileges granted by its **setuid** status; but it may wish to perform selected operations with all its privileges. Standard UNIX implementations achieve this capacity only by swapping the real and effective UIDs. When this is done, the previous effective UID is remembered, but the program's real UID does not always correspond to the UID of the user running the program. Saved UIDs allow a process to set its effective UID to its real UID and then return to

the previous value of its effective UID without having to modify the real UID at any time.

The second enhancement provided by Linux is the addition of a process characteristic that grants just a subset of the rights of the effective UID. The **fsuid** and **fsgid** process properties are used when access rights are granted to files. The appropriate property is set every time the effective UID or GID is set. However, the fsuid and fsgid can be set independently of the effective ids, allowing a process to access files on behalf of another user without taking on the identity of that other user in any other way. Specifically, server processes can use this mechanism to serve files to a certain user without becoming vulnerable to being killed or suspended by that user.

Finally, Linux provides a mechanism for flexible passing of rights from one program to another—a mechanism that has become common in modern versions of UNIX. When a local network socket has been set up between any two processes on the system, either of those processes may send to the other process a file descriptor for one of its open files; the other process receives a duplicate file descriptor for the same file. This mechanism allows a client to pass access to a single file selectively to some server process without granting that process any other privileges. For example, it is no longer necessary for a print server to be able to read all the files of a user who submits a new print job. The print client can simply pass the server file descriptors for any files to be printed, denying the server access to any of the user's other files.

## 20.12 Summary

- Linux is a modern, free operating system based on UNIX standards. It has been designed to run efficiently and reliably on common PC hardware; it also runs on a variety of other platforms, such as mobile phones. It provides a programming interface and user interface compatible with standard UNIX systems and can run a large number of UNIX applications, including an increasing number of commercially supported applications.
- Linux has not evolved in a vacuum. A complete Linux system includes many components that were developed independently of Linux. The core Linux operating-system kernel is entirely original, but it allows much existing free UNIX software to run, resulting in an entire UNIX-compatible operating system free from proprietary code.
- The Linux kernel is implemented as a traditional monolithic kernel for performance reasons, but it is modular enough in design to allow most drivers to be dynamically loaded and unloaded at run time.
- Linux is a multiuser system, providing protection between processes and running multiple processes according to a time-sharing scheduler. Newly created processes can share selective parts of their execution environment with their parent processes, allowing multithreaded programming.
- Interprocess communication is supported by both System V mechanisms—message queues, semaphores, and shared memory—and BSD's socket interface. Multiple networking protocols can be accessed simultaneously through the socket interface.

- The memory-management system uses page sharing and copy-on-write to minimize the duplication of data shared by different processes. Pages are loaded on demand when they are first referenced and are paged back out to backing store according to an LFU algorithm if physical memory needs to be reclaimed.
- To the user, the file system appears as a hierarchical directory tree that obeys UNIX semantics. Internally, Linux uses an abstraction layer to manage multiple file systems. Device-oriented, networked, and virtual file systems are supported. Device-oriented file systems access disk storage through a page cache that is unified with the virtual memory system.

## Practice Exercises

- 20.1 Dynamically loadable kernel modules give flexibility when drivers are added to a system, but do they have disadvantages too? Under what circumstances would a kernel be compiled into a single binary file, and when would it be better to keep it split into modules? Explain your answer.
- 20.2 Multithreading is a commonly used programming technique. Describe three different ways to implement threads, and compare these three methods with the Linux `clone()` mechanism. When might using each alternative mechanism be better or worse than using clones?
- 20.3 The Linux kernel does not allow paging out of kernel memory. What effect does this restriction have on the kernel's design? What are two advantages and two disadvantages of this design decision?
- 20.4 Discuss three advantages of dynamic (shared) linkage of libraries compared with static linkage. Describe two cases in which static linkage is preferable.
- 20.5 Compare the use of networking sockets with the use of shared memory as a mechanism for communicating data between processes on a single computer. What are the advantages of each method? When might each be preferred?
- 20.6 At one time, UNIX systems used disk-layout optimizations based on the rotation position of disk data, but modern implementations, including Linux, simply optimize for sequential data access. Why do they do so? Of what hardware characteristics does sequential access take advantage? Why is rotational optimization no longer so useful?

## Further Reading

The Linux system is a product of the Internet; as a result, much of the available documentation on Linux is available in some form on the Internet. The following key sites reference most of the useful information available:

- The *Linux Cross-Reference Page (LXR)* (<http://lxr.linux.no>) maintains current listings of the Linux kernel, browsable via the web and fully cross-referenced.
- The *Kernel Hackers' Guide* provides a helpful overview of the Linux kernel components and internals and is located at <http://tldp.org/LDP/tlk/tlk.html>.
- The *Linux Weekly News (LWN)* (<http://lwn.net>) provides weekly Linux-related news, including a very well researched subsection on Linux kernel news.

Many mailing lists devoted to Linux are also available. The most important are maintained by a mailing-list manager that can be reached at the e-mail address [majordomo@vger.rutgers.edu](mailto:majordomo@vger.rutgers.edu). Send e-mail to this address with the single line “help” in the mail’s body for information on how to access the list server and to subscribe to any lists.

Finally, the Linux system itself can be obtained over the Internet. Complete Linux distributions are available from the home sites of the companies concerned, and the Linux community also maintains archives of current system components at several places on the Internet. The most important is <ftp://ftp.kernel.org/pub/linux>.

In addition to investigating Internet resources, you can read about the internals of the Linux kernel in [Mauerer (2008)] and [Love (2010)].

The /proc file system was introduced in <http://lucasvr.gobolinux.org/etc/Killian84-Procfs-USENIX.pdf>, and expanded in [http://https://www.usenix.org/sites/default/files/usenix\\_winter91\\_faulkner.pdf](http://https://www.usenix.org/sites/default/files/usenix_winter91_faulkner.pdf).

## Bibliography

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

[Mauerer (2008)] W. Mauerer, *Professional Linux Kernel Architecture*, John Wiley and Sons (2008).

## Chapter 20 Exercises

- 20.7 What are the advantages and disadvantages of writing an operating system in a high-level language, such as C?
- 20.8 In what circumstances is the system-call sequence `fork()` `exec()` most appropriate? When is `vfork()` preferable?
- 20.9 What socket type should be used to implement an intercomputer file-transfer program? What type should be used for a program that periodically tests to see whether another computer is up on the network? Explain your answer.
- 20.10 Linux runs on a variety of hardware platforms. What steps must Linux developers take to ensure that the system is portable to different processors and memory-management architectures and to minimize the amount of architecture-specific kernel code?
- 20.11 What are the advantages and disadvantages of making only some of the symbols defined inside a kernel accessible to a loadable kernel module?
- 20.12 What are the primary goals of the conflict-resolution mechanism used by the Linux kernel for loading kernel modules?
- 20.13 Discuss how the `clone()` operation supported by Linux is used to support both processes and threads.
- 20.14 Would you classify Linux threads as user-level threads or as kernel-level threads? Support your answer with the appropriate arguments.
- 20.15 What extra costs are incurred in the creation and scheduling of a process, compared with the cost of a cloned thread?
- 20.16 How does Linux's Completely Fair Scheduler (CFS) provide improved fairness over a traditional UNIX process scheduler? When is the fairness guaranteed?
- 20.17 What are the two configurable variables of the Completely Fair Scheduler (CFS)? What are the pros and cons of setting each of them to very small and very large values?
- 20.18 The Linux scheduler implements "soft" real-time scheduling. What features necessary for certain real-time programming tasks are missing? How might they be added to the kernel? What are the costs (downsides) of such features?
- 20.19 Under what circumstances would a user process request an operation that results in the allocation of a demand-zero memory region?
- 20.20 What scenarios would cause a page of memory to be mapped into a user program's address space with the copy-on-write attribute enabled?
- 20.21 In Linux, shared libraries perform many operations central to the operating system. What is the advantage of keeping this functionality out of the kernel? Are there any drawbacks? Explain your answer.

- 20.22** What are the benefits of a journaling file system such as Linux's ext3? What are the costs? Why does ext3 provide the option to journal only metadata?
- 20.23** The directory structure of a Linux operating system could include files corresponding to several different file systems, including the Linux /proc file system. How might the need to support different file-system types affect the structure of the Linux kernel?
- 20.24** In what ways does the Linux setuid feature differ from the setuid feature SVR4?
- 20.25** The Linux source code is freely and widely available over the Internet and from CD-ROM vendors. What are three implications of this availability for the security of the Linux system?

# Windows 10



## Updated by Alex Ionescu

The Microsoft Windows 10 operating system is a preemptive multitasking client operating system for microprocessors implementing the Intel IA-32, AMD64, ARM, and ARM64 instruction set architectures (ISAs). Microsoft's corresponding server operating system, Windows Server 2016, is based on the same code as Windows 10 but supports only the 64-bit AMD64 ISAs. Windows 10 is the latest in a series of Microsoft operating systems based on its NT code, which replaced the earlier systems based on Windows 95/98. In this chapter, we discuss the key goals of Windows 10, the layered architecture of the system that has made it so easy to use, the file system, the networking features, and the programming interface.

### CHAPTER OBJECTIVES

- Explore the principles underlying Windows 10's design and the specific components of the system.
- Provide a detailed discussion of the Windows 10 file system.
- Illustrate the networking protocols supported in Windows 10.
- Describe the interface available in Windows 10 to system and application programmers.
- Describe the important algorithms implemented with Windows 10.

## 21.1 History

In the mid-1980s, Microsoft and IBM cooperated to develop the [OS/2 operating system](#), which was written in assembly language for single-processor Intel 80286 systems. In 1988, Microsoft decided to end the joint effort with IBM and develop its own “new technology” (or NT) portable operating system to

support both the OS/2 and POSIX application programming interfaces (APIs). In October 1988, Dave Cutler, the architect of the DEC VAX/VMS operating system, was hired and given the charter of building Microsoft's new operating system.

Originally, the team planned to use the OS/2 API as NT's native environment, but during development, NT was changed to use a new 32-bit Windows API (called Win32), based on the popular 16-bit API used in Windows 3.0. The first versions of NT were Windows NT 3.1 and Windows NT 3.1 Advanced Server. (At that time, 16-bit Windows was at Version 3.1.) Windows NT Version 4.0 adopted the Windows 95 user interface and incorporated Internet web-server and web-browser software. In addition, user-interface routines and all graphics code were moved into the kernel to improve performance (with the side effect of decreased system reliability and significant loss of security). Although previous versions of NT had been ported to other microprocessor architectures (including a brief 64-bit port to Alpha AXP 64), the Windows 2000 version, released in February 2000, supported only IA-32-compatible processors due to marketplace factors. Windows 2000 incorporated significant changes. It added Active Directory (an X.500-based directory service), better networking and laptop support, support for plug-and-play devices, a distributed file system, and support for more processors and more memory.

### 21.1.1 Windows XP, Vista, and 7

In October 2001, Windows XP was released as both an update to the Windows 2000 desktop operating system and a replacement for Windows 95/98. In April 2003, the server edition of Windows XP (called Windows Server 2003) became available. Windows XP updated the graphical user interface (GUI) with a visual design that took advantage of more recent hardware advances and many new *ease-of-use features*. Numerous features were added to automatically repair problems in applications and the operating system itself. Because of these changes, Windows XP provided better networking and device experience (including zero-configuration wireless, instant messaging, streaming media, and digital photography/video). Windows Server 2003 provided dramatic performance improvements for large multiprocessor systems, as well as better reliability and security than earlier Windows operating systems.

The long-awaited update to Windows XP, called Windows Vista, was released in January 2007, but it was not well received. Although Windows Vista included many improvements that later continued into Windows 7, these improvements were overshadowed by Windows Vista's perceived sluggishness and compatibility problems. Microsoft responded to criticisms of Windows Vista by improving its engineering processes and working more closely with the makers of Windows hardware and applications.

The result was Windows 7, which was released in October 2009, along with corresponding server edition called Windows Server 2008 R2. Among the significant engineering changes was the increased use of **event tracing** rather than counters or profiling to analyze system behavior. Tracing runs constantly in the system, watching hundreds of scenarios execute. Scenarios include process startup and exit, file copy, and web-page load, for example. When one of these scenarios fails, or when it succeeds but does not perform well, the traces can be analyzed to determine the cause.

### 21.1.2 Windows 8

Three years later, in October 2012—amid an industry-wide pivot toward mobile computing and the world of [apps](#)—Microsoft released Windows 8, which represented the most significant change to the operating system since Windows XP. Windows 8 included a new user interface (named [Metro](#)) and a new programming model API (named [WinRT](#)). It also included a new way of managing applications (which ran under a new sandbox mechanism) through a [package system](#) that exclusively supported the new [Windows Store](#), a competitor to the Apple App Store and the Android Store. Additionally, Windows 8 included a plethora of security, boot, and performance improvements. At the same time, support for “subsystems,” a concept we’ll describe further later in the chapter, was removed.

To support the new mobile world, Windows 8 was ported to the 32-bit ARM ISA for the first time and included multiple changes to the power management and hardware extensibility features of the kernel (discussed later in this chapter). Microsoft marketed two versions of this port. One version, called Windows RT, ran both Windows Store–packaged applications and some Microsoft-branded “classic” applications, such as Notepad, Internet Explorer, and most importantly, Office. The other version, called Windows Phone, could only run Windows Store–packaged applications.

For the first time ever, Microsoft released its own branded mobile hardware, under the “Surface” brand, which included the Surface RT, a tablet device that exclusively ran the Windows RT operating system. A bit later, Microsoft bought Nokia and began releasing Microsoft-branded phones as well, running Windows Phone.

Unfortunately, Windows 8 was a market failure, for several reasons. On the one hand, Metro focused on a tablet-oriented interface that forced users accustomed to older Windows operating systems to completely change the way they worked on their desktop computers. Windows 8, for example, replaced the start menu with touchscreen features, replaced shortcuts with animated “tiles,” and offered little or no keyboard input support. On the other hand, the dearth of applications in the Windows Store, which was the only way to obtain apps for Microsoft’s phone and tablet, led to the market failure of these devices as well, causing the company to eventually phase out the Surface RT device and write off the Nokia purchase.

Microsoft quickly sought to address many of these issues with the release of Windows 8.1 in October 2013. This release addressed many of the usability flaws of Windows 8 on nonmobile devices, bringing back more usability through a traditional keyboard and mouse, and provided ways to avoid the tile-based Metro interface. It also continued to improve on the many security, performance, and reliability changes introduced in Windows 8. Although this release was better received, the continued lack of applications in the Windows Store was a problem for the operating system’s mobile market penetration, while desktop and server application programmers felt abandoned due to a lack of improvements in their area.

### 21.1.3 Windows 10

With the release of [Windows 10](#) in July 2015 and its server companion, Windows Server 2016, in October 2016, Microsoft shifted to a “Windows-

as-a-Service” (WaaS) model (with included periodic functionality improvements). Windows 10 receives monthly incremental improvements called “feature rollups,” as well as eight-month feature releases called “updates.” Additionally, each upcoming release is made available to the public through the Windows Insider Program, or WIP, which releases versions on an almost weekly basis. Like cloud services and websites such as Facebook and Google, the new operating system uses live telemetry (sending debug information back to Microsoft) and tracing to dynamically enable and disable certain features for A/B testing (comparing how version “A” executes compared to similar version “B”), tries out new features while watching for compatibility issues, and aggressively adds or removes support for modern or legacy hardware. These dynamic configuration and testing features are what make this release an “as-a-service” implementation.

Windows 10 reintroduced the start menu, restored keyboard support, and deemphasized full-screen applications and live tiles. From the user’s perspective, these changes brought back the ease of use that users expected from Windows-based desktop operating systems. Additionally, Metro (which was renamed **Modern**) was redesigned so that Windows Store–packaged applications could be run on the regular desktop side by side with legacy applications. Finally, a new mechanism called the **Windows Desktop Bridge** made it possible to place Win32 applications in the Windows Store, mitigating the lack of applications written specifically for the newer systems. Meanwhile, Microsoft added support for C++11, C++14, and C++17 in the Visual Studio product, and many new APIs were added to the traditional Win32 programming API. A related change in Windows 10 was the release of the Unified Windows Platform (UWP) architecture, which allows applications to be written in such a way that they can execute on Windows for Desktop, Windows for IoT, XBOX One, Windows Phone, and Windows 10 Mixed Reality (previously known as Windows Holographic).

Windows 10 also replaced the concept of multiple subsystems, which had been removed in Windows 8 (as mentioned earlier), with a new mechanism called **Pico Providers**. This mechanism allows unmodified binaries belonging to a different operating system to run natively on Windows 10. In the “Anniversary Update” released in August 2016, this functionality was used to provide the Windows Subsystem for Linux, which can be used to run Linux ELF binaries in an entirely unmodified Ubuntu user-space environment.

In response to increased competitive pressures in the mobile and cloud-computing worlds, Microsoft also made power, performance, and scalability improvements in Windows 10, enabling it to run on a larger number of devices. In fact, a version called Windows 10 IoT Edition is specifically designed for environments such as the Raspberry Pi, while support for cloud-computing technologies such as containerization is built in through Docker for Windows. In Windows 10, the Microsoft Hyper-V virtualization technology is also built in, providing additional security and native support for running virtual machines. A special version of Windows Server, called Windows Server Nano, was also released. This extremely low-overhead server operating system is suited for containerized applications and other cloud-computing usages.

Windows 10 is a multiuser operating system, supporting simultaneous access through distributed services or through multiple instances of the GUI

via Windows Terminal Services. The server editions of Windows 10 support simultaneous terminal server sessions from Windows desktop systems. The desktop editions of terminal server multiplex the keyboard, mouse, and monitor between virtual terminal sessions for each logged-on user. This feature, called *fast user switching*, allows users to preempt each other at the console of a PC without having to log off and log on.

Let's return briefly to developments in the Windows GUI. We noted earlier that the GUI implementation moved into kernel mode in Windows NT 4.0 to improve performance. Further performance gains were made with the creation of a new user-mode component in Windows Vista, called the **Desktop Window Manager (DWM)**. DWM provides the Windows interface look and feel on top of the Windows DirectX graphic software. DirectX continues to run in the kernel, as does the code (Win32k) implementing Windows' windowing and graphics model (User and GDI). Windows 7 made substantial changes to the DWM, significantly reducing its memory footprint and improving its performance, while Windows 10 made further improvements, especially in the areas of performance and security. Furthermore, Windows DirectX 11 and 12 include GPGPU mechanisms (general-purpose computing on GPU hardware) through **DirectCompute**, and many parts of Windows have been updated to take advantage of this high-performance graphics model. Through a new rendering layer called **CoreUI**, even legacy applications can now take advantage of DirectX-based rendering (creation of the final screen contents).

Windows XP was the first version of Windows to ship a 64-bit version (for the IA64 in 2003 and the AMD64 in 2005). Internally, the native NT file system (NTFS) and many of the Win32 APIs have always used 64-bit integers where appropriate. The major extension to 64-bit in Windows XP was meant as support for large virtual addresses. In addition, 64-bit editions of Windows support much larger physical memory, with the latest Windows Server 2016 release supporting up to 24 TB of RAM. By the time Windows 7 shipped, the AMD64 ISA had become available on almost all CPUs from both Intel and AMD. In addition, by that time, physical memory on client systems frequently exceeded the 4-GB limit of the IA-32. As a result, the 64-bit version of Windows 10 is now almost exclusively installed on client systems, apart from IoT and mobile systems. Because the AMD64 architecture supports high-fidelity IA-32 compatibility at the level of individual processes, 32- and 64-bit applications can be freely mixed in a single system. Interestingly, a similar pattern is now emerging on mobile systems. Apple iOS is the first mobile operating system to support the ARM64 architecture, which is the 64-bit ISA extension of ARM (also called AArch64). A future Windows 10 release will also officially ship with an ARM64 port designed for a new class of hardware, with compatibility for IA-32 architecture applications achieved through emulation and dynamic JIT recompilation.

In the rest of our description of Windows 10, we do not distinguish between the client editions and the corresponding server editions. They are based on the same core components and run the same binary files for the kernel and most drivers. Similarly, although Microsoft ships a variety of different editions of each release to address different market price points, few of the differences between editions are reflected in the core of the system. In this chapter, we focus primarily on the core components of Windows 10.

## 21.2 Design Principles

Microsoft's design goals for Windows included security, reliability, compatibility, high performance, extensibility, portability, and international support. Some additional goals, such as energy efficiency and dynamic device support, have recently been added to this list. Next, we discuss each of these goals and how each is achieved in Windows 10.

### 21.2.1 Security

Windows Vista and later security goals required more than just adherence to the design standards that had enabled Windows NT 4.0 to receive a C2 security classification from the U.S. government. (A C2 classification signifies a moderate level of protection from defective software and malicious attacks. Classifications were defined by the Department of Defense Trusted Computer System Evaluation Criteria, also known as the [Orange Book](#).) Extensive code review and testing were combined with sophisticated automatic analysis tools to identify and investigate potential defects that might represent security vulnerabilities. Additionally, *bug bounty* participation programs allow external researchers and security professionals to identify, and submit, previously unknown security issues in Windows. In exchange, they receive monetary payment as well as credit in monthly security rollups, which are released by Microsoft to keep Windows 10 as secure as possible.

Windows traditionally based security on discretionary access controls. System objects, including files, registry keys, and kernel synchronization objects, are protected by [access-control lists \(ACLs\)](#) (see Section 13.4.2). ACLs are vulnerable to user and programmer errors, however, as well as to the most common attacks on consumer systems, in which the user is tricked into running code, often while browsing the Web. Windows Vista introduced a mechanism called [integrity levels](#) that acts as a rudimentary [capability](#) system for controlling access. Objects and processes are marked as having no, low, medium, or high system integrity. The integrity level determines what rights the objects and processes will have. For example, Windows does not allow a process to modify an object with a higher integrity level (based on its *mandatory policy*), no matter what the setting of the ACL. Additionally, a process cannot read the memory of a higher-integrity process, no matter the ACL.

Windows 10 further strengthened the security model by introducing a combination of attribute-based access control (ABAC) and claim-based access control (CABC). Both features are used to implement dynamic access control (DAC) on server editions, as well as to support the capability-based system used by Windows Store applications and by Modern and packaged applications. With attributes and claims, system administrators need not rely on a user's name (or the group the user belongs to) as the only means that the security system can use to filter access to objects such as files. Properties of the user—such as, say, seniority in the organization, salary, and so on—can also be considered. These properties are encoded as *attributes*, which are paired with conditional access control entries in the ACL, such as “Seniority >= 10 Years.”

Windows uses encryption as part of common protocols such as those used to communicate securely with websites. Encryption is also used to protect user files stored on secondary storage. Windows 7 and later versions allow users to

easily encrypt entire volumes, as well as removable storage devices such as USB flash drives, with a feature called BitLocker. If a computer with an encrypted volume is stolen, the thieves will need very sophisticated technology (such as an electron microscope) to gain access to any of the computer's files, and it will be impossible for them to do so if the user has also configured an external USB-based token (unless the USB token was also stolen).

These types of security features focus on user and data security, but they are vulnerable to highly privileged programs that parse arbitrary content and that can be tricked due to programming errors into executing malicious code. Therefore, Windows also includes security measures often referred to as "exploit mitigations." These measures include wide-scope mitigations such as **address-space layout randomization (ASLR)**, **Data Execution Prevention (DEP)**, **Control-Flow Guard (CFG)**, and **Arbitrary Code Guard (ACG)**, as well as narrow-scope (targeted) mitigations specific to various exploitation techniques (which are outside the scope of this chapter).

Since 2001, chips from both Intel and AMD have allowed memory pages to be marked so that they cannot contain executable instruction code. The Windows DEP feature marks stacks and memory heaps (as well as all other data-only allocations) so that they cannot be used to execute code. This prevents attacks in which a program bug allows a buffer to overflow and then is tricked into executing the contents of the buffer. Additionally, starting with Windows 8.1, all kernel data-only memory allocations have been marked similarly.

Because DEP prevents attacker-controlled data from being executed as code, malicious developers moved on to **code reuse** attacks, in which existing executable code inside the program is reused in unexpected ways. (Only certain parts of the code are executed, and the flow is redirected from one instruction stream to another.) ASLR thwarts many forms of such attacks by randomizing the location of executable (and data) regions of memory, making it harder for code-reuse attacks to know where existing code is located. This safeguard makes it likely that a system under attack by a remote attacker will fail or crash.

No mitigation is perfect, however, and ASLR is no exception. For example, it may be ineffective against local attacks (in which some application is tricked into loading content from secondary storage, for example), as well as so-called **information leak** attacks (in which a program is tricked into revealing part of its address space). To address such problems, Windows 8.1 introduced a technology called CFG, which was much improved in Windows 10. CFG works with the compiler, the linker, the loader, and the memory manager to validate the destination address of any indirect branch (such as a call or jump) against a list of valid function prologues. If a program is tricked into redirecting control flow elsewhere through such an instruction, it crashes.

If attackers cannot bring executable data into an attack, nor reuse existing code, they may attempt to cause a program to allocate, on its own, executable and writeable code, which can then be filled by the attacker. Alternatively, the attackers might modify existing writeable data and mark it as executable data. Windows 10's ACG mitigation prohibits either of these operations. Once executable code is loaded, it can never be modified again, and once data is loaded, it can never be marked as executable.

Windows 10 has over thirty security mitigations in addition to those described here. This set of security features has made traditional attacks more

difficult, perhaps explaining in part why crimeware applications, such as adware, credit card fraudware, and ransomware, have become so prevalent. These types of attacks rely on users to willingly and manually cause harm to their own computers (such as by double-clicking on applications against warning, or inputting their credit card number in a fake banking page). No operating system can be designed to militate against the gullibility and curiosity of human beings. Recently, Microsoft has started working directly with chip manufacturers, such as Intel, to build security mitigations directly into the ISA. One such mitigation, for example, is **Control-flow Enforcement Technology (CET)**, which is a hardware implementation of CFG that also protects against return-oriented-programming (ROP) attacks by using hardware shadow stacks. A shadow stack contains the set of return addresses as stored when a routine is called. The addresses are checked for a mismatch before the return is executed. A mismatch means the stack has been compromised and action should be taken.

Another important aspect of security is integrity. Windows offers several **digital signature** facilities as part of its code integrity features. Windows uses digital signatures to *sign* operating system binaries so that it can verify that the files were produced by Microsoft or another known company. In non-IA-32 versions of Windows, the **code integrity** module is activated at boot to ensure that all the loaded modules in the kernel have valid signatures, assuring that they have not been tampered with. Additionally, ARM versions of Windows 8 extend the code integrity module with user-mode code integrity checks, which validate that all user programs have been signed by Microsoft or delivered through the Windows Store. A special version of Windows 10 (Windows 10 S, mostly meant for the education market) provides similar signing checks on all IA-32 and AMD64 systems. Digital signatures are also used as part of Code Integrity Guard, which allows applications to defend themselves against loading executable code from secondary storage that has not been appropriately signed. For example, an attacker might replace third-party binary with his own, but the digital signature would fail, and Code Integrity Guard would not load the binary into the processes' address space.

Finally, enterprise versions of Windows 10 make it possible to opt in to a new security feature called **Device Guard**. This mechanism allows organizations to customize the digital signing requirements of their computer systems, as well as blacklist and whitelist individual signing certificates or even binary hashes. For example, an organization could choose to allow only user-mode programs signed by Microsoft, Google, or Adobe to launch on their enterprise computers.

### 21.2.2 Reliability

Windows matured greatly as an operating system in its first ten years, leading to Windows 2000. At the same time, its reliability increased due to such factors as maturity in the source code, extensive stress testing of the system, improved CPU architectures, and automatic detection of many serious errors in drivers from both Microsoft and third parties. Windows has subsequently extended the tools for achieving reliability to include automatic analysis of source code for errors, tests to detect validation failures, and an application version of the

driver verifier that applies dynamic checking for many common user-mode programming errors. Other improvements in reliability have resulted from moving more code out of the kernel and into user-mode services. Windows provides extensive support for writing drivers in user mode. System facilities that were once in the kernel and are now in user mode include the renderer for third-party fonts and much of the software stack for audio.

One of the most significant improvements in the Windows experience came from adding memory diagnostics as an option at boot time. This addition is especially valuable because so few consumer PCs have error-correcting memory. Bad RAM that lacks error correction and detection can change the data it stores—a change undetected by the hardware. The result is frustratingly erratic behavior in the system. The availability of memory diagnostics can warn users of a RAM problem. Windows 10 took this even further by introducing run-time memory diagnostics. If a machine encounters a kernel-mode crash more than five times in a row, and the crashes cannot be pinpointed to a specific cause or component, the kernel will use idle periods to move memory contents, flush system caches, and write repeated memory-testing patterns in all memory—all to preemptively discover if RAM is damaged. Users can then be informed of any issues without the need to reboot into the memory diagnostics tool at boot time.

Windows 7 also introduced a fault-tolerant memory heap. The heap learns from application crashes and automatically adjusts memory operations carried out by an application that has crashed. This makes the application more reliable even if it contains common bugs such as using memory after freeing it or accessing past the end of the allocation. Because such bugs can be exploited by attackers, Windows 7 also includes a mitigation for developers to block this feature and immediately crash any application with heap corruption. This is a very practical representation of the dichotomy that exists between the needs of security and the needs of user experience.

Achieving high reliability in Windows is particularly challenging because almost two billion systems run Windows. Even reliability problems that affect only a small percentage of these systems still impact tremendous numbers of users. The complexity of the Windows ecosystem also adds to the challenges. Millions of instances of applications, drivers, and other software are constantly being downloaded and run on Windows systems. Of course, there is also a constant stream of malware attacks. As Windows itself has become harder to attack directly, exploits increasingly target popular applications.

To cope with these challenges, Microsoft is increasingly relying on communications from customer machines to collect data from the ecosystem. Machines are sampled to see how they are performing, what software they are running, and what problems they are encountering. They automatically send data to Microsoft when their software, their drivers, or the kernel itself crashes or hangs. Features are measured to indicate how often they are used. Legacy behavior (methods no longer recommended for use by Microsoft) is sometimes disabled, and alerts are sent if attempts are made to use it again. The result is that Microsoft is building an ever-improving picture of what is happening in the Windows ecosystem that allows continuous improvements through software updates as well as providing data to guide future releases of Windows.

### 21.2.3 Windows and Application Compatibility

As mentioned, Windows XP was both an update of Windows 2000 and a replacement for Windows 95/98. Windows 2000 focused primarily on compatibility for business applications. The requirements for Windows XP included much higher compatibility with the consumer applications that ran on Windows 95/98. Application compatibility is difficult to achieve, for several reasons. For example, applications may check for a specific version of Windows, may depend to some extent on the quirks of the implementation of APIs, or may have latent application bugs that were masked in the previous system. Applications may also have been compiled for a different instruction set or have different expectations when run on today's multi-gigahertz, multicore systems. Windows 10 continues to focus on compatibility issues by implementing several strategies to run applications despite incompatibilities.

Like Windows XP, Windows 10 has a compatibility layer, called the shim engine, that sits between applications and the Win32 APIs. This engine can make Windows 10 look (almost) bug-for-bug compatible with previous versions of Windows. Windows 10 ships with a shim database of over 6,500 entries, describing particular quirks and tweaks that must be made for older applications. Furthermore, through the Application Compatibility Toolkit, users and administrators can build their own shim databases. Windows 10's **SwitchBranch** mechanism allows developers to choose which Windows version they'd like the Win32 API to emulate, including all the quirks and/or bugs of a previous API. The Task Manager's "Operating System Context" column shows what SwitchBranch operating-system version each application is running under.

Windows 10, like earlier NT releases, maintains support for running many 16-bit applications using a *thunking*, or conversion, layer—called Windows-on-Windows-32 (WoW32)—that translates 16-bit API calls into equivalent 32-bit calls. Similarly, the 64-bit version of Windows 10 provides a thunking layer, WoW64, that translates 32-bit API calls into native 64-bit calls. Finally, the ARM64 version of Windows 10 provides a dynamic JIT recompiler, translating IA-32 code, called WoWA64.

The original Windows subsystem model allows multiple operating-system personalities to be supported, as long as the applications are rebuilt as Portable Executable (PE) applications with a Microsoft compiler such as Visual Studio and source code is available. As noted earlier, although the API designed for Windows is the Win32API, some earlier editions of Windows supported a POSIX subsystem. POSIX is a standard specification for UNIX that allows UNIX-compatible software to be recompiled and run without modification on any POSIX-compatible operating system. Unfortunately, as Linux has matured, it has drifted farther and farther away from POSIX compatibility, and many modern Linux applications now rely on Linux-specific system calls and improvements to *glibc* that are not standardized. Additionally, it becomes impractical to ask users (or even enterprises) to recompile with Visual Studio every single Linux application that they'd like to use. Indeed, compiler differences among GCC, CLang, and Microsoft's C/C++ compiler often make doing so impossible. Therefore, even though the subsystem model still exists at an architectural level, the only subsystem on Windows going forward will be the Win32 subsystem itself, and compatibility with other operating systems is achieved through a new model that uses Pico Providers instead.

This significantly more powerful model extends the kernel via the ability to forward, or proxy, every system call, exception, fault, thread creation and termination, and process creation, along with a few other internal operations, to a secondary external driver (the Pico Provider itself). This secondary driver now becomes the owner of all such operations. While still using Windows 10's scheduler and memory manager (similar to a microkernel), it can implement its own ABI, system-call interface, executable file format parser, page fault handling, caching, I/O model, security model, and more.

Windows 10 includes one such Pico Provider, called LxCore, that is a multi-megabyte reimplementation of the Linux kernel. (Note that it is not Linux, and it does not share any code with Linux.) This driver is used by the “Windows Subsystem for Linux” feature, which can be used to load unmodified Linux ELF binaries without the need for source code or recompilation as PE binaries. Windows 10 users can run an unmodified Ubuntu user-mode file system (and, more recently, OpenSUSE and CentOS), servicing it with the apt-get package management command and running packages as normal. Note that the kernel reimplementation is not complete—many system calls are missing, as is access to most devices, since no Linux kernel drivers can load. Notably, while networking is fully supported, as well as serial devices, no GUI/frame-buffer access is possible.

As a final compatibility measure, Windows 8.1 and later versions also include the **Hyper-V for Client** feature. This allows applications to get bug-for-bug compatibility with Windows XP, Linux, and even DOS by running these operating systems inside a virtual machine.

#### 21.2.4 Performance

Windows was designed to provide high performance on desktop systems (which are largely constrained by I/O performance), server systems (where the CPU is often the bottleneck), and large multithreaded and multiprocessor environments (where locking performance and cache-line management are keys to scalability). To satisfy performance requirements, NT used a variety of techniques, such as asynchronous I/O, optimized protocols for networks, kernel-based graphics rendering, and sophisticated caching of file-system data. The memory-management and synchronization algorithms were designed with an awareness of the performance considerations related to cache lines and multiprocessors.

Windows NT was designed for symmetrical multiprocessing (SMP); on a multiprocessor computer, several threads can run at the same time, even in the kernel. On each CPU, Windows NT uses priority-based preemptive scheduling of threads. Except while executing in the dispatcher or at interrupt level, threads in any process running in Windows can be preempted by higher-priority threads. Thus, the system responds quickly (see Chapter 5).

Windows XP further improved performance by reducing the code-path length in critical functions and implementing more scalable locking protocols, such as queued spinlocks and pushlocks. (**Pushlocks** are like optimized spinlocks with read–write lock features.) The new locking protocols helped reduce system bus cycles and included lock-free lists and queues, atomic read–modify–write operations (like `interlocked increment`), and other advanced synchronization techniques. These changes were needed because Windows XP

added support for simultaneous multithreading (SMT), as well as a massively parallel pipelining technology that Intel had commercialized under the marketing name **Hyper Threading**. Because of this new technology, average home machines could appear to have two processors. A few years later, the introduction of multicore systems made multiprocessor systems the norm.

Next, Windows Server 2003, targeted toward large multiprocessor servers, was released, using even better algorithms and making a shift toward per-processor data structures, locks, and caches, as well as using page coloring and supporting NUMA machines. (Page coloring is a performance optimization to ensure that accesses to contiguous pages in virtual memory optimize use of the processor cache.) Windows XP 64-bit Edition was based on the Windows Server 2003 kernel so that early 64-bit adopters could take advantage of these improvements.

By the time Windows 7 was developed, several major changes had come to computing. The number of CPUs and the amount of physical memory available in the largest multiprocessors had increased substantially, so quite a lot of effort was put into further improving operating-system scalability.

The implementation of multiprocessing support in Windows NT used bitmasks to represent collections of processors and to identify, for example, which set of processors a particular thread could be scheduled on. These bitmasks were defined as fitting within a single word of memory, limiting the number of processors supported within a system to 64 on a 64-bit system and 32 on a 32-bit system. Thus, Windows 7 added the concept of **processor groups** to represent a collection of up to 64 processors. Multiple processor groups could be created, accommodating a total of more than 64 processors. Note that Windows calls a schedulable portion of a processor's execution unit a *logical processor*, as distinct from a physical processor or core. When we refer to a "processor" or "CPU" in this chapter, we really mean a "logical processor" from Windows's point of view. Windows 7 supported up to four processor groups, for a total of 256 logical processors, while Windows 10 now supports up to 20 groups, with a total of no more than 640 logical processors (therefore, not all groups can be fully filled).

All these additional CPUs created a great deal of contention for the locks used for scheduling CPUs and memory. Windows 7 broke these locks apart. For example, before Windows 7, a single lock was used by the Windows scheduler to synchronize access to the queues containing threads waiting for events. In Windows 7, each object has its own lock, allowing the queues to be accessed concurrently. Similarly, the global object manager lock, the cache manager VACB lock, and the memory manager PFN lock formerly synchronized access to large, global data structures. All were decomposed into more locks on smaller data structures. Also, many execution paths in the scheduler were rewritten to be lock-free. This change resulted in improved scalability performance for Windows 7 even on systems with 256 logical CPUs.

Other changes were due to the increasing importance of support for parallel computing. For years, the computer industry has been dominated by Moore's Law (see Section 1.1.3), leading to higher densities of transistors that manifest themselves as faster clock rates for each CPU. Moore's Law continues to hold true, but limits have been reached that prevent CPU clock rates from increasing further. Instead, transistors are being used to build more and more CPUs into each chip. New programming models for achieving paral-

lel execution, such as Microsoft’s Concurrency RunTime (ConCRT) and Parallel Processing Library (PPL), as well as Intel’s Threading Building Blocks (TBB), are being used to express parallelism in C++ programs. Additionally, a vendor-neutral standard called OpenMP is supported by almost all compilers. Although Moore’s Law has governed computing for forty years, it now seems that Amdahl’s Law, which governs parallel computing (see Section 4.2), will rule the future.

Finally, power considerations have complicated design decisions around high-performance computing—especially in mobile systems, where battery life might trump performance needs, but also in cloud/server environments, where the cost of electricity might outweigh the need for the fastest possible computational result. Accordingly, Windows 10 now supports features that may sometimes sacrifice raw performance for better power efficiency. Examples include Core Parking, which puts an idle system into a sleep state, and Heterogeneous Multi Processing (HMP), which allocates tasks efficiently among cores.

To support task-based parallelism, the AMD64 ports of Windows 7 and later versions provide a new form of **user-mode scheduling (UMS)**. UMS allows programs to be decomposed into tasks, and the tasks are then scheduled on the available CPUs by a scheduler that operates in user mode rather than in the kernel.

The advent of multiple CPUs on the smallest computers is only part of the shift taking place to parallel computing. Graphics processing units (GPUs) accelerate the computational algorithms needed for graphics by using **SIMD** architectures to execute a single instruction for multiple data at the same time. This has given rise to the use of GPUs for general computing, not just graphics. Operating-system support for software like OpenCL and CUDA is allowing programs to take advantage of the GPUs. Windows supports the use of GPUs through software in its DirectX graphics support. This software, called DirectCompute, allows programs to specify **computational kernels** using the “high-level shader language” programming model used by SIMD hardware. The computational kernels run very quickly on the GPU and return their results to the main computation running on the CPU. In Windows 10, the native graphics stack and many new Windows applications make use of DirectCompute, and new versions of Task Manager track GPU processor and memory usage, with DirectX now having its own GPU thread scheduler and GPU memory manager.

### 21.2.5 Extensibility

**Extensibility** refers to the capability of an operating system to keep up with advances in computing technology. To facilitate change over time, the developers implemented Windows using a layered architecture. The lowest-level kernel “executive” runs in kernel mode and provides the basic system services and abstractions that support shared use of the system. On top of the executive, several services operate in user mode. Among them were the environment subsystems that emulated different operating systems, which are deprecated today. Even in the kernel, Windows uses a layered architecture, with loadable drivers in the I/O system, so new file systems, new kinds of I/O devices, and new kinds of networking can be added while the system is running. Drivers

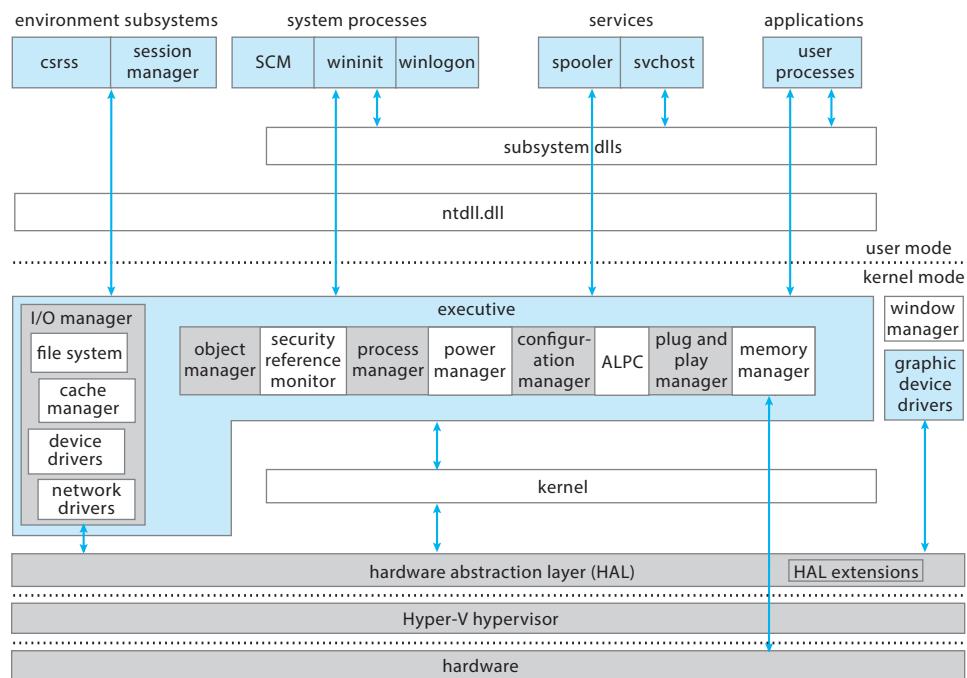


Figure 21.1 Windows block diagram.

aren't limited to providing I/O functionality, however. As we've seen, a Pico Provider is also a type of loadable driver (as are most anti-malware drivers). Through Pico Providers and the modular structure of the system, additional operating system support can be added without affecting the executive. Figure 21.1 shows the architecture of the Windows 10 kernel and subsystems.

Windows also uses a client–server model like the Mach operating system and supports distributed processing through **remote procedure calls (RPCs)** as defined by the Open Software Foundation. These RPCs take advantage of an executive component, called the **advanced local procedure call (ALPC)**, that implements highly scalable communication between separate processes on a local machine. A combination of TCP/IP packets and named pipes over the SMB protocol is used for communication between processes across a network. On top of RPC, Windows implements the Distributed Common Object Model (DCOM) infrastructure, as well as the Windows Management Instrumentation (WMI) and Windows Remote Management (WinRM) mechanism, all of which can be used to rapidly extend the system with new services and management capabilities.

### 21.2.6 Portability

An operating system is **portable** if it can be moved from one CPU architecture to another with relatively few changes. Windows was designed to be portable. Like the UNIX operating system, Windows is written primarily in C and C++. There is relatively little architecture-specific source code and very little assem-

bly code. Porting Windows to a new architecture mostly affects the Windows kernel, since the user-mode code in Windows is almost exclusively written to be architecture independent. To port Windows, the kernel's architecture-specific code must be rewritten for the target CPU, and sometimes conditional compilation is needed in other parts of the kernel because of changes in major data structures, such as the page-table format. The entire Windows system must then be recompiled for the new CPU instruction set.

Operating systems are sensitive not only to CPU architecture but also to CPU support chips and hardware boot programs. The CPU and support chips are collectively known as the **chipset**. These chipsets and the associated boot code determine how interrupts are delivered, describe the physical characteristics of each system, and provide interfaces to deeper aspects of the CPU architecture, such as error recovery and power management. It would be burdensome to have to port Windows to each type of support chip as well as to each CPU architecture. Instead, Windows isolates most of the chipset-dependent code in a dynamic link library (DLL), called the **hardware-abstraction layer (HAL)**, that is loaded with the kernel.

The Windows kernel depends on the HAL interfaces rather than on the underlying chipset details. This allows the single set of a kernel and driver binaries for a particular CPU to be used with different chipsets simply by loading a different version of the HAL. Originally, to support the many architectures that Windows ran on, and the many computer companies and designs in the market, over 450 different HALs existed. Over time, the advent of standards such as the Advanced Configuration and Power Interface (ACPI), the increasing similarity of components available in the marketplace, and the merging of computer manufacturers led to changes; today, the AMD64 port of Windows 10 comes with a single HAL. Interestingly, though, no such developments have yet occurred in the market for mobile devices. Today, Windows supports a limited number of ARM chipsets—and must have the appropriate HAL code for each of them. To avoid going back to a model of multiple HALs, Windows 8 introduced the concept of HAL Extensions, which are DLLs that are loaded dynamically by the HAL based on the detected SoC (system on a chip) components, such as the interrupt controller, timer manager, and DMA controller.

Over the years, Windows has been ported to a number of different CPU architectures: Intel IA-32-compatible 32-bit CPUs, AMD64-compatible and IA64 64-bit CPUs, and DEC Alpha, DEC Alpha AXP64, MIPS, and PowerPC CPUs. Most of these CPU architectures failed in the consumer desktop market. When Windows 7 shipped, only the IA-32 and AMD64 architectures were supported on client computers, along with AMD64 on servers. With Windows 8, 32-bit ARM was added, and Windows 10 now supports ARM64 as well.

### 21.2.7 International Support

Windows was designed for international and multinational use. It provides support for different locales via the **national-language-support (NLS)** API. The NLS API provides specialized routines to format dates, time, and money in accordance with national customs. String comparisons are specialized to account for varying character sets. UNICODE is Windows's native character code, specifically in its UTL-16LE encoding format (which is different from

Linux's and the Web's standard UTF-8). Windows supports ANSI characters by converting them to UNICODE characters before manipulating them (8-bit to 16-bit conversion).

System text strings are kept in resource tables inside files that can be replaced to localize the system for different languages. Before Windows Vista, Microsoft shipped these resource tables inside the DLLs themselves, which meant that different executable binaries existed for each different version of Windows and only one language was available at a single time. With Windows Vista's [multiple user interface \(MUI\)](#) support, multiple locales can be used concurrently, which is important to multilingual individuals and businesses. This was achieved by moving all of the resource tables into separate .mui files that live in the appropriate language directory alongside the .dll file, with support in the loader to pick the appropriate file based on the currently selected language.

### 21.2.8 Energy Efficiency

Increasing energy efficiency causes batteries to last longer for laptops and Internet-only netbooks, saves significant operating costs for power and cooling of data centers, and contributes to green initiatives aimed at lowering energy consumption by businesses and consumers. For some time, Windows has implemented several strategies for decreasing energy use. The CPUs are moved to lower power states—for example, by lowering clock frequency—whenever possible. In addition, when a computer is not being actively used, Windows may put the entire computer into a low-power state (sleep) or may even save all of memory to secondary storage and shut the computer off (hibernation). When the user returns, the computer powers up and continues from its previous state, so the user does not need to reboot and restart applications.

The longer a CPU can stay unused, the more energy can be saved. Because computers are so much faster than human beings, a lot of energy can be saved just while humans are thinking. The problem is that many programs are polled to wait for activity, and software timers are frequently expiring, keeping the CPU from staying idle long enough to save much energy.

Windows 7 extends CPU idle time by delivering clock-tick interrupts only to logical CPU 0 and all other currently active CPUs (skipping idle ones) and by coalescing eligible software timers into smaller numbers of events. On server systems, it also “parks” entire CPUs when systems are not heavily loaded. Additionally, timer expiration is not distributed, and a single CPU is typically in charge of handling all software timer expirations. A thread that was running on, say, logical CPU 3 does not cause CPU 3 to wake up and service this expiration if it is currently idle when another, nonsleeping CPU could handle it instead.

While these measures helped, they were not enough to increase battery life in mobile systems such as phones, which have a fraction of the battery capacity of laptops. Windows 8 thus introduced a number of features to further optimize battery life. First, the WinRT programming model does not allow for precise timers with a guaranteed expiration time. All timers registered through the new API are candidates for coalescing, unlike Win32 timers, which had to be manually opted in. Next, the concept of a [dynamic tick](#) was introduced, in

which CPU0 is no longer the **clock owner**, and the last-active CPU takes on this responsibility.

More significantly, the entire Metro/Modern/UWP application model delivered through the Windows Store includes a feature, the **Process Lifetime Manager (PLM)**, that automatically suspends all of the threads in a process that has been idle for more than a few seconds. This not only mitigates the constant polling behavior of many applications, but also removes the ability for UWP applications to do their own background work (such as querying the GPS location), forcing them to deal with a system of **brokers** that efficiently coalesce audio, location, download, and other requests and can cache data while the process is suspended.

Finally, using a new component called the **Desktop Activity Moderator (DAM)**, Windows 8 and later versions support a new type of system state called **Connected Standby**. Imagine putting a computer to sleep—this action takes several seconds, after which everything on the computer appears to disappear, with all the hardware turning off. Pressing a button on the keyboard wakes up the computer, which takes a few additional seconds, and everything resumes. On a phone or tablet, however, putting the device to sleep is not expected to take seconds—users want their screen to turn off immediately. But if Windows merely turned off the screen, all programs would continue running, and legacy Win32 applications, lacking a PLM and timer coalescing, would continue to poll, perhaps even waking up the screen again. Battery life would drain significantly.

Connected Standby addresses this problem by virtually freezing the computer when the power button is pressed or the screen turns off—without really putting the computer to sleep. The hardware clock is stopped, all processes and services are suspended, and all timer expirations are delayed 30 minutes. The net effect, even though the computer is still running, is that it runs in such a almost-total state of idleness that the processor and peripherals can effectively run in their lowest power state. Special hardware and firmware are required to fully support this mode; for example, the Surface-branded tablet hardware includes this capability.

### 21.2.9 Dynamic Device Support

Early in the history of the PC industry, computer configurations were fairly static, although new devices might occasionally be plugged into the serial, printer, or game ports on the back of a computer. The next steps toward dynamic configuration of PCs were laptop docks and PCMCIA cards. Using such a device, a PC could quickly be connected to or disconnected from a full set of peripherals. Contemporary PCs are designed to enable users to plug and unplug a huge host of peripherals frequently.

Support for dynamic configuration of devices is continually evolving in Windows. The system can automatically recognize devices when they are plugged in and can find, install, and load the appropriate drivers—often without user intervention. When devices are unplugged, the drivers automatically unload, and system execution continues without disrupting other software. Additionally, Windows Update permits downloading of third-party drivers

directly through Microsoft, avoiding the usage of installation DVDs or having the user scour the manufacturer's website.

Beyond peripherals, Windows Server also supports dynamic hot-add and hot-replace of CPUs and RAM, as well as dynamic hot-remove of RAM. These features allow the components to be added, replaced, or removed without system interruption. While of limited use in physical servers, this technology is key to dynamic scalability in cloud computing, especially in Infrastructure-as-a-Service (IaaS) and cloud computing environments. In these scenarios, a physical machine can be configured to support a limited number of its processors based on a service fee, which can then be dynamically upgraded, without requiring a reboot, through a compatible hypervisor such as Hyper-V and a simple slider in the owner's user interface.

## 21.3 System Components

The architecture of Windows is a layered system of modules operating at specific privilege levels, as shown earlier in Figure 21.1. By default, these privilege levels are first implemented by the processor (providing a “vertical” privilege isolation between user mode and kernel mode). Windows 10 can also use its Hyper-V hypervisor to provide an orthogonal (logically independent) security model through **Virtual Trust Levels (VTLs)**. When users enable this feature, the system operates in a Virtual Secure Mode (VSM). In this mode, the layered privileged system now has two implementations, one called the **Normal World**, or VTL 0, and one called the **Secure World**, or VTL 1. Within each of these worlds, we find a user mode and a kernel mode.

Let's look at this structure in somewhat more detail.

- In the Normal World, in kernel mode are (1) the HAL and its extensions and (2) the kernel and its executive, which load drivers and DLL dependencies. In user mode are a collection of system processes, the Win32 environment subsystem, and various services.
- In the Secure World, if VSM is enabled, are a secure kernel and executive (within which a secure micro-HAL is embedded). A collection of isolated **Trustlets** (discussed later) run in secure user mode.
- Finally, the bottommost layer in Secure World runs in a special processor mode (called, for example, VMX Root Mode on Intel processors), which contains the Hyper-V hypervisor component, which uses hardware virtualization to construct the Normal-to-Secure-World boundary. (The user-to-kernel boundary is provided by the CPU natively.)

One of the chief advantages of this type of architecture is that interactions between modules, and between privilege levels, are kept simple, and that isolation needs and security needs are not necessarily conflated through privilege. For example, a secure, protected component that stores passwords can itself be unprivileged. In the past, operating-system designers chose to meet isolation needs by making the secure component highly privileged, but this results in a net loss for the security of the system when this component is compromised.

The remainder of this section describes these layers and subsystems.

### 21.3.1 Hyper-V Hypervisor

The hypervisor is the first component initialized on a system with VSM enabled, which happens as soon as the user enables the Hyper-V component. It is used both to provide hardware virtualization features for running separate virtual machines and to provide the VTL boundary and related access to the hardware's Second Level Address Translation (SLAT) functionality (discussed shortly). The hypervisor uses a CPU-specific virtualization extension, such as AMD's Pacifica (SVMX) or Intel's Vanderpool (VT-x), to intercept any interrupt, exception, memory access, instruction, port, or register access that it chooses and deny, modify, or redirect the effect, source, or destination of the operation. It also provides a [hypervisor](#) interface, which enables it to communicate with the kernel in VTL 0, the secure kernel in VTL 1, and all other running virtual machine kernels and secure kernels.

### 21.3.2 Secure Kernel

The secure kernel acts as the kernel-mode environment of isolated (VTL 1) user-mode Trustlet applications (applications that implement parts of the Windows security model). It provides the same system-call interface that the kernel does, so that all interrupts, exceptions, and attempts to enter kernel mode from a VTL 1 Trustlet result in entering the secure kernel instead. However, the secure kernel is not involved in context switching, thread scheduling, memory management, interprocess-communication, or any of the other standard kernel tasks. Additionally, no kernel-mode drivers are present in VTL 1. In an attempt to reduce the attack surface of the Secure World, these complex implementations remain the responsibility of Normal World components. Thus, the secure kernel acts as a type of "proxy kernel" that hands off the management of its resources, paging, scheduling, and more, to the regular kernel services in VTL 0. This does make the Secure World vulnerable to denial-of-service attacks, but that is a reasonable tradeoff of the security design, which values data privacy and integrity over service guarantees.

In addition to forwarding system calls, the secure kernel's other responsibility is providing access to the hardware secrets, the trusted platform module (TPM), and code integrity policies that were captured at boot. With this information, Trustlets can encrypt and decrypt data with keys that the Normal World cannot obtain and can sign and attest (co-sign by Microsoft) reports with integrity tokens that cannot be faked or replicated outside of the Secure World. Using a CPU feature called Second Level Address Translation (SLAT), the secure kernel also provides the ability to allocate virtual memory in such a way that the physical pages backing it cannot be seen at all from the Normal World. Windows 10 uses these capabilities to provide additional protection of enterprise credentials through a feature called Credential Guard.

Furthermore, when Device Guard (mentioned earlier) is activated, it takes advantage of VTL 1 capabilities by moving all digital signature checking into the secure kernel. This means that even if attacked through a software vulnerability, the normal kernel cannot be forced to load unsigned drivers, as the VTL 1 boundary would have to be breached for that to occur. On a Device Guard-protected system, for a kernel-mode page in VTL 0 to be authorized for execution, the kernel must first ask permission from the secure kernel, and only the secure kernel can grant this page executable access. More secure deployments

(such as in embedded or high-risk systems) can require this level of signature validation for user-mode pages as well.

Additionally, work is being done to allow special classes of hardware devices, such as USB webcams and smartcard readers, to be directly managed by user-mode drivers running in VTL 1 (using the UMDF framework described later), allowing biometric data to be securely captured in VTL 1 without any component in the Normal World being able to intercept it. Currently, the only Trustlets allowed are those that provide the Microsoft-signed implementation of Credential Guard and virtual-TPM support. Newer versions of Windows 10 will also support **VSM Enclaves**, which will allow validly signed (but not necessarily Microsoft-signed) third-party code wishing to perform its own cryptographic calculations to do so. Software enclaves will allow regular VTL 0 applications to “call into” an enclave, which will run executable code on top of input data and return presumably encrypted output data.

For more information on the secure kernel, see <https://blogs.technet.microsoft.com/ash/2016/03/02/windows-10-device-guard-and-credential-guard-demystified/>.

### 21.3.3 Hardware-Abstraction Layer

The HAL is the layer of software that hides hardware chipset differences from upper levels of the operating system. The HAL exports a virtual hardware interface that is used by the kernel dispatcher, the executive, and the device drivers. Only a single version of each device driver is required for each CPU architecture, no matter what support chips might be present. Device drivers map devices and access them directly, but the chipset-specific details of mapping memory, configuring I/O buses, setting up DMA, and coping with motherboard-specific facilities are all provided by the HAL interfaces.

### 21.3.4 Kernel

The kernel layer of Windows has the following main responsibilities: thread scheduling and context switching, low-level processor synchronization, interrupt and exception handling, and switching between user mode and kernel mode through the system-call interface. Additionally, the kernel layer implements the initial code that takes over from the boot loader, formalizing the transition into the Windows operating system. It also implements the initial code that safely crashes the kernel in case of an unexpected exception, assertion, or other inconsistency. The kernel is mostly implemented in the C language, using assembly language only when absolutely necessary to interface with the lowest level of the hardware architecture and when direct register access is needed.

#### 21.3.4.1 Dispatcher

The dispatcher provides the foundation for the executive and the subsystems. Most of the dispatcher is never paged out of memory, and its execution is never preempted. Its main responsibilities are thread scheduling and context switching, implementation of synchronization primitives, timer management, software interrupts (asynchronous and deferred procedure calls), interprocessor interrupts (IPIs) and exception dispatching. It also manages hardware and

software interrupt prioritization under the system of **interrupt request levels (IRQLs)**.

#### 21.3.4.2 Switching Between User-Mode and Kernel-Mode Threads

What the programmer thinks of as a thread in traditional Windows is actually a thread with two modes of execution: a **user-mode thread (UT)** and a **kernel-mode thread (KT)**. The thread has two stacks, one for UT execution and the other for KT. A UT requests a system service by executing an instruction that causes a trap to kernel mode. The kernel layer runs a trap handler that switches UT stack to its KT sister and changes CPU mode to kernel. When thread in KT mode has completed its kernel execution and is ready to switch back to the corresponding UT, the kernel layer is called to make the switch to the UT, which continues its execution in user mode. The KT switch also happens when an interrupt occurs.

Windows 7 modifies the behavior of the kernel layer to support user-mode scheduling of the UTs. User-mode schedulers in Windows 7 support cooperative scheduling. A UT can explicitly yield to another UT by calling the user-mode scheduler; it is not necessary to enter the kernel. User-mode scheduling is explained in more detail in Section 21.7.3.7.

In Windows, the dispatcher is not a separate thread running in the kernel. Rather, the dispatcher code is executed by the KT component of a UT thread. A thread goes into kernel mode in the same circumstances that, in other operating systems, cause a kernel thread to be called. These same circumstances will cause the KT to run through the dispatcher code after its other operations, determining which thread to run next on the current core.

#### 21.3.4.3 Threads

Like many other modern operating systems, Windows uses threads as the key schedulable unit of executable code, with processes serving as containers of threads. Therefore, each process must have at least one thread, and each thread has its own scheduling state, including actual priority, processor affinity, and CPU usage information.

There are eight possible thread states: `initializing`, `ready`, `deferred-ready`, `standby`, `running`, `waiting`, `transition`, and `terminated`. `ready` indicates that the thread is waiting to execute, while `deferred-ready` indicates that the thread has been selected to run on a specific processor but has not yet been scheduled. A thread is `running` when it is executing on a processor core. It runs until it is preempted by a higher-priority thread, until it terminates, until its allotted execution time (quantum) ends, or until it waits on a dispatcher object, such as an event signaling I/O completion. If a thread is preempting another thread on a different processor, it is placed in the `standby` state on that processor, which means it is the next thread to run.

Preemption is instantaneous—the current thread does not get a chance to finish its quantum. Therefore, the processor sends a software interrupt—in this case, a **deferred procedure call (DPC)**—to signal to the other processor that a thread is in the `standby` state and should be immediately picked up for execution. Interestingly, a thread in the `standby` state can itself be preempted if yet another processor finds an even higher-priority thread to run in this processor. At that point, the new higher-priority thread will go to `standby`,

and the previous thread will go to the ready state. A thread is in the waiting state when it is waiting for a dispatcher object to be signaled. A thread is in the transition state while it waits for resources necessary for execution; for example, it may be waiting for its kernel stack to be paged in from secondary storage. A thread enters the terminated state when it finishes execution, and a thread begins in the initializing state as it is being created, before becoming ready for the first time.

The dispatcher uses a 32-level priority scheme to determine the order of thread execution. Priorities are divided into two classes: variable class and static class. The variable class contains threads having priorities from 1 to 15, and the static class contains threads with priorities ranging from 16 to 31. The dispatcher uses a linked list for each scheduling priority; this set of lists is called the **dispatcher database**. The database uses a bitmap to indicate the presence of at least one entry in the list associated with the priority of the bit's position. Therefore, instead of having to traverse the set of lists from highest to lowest until it finds a thread that is ready to run, the dispatcher can simply find the list associated with the highest bit set.

Prior to Windows Server 2003, the dispatcher database was global, resulting in heavy contention on large CPU systems. In Windows Server 2003 and later versions, the global database was broken apart into per-processor databases, with per-processor locks. With this new model, a thread will only be in the database of its **ideal processor**. It is thus guaranteed to have a processor affinity that includes the processor on whose database it is located. The dispatcher can now simply pick the first thread in the list associated with the highest bit set and does not have to acquire a global lock. Dispatching is therefore a constant-time operation, parallelizable across all CPUs on the machine.

On a single-processor system, if no ready thread is found, the dispatcher executes a special thread called the *idle thread*, whose role is to begin the transition to one of the CPU's initial sleep states. Priority class 0 is reserved for the idle thread. On a multiprocessor system, before executing the idle thread, the dispatcher looks at the dispatcher databases of other nearby processors, taking caching topologies and NUMA node distances into consideration. This operation requires acquiring the locks of other processor cores in order to safely inspect their lists. If no thread can be stolen from a nearby core, the dispatcher looks at the next nearest core, and so on. If no threads can be stolen at all, then the processor executes the idle thread. Therefore, in a multiprocessor system, each CPU will have its own idle thread.

Putting each thread on only the dispatcher database of its ideal processor causes a locality problem. Imagine a CPU executing a thread at priority 2 in a CPU-bound way, while another CPU is executing a thread at priority 18, also CPU-bound. Then, a thread at priority 17 becomes ready. If the ideal processor of this thread is the first CPU, the thread preempts the current running thread. But if the ideal processor is the latter CPU, it goes into the ready queue instead, waiting for its turn to run (which won't happen until the priority 17 thread gives up the CPU by terminating or entering a wait state).

Windows 7 introduced a load-balancer algorithm to address this situation, but it was a heavy-handed and disruptive approach to the locality issue. Windows 8 and later versions solved the problem in a more nuanced way. Instead of a global database as in Windows XP and earlier versions, or a per-processor

database as in Windows Server 2003 and later versions, the newer Windows versions combine these approaches to form a **shared ready queue** among a group of some, but not all, processors. The number of CPUs that form one shared group depends on the topology of the system, as well as on whether it is a server or client system. The number is chosen to keep contention low on very large processor systems, while avoiding locality (and thus latency and contention) issues on smaller client systems. Additionally, processor affinities are still respected, so that a processor in a given group is guaranteed that all threads in the shared ready queue are appropriate—it never needs to “skip” over a thread, keeping the algorithm constant time.

Windows has a timer expire every 15 milliseconds to create a clock “tick” to examine system states, update the time, and do other housekeeping. That tick is received by the thread on every non-idle core. The interrupt handler (being run by the thread, now in KT mode) determines if the thread’s quantum has expired. When a thread’s time quantum runs out, the clock interrupt queues a quantum-end DPC to the processor. Queuing the DPC results in a software interrupt when the processor returns to normal interrupt priority. The software interrupt causes the thread to run dispatcher code in KT mode to reschedule the processor to execute the next ready thread at the preempted thread’s priority level in a round-robin fashion. If no other thread at this level is ready, a lower-priority ready thread is not chosen, because a higher-priority ready thread already exists—the one that exhausted its quantum in the first place. In this situation, the quantum is simply restored to its default value, and the same thread executes once again. Therefore, Windows always executes the highest-priority ready thread.

When a variable-priority thread is awakened from a wait operation, the dispatcher may boost its priority. The amount of the boost depends on the type of wait associated with the thread. If the wait was due to I/O, then the boost depends on the device for which the thread was waiting. For example, a thread waiting for sound I/O would get a large priority increase, whereas a thread waiting for a disk operation would get a moderate one. This strategy enables I/O-bound threads to keep the I/O devices busy while permitting compute-bound threads to use spare CPU cycles in the background.

Another type of boost is applied to threads waiting on mutex, semaphore, or event synchronization objects. This boost is usually a hard-coded value of one priority level, although kernel drivers have the option of making a different change. (For example, the kernel-mode GUI code applies a boost of two priority levels to all GUI threads waking up to process window messages.) This strategy is used to reduce the latency between when a lock or other notification mechanism is signaled and when the next waiter in line executes in response to the state change.

In addition, the thread associated with the user’s active GUI window receives a priority boost of two whenever it wakes up for any reason, on top of any other existing boost, to enhance its response time. This strategy, called the **foreground priority separation boost**, tends to give good response times to interactive threads.

Finally, Windows Server 2003 added a lock-handoff boost for certain classes of locks, such as critical sections. This boost is similar to the mutex, semaphore, and event boost, except that it tracks ownership. Instead of boosting the waking thread by a hard-coded value of one priority level, it boosts to one priority

level above that of the current owner (the one releasing the lock). This helps in situations where, for example, a thread at priority 12 is releasing a mutex, but the waiting thread is at priority 8. If the waiting thread receives a boost only to 9, it will not be able to preempt the releasing thread. But if it receives a boost to 13, it can preempt and instantly acquire the critical section.

Because threads may run with boosted priorities when they wake up from waits, the priority of a thread is lowered at the end of every quantum as long as the thread is above its base (initial) priority. This is done according to the following rule: For I/O threads and threads boosted due to waking up because of an event, mutex, or semaphore, one priority level is lost at quantum end. For threads boosted due to the lock-handoff boost or the foreground priority separation boost, the entire value of the boost is lost. Threads that have received boosts of both types will obey both of these rules (losing one level of the first boost, as well as the entirety of the second boost). Lowering the thread's priority makes sure that the boost is applied only for latency reduction and for keeping I/O devices busy, not to give undue execution preference to compute-bound threads.

#### 21.3.4.4 Thread Scheduling

Scheduling occurs when a thread enters the `ready` or `waiting` state, when a thread terminates, or when an application changes a thread's processor affinity. As we have seen throughout the text, a thread could become ready at any time. If a higher-priority thread becomes ready while a lower-priority thread is running, the lower-priority thread is preempted immediately. This preemption gives the higher-priority thread instant access to the CPU, without waiting on the lower-priority thread's quantum to complete.

It is the lower-priority thread itself, performing some event that caused it to operate in the dispatcher, that wakes up the waiting thread and immediately context-switches to it while placing itself back in the `ready` state. This model essentially distributes the scheduling logic throughout dozens of Windows kernel functions and makes each currently running thread behave as the scheduling entity. In contrast, other operating systems rely on an external "scheduler thread" triggered periodically based on a timer. The advantage of the Windows approach is latency reduction, with the cost of added overhead inside every I/O and other state-changing operation, which causes the current thread to perform scheduler work.

Windows is not a hard-real-time operating system, however, because it does not guarantee that any thread, even the highest-priority one, will start to execute within a particular time limit or have a guaranteed period of execution. Threads are blocked indefinitely while DPCs and **interrupt service routines (ISRs)** are running (as further discussed below), and they can be preempted at any time by a higher-priority thread or be forced to round-robin with another thread of equal priority at quantum end.

Traditionally, the Windows scheduler uses sampling to measure CPU utilization by threads. The system timer fires periodically, and the timer interrupt handler takes note of what thread is currently scheduled and whether it is executing in user or kernel mode when the interrupt occurred. This sampling technique originally came about because either the CPU did not have a high-resolution clock or the clock was too expensive or unreliable to access

frequently. Although efficient, sampling is inaccurate and leads to anomalies such as charging the entire duration of the clock (15 milliseconds) to the currently running thread (or DPC or ISR). Therefore, the system ends up completely ignoring some number of milliseconds—say, 14.999—that could have been spent idle, running other threads, running other DPCs and ISRs, or a combination of all of these operations. Additionally, because quantum is measured based on clock ticks, this causes the premature round-robin selection of a new thread, even though the current thread may have run for only a fraction of the quantum.

Starting with Windows Vista, execution time is also tracked using the hardware **timestamp counter (TSC)** included in all processors since the Pentium Pro. Using the TSC results in more accurate accounting of CPU usage (for applications that use it—note that Task Manager does not) and also causes the scheduler not to switch out threads before they have run for a full quantum. Additionally, Windows 7 and later versions track, and charge, the TSC to ISRs and DPCs, resulting in more accurate “Interrupt Time” measurements as well (again, for tools that use this new measurement). Because all possible execution time is now accounted for, it is possible to add it to idle time (which is also tracked using the TSC) and accurately compute the exact number of CPU cycles out of all possible CPU cycles in a given period (due to the fact that modern processors have dynamically shifting frequencies), resulting in cycle-accurate CPU usage measurements. Tools such as Microsoft’s SysInternals Process Explorer use this mechanism in their user interface.

#### 21.3.4.5 Implementation of Synchronization Primitives

Windows uses a number of **dispatcher objects** to control dispatching and synchronization in the system. Examples of these objects include the following:

- The **event** is used to record an event occurrence and to synchronize this occurrence with some action. Notification events signal all waiting threads, and synchronization events signal a single waiting thread.
- The **mutex** provides kernel-mode or user-mode mutual exclusion associated with the notion of ownership.
- The **semaphore** acts as a counter or gate to control the number of threads that access a resource.
- The **thread** is the entity that is scheduled by the kernel dispatcher. It is associated with a process, which encapsulates a virtual address space, list of open resources, and more. The thread is signaled when the thread exits, and the process, when the process exits (that is, when all of its threads have exited).
- The **timer** is used to keep track of time and to signal timeouts when operations take too long and need to be interrupted or when a periodic activity needs to be scheduled. Just like events, timers can operate in notification mode (signal all) or synchronization mode (signal one).

All of the dispatcher objects can be accessed from user mode via an open operation that returns a handle. The user-mode code waits on handles to

synchronize with other threads as well as with the operating system (see Section 21.7.1).

#### 21.3.4.6 Interrupt Request Levels (IRQLs)

Both hardware and software interrupts are prioritized and are serviced in priority order. There are 16 interrupt request levels (IRQLs) on all Windows ISAs except the legacy IA-32, which uses 32. The lowest level, IRQL 0, is called the PASSIVE\_LEVEL and is the default level at which all threads execute, whether in kernel or user mode. The next levels are the software interrupt levels for APCs and DPCs. Levels 3 to 10 are used to represent hardware interrupts based on selections made by the PnP manager with the help of the HAL and the PCI/ACPI bus drivers. Finally, the uppermost levels are reserved for the clock interrupt (used for quantum management) and IPI delivery. The last level, HIGH\_LEVEL, blocks all maskable interrupts and is typically used when crashing the system in a controlled manner.

The Windows IRQLs are defined in Figure 21.2.

#### 21.3.4.7 Software Interrupts: Asynchronous and Deferred Procedure Calls

The dispatcher implements two types of software interrupts: **asynchronous procedure calls (APCs)** and deferred procedure calls (DPCs, mentioned earlier). APCs are used to suspend or resume existing threads, terminate threads, deliver notifications that an asynchronous I/O has completed, and extract or modify the contents of the CPU registers (the context) from a running thread. APCs are queued to specific threads and allow the system to execute both system and user code within a process's context. User-mode execution of an APC cannot occur at arbitrary times, but only when the thread is waiting and is marked *alertable*. Kernel-mode execution of an APC, in contrast, instantaneously executes in the context of a running thread because it is delivered as a software interrupt running at IRQL 1 (APC\_LEVEL), which is higher than the default IRQL 0 (PASSIVE\_LEVEL). Additionally, even if a thread is waiting in kernel mode, the wait can be broken by the APC and resumed once the APC completes execution.

interrupt levels	types of interrupts
31	machine check or bus error
30	power fail
29	interprocessor notification (request another processor to act; e.g., dispatch a process or update the TLB)
28	clock (used to keep track of time)
27	profile
3–26	traditional PC IRQ hardware interrupts
2	dispatch and deferred procedure call (DPC) (kernel)
1	asynchronous procedure call (APC)
0	passive

Figure 21.2 Windows x86 interrupt-request levels (IRLQs).

DPCs are used to postpone interrupt processing. After handling all urgent device-interrupt processing, the ISR schedules the remaining processing by queuing a DPC. The associated software interrupt runs at IRQL 2 (DPC\_LEVEL), which is lower than all other hardware/I/O interrupt levels. Thus, DPCs do not block other device ISRs. In addition to deferring device-interrupt processing, the dispatcher uses DPCs to process timer expirations and to interrupt current thread execution at the end of the scheduling quantum.

Because IRQL 2 is higher than 0 (PASSIVE) and 1 (APC), execution of DPCs prevents standard threads from running on the current processor and also keeps APCs from signaling the completion of I/O. Therefore, it is important for DPC routines not to take an extended amount of time. As an alternative, the executive maintains a pool of worker threads. DPCs can queue work items to the worker threads, where they will be executed using normal thread scheduling at IRQL 0. Because the dispatcher itself runs at IRQL 2, and because paging operations require waiting on I/O (and that involves the dispatcher), DPC routines are restricted in that they cannot take page faults, call pageable system services, or take any other action that might result in an attempt to wait for a dispatcher object to be signaled. Unlike APCs, which are targeted to a thread, DPC routines make no assumptions about what process context the processor is executing, since they execute in the same context as the currently executing thread, which was interrupted.

#### 21.3.4.8 Exceptions, Interrupts, and IPIs

The kernel dispatcher also provides trap handling for exceptions and interrupts generated by hardware or software. Windows defines several architecture-independent exceptions, including:

- Integer or floating-point overflow
- Integer or floating-point divide by zero
- Illegal instruction
- Data misalignment
- Privileged instruction
- Access violation
- Paging file quota exceeded
- Debugger breakpoint

The trap handlers deal with the hardware-level exceptions (called **traps**) and call the elaborate exception-handling code performed by the kernel's exception dispatcher. The **exception dispatcher** creates an exception record containing the reason for the exception and finds an exception handler to deal with it.

When an exception occurs in kernel mode, the exception dispatcher simply calls a routine to locate the exception handler. If no handler is found, a fatal system error occurs and the user is left with the infamous “blue screen of death” that signifies system failure. In Windows 10, this is now a friendlier “sad face of sorrow” with a QR code, but the blue color remains.

Exception handling is more complex for user-mode processes, because the Windows error reporting (WER) service sets up an ALPC error port for every process, on top of the Win32 environment subsystem, which sets up an ALPC exception port for every process it creates. (For details on ports, see Section 21.3.5.4.) Furthermore, if a process is being debugged, it gets a debugger port. If a debugger port is registered, the exception handler sends the exception to the port. If the debugger port is not found or does not handle that exception, the dispatcher attempts to find an appropriate exception handler. If none exists, it contacts the default unhandled exception handler, which will notify WER of the process crash so that a crash dump can be generated and sent to Microsoft. If there is a handler, but it refuses to handle the exception, the debugger is called again to catch the error for debugging. If no debugger is running, a message is sent to the process's exception port to give the environment subsystem a chance to react to the exception. Finally, a message is sent to WER through the error port, in the case where the unhandled exception handler may not have had a chance to do so, and then the kernel simply terminates the process containing the thread that caused the exception.

WER will typically send the information back to Microsoft for further analysis, unless the user has opted out or is using a local error-reporting server. In some cases, Microsoft's automated analysis may be able to recognize the error immediately and suggest a fix or workaround.

The interrupt dispatcher in the kernel handles interrupts by calling either an interrupt service routine (ISR) supplied by a device driver or a kernel trap-handler routine. The interrupt is represented by an **interrupt object** that contains all the information needed to handle the interrupt. Using an interrupt object makes it easy to associate interrupt-service routines with an interrupt without having to access the interrupt hardware directly.

Different processor architectures have different types and numbers of interrupts. For portability, the interrupt dispatcher maps the hardware interrupts into a standard set.

The kernel uses an **interrupt-dispatch table** to bind each interrupt level to a service routine. In a multiprocessor computer, Windows keeps a separate interrupt-dispatch table (IDT) for each processor core, and each processor's IRQL can be set independently to mask out interrupts. All interrupts that occur at a level equal to or less than the IRQL of a processor are blocked until the IRQL is lowered by a kernel-level thread or by an ISR returning from interrupt processing. Windows takes advantage of this property and uses software interrupts to deliver APCs and DPCs, to perform system functions such as synchronizing threads with I/O completion, to start thread execution, and to handle timers.

### 21.3.5 Executive

The Windows executive provides a set of services that all environment subsystems use. To give you a good basic overview, we discuss the following services here: object manager, virtual memory manager, process manager, advanced local procedure call facility, I/O manager, cache manager, security reference monitor, plug-and-play and power managers, registry, and startup. Note, though, that the Windows executive includes more than two dozen services in total.

The executive is organized according to object-oriented design principles. An **object type** in Windows is a system-defined data type that has a set of attributes (data values) and a set of methods (for example, functions or operations) that help define its behavior. An **object** is an instance of an object type. The executive performs its job by using a set of objects whose attributes store the data and whose methods perform the activities.

#### 21.3.5.1 Object Manager

For managing kernel-mode entities, Windows uses a generic set of interfaces that are manipulated by user-mode programs. Windows calls these entities *objects*, and the executive component that manipulates them is the **object manager**. Examples of objects are files, registry keys, devices, ALPC ports, drivers, mutexes, events, processes, and threads. As we saw earlier, some of these, such as mutexes and processes, are dispatcher objects, which means that threads can block in the dispatcher waiting for any of these objects to be signaled. Additionally, most of the non-dispatcher objects include an internal dispatcher object, which is signaled by the executive service controlling it. For example, file objects have an event object embedded, which is signaled when a file is modified.

User-mode and kernel-mode code can access these objects using an opaque value called a **handle**, which is returned by many APIs. Each process has a **handle table** containing entries that track the objects used by the process. There is a “system process” (see Section 21.3.5.11) that has its own handle table, which is protected from user code and is used when kernel-mode code is manipulating handles. The handle tables in Windows are represented by a tree structure, which can expand from holding 1,024 handles to holding over 16 million. In addition to using handles, kernel-mode code can also access an object by using **referenced pointer**, which it must obtain by calling a special API. When handles are used, they must eventually be closed, to avoid keeping an active reference on the object. Similarly, when kernel code uses a referenced pointer, it must use a special API to drop the reference.

A handle can be obtained by creating an object, by opening an existing object, by receiving a duplicated handle, or by inheriting a handle from a parent process. To work around the issue that developers may forget to close their handles, all of the open handles of a process are implicitly closed when it exits or is terminated. However, since kernel handles belong to the system-wide handle table, when a driver unloads, its handles are not automatically closed, and this can lead to resource leaks on the system.

Since the object manager is the only entity that generates object handles, it is the natural place to centralize calling the security reference monitor (SRM) (see Section 21.3.5.7) to check security. When an attempt is made to open an object, the object manager calls the SRM to check whether a process or thread has the right to access the object. If the access check is successful, the resulting rights (encoded as an **access mask**) are cached in the handle table. Therefore, the opaque handle both represents the object in the kernel and identifies the access that was granted to the object. This important optimization means that whenever a file is written to (which could happen hundreds of times a second), security checks are completely skipped, since the handle is already encoded as

a “write” handle. Conversely, if a handle is a “read” handle, attempts to write to the file would instantly fail, without requiring a security check.

The object manager also enforces quotas, such as the maximum amount of memory a process may use, by charging a process for the memory occupied by all its referenced objects and refusing to allocate more memory when the accumulated charges exceed the process’s quota.

Because objects can be referenced through handles from user and kernel mode, and referenced through pointers from kernel mode, the object manager has to keep track of two counts for each object: the number of handles for the object and the number of references. The handle count is the number of handles that refer to the object in all of the handle tables (including the system handle table). The reference count is the sum of all handles (which count as references) plus all pointer references done by kernel-mode components. The count is incremented whenever a new pointer is needed by the kernel or a driver and decremented when the component is done with the pointer. The purpose of these reference counts is to ensure that an object is not freed while it still has a reference, but can still release some of its data (such as the name and security descriptor) when all handles are closed (since kernel-mode components don’t need this information).

The object manager maintains the Windows internal name space. In contrast to UNIX, which roots the system name space in the file system, Windows uses an abstract object manager name space that is only visible in memory or through specialized tools such as the debugger. Instead of file-system directories, the hierarchy is maintained by a special kind of object called a **directory object** that contains a hash bucket of other objects (including other directory objects). Note that some objects don’t have names (such as threads), and even for other objects, whether an object has a name is up to its creator. For example, a process would only name a mutex if it wanted other processes to find, acquire, or inquire about the state of the mutex.

Because processes and threads are created without names, they are referenced through a separate numerical identifier, such as a process ID (PID) or thread (TID). The object manager also supports symbolic links in the name space. As an example, DOS drive letters are implemented using symbolic links; \Global??\C: is a symbolic link to the device object \Device\HarddiskVolumeN, representing a mounted file-system volume in the \Device directory.

Each object, as mentioned earlier, is an instance of an **object type**. The object type specifies how instances are to be allocated, how data fields are to be defined, and how the standard set of virtual functions used for all objects are to be implemented. The standard functions implement operations such as mapping names to objects, closing and deleting, and applying security checks. Functions that are specific to a particular type of object are implemented by system services designed to operate on that particular object type, not by the methods specified in the object type.

The `parse()` function is the most interesting of the standard object functions. It allows the implementation of an object to override the default naming behavior of the object manager (which is to use the virtual object directories). This ability is useful for objects that have their own internal namespace, especially when the namespace might need to be retained between boots. The

I/O manager (for file objects) and the configuration manager (for registry key objects) are the most notable users of parse functions.

Returning to our Windows naming example, device objects used to represent file-system volumes provide a parse function. This allows a name like `\Global??\C:\foo\bar.doc` to be interpreted as the file `\foo\bar.doc` on the volume represented by the device object `HarddiskVolume2`. We can illustrate how naming, parse functions, objects, and handles work together by looking at the steps to open the file in Windows:

1. An application requests that a file named `C:\foo\bar.doc` be opened.
2. The object manager finds the device object `HarddiskVolume2`, looks up the parse procedure (for example, `IopParseDevice`) from the object's type, and invokes it with the file's name relative to the root of the file system.
3. `IopParseDevice()` looks up the file system that owns the volume `HardDiskVolume2` and then calls into the file system, which looks up how to access `\foo\bar.doc` on the volume, performing its own internal parsing of the `foo` directory to find the `bar.doc` file. The file system then allocates a file object and returns it to the I/O manager's parse routine.
4. When the file system returns, the object manager allocates an entry for the file object in the handle table for the current process and returns the handle to the application.

If the file cannot successfully be opened, `IopParseDevice` returns an error indication to the application.

#### 21.3.5.2 Virtual Memory Manager

The executive component that manages the virtual address space, physical memory allocation, and paging is the **memory manager (MM)**. The design of the MM assumes that the underlying hardware supports virtual-to-physical mapping, a paging mechanism, and transparent cache coherence on multiprocessor systems, as well as allowing multiple page-table entries to map to the same physical page frame. The MM in Windows uses a page-based management scheme based on the page sizes supported by hardware (4 KB, 2 MB, and 1 GB). Pages of data allocated to a process that are not in physical memory are either stored in the **paging file** on secondary storage or mapped directly to a regular file on a local or remote file system. A page can also be marked zero-fill-on-demand, which initializes the page with zeros before it is mapped, thus erasing the previous contents.

On 32-bit processors such as IA-32 and ARM, each process has a 4-GB virtual address space. By default, the upper 2 GB are mostly identical for all processes and are used by Windows in kernel mode to access the operating-system code and data structures. For 64-bit architectures such as the AMD64 architecture, Windows provides a 256-TB per-process virtual address space, divided into two 128-TB regions for user mode and kernel mode. (These restrictions are based on hardware limitations that will soon be lifted. Intel has announced that its future

processors will support up to 128 PB of virtual address space, out of the 16 EB theoretically available.)

The availability of the kernel's code in each process's address space is important, and commonly found in many other operating systems as well. Generally, virtual memory is used to map the kernel code into the address space of each process. Then, when say a system call is executed or an interrupt is received, the context switch to allow the current core to run that code is lighter-weight than it would otherwise be without this mapping. Specifically, no memory-management registers need to be saved and restored, and the cache does not get invalidated. The net result is much faster movement between user and kernel code, compared to older architectures that keep kernel memory separate and not available within the process address space.

The Windows MM uses a two-step process to allocate virtual memory. The first step *reserves* one or more pages of virtual addresses in the process's virtual address space. The second step *commits* the allocation by assigning virtual memory space (physical memory or space in the paging files). Windows limits the amount of virtual memory space a process consumes by enforcing a quota on committed memory. A process de-commits memory that it is no longer using to free up virtual memory space for use by other processes. The APIs used to reserve virtual addresses and commit virtual memory take a handle on a process object as a parameter. This allows one process to control the virtual memory of another.

Windows implements shared memory by defining a **section object**. After getting a handle to a section object, a process maps the memory of the section to a range of addresses, called a **view**. A process can establish a view of the entire section or only the portion it needs. Windows allows sections to be mapped not just into the current process but into any process for which the caller has a handle.

Sections can be used in many ways. A section can be backed by secondary storage either in the system-paging file or in a regular file (a memory-mapped file). A section can be *based*, meaning that it appears at the same virtual address for all processes attempting to access it. Sections can also represent physical memory, allowing a 32-bit process to access more physical memory than can fit in its virtual address space. Finally, the memory protection of pages in the section can be set to read only, read-write, read-write-execute, execute only, no access, or copy-on-write.

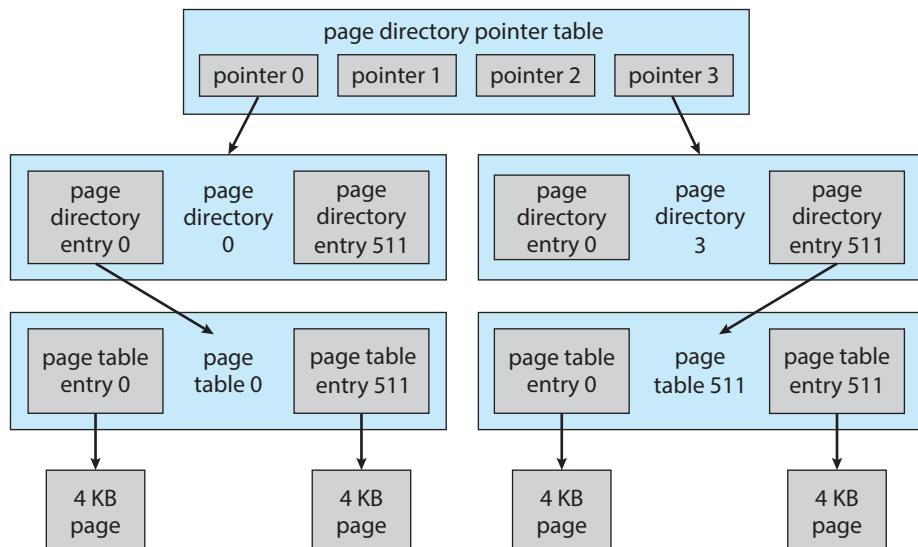
Let's look more closely at the last two of these protection settings:

- A *no-access page* raises an exception if accessed. The exception can be used, for example, to check whether a faulty program iterates beyond the end of an array or simply to detect that the program attempted to access virtual addresses that are not committed to memory. User- and kernel-mode stacks use no-access pages as **guard pages** to detect stack overflows. Another use is to look for heap buffer overruns. Both the user-mode memory allocator and the special kernel allocator used by the device verifier can be configured to map each allocation onto the end of a page, followed by a no-access page to detect programming errors that access beyond the end of an allocation.

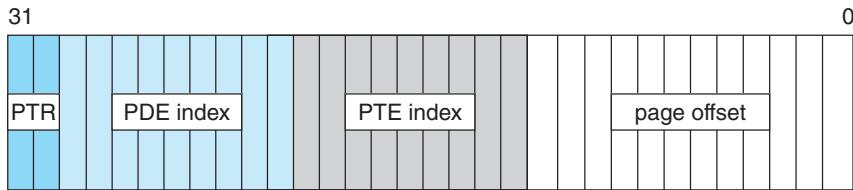
- The *copy-on-write mechanism* enables the MM to use physical memory more efficiently. When two processes want independent copies of data from the same section object, the MM places a single shared copy into virtual memory and activates the copy-on-write property for that region of memory. If one of the processes tries to modify data in a copy-on-write page, the MM makes a private copy of the page for the process.

The virtual address translation on most modern processors uses a multi-level page table. For IA-32 (operating in Physical Address Extension, or PAE, mode) and AMD64 processors, each process has a **page directory** that contains 512 **page-directory entries** (PDEs), each 8 bytes in size. Each PDE points to a **PTE table** that contains 512 **page-table entries** (PTEs), each 8 bytes in size. Each PTE points to a 4-KB **page frame** in physical memory. For a variety of reasons, the hardware requires that the page directories or PTE tables at each level of a multilevel page table occupy a single page. Thus, the number of PDEs or PTEs that fit in a page determines how many virtual addresses are translated by that page. See Figure 21.3 for a diagram of this structure.

The structure described so far can be used to represent only 1 GB of virtual address translation. For IA-32, a second page-directory level is needed, containing only four entries, as shown in the diagram. On 64-bit processors, more entries are needed. For AMD64, the processor can fill all the remaining entries in the second page-directory level and thus obtain 512 GB of virtual address space. Therefore, to support the 256 TB that are required, the processor needs a third page-directory level (called the PML4), which also has 512 entries, each pointing to the lower-level directory. As mentioned earlier, future processors announced by Intel will support 128 PB, requiring a fourth page-directory level (PML5). Thanks to this hierarchical mechanism, the total size of all page-table pages needed to fully represent a 32-bit virtual address space for a process is



**Figure 21.3** Page-table layout.



**Figure 21.4** Virtual-to-physical address translation on IA-32.

only 8 MB. Additionally, the MM allocates pages of PDEs and PTEs as needed and moves page-table pages to secondary storage when not in use, so that the actual physical memory overhead of the paging structures for each process is usually approximately 2 KB. The page-table pages are faulted back into memory when referenced.

We next consider how virtual addresses are translated into physical addresses on IA-32-compatible processors. A 2-bit value can represent the values 0, 1, 2, 3. A 9-bit value can represent values from 0 to 511; a 12-bit value, values from 0 to 4,095. Thus, a 12-bit value can select any byte within a 4-KB page of memory. A 9-bit value can represent any of the 512 PDEs or PTEs in a page directory or PTE-table page. As shown in Figure 21.4, translating a virtual address pointer to a byte address in physical memory involves breaking the 32-bit pointer into four values, starting from the most significant bits:

- Two bits are used to index into the four PDEs at the top level of the page table. The selected PDE will contain the physical page number for each of the four page-directory pages that map 1 GB of the address space.
- Nine bits are used to select another PDE, this time from a second-level page directory. This PDE will contain the physical page numbers of up to 512 PTE-table pages.
- Nine bits are used to select one of 512 PTEs from the selected PTE-table page. The selected PTE will contain the physical page number for the byte we are accessing.
- Twelve bits are used as the byte offset into the page. The physical address of the byte we are accessing is constructed by appending the lowest 12 bits of the virtual address to the end of the physical page number we found in the selected PTE.

Note that the number of bits in a physical address may be different from the number of bits in a virtual address. For example, when PAE is enabled (the only mode supported by Windows 8 and later versions), the IA-32 MMU is extended to the larger 64-bit PTE size, while the hardware supports 36-bit physical addresses, granting access to up to 64 GB of RAM, even though a single process can only map an address space up to 4 GB in size. Today, on the AMD64 architecture, server versions of Windows support very, very large physical addresses—more than we can possibly use or even buy (24 TB as of the latest release). (Of course, at one time 4 GB seemed optimistically large for physical memory.)

To improve performance, the MM maps the page-directory and PTE-table pages into the same contiguous region of virtual addresses in every process. This self-map allows the MM to use the same pointer to access the current PDE or PTE corresponding to a particular virtual address no matter what process is running. The self-map for the IA-32 takes a contiguous 8-MB region of kernel virtual address space; the AMD64 self-map occupies 512 GB. Although the self-map occupies significant address space, it does not require any additional virtual memory pages. It also allows the page table's pages to be automatically paged in and out of physical memory.

In the creation of a self-map, one of the PDEs in the top-level page directory refers to the page-directory page itself, forming a “loop” in the page-table translations. The virtual pages are accessed if the loop is not taken, the PTE-table pages are accessed if the loop is taken once, the lowest-level page-directory pages are accessed if the loop is taken twice, and so forth.

The additional levels of page directories used for 64-bit virtual memory are translated in the same way except that the virtual address pointer is broken up into even more values. For the AMD64, Windows uses four full levels, each of which maps 512 pages, or  $9 + 9 + 9 + 9 + 12 = 48$  bits of virtual address.

To avoid the overhead of translating every virtual address by looking up the PDE and PTE, processors use **translation look-aside buffer (TLB)** hardware, which contains an associative memory cache for mapping virtual pages to PTEs. The TLB is part of the **memory-management unit (MMU)** within each processor. The MMU needs to “walk” (navigate the data structures of) the page table in memory only when a needed translation is missing from the TLB.

The PDEs and PTEs contain more than just physical page numbers. They also have bits reserved for operating-system use and bits that control how the hardware uses memory, such as whether hardware caching should be used for each page. In addition, the entries specify what kinds of access are allowed for both user and kernel modes.

A PDE can also be marked to say that it should function as a PTE rather than a PDE. On a IA-32, the first 11 bits of the virtual address pointer select a PDE in the first two levels of translation. If the selected PDE is marked to act as a PTE, then the remaining 21 bits of the pointer are used as the offset of the byte. This results in a 2-MB size for the page. Mixing and matching 4-KB and 2-MB page sizes within the page table is easy for the operating system and can significantly improve the performance of some programs. The improvement results from reducing how often the MMU needs to reload entries in the TLB, since one PDE mapping 2 MB replaces 512 PTEs, each mapping 4 KB. Newer AMD64 hardware even supports 1-GB pages, which operate in a similar fashion.

Managing physical memory so that 2-MB pages are available when needed is difficult, as they may continually be broken up into 4-KB pages, causing external fragmentation of memory. Also, the large pages can result in very significant internal fragmentation. Because of these problems, it is typically only Windows itself, along with large server applications, that use large pages to improve the performance of the TLB. They are better suited to do so because operating-system and server applications start running when the system boots, before memory has become fragmented.

Windows manages physical memory by associating each physical page with one of seven states: free, zeroed, modified, standby, bad, transition, or valid.

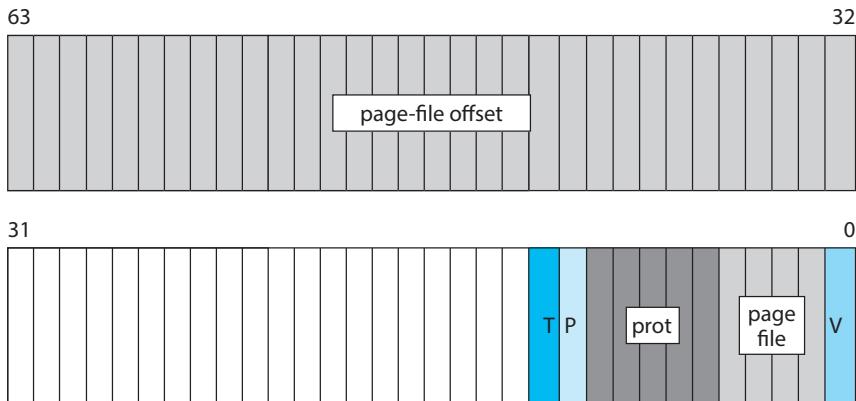
- A *free* page is an available page that has stale or uninitialized content.
- A *zeroed* page is a free page that has been zeroed out and is ready for immediate use to satisfy zero-on-demand faults.
- A *modified* page has been written by a process and must be sent to secondary storage before it is usable by another process.
- A *standby* page is a copy of information already stored on secondary storage. Standby pages may be pages that were not modified, modified pages that have already been written to secondary storage, or pages that were prefetched because they were expected to be used soon.
- A *bad* page is unusable because a hardware error has been detected.
- A *transition* page is on its way from secondary storage to a page frame allocated in physical memory.
- A *valid* page either is part of the working set of one or more processes and is contained within these processes' page tables, or is being used by the system directly (such as to store the nonpaged pool).

While valid pages are contained in processes' page tables, pages in other states are kept in separate lists according to state type. Additionally, to improve performance and protect against aggressive recycling of the standby pages, Windows Vista and later versions implement eight prioritized standby lists. The lists are constructed by linking the corresponding entries in the **page frame number (PFN)** database, which includes an entry for each physical memory page. The PFN entries also include information such as reference counts, locks, and NUMA information. Note that the PFN database represents pages of physical memory, whereas the PTEs represent pages of virtual memory.

When the valid bit in a PTE is zero, hardware ignores all the other bits, and the MM can define them for its own use. Invalid pages can have a number of states represented by bits in the PTE. Page-file pages that have never been faulted in are marked zero-on-demand. Pages mapped through section objects encode a pointer to the appropriate section object. PTEs for pages that have been written to the page file contain enough information to locate the page on secondary storage, and so forth. The structure of the page-file PTE is shown in Figure 21.5. The T, P, and V bits are all zero for this type of PTE. The PTE includes 5 bits for page protection, 32 bits for page-file offset, and 4 bits to select the paging file. There are also 20 bits reserved for additional bookkeeping.

Windows uses a per-working-set, least recently used (LRU) replacement policy to take pages from processes as appropriate. When a process is started, it is assigned a default minimum working-set size, at which point the MM starts to track the age of the pages in each working set. The working set of each process is allowed to grow until the amount of remaining physical memory starts to run low. Eventually, when the available memory runs critically low, the MM trims the working set to remove older pages.

The age of a page depends not on how long it has been in memory but on when it was last referenced. The MM makes this determination by periodically passing through the working set of each process and incrementing the age for pages that have not been marked in the PTE as referenced since the last pass. When it becomes necessary to trim the working sets, the MM uses heuristics to



**Figure 21.5** Page-file page-table entry. The valid bit is zero.

decide how much to trim from each process and then removes the oldest pages first.

A process can have its working set trimmed even when plenty of memory is available, if it was given a *hard limit* on how much physical memory it could use. In Windows 7 and later versions, the MM also trims processes that are growing rapidly, even if memory is plentiful. This policy change significantly improved the responsiveness of the system for other processes.

Windows tracks working sets not only for user-mode processes but also for various kernel-mode regions, which include the file cache and the pageable kernel heap. Pageable kernel and driver code and data have their own working sets, as does each TS session. The distinct working sets allow the MM to use different policies to trim the different categories of kernel memory.

The MM does not fault in only the page immediately needed. Research shows that the memory referencing of a thread tends to have a **locality** property. That is, when a page is used, it is likely that adjacent pages will be referenced in the near future. (Think of iterating over an array or fetching sequential instructions that form the executable code for a thread.) Because of locality, when the MM faults in a page, it also faults in a few adjacent pages. This prefetching tends to reduce the total number of page faults and allows reads to be clustered to improve I/O performance.

In addition to managing committed memory, the MM manages each process's reserved memory, or virtual address space. Each process has an associated tree that describes the ranges of virtual addresses in use and what the uses are. This allows the MM to fault in page-table pages as needed. If the PTE for a faulting address is uninitialized, the MM searches for the address in the process's tree of **virtual address descriptors (VADs)** and uses this information to fill in the PTE and retrieve the page. In some cases, a PTE-table page may not exist; such a page must be transparently allocated and initialized by the MM. In other cases, the page may be shared as part of a section object, and the VAD will contain a pointer to that section object. The section object contains information on how to find the shared virtual page so that the PTE can be initialized to point to it directly.

Starting with Vista, the Windows MM includes a component called SuperFetch. This component combines a user-mode service with specialized kernel-

mode code, including a file-system filter, to monitor all paging operations on the system. Each second, the service queries a trace of all such operations and uses a variety of agents to monitor application launches, fast user switches, standby/sleep/hibernate operations, and more as a means of understanding the system's usage patterns. With this information, it builds a statistical model, using Markov chains, of which applications the user is likely to launch when, in combination with what other applications, and what portions of these applications will be used. For example, SuperFetch can train itself to understand that the user launches Microsoft Outlook in the mornings mostly to read e-mail but composes e-mails later, after lunch. It can also understand that once Outlook is in the background, Visual Studio is likely to be launched next, and that the text editor is going to be in high demand, with the compiler demanded a little less frequently, the linker even less frequently, and the documentation code hardly ever. With this data, SuperFetch will prepopulate the standby list, making low-priority I/O reads from secondary storage at idle times to load what it thinks the user is likely to do next (or another user, if it knows a fast user switch is likely). Additionally, by using the eight prioritized standby lists that Windows offers, each such prefetched paged can be cached at a level that matches the statistical likelihood that it will be needed. Thus, unlikely-to-be-demanded pages can cheaply and quickly be evicted by an unexpected need for physical memory, while likely-to-be-demanded-soon pages can be kept in place for longer. Indeed, SuperFetch may even force the system to trim working sets of other processes before touching such cached pages.

SuperFetch's monitoring does create considerable system overhead. On mechanical (rotational) drives, which have seek times in the milliseconds, this cost is balanced by the benefit of avoiding latencies and multisecond delays in application launch times. On server systems, however, such monitoring is not beneficial, given the random multiuser workloads and the fact that throughput is more important than latency. Further, the combined latency improvements and bandwidth on systems with fast, efficient nonvolatile memory, such as SSDs, make the monitoring less beneficial for those systems as well. In such situations, SuperFetch disables itself, freeing up a few spare CPU cycles.

Windows 10 brings another large improvement to the MM by introducing a component called the compression store manager. This component creates a compressed store of pages in the working set of the **memory compression process**, which is a type of system process. When shareable pages go on the standby list and available memory is low (or certain other internal algorithm decisions are made), pages on the list will be compressed instead of evicted. This can also happen to modified pages targeted for eviction to secondary storage—both by reducing memory pressure, perhaps avoiding the write in the first place, and by causing the written pages to be compressed, thus consuming less page file space and taking less I/O to page out. On today's fast multiprocessor systems, often with built-in hardware compression algorithms, the small CPU penalty is highly preferable to the potential secondary storage I/O cost.

#### 21.3.5.3 Process Manager

The Windows process manager provides services for creating, deleting, interrogating, and managing processes, threads, and jobs. It has no knowledge

about parent–child relationships or process hierarchies, although it can group processes in jobs, and the latter can have hierarchies that must then be maintained. The process manager is also not involved in the scheduling of threads, other than setting the priorities and affinities of the threads in their owner processes. Additionally, through jobs, the process manager can effect various changes in scheduling attributes (such as throttling ratios and quantum values) on threads. Thread scheduling proper, however, takes place in the kernel dispatcher.

Each process contains one or more threads. Processes themselves can be collected into larger units called **job objects**. The original use of job objects was to place limits on CPU usage, working-set size, and processor affinities that control multiple processes at once. Job objects were thus used to manage large data-center machines. In Windows XP and later versions, job objects were extended to provide security-related features, and a number of third-party applications such as Google Chrome began using jobs for this purpose. In Windows 8, a massive architectural change allowed jobs to influence scheduling through generic CPU throttling as well as per-user-session-aware fairness throttling/balancing. In Windows 10, throttling support was extended to secondary storage I/O and network I/O as well. Additionally, Windows 8 allowed job objects to nest, creating hierarchies of limits, ratios, and quotas that the system must accurately compute. Additional security and power management features were given to job objects as well.

As a result, all Windows Store applications and all UWP application processes run in jobs. The DAM, introduced earlier, implements Connected Standby support using jobs. Finally, Windows 10’s support for Docker Containers, a key part of its cloud offerings, uses job objects, which it calls **silos**. Thus, jobs have gone from being an esoteric data-center resource management feature to a core mechanism of the process manager for multiple features.

Due to Windows’s layered architecture and the presence of environment subsystems, process creation is quite complex. An example of process creation in the Win32 environment under Windows 10 is as follows. Note that the launching of UWP “Modern” Windows Store applications (which are called **packaged applications**, or “AppX”) is significantly more complex and involves factors outside the scope of this discussion.

1. A Win32 application calls `CreateProcess()`.
2. A number of parameter conversions and behavioral conversions are done from the Win32 world to the NT world.
3. `CreateProcess()` then calls the `NtCreateUserProcess()` API in the process manager of the NT executive to actually create the process and its initial thread.
4. The process manager calls the object manager to create a process object and returns the object handle to Win32. It then calls the memory manager to initialize the address space of the new process, its handle table, and other key data structures, such as the process environment block (PEBL) (which contains internal process management data).

5. The process manager calls the object manager again to create a thread object and returns the handle to Win32. It then calls the memory manager to create the thread environment block (TEB) and the dispatcher to initialize the scheduling attributes of the thread, setting its state to initializing.
6. The process manager creates the initial thread startup context (which will eventually point to the `main()` routine of the application), asks the scheduler to mark the thread as ready, and then immediately suspends it, putting it into a waiting state.
7. A message is sent to the Win32 subsystem to notify it that the process is being created. The subsystem performs additional Win32-specific work to initialize the process, such as computing its shutdown level and drawing the animated hourglass or “donut” mouse cursor.
8. Back in `CreateProcess()`, inside the parent process, the `ResumeThread()` API is called to wake up the process’s initial thread. Control returns to the parent.
9. Now, inside the initial thread of the new process, the user-mode link loader takes control (inside `ntdll.dll`, which is automatically mapped into all processes). It loads all the library dependencies (DLLs) of the application, creates its initial heap, sets up exception handling and application compatibility options, and eventually calls the `main()` function of the application.

The Windows APIs for manipulating virtual memory and threads and for duplicating handles take a process handle, so their subsystem and other services, when notified of process creation, can perform operations on behalf of the new process without having to execute directly in the new process’s context. Windows also supports a UNIX `fork()` style of process creation. A number of features—including [process reflection](#), which is used by the Windows error reporting (WER) infrastructure during process crashes, as well as the Windows subsystem for Linux’s implementation of the Linux `fork()` API—depend on this capability.

The debugger support in the process manager includes the APIs to suspend and resume threads and to create threads that begin in suspended mode. There are also process-manager APIs that get and set a thread’s register context and access another process’s virtual memory. Threads can be created in the current process; they can also be injected into another process. The debugger makes use of thread injection to execute code within a process being debugged. Unfortunately, the ability to allocate, manipulate, and inject both memory and threads across processes is often misused by malicious programs.

While running in the executive, a thread can temporarily attach to a different process. [Thread attach](#) is used by kernel worker threads that need to execute in the context of the process originating a work request. For example, the MM might use thread attach when it needs access to a process’s working set or page tables, and the I/O manager might use it in updating the status variable in a process for asynchronous I/O operations.

#### 21.3.5.4 Facilities for Client–Server Computing

Like many other modern operating systems, Windows uses a client–server model throughout, primarily as a layering mechanism, which allows putting common functionality into a “service” (the equivalent of a daemon in UNIX terms), as well as splitting out content-parsing code (such as a PDF reader or Web browser) from system-action-capable code (such as the Web browser’s capability to save a file on secondary storage or the PDF reader’s ability to print out a document). For example, on a recent Windows 10 operating system, opening the New York Times website with the Microsoft Edge browser will likely result in 12 to 16 different processes in a complex organization of “broker,” “renderer/parser,” “JITTer,” services, and clients.

The most basic such “server” on a Windows computer is the Win32 environment subsystem, which is the server that implements the operating-system personality of the Win32 API inherited from the Windows 95/98 days. Many other services, such as user authentication, network facilities, printer spooling, Web services, network file systems, and plug-and-play, are also implemented using this model. To reduce the memory footprint, multiple services are often collected into a few processes running the `svchost.exe` program. Each service is loaded as a dynamic-link library (DLL), which implements the service by relying on user-mode thread-pool facilities to share threads and wait for messages (see Section 21.3.5.3). Unfortunately, this pooling originally resulted in poor user experience in troubleshooting and debugging runaway CPU usage and memory leaks, and it weakened the overall security of each service. Therefore, in recent versions of Windows 10, if the system has over 2 GB of RAM, each DLL service runs in its own individual `svchost.exe` process.

In Windows, the recommended paradigm for implementing client–server computing is to use RPCs to communicate requests, because of their inherent security, serialization services, and extensibility features. The Win32 API supports the Microsoft standard of the DCE-RPC protocol, called MS-RPC, as described in Section 21.6.2.7.

RPC uses multiple transports (for example, named pipes and TCP/IP) that can be used to implement RPCs between systems. When an RPC occurs only between a client and a server on the local system, ALPC can be used as the transport. Furthermore, because RPC is heavyweight and has multiple system-level dependencies (including the WINXXIII environment subsystem itself), many native Windows services, as well as the kernel, directly use ALPC, which is not available (nor suitable) for third-party programmers.

ALPC is a message-passing mechanism similar to UNIX domain sockets and Mach IPC. The server process publishes a globally visible connection-port object. When a client wants services from the server, it opens a handle to the server’s connection-port object and sends a connection request to the port. If the server accepts the connection, then ALPC creates a pair of communication-port objects, providing the client’s connect API with its handle to the pair, and then providing the server’s accept API with the other handle to the pair.

At this point, messages can be sent across communication ports as either datagrams, which behave like UDP and require no reply, or requests, which must receive a reply. The client and server can then use either synchronous messaging, in which one side is always blocking (waiting for a request or expecting a reply), or asynchronous messaging, in which the thread-pool

mechanism can be used to perform work whenever a request or reply is received, without the need for a thread to block for a message. For servers located in kernel mode, communication ports also support a callback mechanism, which allows an immediate switch to the kernel side (KT) of the user-mode thread (UT), immediately executing the server's handler routine.

When an ALPC message is sent, one of two message-passing techniques can be chosen.

1. The first technique is suitable for small to medium-sized messages (below 64 KB). In this case, the port's kernel message queue is used as intermediate storage, and the messages are copied from one process, to the kernel, to the other process. The disadvantage of this technique is the double buffering, as well as the fact that messages remain in kernel memory until the intended receiver consumes them. If the receiver is highly contended or currently unavailable, this may result in megabytes of kernel-mode memory being locked up.
2. The second technique is for larger messages. In this case, a shared-memory section object is created for the port. Messages sent through the port's message queue contain a "message attribute," called a [data view attribute](#), that refers to the section object. The receiving side "exposes" this attribute, resulting in a virtual address mapping of the section object and a sharing of physical memory. This avoids the need to copy large messages or to buffer them in kernel-mode memory. The sender places data into the shared section, and the receiver sees them directly, as soon as it consumes a message.

Many other possible ways of implementing client–server communication exist, such as by using mailslots, pipes, sockets, section objects paired with events, window messages, and more. Each one has its uses, benefits, and disadvantages. RPC and ALPC remain the most fully featured, safe, secure, and feature-rich mechanisms for such communication, however, and they are the mechanisms used by the vast majority of Windows processes and services.

#### 21.3.5.5 I/O Manager

The [I/O manager](#) is responsible for all device drivers on the system, as well as for implementing and defining the communication model that allows drivers to communicate with each other, with the kernel, and with user-mode clients and consumers. Additionally, as in UNIX-based operating systems, I/O is always targeted to a [file object](#), even if the device is not a file system. The I/O manager in Windows allows device drivers to be "filtered" by other drivers, creating a [device stack](#) through which I/O flows and which can be used to modify, extend, or enhance the original request. Therefore, the I/O manager always keeps track of which device drivers and filter drivers are loaded.

Due to the importance of file-system drivers, the I/O manager has special support for them and implements interfaces for loading and managing file systems. It works with the MM to provide memory-mapped file I/O and controls the Windows cache manager, which handles caching for the entire I/O system. The I/O manager is fundamentally asynchronous, providing synchronous I/O by explicitly waiting for an I/O operation to complete. The I/O manager

provides several models of asynchronous I/O completion, including setting of events, updating of a status variable in the calling process, delivery of APCs to initiating threads, and use of I/O completion ports, which allow a single thread to process I/O completions from many other threads. It also manages buffers for I/O requests.

Device drivers are arranged in a list for each device (called a driver or I/O stack). A driver is represented in the system as a **driver object**. Because a single driver can operate on multiple devices, the drivers are represented in the I/O stack by a **device object**, which contains a link to the driver object. Additionally, nonhardware drivers can use device objects as a way to expose different interfaces. As an example, there are TCP6, UDP6, UDP, TCP, RawIp, and RawIp6 device objects owned by the TCP/IP driver object, even though these don't represent physical devices. Similarly, each volume on secondary storage is its own device object, owned by the volume manager driver object.

Once a handle is opened to a device object, the I/O manager always creates a file object and returns a file handle instead of a device handle. It then converts the requests it receives (such as create, read, and write) into a standard form called an **I/O request packet (IRP)**. It forwards the IRP to the first driver in the targeted I/O stack for processing. After a driver processes the IRP, it calls the I/O manager either to forward the IRP to the next driver in the stack or, if all processing is finished, to complete the operation represented by the IRP.

The I/O request may be completed in a context different from the one in which it was made. For example, if a driver is performing its part of an I/O operation and is forced to block for an extended time, it may queue the IRP to a worker thread to continue processing in the system context. In the original thread, the driver returns a status indicating that the I/O request is pending so that the thread can continue executing in parallel with the I/O operation. An IRP may also be processed in interrupt-service routines and completed in an arbitrary process context. Because some final processing may need to take place in the context that initiated the I/O, the I/O manager uses an APC to do final I/O-completion processing in the process context of the originating thread.

The I/O stack model is very flexible. As a driver stack is built, various drivers have the opportunity to insert themselves into the stack as **filter drivers**. Filter drivers can examine and potentially modify each I/O operation. Volume snapshotting (**shadow copies**) and disk encryption (**BitLocker**) are two built-in examples of functionality implemented using filter drivers that execute above the volume manager driver in the stack. File-system filter drivers execute above the file system and have been used to implement functionalities such as hierarchical storage management, single instancing of files for remote boot, and dynamic format conversion. Third parties also use file-system filter drivers to implement anti-malware tools. Due to the large number of file-system filters, Windows Server 2003 and later versions now include a **filter manager** component, which acts as the sole file-system filter and which loads **minifilter** ordered by specific **altitudes** (relative priorities). This model allows filters to transparently cache data and repeated queries without having to know about each other's requests. It also provides stricter load ordering.

Device drivers for Windows are written to the Windows Driver Model (WDM) specification. This model lays out all the requirements for device drivers, including how to layer filter drivers, share common code for han-

dling power and plug-and-play requests, build correct cancellation logic, and so forth.

Because of the richness of the WDM, writing a full WDM device driver for each new hardware device can involve a great deal of work. In some cases, the port/miniport model makes it unnecessary to do this for certain hardware devices. Within a range of devices that require similar processing, such as audio drivers, storage controllers, or Ethernet controllers, each instance of a device shares a common driver for that class, called a **port driver**. The port driver implements the standard operations for the class and then calls device-specific routines in the device's **miniport driver** to implement device-specific functionality. The physical-link layer of the network stack is implemented in this way, with the `ndis.sys` port driver implementing much of the generic network processing functionality and calling out to the network miniport drivers for specific hardware commands related to sending and receiving network frames (such as Ethernet).

Similarly, the WDM includes a class/miniclass model. Here, a certain class of devices can be implemented in a generic way by a single class driver, with callouts to a miniclass for specific hardware functionality. For example, the Windows disk driver is a class driver, as are drivers for CD/DVDs and tape drives. The keyboard and mouse driver are class drivers as well. These types of devices don't need a miniclass, but the battery class driver, for example, does require a miniclass for each of the various external uninterruptible power supplies (UPSs) sold by vendors.

Even with the port/miniport and class/miniclass model, significant kernel-facing code must be written. And this model is not useful for custom hardware or for logical (nonhardware) drivers. Starting with Windows 2000 Service Pack 4, kernel-mode drivers can be written using the **Kernel-Mode Driver Framework (KMDF)**, which provides a simplified programming model for drivers on top of WDM. Another option is the **User-Mode Driver Framework (UMDF)**, which allows drivers to be written in user mode through a **reflecto** driver in the kernel that forwards the requests through the kernel's I/O stack. These two frameworks make up the **Windows Driver Foundation** model, which has reached Version 2.1 in Windows 10 and contains a fully compatible API between KMDF and UMDF. It has been fully open-sourced on GitHub.

Because many drivers do not need to operate in kernel mode, and it is easier to develop and deploy drivers in user mode, UMDF is strongly recommended for new drivers. It also makes the system more reliable, because a failure in a user-mode driver does not cause a kernel (system) crash.

### 21.3.5.6 Cache Manager

In many operating systems, caching is done by the block device system, usually at the physical/block level. Instead, Windows provides a centralized caching facility that operates at the logical/virtual file level. The **cache manager** works closely with the MM to provide cache services for all components under the control of the I/O manager. This means that the cache can operate on anything from remote files on a network share to logical files on a custom file system. The size of the cache changes dynamically according to how much free memory is available in the system; it can grow as large as 2 TB on a 64-bit system.

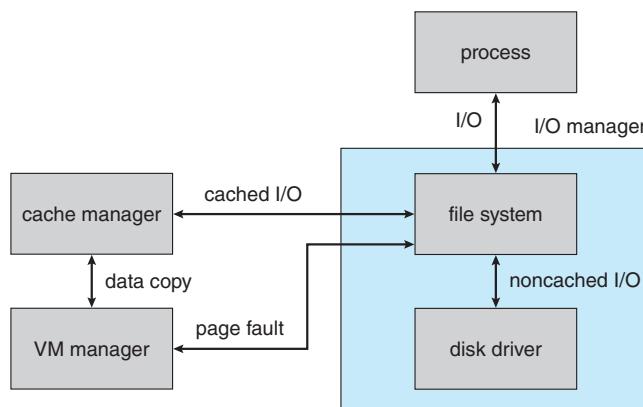
The cache manager maintains a private working set rather than sharing the system process's working set, which allows trimming to page out cached files more effectively. To build the cache, the cache manager memory-maps files into kernel memory and then uses special interfaces to the MM to fault pages into or trim them from this private working set, which lets it take advantage of additional caching facilities provided by the memory manager.

The cache is divided into blocks of 256 KB. Each cache block can hold a view (that is, a memory-mapped region) of a file. Each cache block is described by a **virtual address control block (VACB)** that stores the virtual address and file offset for the view, as well as the number of processes using the view. The VACBs reside in arrays maintained by the cache manager, and there are arrays for critical as well as low-priority cached data to improve performance in situations of memory pressure.

When the I/O manager receives a file's user-level read request, the I/O manager sends an IRP to the I/O stack for the volume on which the file resides. For files that are marked as cacheable, the file system calls the cache manager to look up the requested data in its cached file views. The cache manager calculates which entry of that file's VACB index array corresponds to the byte offset of the request. The entry either points to the view in the cache or is invalid. If it is invalid, the cache manager allocates a cache block (and the corresponding entry in the VACB array) and maps the view into the cache block. The cache manager then attempts to copy data from the mapped file to the caller's buffer. If the copy succeeds, the operation is completed.

If the copy fails, it does so because of a page fault, which causes the MM to send a noncached read request to the I/O manager. The I/O manager sends another request down the driver stack, this time requesting a paging operation, which bypasses the cache manager and reads the data from the file directly into the page allocated for the cache manager. Upon completion, the VACB is set to point at the page. The data, now in the cache, are copied to the caller's buffer, and the original I/O request is completed. Figure 21.6 shows an overview of these operations.

When possible, for synchronous operations on cached files, I/O is handled by the **fast I/O mechanism**. This mechanism parallels the normal IRP-based I/O



**Figure 21.6** File I/O.

but calls into the driver stack directly rather than passing down an IRP, which saves memory and time. Because no IRP is involved, the operation should not block for an extended period of time and cannot be queued to a worker thread. Therefore, when the operation reaches the file system and calls the cache manager, the operation fails if the information is not already in the cache. The I/O manager then attempts the operation using the normal IRP path.

A kernel-level read operation is similar, except that the data can be accessed directly from the cache rather than being copied to a buffer in user space. To use file-system metadata (data structures that describe the file system), the kernel uses the cache manager's mapping interface to read the metadata. To modify the metadata, the file system uses the cache manager's pinning interface. **Pinning** a page locks the page into a physical-memory page frame so that the MM manager cannot move the page or page it out. After updating the metadata, the file system asks the cache manager to unpin the page. A modified page is marked dirty, and so the MM flushes the page to secondary storage.

To improve performance, the cache manager keeps a small history of read requests and from this history attempts to predict future requests. If the cache manager finds a pattern in the previous three requests, such as sequential access forward or backward, it prefetches data into the cache before the next request is submitted by the application. In this way, the application may find its data already cached and not need to wait for secondary storage I/O.

The cache manager is also responsible for telling the MM to flush the contents of the cache. The cache manager's default behavior is write-back caching: it accumulates writes for 4 to 5 seconds and then wakes up the cache-writer thread. When write-through caching is needed, a process can set a flag when opening the file, or can call an explicit cache-flush function.

A fast-writing process could potentially fill all the free cache pages before the cache-writer thread had a chance to wake up and flush the pages to secondary storage. The cache writer prevents a process from flooding the system in the following way. When the amount of free cache memory becomes low, the cache manager temporarily blocks processes attempting to write data and wakes the cache-writer thread to flush pages to secondary storage. If the fast-writing process is actually a network redirector for a network file system, blocking it for too long could cause network transfers to time out and be retransmitted. This retransmission would waste network bandwidth. To prevent such waste, network redirectors can instruct the cache manager to limit the backlog of writes in the cache.

Because a network file system needs to move data between secondary storage and the network interface, the cache manager also provides a DMA interface to move the data directly. Moving data directly avoids the need to copy data through an intermediate buffer.

#### 21.3.5.7 Security Reference Monitor

Centralizing management of system entities in the object manager enables Windows to use a uniform mechanism to perform run-time access validation and audit checks for every user-accessible entity in the system. Additionally, even entities not managed by the object manager may have access to the API routines for performing security checks. Whenever a thread opens a handle to a protected data structure (such as an object), the **security reference monitor**

(**SRM**) checks the effective security token and the object's security descriptor, which contains two access-control lists—the discretionary access control list (DACL) and the system access control list (SACL)—to see whether the process has the necessary access rights. The effective security token is typically the token of the thread's process, but it can also be the token of the thread itself, as described below.

Each process has an associated **security token**. When the login process (`lsass.exe`) authenticates a user, the security token is attached to the user's first process (`userinit.exe`) and copied for each of its child processes. The token contains the **security identity (SID)** of the user, the SIDs of the groups the user belongs to, the privileges the user has, the integrity level of the process, the attributes and claims associated with the user, and any relevant capabilities. By default, threads don't have their own explicit tokens, causing them to share the common token of the process. However, using a mechanism called **impersonation**, a thread running in a process with a security token belonging to one user can set a thread-specific token belonging to another user to impersonate that user. At this point, the effective token becomes the token of the thread, and all operations, quotas, and limitations are subject to that user's token. The thread can later choose to “revert” to its old identity by removing the thread-specific token, so that the effective token is once again that of the process.

This impersonation facility is fundamental to the client–server model, where services must act on behalf of a variety of clients with different security IDs. The right to impersonate a user is most often delivered as part of a connection from a client process to a server process. Impersonation allows the server to access system services as if it were the client in order to access or create objects and files on behalf of the client. The server process must be trustworthy and must be carefully written to be robust against attacks. Otherwise, one client could take over a server process and then impersonate any user who made a subsequent client request. Windows provides APIs to support impersonation at the ALPC (and thus RPC and DCOM) layer, the named pipe layer, and the Winsock layer.

The SRM is also responsible for manipulating the privileges in security tokens. Special privileges are required for users to change the system time, load a driver, or change firmware environment variables. Additionally, certain users can have powerful privileges that override default access control rules. These include users who must perform backup or restore operations on file systems (allowing them to bypass read/write restrictions), debug processes (allowing them to bypass security features), and so forth.

The integrity level of the code executing in a process is also represented by a token. Integrity levels are a type of mandatory labeling mechanism, as mentioned earlier. By default, a process cannot modify an object with an integrity level higher than that of the code executing in the process, whatever other permissions have been granted. In addition, it cannot read from another process object at a higher integrity level. Objects can also protect themselves from read access by manually changing the mandatory policy associated with their security descriptor. Inside an object (such as a file or a process), the integrity level is stored in the SACL, which distinguishes it from typical discretionary user and group permissions, stored in the DACL.

Integrity levels were introduced to make it harder for code to take over a system by attacking external-content-parsing software, like a browser or

PDF reader, because such software is expected to run at a low integrity level. For example, Microsoft Edge runs at “low integrity,” as do Adobe Reader and Google Chrome. A regular application, such as Microsoft Word, runs at “medium integrity.” Finally, you can expect an application run by an administrator or a setup program to run at “high integrity.”

Creating applications to run at lower integrity levels places a burden on the developers to implement this security feature, because they must create a client–server model to support a broker and parser or renderer, as mentioned earlier. In order to streamline this security model, Windows 8 introduced the [Application Container](#), often just called “AppContainer,” which is a special extension of the token object. When running under an AppContainer, an application automatically has its process token adjusted in the following ways:

1. The token’s integrity level is set to low. This means that the application cannot write to or modify most objects (files, keys, processes) on the system, nor can it read from any other process on the system.
2. All groups and the user SID are disabled (ignored) in the token. Let’s say that the application was launched by user Anne, who belongs to the World group. Any files accessible to Anne or World will be inaccessible to this application.
3. All privileges except a handful are removed from the token. This prevents powerful system calls or system-wide operations from being permitted.
4. A special AppContainer SID is added to the token, which corresponds to the SHA-256 hash of the application’s package identifier. This is the only valid security identifier in the token, so any object wishing to be directly accessible to this application needs to explicitly give the AppContainer SID read or write access.
5. A set of capability SIDs are added to the token, based on the application’s manifest file. When the application is first installed, these capabilities are shown to the user, who must agree to them before the application is deployed.

We can see that the AppContainer mechanism changes the security model from a discretionary system where access to protected resources is defined by users and groups to a mandatory system where each application has its own unique security identity and access occurs on a per-application basis. This separation of privileges and permissions is a great leap forward in security, but it places a potential burden on resource access. Capabilities and brokers help to alleviate this burden.

Capabilities are used by system brokers implemented by Windows to perform various actions on behalf of packaged applications. For example, assume that Harold’s packaged application has no access to Harold’s file system, since the Harold SID is disabled. In this situation, a broker might check for the `Play User Media` capability and allow the music player process to read any MP3 files located in Harold’s My Music directory. Thus, Harold will not be forced to mark all of his files with the AppContainer SID of his favorite media player application, as long as the application has the `Play User Media` capability and Harold agreed to it when he downloaded the application.

A final responsibility of the SRM is logging security audit events. The ISO standard **Common Criteria** (the international successor to the Orange Book standard developed by the United States Department of Defense) requires that a secure system have the ability to detect and log all attempts to access system resources so that it can more easily trace attempts at unauthorized access. Because the SRM is responsible for making access checks, it generates most of the audit records, which are then written by `lsass.exe` into the security-event log.

#### 21.3.5.8 Plug-and-Play Manager

The operating system uses the **plug-and-play (PnP) manager** to recognize and adapt to changes in hardware configuration. PnP devices use standard protocols to identify themselves to the system. The PnP manager automatically recognizes installed devices and detects changes in devices as the system operates. The manager also keeps track of hardware resources used by a device, as well as potential resources that could be used, and takes care of loading the appropriate drivers. This management of hardware resources—primarily interrupts, DMA channels, and I/O memory ranges—has the goal of determining a hardware configuration in which all devices are able to operate successfully. The PnP manager and the Windows Driver Model see drivers as either **bus drivers**, which detect and enumerate the devices on a bus (such as PCI or USB), or **function drivers**, which implement the functionality of a particular device on the bus.

The PnP manager handles dynamic reconfiguration as follows. First, it gets a list of devices from each bus driver. It loads the drivers and sends an **add-device** request to the appropriate driver for each device. Working in tandem with special **resource arbiters** owned by the various bus drivers, the PnP manager then figures out the optimal resource assignments and sends a **start-device** request to each driver specifying the resource assignments for the related devices. If a device needs to be reconfigured, the PnP manager sends a **query-stop** request, which asks the driver whether the device can be temporarily disabled. If the driver can disable the device, then all pending operations are completed, and new operations are prevented from starting. Finally, the PnP manager sends a **stop** request and can then reconfigure the device with a new **start-device** request.

The PnP manager also supports other requests. For example, **query-remove**, which operates similarly to **query-stop**, is employed when a user is getting ready to eject a removable device, such as a USB storage device. The **surprise-remove** request is used when a device fails or, more often, when a user removes a device without telling the system to stop it first. Finally, the **remove** request tells the driver to stop using a device permanently.

Many programs in the system are interested in the addition or removal of devices, so the PnP manager supports notifications. Such a notification, for example, gives the file manager the information it needs to update its list of secondary storage volumes when a new storage device is attached or removed.

Installing devices can also result in starting new services on the system. Previously, such services frequently set themselves up to run whenever the system booted and continued to run even if the associated device was never plugged into the system, because they had to be running in order to receive the

PnP notification. Windows 7 introduced a **service-trigger** mechanism in the **service control manager (SCM)** (`services.exe`), which manages the system services. With this mechanism, services can register themselves to start only when SCM receives a notification from the PnP manager that the device of interest has been added to the system.

### 21.3.5.9 Power Manager

Windows works with the hardware to implement sophisticated strategies for energy efficiency, as described in Section 21.2.8. The policies that drive these strategies are implemented by the **power manager**. The power manager detects current system conditions, such as the load on CPUs or I/O devices, and improves energy efficiency by reducing the performance and responsiveness of the system when need is low. The power manager can also put the entire system into a very efficient *sleep* mode and can even write all the contents of memory to secondary storage and turn off the power to allow the system to go into *hibernation*.

The primary advantage of sleep is that the system can enter that state fairly quickly, perhaps just a few seconds after the lid closes on a laptop. The return from sleep is also fairly quick. The power is turned down to a low level on the CPUs and I/O devices, but the memory continues to be powered enough that its contents are not lost. As noted earlier, however, on mobile devices, these few seconds still add up to an unreasonable user experience, so the power manager works with the Desktop Activity Moderator to kick off the Connected Standby state as soon as the screen is turned off. Connected Standby virtually freezes the computer but does not really put the computer to sleep.

Hibernation takes considerably longer to enter than sleep because the entire contents of memory must be transferred to secondary storage before the system is turned off. However, the fact that the system is, in fact, turned off is a significant advantage. If there is a loss of power to the system, as when the battery is swapped on a laptop or a desktop system is unplugged, the saved system data will not be lost. Unlike shutdown, hibernation saves the currently running system so a user can resume where she left off. Furthermore, because hibernation does not require power, a system can remain in hibernation indefinitely. Therefore, this feature is extremely useful on desktops and server systems, and it is also used on laptops when the battery hits a critical level (because putting the system to sleep when the battery is low might result in the loss of all data if the battery runs out of power while in the sleep state).

In Windows 7, the power manager also includes a processor power manager (PPM), which specifically implements strategies such as core parking, CPU throttling and boosting, and more. In addition, Windows 8 introduced the **power framework (PoFX)**, which works with function drivers to implement specific functional power states. This means that devices can expose their internal power management (clock speeds, current/power draws, and so forth) to the system, which can then use the information for fine-grained control of the devices. Thus, for example, instead of simply turning a device on or off, the system can turn specific components on or off.

Like the PnP manager, the power manager provides notifications to the rest of the system about changes in the power state. Some applications want to know when the system is about to be shut down so they can start saving their

states to secondary storage, and, as mentioned earlier, the DAM needs to know when the screen is turned off and on again.

#### 21.3.5.10 Registry

Windows keeps much of its configuration information in internal repositories of data, called **hives**, that are managed by the Windows configuration manager, commonly known as the **registry**. The configuration manager is implemented as a component of the executive.

There are separate hives for system information, each user's preferences, software information, security, and boot options. Additionally, as part of the new application and security model introduced by AppContainers and UWPModern/Metro packaged applications in Windows 8, each such application has its own separate hive, called an application hive.

The registry represents the configuration state in each hive as a hierarchical namespace of keys (directories), each of which can contain a set of arbitrarily sized values. In the Win32 API, these values have a specific "type," such as UNICODE string, 32-bit integer, or untyped binary data, but the registry itself treats all values the same, leaving it up to the higher API layers to infer a structure based on type and size. Therefore, for example, nothing prevents a "32-bit integer" from being a 999-byte UNICODE string.

In theory, new keys and values are created and initialized as new software is installed, and then they are modified to reflect changes in the configuration of that software. In practice, the registry is often used as a general-purpose database, as an interprocess-communication mechanism, and for many other such inventive purposes.

Restarting applications, or even the system, every time a configuration change was made would be a nuisance. Instead, programs rely on various kinds of notifications, such as those provided by the PnP and power managers, to learn about changes in the system configuration. The registry also supplies notifications; threads can register to be notified when changes are made to some part of the registry. The threads can thus detect and adapt to configuration changes recorded in the registry. Furthermore, registry keys are objects managed by the object manager, and they expose an event object to the dispatcher. This allows threads to put themselves in a waiting state associated with the event, which the configuration manager will signal if the key (or any of its values) is ever modified.

Whenever significant changes are made to the system, such as when updates to the operating system or drivers are installed, there is a danger that the configuration data may be corrupted (for example, if a working driver is replaced by a nonworking driver or an application fails to install correctly and leaves partial information in the registry). Windows creates a **system restore point** before making such changes. The restore point contains a copy of the hives before the change and can be used to return to this version of the hives in order to get a corrupted system working again.

To improve the stability of the registry configuration, the registry also implements a variety of "self-healing" algorithms, which can detect and fix certain cases of registry corruption. Additionally, the registry internally uses a two-phase commit transactional algorithm, which prevents corruption to individual keys or values as they are being updated. While these mechanisms

guarantee the integrity of small portions of the registry or individual keys and values, they have not supplanted the system restore facility for recovering from damage to the registry configuration caused by a failure during a software installation.

### 21.3.5.11 Booting

The booting of a Windows PC begins when the hardware powers on and firmware begins executing from ROM. In older machines, this firmware was known as the BIOS, but more modern systems use UEFI (the Unified Extensible Firmware Interface), which is faster, is more modern, and makes better use of the facilities in contemporary processors. Additionally, UEFI includes a feature called **Secure Boot** that provides integrity checks through digital signature verification of all firmware and boot-time components. This digital signature check guarantees that only Microsoft's boot-time components and the vendor's firmware are present at boot time, preventing any early third-party code from loading.

The firmware runs **power-on self-test (POST)** diagnostics, identifies many of the devices attached to the system and initializes them to a clean power-up state, and then builds the description used by ACPI. Next, the firmware finds the system boot device, loads the Windows boot manager program (`bootmgfw.efi` on UEFI systems), and begins executing it.

In a machine that has been hibernating, the `winresume.efi` program is loaded next. It restores the running system from secondary storage, and the system continues execution at the point it had reached right before hibernating. In a machine that has been shut down, `bootmgfw.efi` performs further initialization of the system and then loads `winload.efi`. This program loads `hal.dll`, the kernel (`ntoskrnl.exe`) and its dependencies, and any drivers needed in booting, and the system hive. `winload` then transfers execution to the kernel.

The procedure is somewhat different on Windows 10 systems where Virtual Secure Mode is enabled (and the hypervisor is turned on). Here, `winload.efi` will instead load `hvloader.exe` or `hvloader.dll`, which initializes the hypervisor first. On Intel systems, this is `hvix64.exe`, while AMD systems use `hvax64.exe`. The hypervisor then sets up VTL 1 (the Secure World) and VTL 0 (the Normal World) and returns to `winload.efi`, which now loads the secure kernel (`securekernel.exe`) and its dependencies. Then the secure kernel's entry point is called, which initializes VTL 1, after which it returns back to the loader at VTL 0, which resumes with the steps described above.

As the kernel initializes itself, it creates several processes. The **idle process** serves as the container of all idle threads, so that system-wide CPU idle time can easily be computed. The **system process** contains all of the internal kernel worker threads and other system threads created by drivers for polling, housekeeping, and other background work. The memory compression process, new to Windows 10, has a working set composed of compressed standby pages used by the store manager to alleviate system pressure and optimize paging. Finally, if VSM is enabled, the **secure system process** represents the fact that the secure kernel is loaded.

The first user-mode process, which is also created by the kernel, is **session manager subsystem (SMSS)**, which is similar to the `init` (initialization)

process in UNIX. SMSS performs further initialization of the system, including establishing the paging files and creating the initial user sessions. Each session represents a logged-on user, except for *session 0*, which is used to run system-wide background processes, such as lsass and services. Each session is given its own instance of an SMSS process, which exits once the session is created. In each of these sessions, this ephemeral SMSS loads the Win32 environment subsystem (`csrss.exe`) and its driver (`win32k.sys`). Then, in each session other than 0, SMSS runs the `winlogon` process, which launches `logonui`. This process captures user credentials in order for `lsass` to log in a user, then launch the `userinit` and `explorer` process, which implements the Windows shell (start menu, desktop, tray icons, notification center, and so on). The following list itemizes some of these aspects of booting:

- SMSS completes system initialization and then starts up one SMSS for session 0 and one SMSS for the first login session (1).
- `wininit` runs in session 0 to initialize user mode and start `lsass` and `services`.
- `lsass`, the security subsystem, implements facilities such as authentication of users. If user credentials are protected by VSM through Credential Guard, then `lsaiso` and `bioiso` are also started as VTL 1 Trustlets by `lsass`.
- `services` contains the service control manager, or SCM, which supervises all background activities in the system, including user-mode services. A number of services will have registered to start when the system boots. Others will be started only on demand or when triggered by an event such as the arrival of a device.
- `csrss` is the Win32 environment subsystem process. It is started in every session—mainly because it handles mouse and keyboard input, which needs to be separated per user.
- `winlogon` is run in each Windows session other than session 0 to log on a user by launching `logonui`, which presents the logon user interface.

Starting with Windows XP, the system optimizes the boot process by prefetching pages from files on secondary storage based on previous boots of the system. Disk access patterns at boot are also used to lay out system files on disk to reduce the number of I/O operations required. Windows 7 reduced the processes necessary to start the system by allowing services to start only when needed, rather than at system start-up. Windows 8 further reduced boot time by parallelizing all driver loads through a pool of worker threads in the PnP subsystem and by supporting UEFI to make boot-time transition more efficient. All of these approaches contributed to a dramatic reduction in system boot time, but eventually little further improvement was possible.

To address boot-time concerns, especially on mobile systems, where RAM and cores are limited, Windows 8 also introduced **Hybrid Boot**. This feature combines hibernation with a simple logoff of the current user. When the user shuts down the system, and all other applications and sessions have exited, the system is returned to the `logonui` prompt and then is hibernated. When the system is turned on again, it resumes very quickly to the logon screen, which

gives drivers a chance to reinitialize devices and gives the appearance of a full boot while work is still occurring.

## 21.4 Terminal Services and Fast User Switching

Windows supports a GUI-based console that interfaces with the user via keyboard, mouse, and display. Most systems also support audio and video. For example, audio input is used by Cortana, Windows's voice-recognition and virtual assistant software, which is powered by machine learning. Cortana makes the system more convenient and can also increase its accessibility for users with motor disabilities. Windows 7 added support for **multi-touch hardware**, allowing users to input data by touching the screen with one or more fingers. Video-input capability is used both for accessibility and for security: Windows Hello is a security feature in which advanced 3D heat-sensing, face-mapping cameras and sensors can be used to uniquely identify the user without requiring traditional credentials. In newer versions of Windows 10, eye-motion sensing hardware—in which mouse input is replaced by information on the position and gaze of the eyeballs—can be used for accessibility. Other future input experiences will likely evolve from Microsoft's **HoloLens** augmented-reality product.

The PC was, of course, envisioned as a *personal computer*—an inherently single-user machine. For some time, however, Windows has supported the sharing of a PC among multiple users. Each user who is logged on using the GUI has a **session** created to represent the GUI environment he will be using and to contain all the processes necessary to run his applications. Windows allows multiple sessions to exist at the same time on a single machine. However, client versions of Windows support only a single console, consisting of all the monitors, keyboards, and mice connected to the PC. Only one session can be connected to the console at a time. From the logon screen displayed on the console, users can create new sessions or attach to an existing session. This allows multiple users to share a single PC without having to log off and on between users. Microsoft calls this use of sessions *fast user switching*. macOS has a similar feature.

A user on one PC can also create a new session or connect to an existing session on another computer, which becomes a **remote desktop**. The terminal services feature (TS) makes the connection through a protocol called Remote Desktop Protocol (RDP). Users often employ this feature to connect to a session on a work PC from a home PC. Remote desktops can also be used for remote troubleshooting scenarios: a remote user can be invited to share a session with the user logged on to the session on the console. The remote user can watch the user's actions and can even be given control of the desktop to help resolve computing problems. This latter use of terminal services uses the “mirroring” feature, where the alternative user is sharing the same session instead of creating a separate one.

Many corporations use corporate systems maintained in data centers to run all user sessions that access corporate resources, rather than allowing users to access those resources from their PCs, by exclusively dedicating these machines as terminal servers. Each server computer may handle hundreds of remote-desktop sessions. This is a form of **thin-client computing**, in which

individual computers rely on a server for many functions. Relying on data-center terminal servers improves the reliability, manageability, and security of corporate computing resources.

## 21.5 File System

The native file system in Windows is NTFS. It is used for all local volumes. However, associated USB thumb drives, flash memory on cameras, and external storage devices may be formatted with the 32-bit FAT file system for portability. FAT is a much older file-system format that is understood by many systems besides Windows, such as the software running on cameras. A disadvantage is that the FAT file system does not restrict file access to authorized users. The only solution for securing data with FAT is to run an application to encrypt the data before storing it on the file system.

In contrast, NTFS uses ACLs to control access to individual files and supports implicit encryption of individual files or entire volumes (using Windows BitLocker feature). NTFS implements many other features as well, including data recovery, fault tolerance, very large files and file systems, multiple data streams, UNICODE names, sparse files, journaling, volume shadow copies, and file compression.

### 21.5.1 NTFS Internal Layout

The fundamental entity in NTFS is the volume. A volume is created by the Windows logical disk management utility and is based on a logical disk partition. A volume may occupy a portion of a device or an entire device, or may span several devices. The volume manager can protect the contents of the volume with various levels of RAID.

NTFS does not deal with individual sectors of a storage device but instead uses **clusters** as the units of storage allocation. The cluster size, which is a power of 2, is configured when an NTFS file system is formatted. The default cluster size is based on the volume size—4 KB for volumes larger than 2 GB. Given the size of today’s storage devices, it may make sense to use cluster sizes larger than the Windows defaults to achieve better performance, although these performance gains will come at the expense of more internal fragmentation.

NTFS uses **logical cluster numbers (LCNs)** as storage addresses. It assigns them by numbering clusters from the beginning of the device to the end. Using this scheme, the system can calculate a physical storage offset (in bytes) by multiplying the LCN by the cluster size.

A file in NTFS is not a simple byte stream as it is in UNIX; rather, it is a structured object consisting of typed **attributes**. Each attribute of a file is an independent byte stream that can be created, deleted, read, and written. Some attribute types are standard for all files, including the file name (or names, if the file has aliases, such as an MS-DOS short name), the creation time, and the security descriptor that specifies the access control list. User data are stored in **data attributes**.

Most traditional data files have an **unnamed** data attribute that contains all the file’s data. However, additional data streams can be created with explicit names. The IProp interfaces of the Component Object Model (discussed later

in this chapter) use a named data stream to store properties on ordinary files, including thumbnails of images. In general, attributes can be added as necessary and are accessed using a *file-name:attribute* syntax. NTFS returns only the size of the unnamed attribute in response to file-query operations, such as when running the `dir` command.

Every file in NTFS is described by one or more records in an array stored in a special file called the master file table (MFT). The size of a record is determined when the file system is created; it ranges from 1 to 4 KB. Small attributes are stored in the MFT record itself and are called *resident attributes*. Large attributes, such as the unnamed bulk data, are called *nonresident attributes* and are stored in one or more contiguous extents on the device. A pointer to each extent is stored in the MFT record. For a small file, even the data attribute may fit inside the MFT record. If a file has many attributes—or if it is highly fragmented, so that many pointers are needed to point to all the fragments—one record in the MFT might not be large enough. In this case, the file is described by a record called the **base file record**, which contains pointers to overflow records that hold the additional pointers and attributes.

Each file in an NTFS volume has a unique ID called a **file reference**. The file reference is a 64-bit quantity that consists of a 48-bit file number and a 16-bit sequence number. The file number is the record number (that is, the array slot) in the MFT that describes the file. The sequence number is incremented every time an MFT entry is reused. The sequence number enables NTFS to perform internal consistency checks, such as catching a stale reference to a deleted file after the MFT entry has been reused for a new file.

### 21.5.1.1 NTFS B+ Tree

As in UNIX, the NTFS namespace is organized as a hierarchy of directories. Each directory uses a data structure called a **B+ tree** to store an index of the file names in that directory. In a B+ tree, the length of every path from the root of the tree to a leaf is the same, and the cost of reorganizing the tree is eliminated. The **index root** of a directory contains the top level of the B+ tree. For a large directory, this top level contains pointers to disk extents that hold the remainder of the tree. Each entry in the directory contains the name and file reference of the file, as well as a copy of the update timestamp and file size taken from the file's resident attributes in the MFT. Copies of this information are stored in the directory so that a directory listing can be efficiently generated. Because all the file names, sizes, and update times are available from the directory itself, there is no need to gather these attributes from the MFT entries for each of the files.

### 21.5.1.2 NTFS Metadata

The NTFS volume's metadata are all stored in files. The first file is the MFT. The second file, which is used during recovery if the MFT is damaged, contains a copy of the first 16 entries of the MFT. The next few files are also special in purpose. They include the following files:

- The **log file** records all metadata updates to the file system.
- The **volume file** contains the name of the volume, the version of NTFS that formatted the volume, and a bit that tells whether the volume may have

been corrupted and needs to be checked for consistency using the chkdsk program.

- The **attribute-definition table** indicates which attribute types are used in the volume and what operations can be performed on each of them.
- The **root directory** is the top-level directory in the file-system hierarchy.
- The **bitmap file** indicates which clusters on a volume are allocated to files and which are free.
- The **boot file** contains the startup code for Windows and must be located at a particular secondary storage device address so that it can be found easily by a simple ROM bootstrap loader. The boot file also contains the physical address of the MFT.
- The **bad-cluster file** keeps track of any bad areas on the volume; NTFS uses this record for error recovery.

Keeping all the NTFS metadata in actual files has a useful property. As discussed in Section 21.3.5.6, the cache manager caches file data. Since all the NTFS metadata reside in files, these data can be cached using the same mechanisms used for ordinary data.

### 21.5.2 Recovery

In many simple file systems, a power failure at the wrong time can damage the file-system data structures so severely that the entire volume is scrambled. Many UNIX file systems, including UFS but not ZFS, store redundant metadata on the storage device, and they recover from crashes by using the `fsck` program to check all the file-system data structures and restore them forcibly to a consistent state. Restoring them often involves deleting damaged files and freeing data clusters that had been written with user data but not properly recorded in the file system's metadata structures. This checking can be a slow process and can result in the loss of significant amounts of data.

NTFS takes a different approach to file-system robustness. In NTFS, all file-system data-structure updates are performed inside transactions. Before a data structure is altered, the transaction writes a log record that contains redo and undo information. After the data structure has been changed, the transaction writes a commit record to the log to signify that the transaction succeeded.

After a crash, the system can restore the file-system data structures to a consistent state by processing the log records, first redoing the operations for committed transactions (to be sure their changes reached the file system data structures) and then undoing the operations for transactions that did not commit successfully before the crash. Periodically (usually every 5 seconds), a checkpoint record is written to the log. The system does not need log records prior to the checkpoint to recover from a crash. They can be discarded, so the log file does not grow without bounds. The first time after system startup that an NTFS volume is accessed, NTFS automatically performs file-system recovery.

This scheme does not guarantee that all the user-file contents are correct after a crash. It ensures only that the file-system data structures (the metadata files) are undamaged and reflect some consistent state that existed prior to the

crash. It would be possible to extend the transaction scheme to cover user files, and Microsoft took some steps to do this in Windows Vista.

The log is stored in the third metadata file at the beginning of the volume. It is created with a fixed maximum size when the file system is formatted. It has two sections: the *logging area*, which is a circular queue of log records, and the *restart area*, which holds context information, such as the position in the logging area where NTFS should start reading during a recovery. In fact, the restart area holds two copies of its information, so recovery is still possible if one copy is damaged during the crash.

The logging functionality is provided by the *log-file service*. In addition to writing the log records and performing recovery actions, the log-file service keeps track of the free space in the log file. If the free space gets too low, the log-file service queues pending transactions, and NTFS halts all new I/O operations. After the in-progress operations complete, NTFS calls the cache manager to flush all data and then resets the log file and performs the queued transactions.

### 21.5.3 Security

The security of an NTFS volume is derived from the Windows object model. Each NTFS file references a security descriptor, which specifies the owner of the file, and an access-control list, which contains the access permissions granted or denied to each user or group listed. Early versions of NTFS used a separate security descriptor as an attribute of each file. Beginning with Windows 2000, the security-descriptor attribute points to a shared copy, with a significant savings in storage space and caching space; many, many files have identical security descriptors.

In normal operation, NTFS does not enforce permissions on traversal of directories in file path names. However, for compatibility with POSIX, these checks can be enabled. The latter option is inherently more expensive, since modern parsing of file path names uses prefix matching rather than directory-by-directory parsing of path names. Prefix matching is an algorithm that looks up strings in a cache and finds the entry with the longest match—for example, an entry for `\foo\bar\dir` would be a match for `\foo\bar\dir2\dir3myfile`. The prefix-matching cache allows path-name traversal to begin much deeper in the tree, saving many steps. Enforcing traversal checks means that the user's access must be checked at each directory level. For instance, a user might lack permission to traverse `\foo\bar`, so starting at the access for `\foo\bar\dir` would be an error.

### 21.5.4 Compression

NTFS can perform data compression on individual files or on all data files in a directory. To compress a file, NTFS divides the file's data into **compression units**, which are blocks of 16 contiguous clusters. When a compression unit is written, a data-compression algorithm is applied. If the result fits into fewer than 16 clusters, the compressed version is stored. When reading, NTFS can determine whether data have been compressed: if they have been, the length of the stored compression unit is less than 16 clusters. To improve performance when reading contiguous compression units, NTFS prefetches and decompresses ahead of the application requests.

For sparse files or files that contain mostly zeros, NTFS uses another technique to save space. Clusters that contain only zeros because they have never been written are not actually allocated or stored on storage devices. Instead, gaps are left in the sequence of virtual-cluster numbers stored in the MFT entry for the file. When reading a file, if NTFS finds a gap in the virtual-cluster numbers, it just zero-fills that portion of the caller's buffer. This technique is also used by UNIX.

### 21.5.5 Mount Points, Symbolic Links, and Hard Links

Mount points are a form of symbolic link specific to directories on NTFS that were introduced in Windows 2000. They provide a mechanism for organizing storage volumes that is more flexible than the use of global names (like drive letters). A mount point is implemented as a symbolic link with associated data containing the true volume name. Ultimately, mount points will supplant drive letters completely, but there will be a long transition due to the dependence of many applications on the drive-letter scheme.

Windows Vista introduced support for a more general form of symbolic links, similar to those found in UNIX. The links can be absolute or relative, can point to objects that do not exist, and can point to both files and directories even across volumes. NTFS also supports **hard links**, where a single file has an entry in more than one directory of the same volume.

### 21.5.6 Change Journal

NTFS keeps a journal describing all changes that have been made to the file system. User-mode services can receive notifications of changes to the journal and then identify what files have changed by reading from the journal. The search indexer service uses the change journal to identify files that need to be re-indexed. The file-replication service uses it to identify files that need to be replicated across the network.

### 21.5.7 Volume Shadow Copies

Windows implements the capability of bringing a volume to a known state and then creating a shadow copy that can be used to back up a consistent view of the volume. This technique is known as *snapshots* in some other file systems. Making a shadow copy of a volume is a form of copy-on-write, where blocks modified after the shadow copy is created are stored in their original form in the copy. Achieving a consistent state for the volume requires the cooperation of applications, since the system cannot know when the data used by the application are in a stable state from which the application could be safely restarted.

The server version of Windows uses shadow copies to efficiently maintain old versions of files stored on file servers. This allows users to see documents as they existed at earlier points in time. A user can thus recover files that were accidentally deleted or simply look at a previous version of the file, all without pulling out backup media.

## 21.6 Networking

Windows supports both peer-to-peer and client–server networking. It also has facilities for network management. The networking components in Windows provide data transport, interprocess communication, file sharing across a network, and the ability to send print jobs to remote printers.

### 21.6.1 Network Interfaces

To describe networking in Windows, we must first mention two of the internal networking interfaces: the [Network Device Interface specification \(NDIS\)](#) and the [Transport Driver Interface \(TDI\)](#). The NDIS interface was developed in 1989 by Microsoft and 3Com to separate network adapters from transport protocols so that either could be changed without affecting the other. NDIS resides at the interface between the data-link and network layers in the ISO model and enables many protocols to operate over many different network adapters. In terms of the ISO model, the TDI is the interface between the transport layer (layer 4) and the session layer (layer 5). This interface enables any session-layer component to use any available transport mechanism. (Similar reasoning led to the streams mechanism in UNIX.) The TDI supports both connection-based and connectionless transport and has functions to send any type of data.

### 21.6.2 Protocols

Windows implements transport protocols as drivers. These drivers can be loaded and unloaded from the system dynamically, although in practice the system typically has to be rebooted after a change. Windows comes with several networking protocols. Next, we discuss a number of these protocols.

#### 21.6.2.1 Server Message Block

The [Server Message Block \(SMB\)](#) protocol was first introduced in MS-DOS 3.1. The system uses the protocol to send I/O requests over the network. The SMB protocol has four message types. Session control messages are commands that start and end a redirector connection to a shared resource at the server. A redirector uses File messages to access files at the server. Printer messages are used to send data to a remote print queue and to receive status information from the queue, and Message messages are used to communicate with another workstation. A version of the SMB protocol was published as the [Common Internet File System \(CIFS\)](#) and is supported on a number of operating systems.

#### 21.6.2.2 Transmission Control Protocol/Internet Protocol

The transmission control protocol/Internet protocol (TCP/IP) suite that is used on the Internet has become the de facto standard networking infrastructure. Windows uses TCP/IP to connect to a wide variety of operating systems and hardware platforms. The Windows TCP/IP package includes the simple network-management protocol (SNMP), the dynamic host-configuration protocol (DHCP), and the older Windows Internet name service (WINS). Windows Vista introduced a new implementation of TCP/IP that supports both IPv4 and IPv6 in the same network stack. This new implementation also supports

offloading of the network stack onto advanced hardware to achieve very high performance for servers.

Windows provides a software firewall that limits the TCP ports that can be used by programs for network communication. Network firewalls are commonly implemented in routers and are a very important security measure. Having a firewall built into the operating system makes a hardware router unnecessary, and it also provides more integrated management and easier use.

#### 21.6.2.3 Point-to-Point Tunneling Protocol

The **Point-to-Point Tunneling Protocol (PPTP)** is a protocol provided by Windows to communicate between remote-access server modules running on Windows server machines and other client systems that are connected over the Internet. The remote-access servers can encrypt data sent over the connection, and they support multiprotocol **virtual private networks (VPNs)** on the Internet.

#### 21.6.2.4 HTTP Protocol

The HTTP protocol is used to get/put information using the World Wide Web. Windows implements HTTP using a kernel-mode driver, so web servers can operate with a low-overhead connection to the networking stack. HTTP is a fairly general protocol that Windows makes available as a transport option for implementing RPC.

#### 21.6.2.5 Web-Distributed Authoring and Versioning Protocol

Web-distributed authoring and versioning (WebDAV) is an HTTP-based protocol for collaborative authoring across a network. Windows builds a WebDAV redirector into the file system. Being built directly into the file system enables WebDAV to work with other file-system features, such as encryption. Personal files can then be stored securely in a public place. Because WebDAV uses HTTP, which is a get/put protocol, Windows has to cache the files locally so programs can use read and write operations on parts of the files.

#### 21.6.2.6 Named Pipes

**Named pipes** are a connection-oriented messaging mechanism. A process can use named pipes to communicate with other processes on the same machine. Since named pipes are accessed through the file-system interface, the security mechanisms used for file objects also apply to named pipes. The SMB protocol supports named pipes, so named pipes can also be used for communication between processes on different systems.

The format of pipe names follows the **Uniform Naming Convention (UNC)**. A UNC name looks like a typical remote file name. The format is \\server\_name\\share\_name\\x\\y\\z, where server\_name identifies a server on the network; share\_name identifies any resource that is made available to network users, such as directories, files, named pipes, and printers; and \\x\\y\\z is a normal file path name.

### 21.6.2.7 Remote Procedure Calls

Remote procedure calls (RPCs), mentioned earlier, are client–server mechanisms that enable an application on one machine to make a procedure call to code on another machine. The client calls a local procedure—a stub routine—which packs its arguments into a message and sends them across the network to a particular server process. The client-side stub routine then blocks. Meanwhile, the server unpacks the message, calls the procedure, packs the return results into a message, and sends them back to the client stub. The client stub unblocks, receives the message, unpacks the results of the RPC, and returns them to the caller. This packing of arguments is sometimes called **marshaling**. The client stub code and the descriptors necessary to pack and unpack the arguments for an RPC are compiled from a specification written in the **Microsoft Interface Definition Language**.

The Windows RPC mechanism follows the widely used distributed-computing-environment standard for RPC messages, so programs written to use Windows RPCs are highly portable. The RPC standard is detailed. It hides many of the architectural differences among computers, such as the sizes of binary numbers and the order of bytes and bits in computer words, by specifying standard data formats for RPC messages.

### 21.6.2.8 Component Object Model

The **Component Object Model (COM)** is a mechanism for interprocess communication that was developed for Windows. A COM object provides a well-defined interface to manipulate the data in the object. For instance, COM is the infrastructure used by Microsoft’s **Object Linking and Embedding (OLE)** technology for inserting spreadsheets into Microsoft Word documents. Many Windows services provide COM interfaces. In addition, a distributed extension called **DCOM** can be used over a network utilizing RPC to provide a transparent method of developing distributed applications.

## 21.6.3 Redirectors and Servers

In Windows, an application can use the Windows I/O API to access files from a remote computer as though they were local, provided that the remote computer is running a CIFS server such as those provided by Windows. A **redirector** is the client-side object that forwards I/O requests to a remote system, where they are satisfied by a server. For performance and security, the redirectors and servers run in kernel mode.

In more detail, access to a remote file occurs as follows:

1. The application calls the I/O manager to request that a file be opened with a file name in the standard UNC format.
2. The I/O manager builds an I/O request packet, as described in Section 21.3.5.5.
3. The I/O manager recognizes that the access is for a remote file and calls a driver called a **Multiple UNC Provider (MUP)**.

4. The MUP sends the I/O request packet asynchronously to all registered redirectors.
5. A redirector that can satisfy the request responds to the MUP. To avoid asking all the redirectors the same question in the future, the MUP uses a cache to remember which redirector can handle this file.
6. The redirector sends the network request to the remote system.
7. The remote-system network drivers receive the request and pass it to the server driver.
8. The server driver hands the request to the proper local file-system driver.
9. The proper device driver is called to access the data.
10. The results are returned to the server driver, which sends the data back to the requesting redirector. The redirector then returns the data to the calling application via the I/O manager.

A similar process occurs for applications that use the Win32 network API, rather than the UNC services, except that a module called a *multi-provider router* is used instead of a MUP.

For portability, redirectors and servers use the TDI API for network transport. The requests themselves are expressed in a higher-level protocol, which by default is the SMB protocol described in Section 21.6.2. The list of redirectors is maintained in the system hive of the registry.

#### 21.6.3.1 Distributed File System

UNC names are not always convenient, because multiple file servers may be available to serve the same content and UNC names explicitly include the name of the server. Windows supports a **distributed file-system (DFS)** protocol that allows a network administrator to serve up files from multiple servers using a single distributed name space.

#### 21.6.3.2 Folder Redirection and Client-Side Caching

To improve the PC experience for users who frequently switch among computers, Windows allows administrators to give users **roaming profile**, which keep users' preferences and other settings on servers. **Folder redirection** is then used to automatically store a user's documents and other files on a server.

This works well until one of the computers is no longer attached to the network, as when a user takes a laptop onto an airplane. To give users off-line access to their redirected files, Windows uses **client-side caching (CSC)**. CSC is also used when the computer is on-line to keep copies of the server files on the local machine for better performance. The files are pushed up to the server as they are changed. If the computer becomes disconnected, the files are still available, and the update of the server is deferred until the next time the computer is online.

### 21.6.4 Domains

Many networked environments have natural groups of users, such as students in a computer laboratory at school or employees in one department in a business. Frequently, we want all the members of the group to be able to access shared resources on their various computers in the group. To manage the global access rights within such groups, Windows uses the concept of a domain. Previously, these domains had no relationship whatsoever to the domain-name system (DNS) that maps Internet host names to IP addresses. Now, however, they are closely related.

Specifically, a Windows domain is a group of Windows workstations and servers that share a common security policy and user database. Since Windows uses the Kerberos protocol for trust and authentication, a Windows domain is the same thing as a Kerberos realm. Windows uses a hierarchical approach for establishing trust relationships between related domains. The trust relationships are based on DNS and allow transitive trusts that can flow up and down the hierarchy. This approach reduces the number of trusts required for  $n$  domains from  $n * (n - 1)$  to  $O(n)$ . The workstations in the domain trust the domain controller to give correct information about the access rights of each user (loaded into the user's access token by `1saas`). All users retain the ability to restrict access to their own workstations, however, no matter what any domain controller may say.

### 21.6.5 Active Directory

**Active Directory** is the Windows implementation of **Lightweight Directory-Access Protocol (LDAP)** services. Active Directory stores the topology information about the domain, keeps the domain-based user and group accounts and passwords, and provides a domain-based store for Windows features that need it, such as **Windows group policy**. Administrators use group policies to establish uniform standards for desktop preferences and software. For many corporate information-technology groups, uniformity drastically reduces the cost of computing.

## 21.7 Programmer Interface

The **Win32 API** is the fundamental interface to the capabilities of Windows. This section describes five main aspects of the Win32 API: access to kernel objects, sharing of objects between processes, process management, interprocess communication, and memory management.

### 21.7.1 Access to Kernel Objects

The Windows kernel provides many services that application programs can use. Application programs obtain these services by manipulating kernel objects. A process gains access to a kernel object named `XXX` by calling the `CreateXXX` function to open a handle to an instance of `XXX`. This handle is unique to the process. Depending on which object is being opened, if the `Create()` function fails, it may return 0, or it may return a special constant named `INVALID_HANDLE_VALUE`. A process can close any handle by calling the

---

```

SECURITY_ATTRIBUTES sa;
sa.nLength = sizeof(sa);
sa.lpSecurityDescriptor = NULL;
sa.bInheritHandle = TRUE;
HANDLE hSemaphore = CreateSemaphore(&sa, 1, 1, NULL);
WCHAR wszCommandline[MAX_PATH];
StringCchPrintf(wszCommandLine, _countof(wszCommandLine),
L"another_process.exe %d", hSemaphore);
CreateProcess(L"another_process.exe", wszCommandLine,
NULL, NULL, TRUE, . . .);

```

---

**Figure 21.7** Code enabling a child to share an object by inheriting a handle.

CloseHandle() function, and the system may delete the object if the count of handles referencing the object in all processes drops to zero.

### 21.7.2 Sharing Objects Between Processes

Windows provides three ways to share objects between processes. The first way is for a child process to inherit a handle to the object. When the parent calls the CreateXXX function, the parent supplies a SECURITIES\_ATTRIBUTES structure with the bInheritHandle field set to TRUE. This field creates an inheritable handle. Next, the child process is created, passing a value of TRUE to the CreateProcess() function's bInheritHandle argument. Figure 21.7 shows a code sample that creates a semaphore handle inherited by a child process.

Assuming the child process knows which handles are shared, the parent and child can achieve interprocess communication through the shared objects. In the example in Figure 21.7, the child process gets the value of the handle from the first command-line argument and then shares the semaphore with the parent process.

The second way to share objects is for one process to give the object a name when the object is created and for the second process to open the name. This method has two drawbacks: Windows does not provide a way to check whether an object with the chosen name already exists, and the object name space is global, without regard to the object type. For instance, two applications may create and share a single object named “foo” when two distinct objects—possibly of different types—were desired.

Named objects have the advantage that unrelated processes can readily share them. The first process calls one of the CreateXXX functions and supplies a name as a parameter. The second process gets a handle to share the object by calling OpenXXX() (or CreateXXX) with the same name, as shown in the example in Figure 21.8.

The third way to share objects is via the DuplicateHandle() function. This method requires some other method of interprocess communication to pass the duplicated handle. Given a handle to a process and the value of a handle within that process, a second process can get a handle to the same object and thus share it. An example of this method is shown in Figure 21.9.

```
// Process A
. . .
HANDLE hSemaphore = CreateSemaphore(NULL, 1, 1, L"MySEM1");
. . .

// Process B
. . .
HANDLE hSemaphore = OpenSemaphore(SEMAPHORE_ALL_ACCESS,
    FALSE, L"MySEM1");
. . .
```

---

**Figure 21.8** Code for sharing an object by name lookup.

### 21.7.3 Process Management

In Windows, a **process** is a loaded instance of an application and a **thread** is an executable unit of code that can be scheduled by the kernel dispatcher. Thus, a process contains one or more threads. A process is created when a thread in some other process calls the `CreateProcess()` API. This routine loads any dynamic link libraries used by the process and creates an initial thread in the process. Additional threads can be created by the `CreateThread()` function.

---

```
// Process A wants to give Process B access to a semaphore

// Process A

DWORD dwProcessBId; // must; from some IPC mechanism
HANDLE hSemaphore = CreateSemaphore(NULL, 1, 1, NULL);
HANDLE hProcess = OpenProcess(PROCESS_DUP_HANDLE, FALSE,
    dwProcessBId);
HANDLE hSemaphoreCopy;
DuplicateHandle(GetCurrentProcess(), hSemaphore,
    hProcess, &hSemaphoreCopy,
    0, FALSE, DUPLICATE_SAME_ACCESS);
// send the value of the semaphore to Process B
// using a message or shared memory object
. . .

// Process B
HANDLE hSemaphore = // value of semaphore from message
// use hSemaphore to access the semaphore
. . .
```

---

**Figure 21.9** Code for sharing an object by passing a handle.

Each thread is created with its own stack, which defaults to 1 MB unless otherwise specified in an argument to `CreateThread()`.

### 21.7.3.1 Scheduling Rule

Priorities in the Win32 environment are based on the native kernel (NT) scheduling model, but not all priority values may be chosen. The Win32 API uses six priority classes:

1. `IDLE_PRIORITY_CLASS` (NT priority level 4)
2. `BELOW_NORMAL_PRIORITY_CLASS` (NT priority level 6)
3. `NORMAL_PRIORITY_CLASS` (NT priority level 8)
4. `ABOVE_NORMAL_PRIORITY_CLASS` (NT priority level 10)
5. `HIGH_PRIORITY_CLASS` (NT priority level 13)
6. `REALTIME_PRIORITY_CLASS` (NT priority level 24)

Processes are typically members of the `NORMAL_PRIORITY_CLASS` unless the parent of the process was of the `IDLE_PRIORITY_CLASS` or another class was specified when `CreateProcess` was called. The priority class of a process is the default for all threads that execute in the process. It can be changed with the `SetPriorityClass()` function or by passing an argument to the `start` command. Only users with the *increase scheduling priority* privilege can move a process into the `REALTIME_PRIORITY_CLASS`. Administrators and power users have this privilege by default.

When a user is switching between interactive processes and workloads, the system needs to schedule the appropriate threads so as to provide good responsiveness, which leads to a shorter quanta of execution. Yet, once the user has chosen a particular process, a good amount of throughput from this particular process is also expected. For this reason, Windows has a special scheduling rule for processes not in the `REALTIME_PRIORITY_CLASS`. Windows distinguishes between the process associated with the foreground window on the screen and the other (background) processes. When a process moves into the foreground, Windows increases the scheduling quantum for all its threads by a factor of 3; CPU-bound threads in the foreground process will run three times longer than similar threads in background processes. Because server systems always operate with a much larger quantum than client systems—a factor of 6—this behavior is not enabled for server systems. For both types of systems, however, the scheduling parameters can be customized through the appropriate system dialog or registry key.

### 21.7.3.2 Thread Priorities

A thread starts with an initial priority determined by its class. The priority can be altered by the `SetThreadPriority()` function. This function takes an argument that specifies a priority relative to the base priority of its class:

- `THREAD_PRIORITY_LOWEST`: base – 2
- `THREAD_PRIORITY_BELOW_NORMAL`: base – 1

- `THREAD_PRIORITY_NORMAL`: base + 0
- `THREAD_PRIORITY_ABOVE_NORMAL`: base + 1
- `THREAD_PRIORITY_HIGHEST`: base + 2

Two other designations are also used to adjust the priority. Recall from Section 21.3.4.3 that the kernel has two priority classes: 16–31 for the static class and 1–15 for the variable class. `THREAD_PRIORITY_IDLE` sets the priority to 16 for static-priority threads and to 1 for variable-priority threads. `THREAD_PRIORITY_TIME_CRITICAL` sets the priority to 31 for real-time threads and to 15 for variable-priority threads.

The kernel adjusts the priority of a variable class thread dynamically depending on whether the thread is I/O bound or CPU bound. The Win32 API provides a method to disable this adjustment via `SetProcessPriorityBoost()` and `SetThreadPriorityBoost()` functions.

### 21.7.3.3 Thread Suspend and Resume

A thread can be created in a *suspended state* or can be placed in a suspended state later by use of the `SuspendThread()` function. Before a suspended thread can be scheduled by the kernel dispatcher, it must be moved out of the suspended state by use of the `ResumeThread()` function. Both functions set a counter so that if a thread is suspended twice, it must be resumed twice before it can run.

### 21.7.3.4 Thread Synchronization

To synchronize concurrent access to shared objects by threads, the kernel provides synchronization objects, such as semaphores and mutexes. These are dispatcher objects, as discussed in Section 21.3.4.3. Threads can also synchronize with kernel services operating on kernel objects—such as threads, processes, and files—because these are also dispatcher objects. Synchronization with kernel dispatcher objects can be achieved by use of the `WaitForSingleObject()` and `WaitForMultipleObjects()` functions; these functions wait for one or more dispatcher objects to be signaled.

Another method of synchronization is available to threads within the same process that want to execute code exclusively. The Win32 **critical section object** is a user-mode mutex object that can often be acquired and released without entering the kernel. On a multiprocessor, a Win32 critical section will attempt to spin while waiting for a critical section held by another thread to be released. If the spinning takes too long, the acquiring thread will allocate a kernel mutex and yield its CPU. Critical sections are particularly efficient because the kernel mutex is allocated only when there is contention and then used only after attempting to spin. Most mutexes in programs are never actually contended, so the savings are significant.

Before using a critical section, some thread in the process must call `InitializeCriticalSection()`. Each thread that wants to acquire the mutex calls `EnterCriticalSection()` and then later calls `LeaveCriticalSection()` to release the mutex. There is also a `TryEnterCriticalSection()` function, which attempts to acquire the mutex without blocking.

For programs that want user-mode reader-writer locks rather than mutexes, Win32 supports **slim reader-writer (SRW) locks**. SRW locks have APIs similar to those for critical sections, such as `InitializeSRWLock`, `AcquireSRWLockXXX`, and `ReleaseSRWLockXXX`, where XXX is either Exclusive or Shared, depending on whether the thread wants write access or only read access to the object protected by the lock. The Win32 API also supports **condition variables**, which can be used with either critical sections or SRW locks.

#### 21.7.3.5 Thread Pool

Repeatedly creating and deleting threads can be expensive for applications and services that perform small amounts of work in each instantiation. The Win32 thread pool provides user-mode programs with three services: a queue to which work requests may be submitted (via the `SubmitThreadpoolWork()` function), an API that can be used to bind callbacks to waitable handles (`RegisterWaitForSingleObject()`), and APIs to work with timers (`CreateThreadpoolTimer()` and `WaitForThreadpoolTimerCallbacks()`) and to bind callbacks to I/O completion queues (`BindIoCompletionCallback()`).

The goal of using a thread pool is to increase performance and reduce memory footprint. Threads are relatively expensive, and each processor can be executing only one thread at a time no matter how many threads are available. The thread pool attempts to reduce the number of runnable threads by slightly delaying work requests (reusing each thread for many requests) while providing enough threads to effectively utilize the machine's CPUs. The wait and I/O- and timer-callback APIs allow the thread pool to further reduce the number of threads in a process, using far fewer threads than would be necessary if a process were to devote separate threads to servicing each waitable handle, timer, or completion port.

#### 21.7.3.6 Fibers

A **fibre** is user-mode code that is scheduled according to a user-defined scheduling algorithm. Fibers are completely a user-mode facility; the kernel is not aware that they exist. The fiber mechanism uses Windows threads as if they were CPUs to execute the fibers. Fibers are cooperatively scheduled, meaning that they are never preempted but must explicitly yield the thread on which they are running. When a fiber yields a thread, another fiber can be scheduled on it by the run-time system (the programming language run-time code).

The system creates a fiber by calling either `ConvertThreadToFiber()` or `CreateFiber()`. The primary difference between these functions is that `CreateFiber()` does not begin executing the fiber that was created. To begin execution, the application must call `SwitchToFiber()`. The application can terminate a fiber by calling `DeleteFiber()`.

Fibers are not recommended for threads that use Win32 APIs rather than standard C-library functions because of potential incompatibilities. Win32 user-mode threads have a **thread-environment block (TEB)** that contains numerous per-thread fields used by the Win32 APIs. Fibers must share the TEB of the thread on which they are running. This can lead to problems when a Win32 interface puts state information into the TEB for one fiber and then the information is

overwritten by a different fiber. Fibers are included in the Win32 API to facilitate the porting of legacy UNIX applications that were written for a user-mode thread model such as Pthreads.

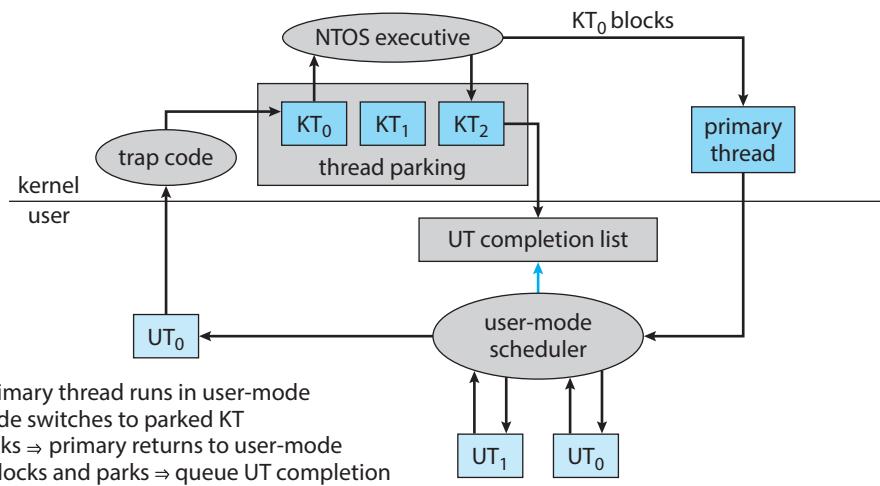
#### 21.7.3.7 User-Mode Scheduling UMS and ConcRT

A new mechanism in Windows 7, user-mode scheduling (UMS), addressed several limitations of fibers. As just noted, fibers are unreliable for executing Win32 APIs because they do not have their own TEBs. When a thread running a fiber blocks in the kernel, the user scheduler loses control of the CPU for a time as the kernel dispatcher takes over scheduling. Problems may result when fibers change the kernel state of a thread, such as the priority or impersonation token, or when they start asynchronous I/O.

UMS provides an alternative model by recognizing that each Windows thread is actually two threads: a kernel thread (KT) and a user thread (UT). Each type of thread has its own stack and its own set of saved registers. The KT and UT appear as a single thread to the programmer because UTs can never block but must always enter the kernel, where an implicit switch to the corresponding KT takes place. UMS uses each UT's TEB to uniquely identify the UT. When a UT enters the kernel, an explicit switch is made to the KT that corresponds to the UT identified by the current TEB. The reason the kernel does not know which UT is running is that UTs can invoke a user-mode scheduler, as fibers do. But in UMS, the scheduler switches UTs, including switching the TEBs.

When a UT enters the kernel, its KT may block. When this happens, the kernel switches to a scheduling thread, which UMS calls a *primary thread*, and uses this thread to reenter the user-mode scheduler so that it can pick another UT to run. Eventually, a blocked KT will complete its operation and be ready to return to user mode. Since UMS has already reentered the user-mode scheduler to run a different UT, UMS queues the UT corresponding to the completed KT to a completion list in user mode. When the user-mode scheduler is choosing a new UT to switch to, it can examine the completion list and treat any UT on the list as a candidate for scheduling. The key features of UMS are depicted in Figure 21.10.

Unlike fibers, UMS is not intended to be used directly by programmers. The details of writing user-mode schedulers can be very challenging, and UMS does not include such a scheduler. Rather, the schedulers come from programming language libraries that build on top of UMS. Microsoft Visual Studio 2010 shipped with Concurrency Runtime (ConcRT), a concurrent programming framework for C++. ConcRT provides a user-mode scheduler together with facilities for decomposing programs into tasks, which can then be scheduled on the available CPUs. ConcRT provides support for `par_for` styles of constructs, as well as rudimentary resource management and task synchronization primitives. However, as of Visual Studio 2013, the UMS scheduling mode is no longer available in ConcRT. Significant performance metrics showed that true parallel programs that are well written do not spend a large amount of time context-switching between their tasks. The benefits that UMS provided in this space did not outweigh the complexity of maintaining a separate scheduler—in some cases, even the default NT scheduler performed better.



**Figure 21.10** User-mode scheduling.

### 21.7.3.8 Winsock

**Winsock** is the Windows sockets API. Winsock is a session-layer interface that is largely compatible with BSD sockets but has some added Windows extensions. It provides a standardized interface to many transport protocols that may have different addressing schemes, so that any Winsock application can run on any Winsock-compliant protocol stack. Winsock underwent a major update in Windows Vista to add tracing, IPv6 support, impersonation, new security APIs, and many other features.

Winsock follows the Windows Open System Architecture (WOSA) model, which provides a standard service provider interface (SPI) between applications and networking protocols. Applications can load and unload *layered protocols* that build additional functionality, such as additional security, on top of the transport protocol layers. Winsock supports asynchronous operations and notifications, reliable multicasting, secure sockets, and kernel mode sockets. It also supports simpler usage models, like the `WSAConnectByName()` function, which accepts the target as strings specifying the name or IP address of the server and the service or port number of the destination port.

### 21.7.4 IPC Using Windows Messaging

Win32 applications handle interprocess communication in several ways. The typical high-performance way is by using local RPCs or named pipes. Another is by using shared kernel objects, such as named section objects, and a synchronization object, such as an event. Yet another is by using the Windows messaging facility—an approach that is particularly popular for Win32 GUI applications. One thread can send a message to another thread or to a window by calling `PostMessage()`, `PostThreadMessage()`, `SendMessage()`, `SendThreadMessage()`, or `SendMessageCallback()`. *Posting* a message and *sending* a message differ in this way: The post routines are asynchronous; they

return immediately, and the calling thread does not know when the message is actually delivered. The send routines are synchronous; they block the caller until the message has been delivered and processed.

In addition to sending a message, a thread can send data with the message. Since processes have separate address spaces, the data must be copied. The system copies data by calling `SendMessage()` to send a message of type `WM_COPYDATA` with a `COPYDATASTRUCT` data structure that contains the length and address of the data to be transferred. When the message is sent, Windows copies the data to a new block of memory and gives the virtual address of the new block to the receiving process.

Every Win32 GUI thread has its own input queue from which it receives messages. If a Win32 application does not call `GetMessage()` to handle events on its input queue, the queue fills up; and after about five seconds, the task manager marks the application as “Not Responding.” Note that message passing is subject to the integrity level mechanism introduced earlier. Thus, a process may not send a message such as `WM_COPYDATA` to a process with a higher integrity level, unless a special Windows API is used to remove the protection (`ChangeWindowMessageFilterEx`).

### 21.7.5 Memory Management

The Win32 API provides several ways for an application to use memory: virtual memory, memory-mapped files, heaps, thread-local storage, and AWE physical memory.

#### 21.7.5.1 Virtual Memory

An application calls `VirtualAlloc()` to reserve or commit virtual memory and `VirtualFree()` to de-commit or release the memory. These functions enable the application to specify the virtual address at which the memory is allocated. (Otherwise, a random address is selected, which is recommended for security reasons.) The functions operate on multiples of the memory page size but, for historical reasons, always return memory allocated on a 64-KB boundary. Examples of these functions appear in Figure 21.11. The `VirtualAllocEx()` and `VirtualFreeEx()` functions can be used to allocate and free memory in a separate process, while `VirtualAllocExNuma()` can be used to leverage memory locality on NUMA systems.

#### 21.7.5.2 Memory-Mapped Files

Another way for an application to use memory is by memory-mapping a file into its address space. Memory mapping is also a convenient way for two processes to share memory: both processes map the same file into their virtual memory. Memory mapping is a multistage process, as you can see in the example in Figure 21.12.

If a process wants to map some address space just to share a memory region with another process, no file is needed. The process calls `CreateFileMapping()` with a file handle of `0xffffffff`, a particular size, and (optionally) a name. The resulting file-mapping object can be shared by inheritance, by name lookup (if it was named), or by handle duplication.

---

```

// reserve 16 MB at the top of our address space
PVOID pBuf = VirtualAlloc(NULL, 0x1000000,
    MEM_RESERVE | MEM_TOP_DOWN, PAGE_READWRITE);
// commit the upper 8 MB of the allocated space
VirtualAlloc((LPVOID)((DWORD_PTR)pBuf + 0x800000), 0x800000,
    MEM_COMMIT, PAGE_READWRITE);
// do something with the memory
. .
// now decommit the memory
VirtualFree((LPVOID)((DWORD_PTR)pBuf + 0x800000), 0x800000,
    MEM_DECOMMIT);
// release all of the allocated address space
VirtualFree(pBuf, 0, MEM_RELEASE);

```

---

**Figure 21.11** Code fragments for allocating virtual memory.

#### 21.7.5.3 Heaps

Heaps provide a third way for applications to use memory, just as with `malloc()` and `free()` in standard C or `new()` and `delete()` in C++. A heap in the Win32 environment is a region of pre-committed address space. When a Win32 process is initialized, it is created with a **default heap**. Since most Win32

---

```

// set the file mapping size to 8MB
DWORD dwSize = 0x800000;
// open the file or create it if it does not exist
HANDLE hFile = CreateFile(L"somefile.ext",
    GENERIC_READ | GENERIC_WRITE,
    FILE_SHARE_READ | FILE_SHARE_WRITE, NULL,
    OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
// create the file mapping
HANDLE hMap = CreateFileMapping(hFile,
    PAGE_READWRITE | SEC_COMMIT, 0, dwSize, L"SHM_1");
// now get a view of the space mapped
PVOID pBuf = MapViewOfFile(hMap, FILE_MAP_ALL_ACCESS,
    0, 0, 0, dwSize);
// do something with the mapped file
. .
// now unmap the file
UnmapViewOfFile(pBuf);
CloseHandle(hMap);
CloseHandle(hFile);

```

---

**Figure 21.12** Code fragments for memory mapping of a file.

applications are multithreaded, access to the heap is synchronized to protect the heap's space-allocation data structures from being damaged by concurrent updates by multiple threads. The advantage of the heap is that it can be used to make allocations as small as 1 byte, because the underlying memory pages have already been committed. Unfortunately, heap memory cannot be shared or marked as read-only, because all heap allocations share the same pages. However, by using `HeapCreate()`, a programmer can create his or her own heap, which can be marked as read-only with `HeapProtect()`, created as an executable heap, or even allocated on a specific NUMA node.

Win32 provides several heap-management functions so that a process can allocate and manage a private heap. These functions are `HeapCreate()`, `HeapAlloc()`, `HeapRealloc()`, `HeapSize()`, `HeapFree()`, and `HeapDestroy()`. The Win32 API also provides the `HeapLock()` and `HeapUnlock()` functions to enable a thread to gain exclusive access to a heap. Note that these functions perform only synchronization; they do not truly “lock” pages against malicious or buggy code that bypasses the heap layer.

The original Win32 heap was optimized for efficient use of space. This led to significant problems with fragmentation of the address space for larger server programs that ran for long periods of time. A new **low-fragmentation heap (LFH)** design introduced in Windows XP greatly reduced the fragmentation problem. The heap manager in Windows 7 and later versions automatically turns on LFH as appropriate. Additionally, the heap is a primary target of attackers using vulnerabilities such as double-free, use-after-free, and other memory-corruption-related attacks. Each version of Windows, including Windows 10, has added more randomness, entropy, and security mitigations to prevent attackers from guessing the ordering, size, location, and content of heap allocations.

#### 21.7.5.4 Thread-Local Storage

A fourth way for applications to use memory is through a **thread-local storage (TLS)** mechanism. Functions that rely on global or static data typically fail to work properly in a multithreaded environment. For instance, the C run-time function `strtok()` uses a static variable to keep track of its current position while parsing a string. For two concurrent threads to execute `strtok()` correctly, they need separate current position variables. TLS provides a way to maintain instances of variables that are global to the function being executed but not shared with any other thread.

TLS provides both dynamic and static methods of creating thread-local storage. The dynamic method is illustrated in Figure 21.13. The TLS mechanism allocates global heap storage and attaches it to the thread environment block (TEB) that Windows allocates to every user-mode thread. The TEB is readily accessible by each thread and is used not just for TLS but for all the per-thread state information in user mode.

#### 21.7.5.5 AWE Memory

A final way for applications to use memory is through the **Address Windowing Extension (AWE)** functionality. This mechanism allows a developer to directly

---

```
// reserve a slot for a variable
DWORD dwVarIndex = T1sAlloc();
// make sure a slot was available
if (dwVarIndex == TLS_OUT_OF_INDEXES)
    return;
// set it to the value 10
T1sSetValue(dwVarIndex, (LPVOID)10);
// get the value
DWORD dwVar = (DWORD)(DWORD_PTR)T1sGetValue(dwVarIndex);
// release the index
T1sFree(dwVarIndex);
```

---

**Figure 21.13** Code for dynamic thread-local storage.

request free physical pages of RAM from the memory manager (through `AllocateUserPhysicalPages()`) and later commit virtual memory on top of the physical pages using `VirtualAlloc()`. By requesting various regions of physical memory (including scatter-gather support), a user-mode application can access more physical memory than virtual address space; this is useful on 32-bit systems, which may have more than 4 GB of RAM). In addition, the application can bypass the memory manager’s caching, paging, and coloring algorithms. Similar to UMS, AWE may thus offer a way for certain applications to extract additional performance or customization beyond what Windows offers by default. SQL Server, for example, uses AWE memory.

To use a thread-local static variable, the application declares the variable as follows to ensure that every thread has its own private copy:

```
_declspec(thread) DWORD cur_pos = 0;
```

## 21.8 Summary

- Microsoft designed Windows to be an extensible, portable operating system—one able to take advantage of new techniques and hardware.
- Windows supports multiple operating environments and symmetric multiprocessing, including both 32-bit and 64-bit processors and NUMA computers.
- The use of kernel objects to provide basic services, along with support for client–server computing, enables Windows to support a wide variety of application environments.
- Windows provides virtual memory, integrated caching, and preemptive scheduling.

- To protect user data and guarantee program integrity, Windows supports elaborate security mechanisms and exploit mitigations and takes advantage of hardware virtualization.
- Windows runs on a wide variety of computers, so users can choose and upgrade hardware to match their budgets and performance requirements without needing to alter the applications they run.
- By including internationalization features, Windows can run in a variety of countries and many languages.
- Windows has sophisticated scheduling and memory-management algorithms for performance and scalability.
- Recent versions of Windows have added power management and fast sleep and wake features, and decreased resource use in several areas to be more useful on mobile systems such as phones and tablets.
- The Windows volume manager and NTFS file system provide a sophisticated set of features for desktop as well as server systems.
- The Win32 API programming environment is feature rich and expansive, allowing programmers to use all of Windows's features in their programs.

## Practice Exercises

- 21.1 What type of operating system is Windows? Describe two of its major features.
- 21.2 List the design goals of Windows. Describe two in detail.
- 21.3 Describe the booting process for a Windows system.
- 21.4 Describe the three main architectural layers of the Windows kernel.
- 21.5 What is the job of the object manager?
- 21.6 What types of services does the process manager provide?
- 21.7 What is a local procedure call?
- 21.8 What are the responsibilities of the I/O manager?
- 21.9 What types of networking does Windows support? How does Windows implement transport protocols? Describe two networking protocols.
- 21.10 How is the NTFS namespace organized?
- 21.11 How does NTFS handle data structures? How does NTFS recover from a system crash? What is guaranteed after a recovery takes place?
- 21.12 How does Windows allocate user memory?
- 21.13 Describe some of the ways in which an application can use memory via the Win32 API.

## Further Reading

[Russinovich et al. (2017)] give a deep overview of Windows 10 and considerable technical detail about system internals and components.

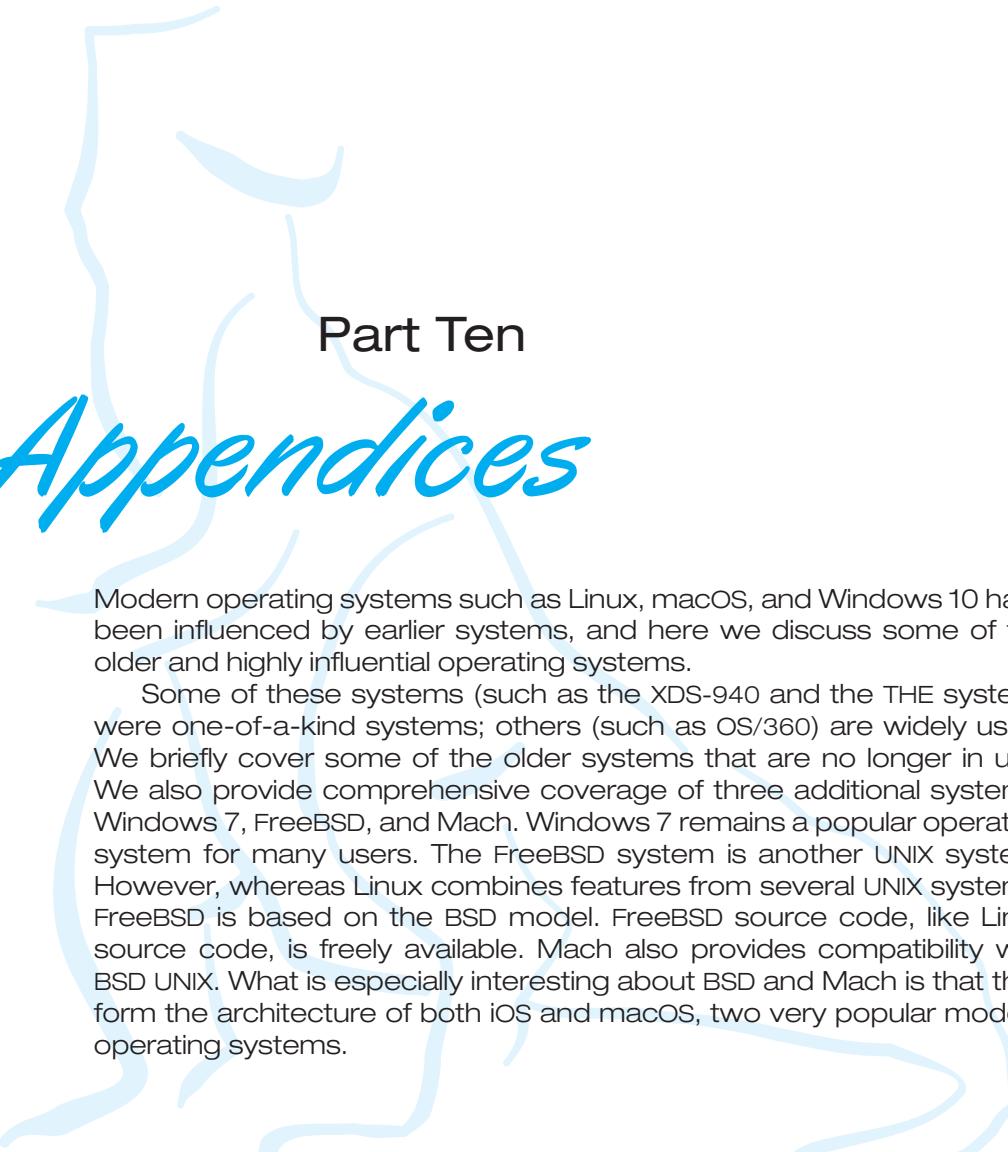
## Bibliography

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

## Chapter 21 Exercises

- 21.14 Under what circumstances would one use the deferred procedure calls facility in Windows?
- 21.15 What is a handle, and how does a process obtain a handle?
- 21.16 Describe the management scheme of the virtual memory manager. How does the VM manager improve performance?
- 21.17 Describe a useful application of the no-access page facility provided in Windows.
- 21.18 Describe the three techniques used for communicating data in a local procedure call. What settings are most conducive to the application of the different message-passing techniques?
- 21.19 What manages caching in Windows? How is the cache managed?
- 21.20 How does the NTFS directory structure differ from the directory structure used in UNIX operating systems?
- 21.21 What is a process, and how is it managed in Windows?
- 21.22 What is the fiber abstraction provided by Windows? How does it differ from the thread abstraction?
- 21.23 How does user-mode scheduling (UMS) in Windows 7 differ from fibers? What are some trade-offs between fibers and UMS?
- 21.24 UMS considers a thread to have two parts, a UT and a KT. How might it be useful to allow UTs to continue executing in parallel with their KTs?
- 21.25 What is the performance trade-off of allowing KTs and UTs to execute on different processors?
- 21.26 Why does the self-map occupy large amounts of virtual address space but no additional virtual memory?
- 21.27 How does the self-map make it easy for the VM manager to move the page-table pages to and from disk? Where are the page-table pages kept on disk?
- 21.28 When a Windows system hibernates, the system is powered off. Suppose you changed the CPU or the amount of RAM on a hibernating system. Do you think that would work? Why or why not?
- 21.29 Give an example showing how the use of a suspend count is helpful in suspending and resuming threads in Windows.





## Part Ten

# Appendices

Modern operating systems such as Linux, macOS, and Windows 10 have been influenced by earlier systems, and here we discuss some of the older and highly influential operating systems.

Some of these systems (such as the XDS-940 and the THE system) were one-of-a-kind systems; others (such as OS/360) are widely used. We briefly cover some of the older systems that are no longer in use. We also provide comprehensive coverage of three additional systems: Windows 7, FreeBSD, and Mach. Windows 7 remains a popular operating system for many users. The FreeBSD system is another UNIX system. However, whereas Linux combines features from several UNIX systems, FreeBSD is based on the BSD model. FreeBSD source code, like Linux source code, is freely available. Mach also provides compatibility with BSD UNIX. What is especially interesting about BSD and Mach is that they form the architecture of both iOS and macOS, two very popular modern operating systems.

# Influential Operating Systems



Now that you understand the fundamental concepts of operating systems (CPU scheduling, memory management, processes, and so on), we are in a position to examine how these concepts have been applied in several older and highly influential operating systems. Some of them (such as the XDS-940 and the THE system) were one-of-a-kind systems; others (such as OS/360) are widely used. The order of presentation highlights the similarities and differences of the systems; it is not strictly chronological or ordered by importance. The serious student of operating systems should be familiar with all these systems.

In the bibliographical notes at the end of the chapter, we include references to further reading about these early systems. The papers, written by the designers of the systems, are important both for their technical content and for their style and flavor.

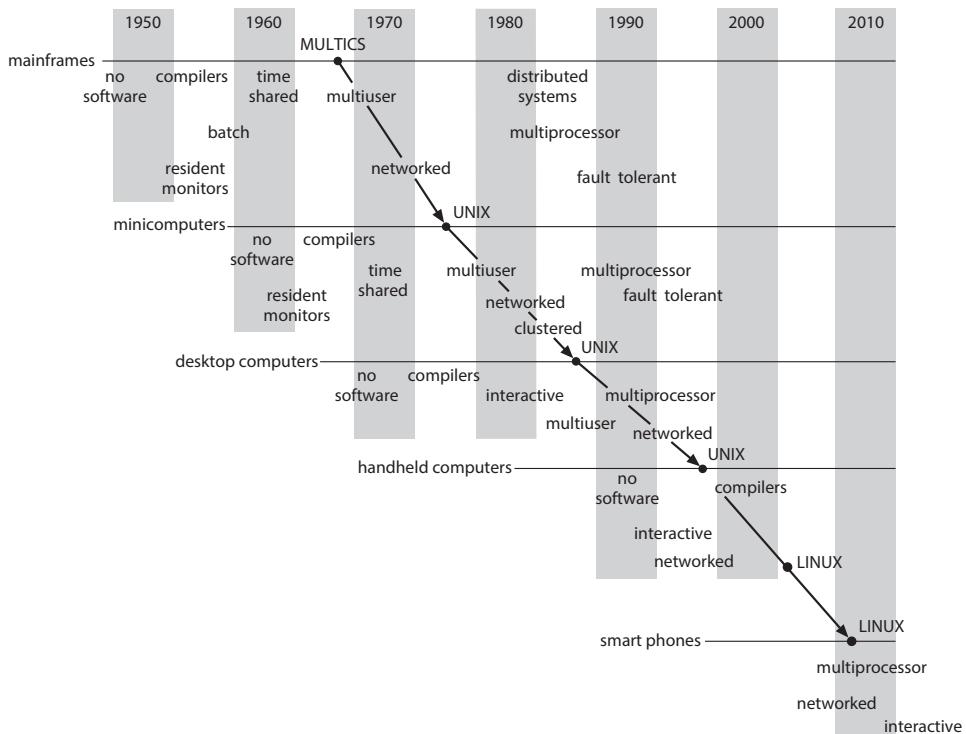
## CHAPTER OBJECTIVES

- Explain how operating-system features migrate over time from large computer systems to smaller ones.
- Discuss the features of several historically important operating systems.

### A.1 Feature Migration

One reason to study early architectures and operating systems is that a feature that once ran only on huge systems may eventually make its way into very small systems. Indeed, an examination of operating systems for mainframes and microcomputers shows that many features once available only on mainframes have been adopted for microcomputers. The same operating-system concepts are thus appropriate for various classes of computers: mainframes, minicomputers, microcomputers, and handhelds. To understand modern operating systems, then, you need to recognize the theme of feature migration and the long history of many operating-system features, as shown in Figure A.1.

A good example of feature migration started with the Multiplexed Information and Computing Services (MULTICS) operating system. MULTICS was devel-



**Figure A.1** Migration of operating-system concepts and features.

oped from 1965 to 1970 at the Massachusetts Institute of Technology (MIT) as a computing **utility**. It ran on a large, complex mainframe computer (the GE-645). Many of the ideas that were developed for MULTICS were subsequently used at Bell Laboratories (one of the original partners in the development of MULTICS) in the design of UNIX. The UNIX operating system was designed around 1970 for a PDP-11 minicomputer. Around 1980, the features of UNIX became the basis for UNIX-like operating systems on microcomputers, and these features are included in several more recent operating systems for microcomputers, such as Microsoft Windows, Windows XP, and the macOS operating system. Linux includes some of these same features, and they can now be found on PDAs.

## A.2 Early Systems

We turn our attention now to a historical overview of early computer systems. We should note that the history of computing starts far before “computers” with looms and calculators. We begin our discussion, however, with the computers of the twentieth century.

Before the 1940s, computing devices were designed and implemented to perform specific, fixed tasks. Modifying one of those tasks required a great deal of effort and manual labor. All that changed in the 1940s when Alan Turing and John von Neumann (and colleagues), both separately and together, worked on the idea of a more general-purpose **stored program** computer. Such a machine

has both a program store and a data store, where the program store provides instructions about what to do to the data.

This fundamental computer concept quickly generated a number of general-purpose computers, but much of the history of these machines is blurred by time and the secrecy of their development during World War II. It is likely that the first working stored-program general-purpose computer was the Manchester Mark 1, which ran successfully in 1949. The first commercial computer—the Ferranti Mark 1, which went on sale in 1951—was its offspring.

Early computers were physically enormous machines run from consoles. The programmer, who was also the operator of the computer system, would write a program and then would operate it directly from the operator's console. First, the program would be loaded manually into memory from the front panel switches (one instruction at a time), from paper tape, or from punched cards. Then the appropriate buttons would be pushed to set the starting address and to start the execution of the program. As the program ran, the programmer/operator could monitor its execution by the display lights on the console. If errors were discovered, the programmer could halt the program, examine the contents of memory and registers, and debug the program directly from the console. Output was printed or was punched onto paper tape or cards for later printing.

### A.2.1 Dedicated Computer Systems

As time went on, additional software and hardware were developed. Card readers, line printers, and magnetic tape became commonplace. Assemblers, loaders, and linkers were designed to ease the programming task. Libraries of common functions were created. Common functions could then be copied into a new program without having to be written again, providing software reusability.

The routines that performed I/O were especially important. Each new I/O device had its own characteristics, requiring careful programming. A special subroutine called a device driver—was written for each I/O device. A device driver knows how the buffers, flags, registers, control bits, and status bits for a particular device should be used. Each type of device has its own driver. A simple task, such as reading a character from a paper-tape reader, might involve complex sequences of device-specific operations. Rather than writing the necessary code every time, the device driver was simply used from the library.

Later, compilers for FORTRAN, COBOL, and other languages appeared, making the programming task much easier but the operation of the computer more complex. To prepare a FORTRAN program for execution, for example, the programmer would first need to load the FORTRAN compiler into the computer. The compiler was normally kept on magnetic tape, so the proper tape would need to be mounted on a tape drive. The program would be read through the card reader and written onto another tape. The FORTRAN compiler produced assembly-language output, which then had to be assembled. This procedure required mounting another tape with the assembler. The output of the assembler would need to be linked to supporting library routines. Finally, the binary object form of the program would be ready to execute. It could be loaded into memory and debugged from the console, as before.

A significant amount of **setup time** could be involved in the running of a job. Each job consisted of many separate steps:

1. Loading the FORTRAN compiler tape
2. Running the compiler
3. Unloading the compiler tape
4. Loading the assembler tape
5. Running the assembler
6. Unloading the assembler tape
7. Loading the object program
8. Running the object program

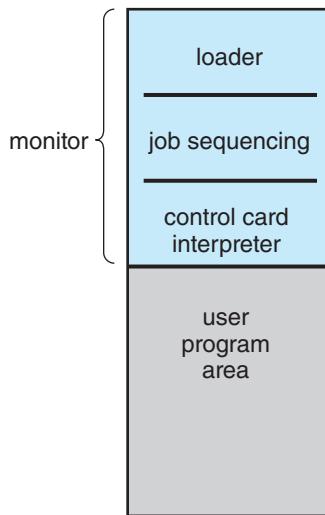
If an error occurred during any step, the programmer/operator might have to start over at the beginning. Each job step might involve the loading and unloading of magnetic tapes, paper tapes, and punch cards.

The job setup time was a real problem. While tapes were being mounted or the programmer was operating the console, the CPU sat idle. Remember that, in the early days, few computers were available, and they were expensive. A computer might have cost millions of dollars, not including the operational costs of power, cooling, programmers, and so on. Thus, computer time was extremely valuable, and owners wanted their computers to be used as much as possible. They needed high **utilization** to get as much as they could from their investments.

### **A.2.2 Shared Computer Systems**

The solution was twofold. First, a professional computer operator was hired. The programmer no longer operated the machine. As soon as one job was finished, the operator could start the next. Since the operator had more experience with mounting tapes than a programmer, setup time was reduced. The programmer provided whatever cards or tapes were needed, as well as a short description of how the job was to be run. Of course, the operator could not debug an incorrect program at the console, since the operator would not understand the program. Therefore, in the case of program error, a dump of memory and registers was taken, and the programmer had to debug from the dump. Dumping the memory and registers allowed the operator to continue immediately with the next job but left the programmer with the more difficult debugging problem.

Second, jobs with similar needs were batched together and run through the computer as a group to reduce setup time. For instance, suppose the operator received one FORTRAN job, one COBOL job, and another FORTRAN job. If she ran them in that order, she would have to set up for FORTRAN (load the compiler tapes and so on), then set up for COBOL, and then set up for FORTRAN again. If she ran the two FORTRAN programs as a batch, however, she could setup only once for FORTRAN, saving operator time.



**Figure A.2** Memory layout for a resident monitor.

But there were still problems. For example, when a job stopped, the operator would have to notice that it had stopped (by observing the console), determine *why* it stopped (normal or abnormal termination), dump memory and register (if necessary), load the appropriate device with the next job, and restart the computer. During this transition from one job to the next, the CPU sat idle.

To overcome this idle time, people developed **automatic job sequencing**. With this technique, the first rudimentary operating systems were created. A small program, called a **resident monitor**, was created to transfer control automatically from one job to the next (Figure A.2). The resident monitor is always in memory (or *resident*).

When the computer was turned on, the resident monitor was invoked, and it would transfer control to a program. When the program terminated, it would return control to the resident monitor, which would then go on to the next program. Thus, the resident monitor would automatically sequence from one program to another and from one job to another.

But how would the resident monitor know which program to execute? Previously, the operator had been given a short description of what programs were to be run on what data. **Control cards** were introduced to provide this information directly to the monitor. The idea is simple. In addition to the program or data for a job, the programmer supplied control cards, which contained directives to the resident monitor indicating what program to run. For example, a normal user program might require one of three programs to run: the FORTRAN compiler (FTN), the assembler (ASM), or the user's program (RUN). We could use a separate control card for each of these:

- \$FTN—Execute the FORTRAN compiler.
- \$ASM—Execute the assembler.
- \$RUN—Execute the user program.

These cards tell the resident monitor which program to run.

We can use two additional control cards to define the boundaries of each job:

\$JOB—First card of a job  
\$END—Final card of a job

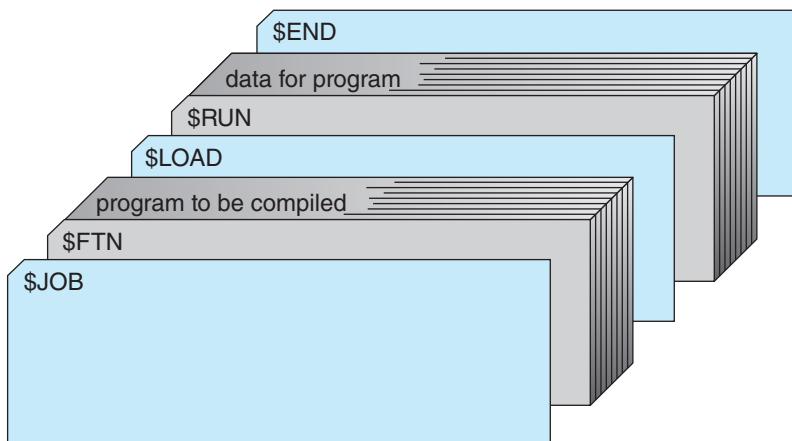
These two cards might be useful in accounting for the machine resources used by the programmer. Parameters can be used to define the job name, account number to be charged, and so on. Other control cards can be defined for other functions, such as asking the operator to load or unload a tape.

One problem with control cards is how to distinguish them from data or program cards. The usual solution is to identify them by a special character or pattern on the card. Several systems used the dollar-sign character (\$) in the first column to identify a control card. Others used a different code. IBM's Job Control Language (JCL) used slash marks (//) in the first two columns. Figure A.3 shows a sample card-deck setup for a simple batch system.

A resident monitor thus has several identifiable parts:

- The **control-card interpreter** is responsible for reading and carrying out the instructions on the cards at the point of execution.
- The **loader** is invoked by the control-card interpreter to load system programs and application programs into memory at intervals.
- The **device drivers** are used by both the control-card interpreter and the loader for the system's I/O devices. Often, the system and application programs are linked to these same device drivers, providing continuity in their operation, as well as saving memory space and programming time.

These batch systems work fairly well. The resident monitor provides automatic job sequencing as indicated by the control cards. When a control card indicates that a program is to be run, the monitor loads the program into memory and transfers control to it. When the program completes, it transfers control



**Figure A.3** Card deck for a simple batch system.

back to the monitor, which reads the next control card, loads the appropriate program, and so on. This cycle is repeated until all control cards are interpreted for the job. Then the monitor automatically continues with the next job.

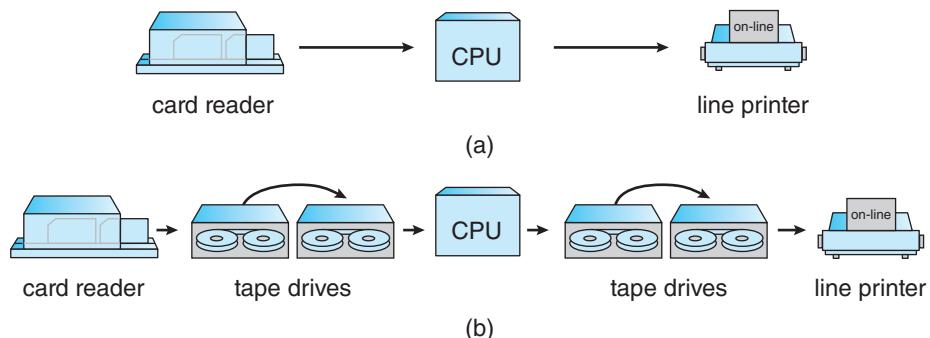
The switch to batch systems with automatic job sequencing was made to improve performance. The problem, quite simply, is that humans are considerably slower than computers. Consequently, it is desirable to replace human operation with operating-system software. Automatic job sequencing eliminates the need for human setup time and job sequencing.

Even with this arrangement, however, the CPU is often idle. The problem is the speed of the mechanical I/O devices, which are intrinsically slower than electronic devices. Even a slow CPU works in the microsecond range, with thousands of instructions executed per second. A fast card reader, in contrast, might read 1,200 cards per minute (or 20 cards per second). Thus, the difference in speed between the CPU and its I/O devices may be three orders of magnitude or more. Over time, of course, improvements in technology resulted in faster I/O devices. Unfortunately, CPU speeds increased even faster, so that the problem was not only unresolved but also exacerbated.

### A.2.3 Overlapped I/O

One common solution to the I/O problem was to replace slow card readers (input devices) and line printers (output devices) with magnetic-tape units. Most computer systems in the late 1950s and early 1960s were batch systems reading from card readers and writing to line printers or card punches. The CPU did not read directly from cards, however; instead, the cards were first copied onto a magnetic tape via a separate device. When the tape was sufficiently full, it was taken down and carried over to the computer. When a card was needed for input to a program, the equivalent record was read from the tape. Similarly, output was written to the tape, and the contents of the tape were printed later. The card readers and line printers were operated *off-line*, rather than by the main computer (Figure A.4).

An obvious advantage of off-line operation was that the main computer was no longer constrained by the speed of the card readers and line printers but was limited only by the speed of the much faster magnetic tape units. The technique of using magnetic tape for all I/O could be applied with any



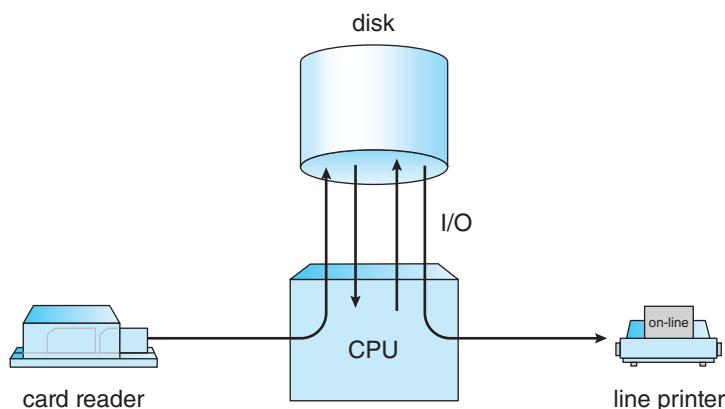
**Figure A.4** Operation of I/O devices (a) on-line and (b) off-line.

similar equipment (such as card readers, card punches, plotters, paper tape, and printers).

The real gain in off-line operation comes from the possibility of using multiple reader-to-tape and tape-to-printer systems for one CPU. If the CPU can process input twice as fast as the reader can read cards, then two readers working simultaneously can produce enough tape to keep the CPU busy. There is a disadvantage, too, however—a longer delay in getting a particular job run. The job must first be read onto tape. Then it must wait until enough additional jobs are read onto the tape to “fill” it. The tape must then be rewound, unloaded, hand-carried to the CPU, and mounted on a free tape drive. This process is not unreasonable for batch systems, of course. Many similar jobs can be batched onto a tape before it is taken to the computer.

Although off-line preparation of jobs continued for some time, it was quickly replaced in most systems. Disk systems became widely available and greatly improved on off-line operation. One problem with tape systems was that the card reader could not write onto one end of the tape while the CPU read from the other. The entire tape had to be written before it was rewound and read, because tapes are by nature **sequential-access devices**. Disk systems eliminated this problem by being **random-access devices**. Because the head is moved from one area of the disk to another, it can switch rapidly from the area on the disk being used by the card reader to store new cards to the position needed by the CPU to read the “next” card.

In a disk system, cards are read directly from the card reader onto the disk. The location of card images is recorded in a table kept by the operating system. When a job is executed, the operating system satisfies its requests for card-reader input by reading from the disk. Similarly, when the job requests the printer to output a line, that line is copied into a system buffer and is written to the disk. When the job is completed, the output is actually printed. This form of processing is called **spooling** (Figure A.5); the name is an acronym for simultaneous peripheral operation on-line. Spooling, in essence, uses the disk as a huge buffer for reading as far ahead as possible on input devices and for storing output files until the output devices are able to accept them.



**Figure A.5** Spooling.

Spooling is also used for processing data at remote sites. The CPU sends the data via communication paths to a remote printer (or accepts an entire input job from a remote card reader). The remote processing is done at its own speed, with no CPU intervention. The CPU just needs to be notified when the processing is completed, so that it can spool the next batch of data.

Spooling overlaps the I/O of one job with the computation of other jobs. Even in a simple system, the spooler may be reading the input of one job while printing the output of a different job. During this time, still another job (or other jobs) may be executed, reading its “cards” from disk and “printing” its output lines onto the disk.

Spooling has a direct beneficial effect on the performance of the system. For the cost of some disk space and a few tables, the computation of one job and the I/O of other jobs can take place at the same time. Thus, spooling can keep both the CPU and the I/O devices working at much higher rates. Spooling leads naturally to multiprogramming, which is the foundation of all modern operating systems.

## A.3 Atlas

The Atlas operating system was designed at the University of Manchester in England in the late 1950s and early 1960s. Many of its basic features that were novel at the time have become standard parts of modern operating systems. Device drivers were a major part of the system. In addition, system calls were added by a set of special instructions called *extra codes*.

Atlas was a batch operating system with spooling. Spooling allowed the system to schedule jobs according to the availability of peripheral devices, such as magnetic tape units, paper tape readers, paper tape punches, line printers, card readers, and card punches.

The most remarkable feature of Atlas, however, was its memory management. **Core memory** was new and expensive at the time. Many computers, like the IBM 650, used a drum for primary memory. The Atlas system used a drum for its main memory, but it had a small amount of core memory that was used as a cache for the drum. Demand paging was used to transfer information between core memory and the drum automatically.

The Atlas system used a British computer with 48-bit words. Addresses were 24 bits but were encoded in decimal, which allowed 1 million words to be addressed. At that time, this was an extremely large address space. The physical memory for Atlas was a 98-KB-word drum and 16-KB words of core. Memory was divided into 512-word pages, providing 32 frames in physical memory. An associative memory of 32 registers implemented the mapping from a virtual address to a physical address.

If a page fault occurred, a page-replacement algorithm was invoked. One memory frame was always kept empty, so that a drum transfer could start immediately. The page-replacement algorithm attempted to predict future memory-accessing behavior based on past behavior. A reference bit for each frame was set whenever the frame was accessed. The reference bits were read into memory every 1,024 instructions, and the last 32 values of these bits were retained. This history was used to define the time since the most recent ref-

erence ( $t_1$ ) and the interval between the last two references ( $t_2$ ). Pages were chosen for replacement in the following order:

1. Any page with  $t_1 > t_2 + 1$  is considered to be no longer in use and is replaced.
2. If  $t_1 \leq t_2$  for all pages, then replace the page with the largest value for  $t_2 - t_1$ .

The page-replacement algorithm assumes that programs access memory in loops. If the time between the last two references is  $t_2$ , then another reference is expected  $t_2$  time units later. If a reference does not occur ( $t_1 > t_2$ ), it is assumed that the page is no longer being used, and the page is replaced. If all pages are still in use, then the page that will not be needed for the longest time is replaced. The time to the next reference is expected to be  $t_2 - t_1$ .

## A.4 XDS-940

The XDS-940 operating system was designed at the University of California at Berkeley in the early 1960s. Like the Atlas system, it used paging for memory management. Unlike the Atlas system, it was a time-shared system. The paging was used only for relocation; it was not used for demand paging. The virtual memory of any user process was made up of 16-KB words, whereas the physical memory was made up of 64-KB words. Each page was made up of 2-KB words. The page table was kept in registers. Since physical memory was larger than virtual memory, several user processes could be in memory at the same time. The number of users could be increased by page sharing when the pages contained read-only reentrant code. Processes were kept on a drum and were swapped in and out of memory as necessary.

The XDS-940 system was constructed from a modified XDS-930. The modifications were typical of the changes made to a basic computer to allow an operating system to be written properly. A user-monitor mode was added. Certain instructions, such as I/O and halt, were defined to be privileged. An attempt to execute a privileged instruction in user mode would trap to the operating system.

A system-call instruction was added to the user-mode instruction set. This instruction was used to create new resources, such as files, allowing the operating system to manage the physical resources. Files, for example, were allocated in 256-word blocks on the drum. A bitmap was used to manage free drum blocks. Each file had an index block with pointers to the actual data blocks. Index blocks were chained together.

The XDS-940 system also provided system calls to allow processes to create, start, suspend, and destroy subprocesses. A programmer could construct a system of processes. Separate processes could share memory for communication and synchronization. Process creation defined a tree structure, where a process is the root and its subprocesses are nodes below it in the tree. Each of the subprocesses could, in turn, create more subprocesses.

## A.5 THE

The THE operating system was designed at the Technische Hogeschool in Eindhoven in the Netherlands in the mid-1960s. It was a batch system running on a Dutch computer, the EL X8, with 32-KB of 27-bit words. The system was mainly noted for its clean design, particularly its layer structure, and its use of a set of concurrent processes employing semaphores for synchronization.

Unlike the processes in the XDS-940 system, the set of processes in the THE system was static. The operating system itself was designed as a set of cooperating processes. In addition, five user processes were created that served as the active agents to compile, execute, and print user programs. When one job was finished, the process would return to the input queue to select another job.

A priority CPU-scheduling algorithm was used. The priorities were recomputed every 2 seconds and were inversely proportional to the amount of CPU time used recently (in the last 8 to 10 seconds). This scheme gave higher priority to I/O-bound processes and to new processes.

Memory management was limited by the lack of hardware support. However, since the system was limited and user programs could be written only in Algol, a software paging scheme was used. The Algol compiler automatically generated calls to system routines, which made sure the requested information was in memory, swapping if necessary. The backing store was a 512-KB-word drum. A 512-word page was used, with an LRU page-replacement strategy.

Another major concern of the THE system was deadlock control. The banker's algorithm was used to provide deadlock avoidance.

Closely related to the THE system is the Venus system. The Venus system was also a layer-structured design, using semaphores to synchronize processes. The lower levels of the design were implemented in microcode, however, providing a much faster system. Paged-segmented memory was used for memory management. In addition, the system was designed as a time-sharing system rather than a batch system.

## A.6 RC 4000

The RC 4000 system, like the THE system, was notable primarily for its design concepts. It was designed in the late 1960s for the Danish 4000 computer by Regnecentralen, particularly by Brinch-Hansen. The objective was not to design a batch system, or a time-sharing system, or any other specific system. Rather, the goal was to create an operating-system nucleus, or kernel, on which a complete operating system could be built. Thus, the system structure was layered, and only the lower levels—comprising the kernel—were provided.

The kernel supported a collection of concurrent processes. A round-robin CPU scheduler was used. Although processes could share memory, the primary communication and synchronization mechanism was the **message system** provided by the kernel. Processes could communicate with each other by exchanging fixed-sized messages of eight words in length. All messages were stored in buffers from a common buffer pool. When a message buffer was no longer required, it was returned to the common pool.

A **message queue** was associated with each process. It contained all the messages that had been sent to that process but had not yet been received. Messages were removed from the queue in FIFO order. The system supported four primitive operations, which were executed atomically:

- `send-message (in receiver, in message, out buffer)`
- `wait-message (out sender, out message, out buffer)`
- `send-answer (out result, in message, in buffer)`
- `wait-answer (out result, out message, in buffer)`

The last two operations allowed processes to exchange several messages at a time.

These primitives required that a process service its message queue in FIFO order and that it block itself while other processes were handling its messages. To remove these restrictions, the developers provided two additional communication primitives that allowed a process to wait for the arrival of the next message or to answer and service its queue in any order:

- `wait-event (in previous-buffer, out next-buffer, out result)`
- `get-event (out buffer)`

I/O devices were also treated as processes. The device drivers were code that converted the device interrupts and registers into messages. Thus, a process would write to a terminal by sending that terminal a message. The device driver would receive the message and output the character to the terminal. An input character would interrupt the system and transfer to a device driver. The device driver would create a message from the input character and send it to a waiting process.

## A.7 CTSS

The Compatible Time-Sharing System (CTSS) was designed at MIT as an experimental time-sharing system and first appeared in 1961. It was implemented on an IBM 7090 and eventually supported up to 32 interactive users. The users were provided with a set of interactive commands that allowed them to manipulate files and to compile and run programs through a terminal.

The 7090 had a 32-KB memory made up of 36-bit words. The monitor used 5-KB words, leaving 27 KB for the users. User memory images were swapped between memory and a fast drum. CPU scheduling employed a multilevel-feedback-queue algorithm. The time quantum for level  $i$  was  $2 * i$  time units. If a program did not finish its CPU burst in one time quantum, it was moved down to the next level of the queue, giving it twice as much time. The program at the highest level (with the shortest quantum) was run first. The initial level of a program was determined by its size, so that the time quantum was at least as long as the swap time.

CTSS was extremely successful and was in use as late as 1972. Although it was limited, it succeeded in demonstrating that time sharing was a con-

venient and practical mode of computing. One result of CTSS was increased development of time-sharing systems. Another result was the development of MULTICS.

## A.8 MULTICS

The MULTICS operating system was designed from 1965 to 1970 at MIT as a natural extension of CTSS. CTSS and other early time-sharing systems were so successful that they created an immediate desire to proceed quickly to bigger and better systems. As larger computers became available, the designers of CTSS set out to create a time-sharing utility. Computing service would be provided like electrical power. Large computer systems would be connected by telephone wires to terminals in offices and homes throughout a city. The operating system would be a time-shared system running continuously with a vast file system of shared programs and data.

MULTICS was designed by a team from MIT, GE (which later sold its computer department to Honeywell), and Bell Laboratories (which dropped out of the project in 1969). The basic GE 635 computer was modified to a new computer system called the GE 645, mainly by the addition of paged-segmentation memory hardware.

In MULTICS, a virtual address was composed of an 18-bit segment number and a 16-bit word offset. The segments were then paged in 1-KB-word pages. The second-chance page-replacement algorithm was used.

The segmented virtual address space was merged into the file system; each segment was a file. Segments were addressed by the name of the file. The file system itself was a multilevel tree structure, allowing users to create their own subdirectory structures.

Like CTSS, MULTICS used a multilevel feedback queue for CPU scheduling. Protection was accomplished through an access list associated with each file and a set of protection rings for executing processes. The system, which was written almost entirely in PL/1, comprised about 300,000 lines of code. It was extended to a multiprocessor system, allowing a CPU to be taken out of service for maintenance while the system continued running.

## A.9 IBM OS/360

The longest line of operating-system development is undoubtedly that of IBM computers. The early IBM computers, such as the IBM 7090 and the IBM 7094, are prime examples of the development of common I/O subroutines, followed by development of a resident monitor, privileged instructions, memory protection, and simple batch processing. These systems were developed separately, often at independent sites. As a result, IBM was faced with many different computers, with different languages and different system software.

The IBM/360—which first appeared in the mid 1960s—was designed to alter this situation. The IBM/360 was designed as a family of computers spanning the complete range from small business machines to large scientific machines. Only one set of software would be needed for these systems, which all used the same operating system: OS/360. This arrangement was intended to

reduce maintenance problems for IBM and to allow users to move programs and applications freely from one IBM system to another.

Unfortunately, OS/360 tried to be all things to all people. As a result, it did none of its tasks especially well. The file system included a type field that defined the type of each file, and different file types were defined for fixed-length and variable-length records and for blocked and unblocked files. Contiguous allocation was used, so the user had to guess the size of each output file. The Job Control Language (JCL) added parameters for every possible option, making it incomprehensible to the average user.

The memory-management routines were hampered by the architecture. Although a base-register addressing mode was used, the program could access and modify the base register, so that absolute addresses were generated by the CPU. This arrangement prevented dynamic relocation; the program was bound to physical memory at load time. Two separate versions of the operating system were produced: OS/MFT used fixed regions and OS/MVT used variable regions.

The system was written in assembly language by thousands of programmers, resulting in millions of lines of code. The operating system itself required large amounts of memory for its code and tables. Operating-system overhead often consumed one-half of the total CPU cycles. Over the years, new versions were released to add new features and to fix errors. However, fixing one error often caused another in some remote part of the system, so that the number of known errors in the system remained fairly constant.

Virtual memory was added to OS/360 with the change to the IBM/370 architecture. The underlying hardware provided a segmented-paged virtual memory. New versions of OS used this hardware in different ways. OS/VS1 created one large virtual address space and ran OS/MFT in that virtual memory. Thus, the operating system itself was paged, as well as user programs. OS/VS2 Release 1 ran OS/MVT in virtual memory. Finally, OS/VS2 Release 2, which is now called MVS, provided each user with his own virtual memory.

MVS is still basically a batch operating system. The CTSS system was run on an IBM 7094, but the developers at MIT decided that the address space of the 360, IBM's successor to the 7094, was too small for MULTICS, so they switched vendors. IBM then decided to create its own time-sharing system, TSS/360. Like MULTICS, TSS/360 was supposed to be a large, time-shared utility. The basic 360 architecture was modified in the model 67 to provide virtual memory. Several sites purchased the 360/67 in anticipation of TSS/360.

TSS/360 was delayed, however, so other time-sharing systems were developed as temporary systems until TSS/360 was available. A time-sharing option (TSO) was added to OS/360. IBM's Cambridge Scientific Center developed CMS as a single-user system and CP/67 to provide a virtual machine to run it on.

When TSS/360 was eventually delivered, it was a failure. It was too large and too slow. As a result, no site would switch from its temporary system to TSS/360. Today, time sharing on IBM systems is largely provided either by TSO under MVS or by CMS under CP/67 (renamed VM).

Neither TSS/360 nor MULTICS achieved commercial success. What went wrong? Part of the problem was that these advanced systems were too large and too complex to be understood. Another problem was the assumption that computing power would be available from a large, remote source. Minicom-

puters came along and decreased the need for large monolithic systems. They were followed by workstations and then personal computers, which put computing power closer and closer to the end users.

## A.10 TOPS-20

DEC created many influential computer systems during its history. Probably the most famous operating system associated with DEC is VMS, a popular business-oriented system that is still in use today as OpenVMS, a product of Hewlett-Packard. But perhaps the most influential of DEC's operating systems was TOPS-20.

TOPS-20 started life as a research project at Bolt, Beranek, and Newman (BBN) around 1970. BBN took the business-oriented DEC PDP-10 computer running TOPS-10, added a hardware memory-paging system to implement virtual memory, and wrote a new operating system for that computer to take advantage of the new hardware features. The result was TENEX, a general-purpose time-sharing system. DEC then purchased the rights to TENEX and created a new computer with a built-in hardware pager. The resulting system was the DECSYSTEM-20 and the TOPS-20 operating system.

TOPS-20 had an advanced command-line interpreter that provided help as needed to users. That, in combination with the power of the computer and its reasonable price, made the DECSYSTEM-20 the most popular time-sharing system of its time. In 1984, DEC stopped work on its line of 36-bit PDP-10 computers to concentrate on 32-bit VAX systems running VMS.

## A.11 CP/M and MS/DOS

Early hobbyist computers were typically built from kits and ran a single program at a time. The systems evolved into more advanced systems as computer components improved. An early "standard" operating system for these computers of the 1970s was **CP/M**, short for Control Program/Monitor, written by Gary Kindall of Digital Research, Inc. CP/M ran primarily on the first "personal computer" CPU, the 8-bit Intel 8080. CP/M originally supported only 64 KB of memory and ran only one program at a time. Of course, it was text-based, with a command interpreter. The command interpreter resembled those in other operating systems of the time, such as the TOPS-10 from DEC.

When IBM entered the personal computer business, it decided to have Bill Gates and company write a new operating system for its 16-bit CPU of choice—the Intel 8086. This operating system, **MS-DOS**, was similar to CP/M but had a richer set of built-in commands, again mostly modeled after TOPS-10. MS-DOS became the most popular personal-computer operating system of its time, starting in 1981 and continuing development until 2000. It supported 640 KB of memory, with the ability to address "extended" and "expanded" memory to get somewhat beyond that limit. It lacked fundamental current operating-system features, however, especially protected memory.

## A.12 Macintosh Operating System and Windows

With the advent of 16-bit CPUs, operating systems for personal computers could become more advanced, feature rich, and usable. The [Apple Macintosh](#) computer was arguably the first computer with a GUI designed for home users. It was certainly the most successful for a while, starting at its launch in 1984. It used a mouse for screen pointing and selecting and came with many utility programs that took advantage of the new user interface. Hard-disk drives were relatively expensive in 1984, so it came only with a 400-KB-capacity floppy drive by default.

The original Mac OS ran only on Apple computers and slowly was eclipsed by Microsoft Windows (starting with Version 1.0 in 1985), which was licensed to run on many different computers from a multitude of companies. As microprocessor CPUs evolved to 32-bit chips with advanced features, such as protected memory and context switching, these operating systems added features that had previously been found only on mainframes and minicomputers. Over time, personal computers became as powerful as those systems and more useful for many purposes. Minicomputers died out, replaced by general- and special-purpose “servers.” Although personal computers continue to increase in capacity and performance, servers tend to stay ahead of them in amount of memory, disk space, and number and speed of available CPUs. Today, servers typically run in data centers or machine rooms, while personal computers sit on or next to desks and talk to each other and servers across a network.

The desktop rivalry between Apple and Microsoft continues today, with new versions of Windows and Mac OS trying to outdo each other in features, usability, and application functionality. Other operating systems, such as AmigaOS and OS/2, have appeared over time but have not been long-term competitors to the two leading desktop operating systems. Meanwhile, Linux in its many forms continues to gain in popularity among more technical users—and even with nontechnical users on systems like the [One Laptop per Child \(OLPC\)](#) children’s connected computer network (<http://laptop.org/>).

## A.13 Mach

The Mach operating system traces its ancestry to the Accent operating system developed at Carnegie Mellon University (CMU). Mach’s communication system and philosophy are derived from Accent, but many other significant portions of the system (for example, the virtual memory system and task and thread management) were developed from scratch.

Work on Mach began in the mid 1980. The operating system was designed with the following three critical goals in mind:

1. Emulate 4.3 BSD UNIX so that the executable files from a UNIX system can run correctly under Mach.
2. Be a modern operating system that supports many memory models, as well as parallel and distributed computing.
3. Have a kernel that is simpler and easier to modify than 4.3 BSD.

Mach's development followed an evolutionary path from BSD UNIX systems. Mach code was initially developed inside the 4.2 BSD kernel, with BSD kernel components replaced by Mach components as the Mach components were completed. The BSD components were updated to 4.3 BSD when that became available. By 1986, the virtual memory and communication subsystems were running on the DEC VAX computer family, including multiprocessor versions of the VAX. Versions for the IBM RT/PC and for SUN 3 workstations followed shortly. Then, 1987 saw the completion of the Encore Multimax and Sequent Balance multiprocessor versions, including task and thread support, as well as the first official releases of the system, Release 0 and Release 1.

Through Release 2, Mach provided compatibility with the corresponding BSD systems by including much of BSD's code in the kernel. The new features and capabilities of Mach made the kernels in these releases larger than the corresponding BSD kernels. Mach 3 moved the BSD code outside the kernel, leaving a much smaller microkernel. This system implements only basic Mach features in the kernel; all UNIX-specific code has been evicted to run in user-mode servers. Excluding UNIX-specific code from the kernel allows the replacement of BSD with another operating system or the simultaneous execution of multiple operating-system interfaces on top of the microkernel. In addition to BSD, user-mode implementations have been developed for DOS, the Macintosh operating system, and OSF/1. This approach has similarities to the virtual machine concept, but here the virtual machine is defined by software (the Mach kernel interface), rather than by hardware. With Release 3.0, Mach became available on a wide variety of systems, including single-processor SUN, Intel, IBM, and DEC machines and multiprocessor DEC, Sequent, and Encore systems.

Mach was propelled to the forefront of industry attention when the Open Software Foundation (OSF) announced in 1989 that it would use Mach 2.5 as the basis for its new operating system, OSF/1. (Mach 2.5 was also the basis for the operating system on the NeXT workstation, the brainchild of Steve Jobs of Apple Computer fame.) The initial release of OSF/1 occurred a year later, and this system competed with UNIX System V, Release 4, the operating system of choice at that time among UNIX International (UI) members. OSF members included key technological companies such as IBM, DEC, and HP. OSF has since changed its direction, and only DEC UNIX is based on the Mach kernel.

Unlike UNIX, which was developed without regard for multiprocessing, Mach incorporates multiprocessing support throughout. This support is also exceedingly flexible, ranging from shared-memory systems to systems with no memory shared between processors. Mach uses lightweight processes, in the form of multiple threads of execution within one task (or address space), to support multiprocessing and parallel computation. Its extensive use of messages as the only communication method ensures that protection mechanisms are complete and efficient. By integrating messages with the virtual memory system, Mach also ensures that messages can be handled efficiently. Finally, by having the virtual memory system use messages to communicate with the daemons managing the backing store, Mach provides great flexibility in the design and implementation of these memory-object-managing tasks. By providing low-level, or primitive, system calls from which more complex functions can be built, Mach reduces the size of the kernel while permitting operating-system emulation at the user level, much like IBM's virtual machine systems.

Today, the only remaining pure Mach implementation is in GNU HURD, a little-used operating system. Mach still lives on, however, in XNU—the kernel driving macOS and the iOS variants. XNU—whose codebase Apple obtained with the acquisition of NeXT and its NeXTSTEP operating system—is a Mach core with a top layer of BSD APIs. Apple continues to support and maintain the Mach APIs (still accessible through specialized system calls known as traps, and via Mach Messages), and the kernel continues evolving with new features to this day.

Some previous editions of *Operating System Concepts* included an entire chapter on Mach. This chapter, as it appeared in the fourth edition, is available on the web (<http://www.os-book.com>).

## A.14 Capability-based Systems—Hydra and CAP

In this section, we survey two capability-based protection systems. These systems differ in their complexity and in the types of policies that can be implemented on them. Neither system is widely used, but both provide interesting proving grounds for protection theories.

### A.14.1 Hydra

Hydra is a capability-based protection system that provides considerable flexibility. The system implements a fixed set of possible access rights, including such basic forms of access as the right to read, write, or execute a memory segment. In addition, a user (of the protection system) can declare other rights. The interpretation of user-defined rights is performed solely by the user's program, but the system provides access protection for the use of these rights, as well as for the use of system-defined rights. These facilities constitute a significant development in protection technology.

Operations on objects are defined procedurally. The procedures that implement such operations are themselves a form of object, and they are accessed indirectly by capabilities. The names of user-defined procedures must be identified to the protection system if it is to deal with objects of the user-defined type. When the definition of an object is made known to Hydra, the names of operations on the type become **auxiliary rights**. Auxiliary rights can be described in a capability for an instance of the type. For a process to perform an operation on a typed object, the capability it holds for that object must contain the name of the operation being invoked among its auxiliary rights. This restriction enables discrimination of access rights to be made on an instance-by-instance and process-by-process basis.

Hydra also provides **rights amplification**. This scheme allows a procedure to be certified as *trustworthy* to act on a formal parameter of a specified type on behalf of any process that holds a right to execute the procedure. The rights held by a trustworthy procedure are independent of, and may exceed, the rights held by the calling process. However, such a procedure must not be regarded as universally trustworthy (the procedure is not allowed to act on other types, for instance), and the trustworthiness must not be extended to any other procedures or program segments that might be executed by a process.

Amplification allows implementation procedures access to the representation variables of an abstract data type. If a process holds a capability to a typed object  $A$ , for instance, this capability may include an auxiliary right to invoke some operation  $P$  but does not include any of the so-called kernel rights, such as read, write, or execute, on the segment that represents  $A$ . Such a capability gives a process a means of indirect access (through the operation  $P$ ) to the representation of  $A$ , but only for specific purposes.

When a process invokes the operation  $P$  on an object  $A$ , however, the capability for access to  $A$  may be amplified as control passes to the code body of  $P$ . This amplification may be necessary to allow  $P$  the right to access the storage segment representing  $A$  so as to implement the operation that  $P$  defines on the abstract data type. The code body of  $P$  may be allowed to read or to write to the segment of  $A$  directly, even though the calling process cannot. On return from  $P$ , the capability for  $A$  is restored to its original, unamplified state. This case is a typical one in which the rights held by a process for access to a protected segment must change dynamically, depending on the task to be performed. The dynamic adjustment of rights is performed to guarantee consistency of a programmer-defined abstraction. Amplification of rights can be stated explicitly in the declaration of an abstract type to the Hydra operating system.

When a user passes an object as an argument to a procedure, we may need to ensure that the procedure cannot modify the object. We can implement this restriction readily by passing an access right that does not have the modification (write) right. However, if amplification may occur, the right to modify may be reinstated. Thus, the user-protection requirement can be circumvented. In general, of course, a user may trust that a procedure performs its task correctly. This assumption is not always correct, however, because of hardware or software errors. Hydra solves this problem by restricting amplifications.

The procedure-call mechanism of Hydra was designed as a direct solution to the *problem of mutually suspicious subsystems*. This problem is defined as follows. Suppose that a program can be invoked as a service by a number of different users (for example, a sort routine, a compiler, a game). When users invoke this service program, they take the risk that the program will malfunction and will either damage the given data or retain some access right to the data to be used (without authority) later. Similarly, the service program may have some private files (for accounting purposes, for example) that should not be accessed directly by the calling user program. Hydra provides mechanisms for directly dealing with this problem.

A Hydra subsystem is built on top of its protection kernel and may require protection of its own components. A subsystem interacts with the kernel through calls on a set of kernel-defined primitives that define access rights to resources defined by the subsystem. The subsystem designer can define policies for use of these resources by user processes, but the policies are enforced by use of the standard access protection provided by the capability system.

Programmers can make direct use of the protection system after acquainting themselves with its features in the appropriate reference manual. Hydra provides a large library of system-defined procedures that can be called by user programs. Programmers can explicitly incorporate calls on these system procedures into their program code or can use a program translator that has been interfaced to Hydra.

### A.14.2 Cambridge CAP System

A different approach to capability-based protection has been taken in the design of the Cambridge CAP system. CAP's capability system is simpler and superficially less powerful than that of Hydra. However, closer examination shows that it, too, can be used to provide secure protection of user-defined objects. CAP has two kinds of capabilities. The ordinary kind is called a **data capability**. It can be used to provide access to objects, but the only rights provided are the standard read, write, and execute of the individual storage segments associated with the object. Data capabilities are interpreted by microcode in the CAP machine.

The second kind of capability is the so-called **software capability**, which is protected, but not interpreted, by the CAP microcode. It is interpreted by a *protected* (that is, privileged) procedure, which may be written by an application programmer as part of a subsystem. A particular kind of rights amplification is associated with a protected procedure. When executing the code body of such a procedure, a process temporarily acquires the right to read or write the contents of a software capability itself. This specific kind of rights amplification corresponds to an implementation of the `seal` and `unseal` primitives on capabilities. Of course, this privilege is still subject to type verification to ensure that only software capabilities for a specified abstract type are passed to any such procedure. Universal trust is not placed in any code other than the CAP machine's microcode. (See the bibliographical notes at the end of the chapter for references.)

The interpretation of a software capability is left completely to the subsystem, through the protected procedures it contains. This scheme allows a variety of protection policies to be implemented. Although programmers can define their own protected procedures (any of which might be incorrect), the security of the overall system cannot be compromised. The basic protection system will not allow an unverified, user-defined, protected procedure access to any storage segments (or capabilities) that do not belong to the protection environment in which it resides. The most serious consequence of an insecure protected procedure is a protection breakdown of the subsystem for which that procedure has responsibility.

The designers of the CAP system have noted that the use of software capabilities allowed them to realize considerable economies in formulating and implementing protection policies commensurate with the requirements of abstract resources. However, subsystem designers who want to make use of this facility cannot simply study a reference manual, as is the case with Hydra. Instead, they must learn the principles and techniques of protection, since the system provides them with no library of procedures.

## A.15 Other Systems

There are, of course, other operating systems, and most of them have interesting properties. The MCP operating system for the Burroughs computer family was the first to be written in a system programming language. It supported segmentation and multiple CPUs. The SCOPE operating system for the CDC 6600 was also a multi-CPU system. The coordination and synchronization of the multiple processes were surprisingly well designed.

History is littered with operating systems that suited a purpose for a time (be it a long or a short time) and then, when faded, were replaced by operating systems that had more features, supported newer hardware, were easier to use, or were better marketed. We are sure this trend will continue in the future.

## Further Reading

Looms and calculators are described in [Frah (2001)] and shown graphically in [Frauenfelder (2005)].

The Manchester Mark 1 is discussed by [Rojas and Hashagen (2000)], and its offspring, the Ferranti Mark 1, is described by [Ceruzzi (1998)].

[Kilburn et al. (1961)] and [Howarth et al. (1961)] examine the Atlas operating system.

The XDS-940 operating system is described by [Lichtenberger and Pirtle (1965)].

The THE operating system is covered by [Dijkstra (1968)] and by [McKeag and Wilson (1976)].

The Venus system is described by [Liskov (1972)].

[Brinch-Hansen (1970)] and [Brinch-Hansen (1973)] discuss the RC 4000 system.

The Compatible Time-Sharing System (CTSS) is presented by [Corbato et al. (1962)].

The MULTICS operating system is described by [Corbato and Vyssotsky (1965)] and [Organick (1972)].

[Mealy et al. (1966)] presented the IBM/360. [Lett and Konigsford (1968)] cover TSS/360.

CP/67 is described by [Meyer and Seawright (1970)] and [Parmelee et al. (1972)].

DEC VMS is discussed by [Kenah et al. (1988)], and TENEX is described by [Bobrow et al. (1972)].

A description of the Apple Macintosh appears in [Apple (1987)]. For more information on these operating systems and their history, see [Freiberger and Swaine (2000)].

The Mach operating system and its ancestor, the Accent operating system, are described by [Rashid and Robertson (1981)]. Mach's communication system is covered by [Rashid (1986)], [Tevanian et al. (1989)], and [Accetta et al. (1986)]. The Mach scheduler is described in detail by [Tevanian et al. (1987a)] and [Black (1990)]. An early version of the Mach shared-memory and memory-mapping system is presented by [Tevanian et al. (1987b)]. A good resource describing the Mach project can be found at <http://www.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html>.

[McKeag and Wilson (1976)] discuss the MCP operating system for the Burroughs computer family as well as the SCOPE operating system for the CDC 6600.

The Hydra system was described by [Wulf et al. (1981)]. The CAP system was described by [Needham and Walker (1977)]. [Organick (1972)] discussed the MULTICS ring-protection system.

## Bibliography

- [Accetta et al. (1986)]** M. Accetta, R. Baron, W. Bolosky, D. B. Golub, R. Rashid, A. Tevanian, and M. Young, “Mach: A New Kernel Foundation for UNIX Development”, *Proceedings of the Summer USENIX Conference* (1986), pages 93–112.
- [Apple (1987)]** Apple Technical Introduction to the Macintosh Family. Addison-Wesley (1987).
- [Black (1990)]** D. L. Black, “Scheduling Support for Concurrency and Parallelism in the Mach Operating System”, *IEEE Computer*, Volume 23, Number 5 (1990), pages 35–43.
- [Bobrow et al. (1972)]** D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, “TENEX, a Paged Time Sharing System for the PDP-10”, *Communications of the ACM*, Volume 15, Number 3 (1972).
- [Brinch-Hansen (1970)]** P. Brinch-Hansen, “The Nucleus of a Multiprogramming System”, *Communications of the ACM*, Volume 13, Number 4 (1970), pages 238–241 and 250.
- [Brinch-Hansen (1973)]** P. Brinch-Hansen, *Operating System Principles*, Prentice Hall (1973).
- [Ceruzzi (1998)]** P. E. Ceruzzi, *A History of Modern Computing*, MIT Press (1998).
- [Corbato and Vyssotsky (1965)]** F. J. Corbato and V. A. Vyssotsky, “Introduction and Overview of the MULTICS System”, *Proceedings of the AFIPS Fall Joint Computer Conference* (1965), pages 185–196.
- [Corbato et al. (1962)]** F. J. Corbato, M. Merwin-Daggett, and R. C. Daley, “An Experimental Time-Sharing System”, *Proceedings of the AFIPS Fall Joint Computer Conference* (1962), pages 335–344.
- [Dijkstra (1968)]** E. W. Dijkstra, “The Structure of the THE Multiprogramming System”, *Communications of the ACM*, Volume 11, Number 5 (1968), pages 341–346.
- [Frah (2001)]** G. Frah, *The Universal History of Computing*, John Wiley and Sons (2001).
- [Frauenfelder (2005)]** M. Frauenfelder, *The Computer—An Illustrated History*, Carlton Books (2005).
- [Freiberger and Swaine (2000)]** P. Freiberger and M. Swaine, *Fire in the Valley—The Making of the Personal Computer*, McGraw-Hill (2000).
- [Howarth et al. (1961)]** D. J. Howarth, R. B. Payne, and F. H. Sumner, “The Manchester University Atlas Operating System, Part II: User’s Description”, *Computer Journal*, Volume 4, Number 3 (1961), pages 226–229.
- [Kenah et al. (1988)]** L. J. Kenah, R. E. Goldenberg, and S. F. Bate, *VAX/VMS Internals and Data Structures*, Digital Press (1988).
- [Kilburn et al. (1961)]** T. Kilburn, D. J. Howarth, R. B. Payne, and F. H. Sumner, “The Manchester University Atlas Operating System, Part I: Internal Organization”, *Computer Journal*, Volume 4, Number 3 (1961), pages 222–225.

- [Lett and Konigsford (1968)]** A. L. Lett and W. L. Konigsford, “TSS/360: A Time-Shared Operating System”, *Proceedings of the AFIPS Fall Joint Computer Conference* (1968), pages 15–28.
- [Lichtenberger and Pirtle (1965)]** W. W. Lichtenberger and M. W. Pirtle, “A Facility for Experimentation in Man-Machine Interaction”, *Proceedings of the AFIPS Fall Joint Computer Conference* (1965), pages 589–598.
- [Liskov (1972)]** B. H. Liskov, “The Design of the Venus Operating System”, *Communications of the ACM*, Volume 15, Number 3 (1972), pages 144–149.
- [McKeag and Wilson (1976)]** R. M. McKeag and R. Wilson, *Studies in Operating Systems*, Academic Press (1976).
- [Mealy et al. (1966)]** G. H. Mealy, B. I. Witt, and W. A. Clark, “The Functional Structure of OS/360”, *IBM Systems Journal*, Volume 5, Number 1 (1966), pages 3–11.
- [Meyer and Seawright (1970)]** R. A. Meyer and L. H. Seawright, “A Virtual Machine Time-Sharing System”, *IBM Systems Journal*, Volume 9, Number 3 (1970), pages 199–218.
- [Needham and Walker (1977)]** R. M. Needham and R. D. H. Walker, “The Cambridge CAP Computer and Its Protection System”, *Proceedings of the Sixth Symposium on Operating System Principles* (1977), pages 1–10.
- [Organick (1972)]** E. I. Organick, *The Multics System: An Examination of Its Structure*, MIT Press (1972).
- [Parmelee et al. (1972)]** R. P. Parmelee, T. I. Peterson, C. C. Tillman, and D. Hatfield, “Virtual Storage and Virtual Machine Concepts”, *IBM Systems Journal*, Volume 11, Number 2 (1972), pages 99–130.
- [Rashid (1986)]** R. F. Rashid, “From RIG to Accent to Mach: The Evolution of a Network Operating System”, *Proceedings of the ACM/IEEE Computer Society, Fall Joint Computer Conference* (1986), pages 1128–1137.
- [Rashid and Robertson (1981)]** R. Rashid and G. Robertson, “Accent: A Communication-Oriented Network Operating System Kernel”, *Proceedings of the ACM Symposium on Operating System Principles* (1981), pages 64–75.
- [Rojas and Hashagen (2000)]** R. Rojas and U. Hashagen, *The First Computers—History and Architectures*, MIT Press (2000).
- [Tevanian et al. (1987a)]** A. Tevanian, Jr., R. F. Rashid, D. B. Golub, D. L. Black, E. Cooper, and M. W. Young, “Mach Threads and the Unix Kernel: The Battle for Control”, *Proceedings of the Summer USENIX Conference* (1987).
- [Tevanian et al. (1987b)]** A. Tevanian, Jr., R. F. Rashid, M. W. Young, D. B. Golub, M. R. Thompson, W. Bolosky, and R. Sanzi, “A UNIX Interface for Shared Memory and Memory Mapped Files Under Mach”, Technical report, Carnegie-Mellon University (1987).
- [Tevanian et al. (1989)]** A. Tevanian, Jr., and B. Smith, “Mach: The Model for Future Unix”, *Byte* (1989).

[Wulf et al. (1981)] W. A. Wulf, R. Levin, and S. P. Harbison, *Hydra/C.mmp: An Experimental Computer System*, McGraw-Hill (1981).

# Windows 7



Updated by Dave Probert

The Microsoft Windows 7 operating system is a 32-/64-bit preemptive multitasking client operating system for microprocessors implementing the Intel IA-32 and AMD64 instruction set architectures (ISAs). Microsoft's corresponding server operating system, Windows Server 2008 R2, is based on the same code as Windows 7 but supports only the 64-bit AMD64 and IA64 (Itanium) ISAs. Windows 7 is the latest in a series of Microsoft operating systems based on its NT code, which replaced the earlier systems based on Windows 95/98. In this appendix, we discuss the key goals of Windows 7, the layered architecture of the system that has made it so easy to use, the file system, the networking features, and the programming interface.

## CHAPTER OBJECTIVES

- Explore the principles underlying Windows 7's design and the specific components of the system.
- Provide a detailed discussion of the Windows 7 file system.
- Illustrate the networking protocols supported in Windows 7.
- Describe the interface available in Windows 7 to system and application programmers.
- Describe the important algorithms implemented with Windows 7.

### B.1 History

In the mid-1980s, Microsoft and IBM cooperated to develop the [OS/2 operating system](#), which was written in assembly language for single-processor Intel 80286 systems. In 1988, Microsoft decided to end the joint effort with IBM and develop its own “new technology” (or NT) portable operating system to support both the OS/2 and POSIX application-programming interfaces (APIs). In

October 1988, Dave Cutler, the architect of the DEC VAX/VMS operating system, was hired and given the charter of building Microsoft's new operating system.

Originally, the team planned to use the OS/2 API as NT's native environment, but during development, NT was changed to use a new 32-bit Windows API (called Win32), based on the popular 16-bit API used in Windows 3.0. The first versions of NT were Windows NT 3.1 and Windows NT 3.1 Advanced Server. (At that time, 16-bit Windows was at Version 3.1.) Windows NT Version 4.0 adopted the Windows 95 user interface and incorporated Internet web-server and web-browser software. In addition, user-interface routines and all graphics code were moved into the kernel to improve performance, with the side effect of decreased system reliability. Although previous versions of NT had been ported to other microprocessor architectures, the Windows 2000 version, released in February 2000, supported only Intel (and compatible) processors due to marketplace factors. Windows 2000 incorporated significant changes. It added Active Directory (an X.500-based directory service), better networking and laptop support, support for plug-and-play devices, a distributed file system, and support for more processors and more memory.

In October 2001, Windows XP was released as both an update to the Windows 2000 desktop operating system and a replacement for Windows 95/98. In 2002, the server edition of Windows XP became available (called Windows .Net Server). Windows XP updated the graphical user interface (GUI) with a visual design that took advantage of more recent hardware advances and many new *ease-of-use features*. Numerous features were added to automatically repair problems in applications and the operating system itself. As a result of these changes, Windows XP provided better networking and device experience (including zero-configuration wireless, instant messaging, streaming media, and digital photography/video), dramatic performance improvements for both the desktop and large multiprocessors, and better reliability and security than earlier Windows operating systems.

The long-awaited update to Windows XP, called Windows Vista, was released in November 2006, but it was not well received. Although Windows Vista included many improvements that later showed up in Windows 7, these improvements were overshadowed by Windows Vista's perceived sluggishness and compatibility problems. Microsoft responded to criticisms of Windows Vista by improving its engineering processes and working more closely with the makers of Windows hardware and applications. The result was **Windows 7**, which was released in October 2009, along with corresponding server editions of Windows. Among the significant engineering changes is the increased use of **execution tracing** rather than counters or profiling to analyze system behavior. Tracing runs constantly in the system, watching hundreds of scenarios execute. When one of these scenarios fails, or when it succeeds but does not perform well, the traces can be analyzed to determine the cause.

Windows 7 uses a client–server architecture (like Mach) to implement two operating-system personalities, Win32 and POSIX, with user-level processes called subsystems. (At one time, Windows also supported an OS/2 subsystem, but it was removed in Windows XP due to the demise of OS/2.) The subsystem architecture allows enhancements to be made to one operating-system personality without affecting the application compatibility of the other. Although the POSIX subsystem continues to be available for Windows 7, the Win32 API has become very popular, and the POSIX APIs are used by only a few sites. The subsystem approach continues to be interesting to study from an operating-system

perspective, but machine-virtualization technologies are now becoming the dominant way of running multiple operating systems on a single machine.

Windows 7 is a multiuser operating system, supporting simultaneous access through distributed services or through multiple instances of the GUI via the Windows terminal services. The server editions of Windows 7 support simultaneous terminal server sessions from Windows desktop systems. The desktop editions of terminal server multiplex the keyboard, mouse, and monitor between virtual terminal sessions for each logged-on user. This feature, called *fast user switching*, allows users to preempt each other at the console of a PC without having to log off and log on.

We noted earlier that some GUI implementation moved into kernel mode in Windows NT 4.0. It started to move into user mode again with Windows Vista, which included the **desktop window manager (DWM)** as a user-mode process. DWM implements the desktop compositing of Windows, providing the Windows *Aero* interface look on top of the Windows DirectX graphic software. DirectX continues to run in the kernel, as does the code implementing Windows' previous windowing and graphics models (Win32k and GDI). Windows 7 made substantial changes to the DWM, significantly reducing its memory footprint and improving its performance.

**Windows XP** was the first version of Windows to ship a 64-bit version (for the IA64 in 2001 and the AMD64 in 2005). Internally, the native NT file system (NTFS) and many of the Win32 APIs have always used 64-bit integers where appropriate—so the major extension to 64-bit in Windows XP was support for large virtual addresses. However, 64-bit editions of Windows also support much larger physical memories. By the time Windows 7 shipped, the AMD64 ISA had become available on almost all CPUs from both Intel and AMD. In addition, by that time, physical memories on client systems frequently exceeded the 4-GB limit of the IA-32. As a result, the 64-bit version of Windows 7 is now commonly installed on larger client systems. Because the AMD64 architecture supports high-fidelity IA-32 compatibility at the level of individual processes, 32- and 64-bit applications can be freely mixed in a single system.

In the rest of our description of Windows 7, we will not distinguish between the client editions of Windows 7 and the corresponding server editions. They are based on the same core components and run the same binary files for the kernel and most drivers. Similarly, although Microsoft ships a variety of different editions of each release to address different market price points, few of the differences between editions are reflected in the core of the system. In this chapter, we focus primarily on the core components of Windows 7.

## B.2 Design Principles

Microsoft's design goals for Windows included security, reliability, Windows and POSIX application compatibility, high performance, extensibility, portability, and international support. Some additional goals, energy efficiency and dynamic device support, have recently been added to this list. Next, we discuss each of these goals and how it is achieved in Windows 7.

### B.2.1 Security

Windows 7 security goals required more than just adherence to the design standards that had enabled Windows NT 4.0 to receive a C2 security classification

from the U.S. government. (A C2 classification signifies a moderate level of protection from defective software and malicious attacks. Classifications were defined by the Department of Defense Trusted Computer System Evaluation Criteria, also known as the [Orange Book](#).) Extensive code review and testing were combined with sophisticated automatic analysis tools to identify and investigate potential defects that might represent security vulnerabilities.

Windows bases security on discretionary access controls. System objects, including files, registry settings, and kernel objects, are protected by [access-control lists \(ACLs\)](#) (see Section 13.4.2). ACLs are vulnerable to user and programmer errors, however, as well as to the most common attacks on consumer systems, in which the user is tricked into running code, often while browsing the web. Windows 7 includes a mechanism called [integrity levels](#) that acts as a rudimentary *capability* system for controlling access. Objects and processes are marked as having low, medium, or high integrity. Windows does not allow a process to modify an object with a higher integrity level, no matter what the setting of the ACL.

Other security measures include [address-space layout randomization \(ASLR\)](#), nonexecutable stacks and heaps, and encryption and [digital signature](#) facilities. ASLR thwarts many forms of attack by preventing small amounts of injected code from jumping easily to code that is already loaded in a process as part of normal operation. This safeguard makes it likely that a system under attack will fail or crash rather than let the attacking code take control.

Recent chips from both Intel and AMD are based on the AMD64 architecture, which allows memory pages to be marked so that they cannot contain executable instruction code. Windows tries to mark stacks and memory heaps so that they cannot be used to execute code, thus preventing attacks in which a program bug allows a buffer to overflow and then is tricked into executing the contents of the buffer. This technique cannot be applied to all programs, because some rely on modifying data and executing it. A column labeled “data execution prevention” in the Windows task manager shows which processes are marked to prevent these attacks.

Windows uses encryption as part of common protocols, such as those used to communicate securely with websites. Encryption is also used to protect user files stored on disk from prying eyes. Windows 7 allows users to easily encrypt virtually a whole disk, as well as removable storage devices such as USB flash drives, with a feature called BitLocker. If a computer with an encrypted disk is stolen, the thieves will need very sophisticated technology (such as an electron microscope) to gain access to any of the computer’s files. Windows uses digital signatures to *sign* operating system binaries so it can verify that the files were produced by Microsoft or another known company. In some editions of Windows, a [code integrity](#) module is activated at boot to ensure that all the loaded modules in the kernel have valid signatures, assuring that they have not been tampered with by an off-line attack.

### **B.2.2 Reliability**

Windows matured greatly as an operating system in its first ten years, leading to Windows 2000. At the same time, its reliability increased due to such factors as maturity in the source code, extensive stress testing of the system, improved CPU architectures, and automatic detection of many serious errors in drivers from both Microsoft and third parties. Windows has subsequently extended

the tools for achieving reliability to include automatic analysis of source code for errors, tests that include providing invalid or unexpected input parameters (known as **fuzzing**) to detect validation failures, and an application version of the driver verifier that applies dynamic checking for an extensive set of common user-mode programming errors. Other improvements in reliability have resulted from moving more code out of the kernel and into user-mode services. Windows provides extensive support for writing drivers in user mode. System facilities that were once in the kernel and are now in user mode include the Desktop Window Manager and much of the software stack for audio.

One of the most significant improvements in the Windows experience came from adding memory diagnostics as an option at boot time. This addition is especially valuable because so few consumer PCs have error-correcting memory. When bad RAM starts to drop bits here and there, the result is frustratingly erratic behavior in the system. The availability of memory diagnostics has greatly reduced the stress levels of users with bad RAM.

Windows 7 introduced a fault-tolerant memory heap. The heap learns from application crashes and automatically inserts mitigations into future execution of an application that has crashed. This makes the application more reliable even if it contains common bugs such as using memory after freeing it or accessing past the end of the allocation.

Achieving high reliability in Windows is particularly challenging because almost one billion computers run Windows. Even reliability problems that affect only a small percentage of users still impact tremendous numbers of human beings. The complexity of the Windows ecosystem also adds to the challenges. Millions of instances of applications, drivers, and other software are being constantly downloaded and run on Windows systems. Of course, there is also a constant stream of malware attacks. As Windows itself has become harder to attack directly, exploits increasingly target popular applications.

To cope with these challenges, Microsoft is increasingly relying on communications from customer machines to collect large amounts of data from the ecosystem. Machines can be sampled to see how they are performing, what software they are running, and what problems they are encountering. Customers can send data to Microsoft when systems or software crashes or hangs. This constant stream of data from customer machines is collected very carefully, with the users' consent and without invading privacy. The result is that Microsoft is building an ever-improving picture of what is happening in the Windows ecosystem that allows continuous improvements through software updates, as well as providing data to guide future releases of Windows.

### B.2.3 Windows and POSIX Application Compatibility

As mentioned, Windows XP was both an update of Windows 2000 and a replacement for Windows 95/98. Windows 2000 focused primarily on compatibility for business applications. The requirements for Windows XP included a much greater compatibility with the consumer applications that ran on Windows 95/98. Application compatibility is difficult to achieve because many applications check for a particular version of Windows, may depend to some extent on the quirks of the implementation of APIs, may have latent application bugs that were masked in the previous system, and so forth. Applications may

also have been compiled for a different instruction set. Windows 7 implements several strategies to run applications despite incompatibilities.

Like Windows XP, Windows 7 has a compatibility layer that sits between applications and the Win32 APIs. This layer makes Windows 7 look (almost) bug-for-bug compatible with previous versions of Windows. Windows 7, like earlier NT releases, maintains support for running many 16-bit applications using a  *thunking*, or conversion, layer that translates 16-bit API calls into equivalent 32-bit calls. Similarly, the 64-bit version of Windows 7 provides a thunking layer that translates 32-bit API calls into native 64-bit calls.

The Windows subsystem model allows multiple operating-system personalities to be supported. As noted earlier, although the API most commonly used with Windows is the Win32 API, some editions of Windows 7 support a POSIX subsystem. POSIX is a standard specification for UNIX that allows most available UNIX-compatible software to compile and run without modification.

As a final compatibility measure, several editions of Windows 7 provide a virtual machine that runs Windows XP inside Windows 7. This allows applications to get bug-for-bug compatibility with Windows XP.

#### B.2.4 High Performance

Windows was designed to provide high performance on desktop systems (which are largely constrained by I/O performance), server systems (where the CPU is often the bottleneck), and large multithreaded and multiprocessor environments (where locking performance and cache-line management are keys to scalability). To satisfy performance requirements, NT used a variety of techniques, such as asynchronous I/O, optimized protocols for networks, kernel-based graphics rendering, and sophisticated caching of file-system data. The memory-management and synchronization algorithms were designed with an awareness of the performance considerations related to cache lines and multiprocessors.

Windows NT was designed for symmetrical multiprocessing (SMP); on a multiprocessor computer, several threads can run at the same time, even in the kernel. On each CPU, Windows NT uses priority-based preemptive scheduling of threads. Except while executing in the kernel dispatcher or at interrupt level, threads in any process running in Windows can be preempted by higher-priority threads. Thus, the system responds quickly (see Chapter 5).

The subsystems that constitute Windows NT communicate with one another efficiently through a **local procedure call (LPC)** facility that provides high-performance message passing. When a thread requests a synchronous service from another process through an LPC, the servicing thread is marked *ready*, and its priority is temporarily boosted to avoid the scheduling delays that would occur if it had to wait for threads already in the queue.

Windows XP further improved performance by reducing the code-path length in critical functions, using better algorithms and per-processor data structures, using memory coloring for **non-uniform memory access (NUMA)** machines, and implementing more scalable locking protocols, such as queued spinlocks. The new locking protocols helped reduce system bus cycles and included lock-free lists and queues, atomic read-modify-write operations (like interlocked increment), and other advanced synchronization techniques.

By the time Windows 7 was developed, several major changes had come to computing. Client/server computing had increased in importance, so an advanced local procedure call (ALPC) facility was introduced to provide higher performance and more reliability than LPC. The number of CPUs and the amount of physical memory available in the largest multiprocessors had increased substantially, so quite a lot of effort was put into improving operating-system scalability.

The implementation of SMP in Windows NT used bitmasks to represent collections of processors and to identify, for example, which set of processors a particular thread could be scheduled on. These bitmasks were defined as fitting within a single word of memory, limiting the number of processors supported within a system to 64. Windows 7 added the concept of **processor groups** to represent arbitrary numbers of CPUs, thus accommodating more CPU cores. The number of CPU cores within single systems has continued to increase not only because of more cores but also because of cores that support more than one logical thread of execution at a time.

All these additional CPUs created a great deal of contention for the locks used for scheduling CPUs and memory. Windows 7 broke these locks apart. For example, before Windows 7, a single lock was used by the Windows scheduler to synchronize access to the queues containing threads waiting for events. In Windows 7, each object has its own lock, allowing the queues to be accessed concurrently. Also, many execution paths in the scheduler were rewritten to be lock-free. This change resulted in good scalability performance for Windows even on systems with 256 hardware threads.

Other changes are due to the increasing importance of support for parallel computing. For years, the computer industry has been dominated by Moore's Law, leading to higher densities of transistors that manifest themselves as faster clock rates for each CPU. Moore's Law continues to hold true, but limits have been reached that prevent CPU clock rates from increasing further. Instead, transistors are being used to build more and more CPUs into each chip. New programming models for achieving parallel execution, such as Microsoft's Concurrency RunTime (ConcRT) and Intel's Threading Building Blocks (TBB), are being used to express parallelism in C++ programs. Where Moore's Law has governed computing for forty years, it now seems that Amdahl's Law, which governs parallel computing, will rule the future.

To support task-based parallelism, Windows 7 provides a new form of **user-mode scheduling (UMS)**. UMS allows programs to be decomposed into tasks, and the tasks are then scheduled on the available CPUs by a scheduler that operates in user mode rather than in the kernel.

The advent of multiple CPUs on the smallest computers is only part of the shift taking place to parallel computing. Graphics processing units (GPUs) accelerate the computational algorithms needed for graphics by using **SIMD** architectures to execute a single instruction for multiple data at the same time. This has given rise to the use of GPUs for general computing, not just graphics. Operating-system support for software like OpenCL and CUDA is allowing programs to take advantage of the GPUs. Windows supports use of GPUs through software in its DirectX graphics support. This software, called DirectCompute, allows programs to specify **computational kernels** using the same HLSL (high-level shader language) programming model used to program the SIMD hardware for **graphics shaders**. The computational kernels run very

quickly on the GPU and return their results to the main computation running on the CPU.

### B.2.5 Extensibility

**Extensibility** refers to the capacity of an operating system to keep up with advances in computing technology. To facilitate change over time, the developers implemented Windows using a layered architecture. The Windows executive runs in kernel mode and provides the basic system services and abstractions that support shared use of the system. On top of the executive, several server subsystems operate in user mode. Among them are **environmental subsystems** that emulate different operating systems. Thus, programs written for the Win32 APIs and POSIX all run on Windows in the appropriate environment. Because of the modular structure, additional environmental subsystems can be added without affecting the executive. In addition, Windows uses loadable drivers in the I/O system, so new file systems, new kinds of I/O devices, and new kinds of networking can be added while the system is running. Windows uses a client–server model like the Mach operating system and supports distributed processing by **remote procedure calls (RPCs)** as defined by the Open Software Foundation.

### B.2.6 Portability

An operating system is **portable** if it can be moved from one CPU architecture to another with few changes. Windows was designed to be portable. Like the UNIX operating system, Windows is written primarily in C and C++. The architecture-specific source code is relatively small, and there is very little use of assembly code. Porting Windows to a new architecture mostly affects the Windows kernel, since the user-mode code in Windows is almost exclusively written to be architecture independent. To port Windows, the kernel's architecture-specific code must be ported, and sometimes conditional compilation is needed in other parts of the kernel because of changes in major data structures, such as the page-table format. The entire Windows system must then be recompiled for the new CPU instruction set.

Operating systems are sensitive not only to CPU architecture but also to CPU support chips and hardware boot programs. The CPU and support chips are collectively known as a **chipset**. These chipsets and the associated boot code determine how interrupts are delivered, describe the physical characteristics of each system, and provide interfaces to deeper aspects of the CPU architecture, such as error recovery and power management. It would be burdensome to have to port Windows to each type of support chip as well as to each CPU architecture. Instead, Windows isolates most of the chipset-dependent code in a dynamic link library (DLL), called the **hardware-abstraction layer (HAL)**, that is loaded with the kernel. The Windows kernel depends on the HAL interfaces rather than on the underlying chipset details. This allows the single set of kernel and driver binaries for a particular CPU to be used with different chipsets simply by loading a different version of the HAL.

Over the years, Windows has been ported to a number of different CPU architectures: Intel IA-32-compatible 32-bit CPUs, AMD64-compatible and IA64 64-bit CPUs, the DEC Alpha, and the MIPS and PowerPC CPUs. Most of these CPU architectures failed in the market. When Windows 7 shipped, only the IA-

32 and AMD64 architectures were supported on client computers, along with AMD64 and IA64 on servers.

### B.2.7 International Support

Windows was designed for international and multinational use. It provides support for different locales via the [national-language-support \(NLS\)](#) API. The NLS API provides specialized routines to format dates, time, and money in accordance with national customs. String comparisons are specialized to account for varying character sets. UNICODE is Windows's native character code. Windows supports ANSI characters by converting them to UNICODE characters before manipulating them (8-bit to 16-bit conversion). System text strings are kept in resource files that can be replaced to localize the system for different languages. Multiple locales can be used concurrently, which is important to multilingual individuals and businesses.

### B.2.8 Energy Efficiency

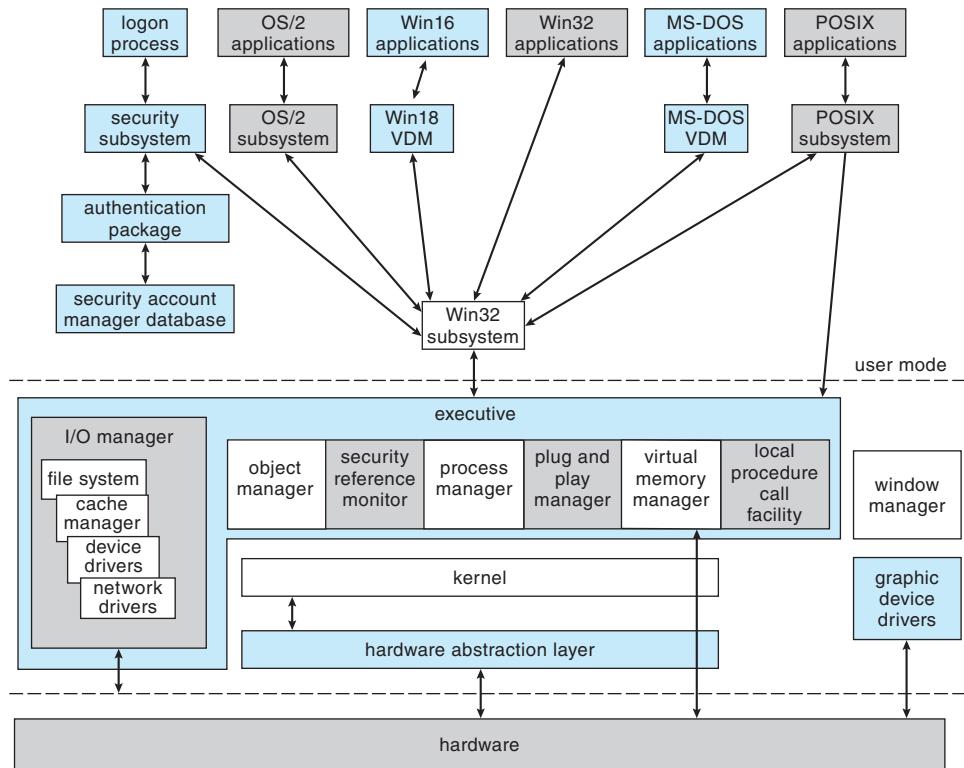
Increasing energy efficiency for computers causes batteries to last longer for laptops and netbooks, saves significant operating costs for power and cooling of data centers, and contributes to green initiatives aimed at lowering energy consumption by businesses and consumers. For some time, Windows has implemented several strategies for decreasing energy use. The CPUs are moved to lower power states—for example, by lowering clock frequency—whenever possible. In addition, when a computer is not being actively used, Windows may put the entire computer into a low-power state (sleep) or may even save all of memory to disk and shut the computer off (hibernation). When the user returns, the computer powers up and continues from its previous state, so the user does not need to reboot and restart applications.

Windows 7 added some new strategies for saving energy. The longer a CPU can stay unused, the more energy can be saved. Because computers are so much faster than human beings, a lot of energy can be saved just while humans are thinking. The problem is that too many programs are constantly polling to see what is happening in the system. A swarm of software timers are firing, keeping the CPU from staying idle long enough to save much energy. Windows 7 extends CPU idle time by skipping clock ticks, coalescing software timers into smaller numbers of events, and “parking” entire CPUs when systems are not heavily loaded.

### B.2.9 Dynamic Device Support

Early in the history of the PC industry, computer configurations were fairly static. Occasionally, new devices might be plugged into the serial, printer, or game ports on the back of a computer, but that was it. The next steps toward dynamic configuration of PCs were laptop docks and PCMCIA cards. A PC could suddenly be connected to or disconnected from a whole set of peripherals. In a contemporary PC, the situation has completely changed. PCs are designed to let users to plug and unplug a huge host of peripherals all the time; external disks, thumb drives, cameras, and the like are constantly coming and going.

Support for dynamic configuration of devices is continually evolving in Windows. The system can automatically recognize devices when they are



**Figure B.1** Windows block diagram.

plugged in and can find, install, and load the appropriate drivers—often without user intervention. When devices are unplugged, the drivers automatically unload, and system execution continues without disrupting other software.

## B.3 System Components

The architecture of Windows is a layered system of modules, as shown in Figure B.1. The main layers are the HAL, the kernel, and the executive, all of which run in kernel mode, and a collection of subsystems and services that run in user mode. The user-mode subsystems fall into two categories: the environmental subsystems, which emulate different operating systems, and the **protection subsystems**, which provide security functions. One of the chief advantages of this type of architecture is that interactions between modules are kept simple. The remainder of this section describes these layers and subsystems.

### B.3.1 Hardware-Abstraction Layer

The HAL is the layer of software that hides hardware chipset differences from upper levels of the operating system. The HAL exports a virtual hardware interface that is used by the kernel dispatcher, the executive, and the device drivers. Only a single version of each device driver is required for

each CPU architecture, no matter what support chips might be present. Device drivers map devices and access them directly, but the chipset-specific details of mapping memory, configuring I/O buses, setting up DMA, and coping with motherboard-specific facilities are all provided by the HAL interfaces.

### B.3.2 Kernel

The kernel layer of Windows has four main responsibilities: thread scheduling, low-level processor synchronization, interrupt and exception handling, and switching between user mode and kernel mode. The kernel is implemented in the C language, using assembly language only where absolutely necessary to interface with the lowest level of the hardware architecture.

The kernel is organized according to object-oriented design principles. An **object type** in Windows is a system-defined data type that has a set of attributes (data values) and a set of methods (for example, functions or operations). An **object** is an instance of an object type. The kernel performs its job by using a set of kernel objects whose attributes store the kernel data and whose methods perform the kernel activities.

#### B.3.2.1 Kernel Dispatcher

The kernel dispatcher provides the foundation for the executive and the sub-systems. Most of the dispatcher is never paged out of memory, and its execution is never preempted. Its main responsibilities are thread scheduling and context switching, implementation of synchronization primitives, timer management, software interrupts (asynchronous and deferred procedure calls), and exception dispatching.

#### B.3.2.2 Threads and Scheduling

Like many other modern operating systems, Windows uses processes and threads for executable code. Each process has one or more threads, and each thread has its own scheduling state, including actual priority, processor affinity, and CPU usage information.

There are six possible thread states: **ready**, **standby**, **running**, **waiting**, **transition**, and **terminated**. Ready indicates that the thread is waiting to run. The highest-priority ready thread is moved to the **standby** state, which means it is the next thread to run. In a multiprocessor system, each processor keeps one thread in a standby state. A thread is **running** when it is executing on a processor. It runs until it is preempted by a higher-priority thread, until it terminates, until its allotted execution time (quantum) ends, or until it waits on a dispatcher object, such as an event signaling I/O completion. A thread is in the **waiting** state when it is waiting for a dispatcher object to be signaled. A thread is in the **transition** state while it waits for resources necessary for execution; for example, it may be waiting for its kernel stack to be swapped in from disk. A thread enters the **terminated** state when it finishes execution.

The dispatcher uses a 32-level priority scheme to determine the order of thread execution. Priorities are divided into two classes: variable class and real-time class. The variable class contains threads having priorities from 1 to 15, and the real-time class contains threads with priorities ranging from 16 to 31.

The dispatcher uses a queue for each scheduling priority and traverses the set of queues from highest to lowest until it finds a thread that is ready to run. If a thread has a particular processor affinity but that processor is not available, the dispatcher skips past it and continues looking for a ready thread that is willing to run on the available processor. If no ready thread is found, the dispatcher executes a special thread called the *idle thread*. Priority class 0 is reserved for the idle thread.

When a thread's time quantum runs out, the clock interrupt queues a quantum-end **deferred procedure call (DPC)** to the processor. Queuing the DPC results in a software interrupt when the processor returns to normal interrupt priority. The software interrupt causes the dispatcher to reschedule the processor to execute the next available thread at the preempted thread's priority level.

The priority of the preempted thread may be modified before it is placed back on the dispatcher queues. If the preempted thread is in the variable-priority class, its priority is lowered. The priority is never lowered below the base priority. Lowering the thread's priority tends to limit the CPU consumption of compute-bound threads versus I/O-bound threads. When a variable-priority thread is released from a wait operation, the dispatcher boosts the priority. The amount of the boost depends on the device for which the thread was waiting. For example, a thread waiting for keyboard I/O would get a large priority increase, whereas a thread waiting for a disk operation would get a moderate one. This strategy tends to give good response times to interactive threads using a mouse and windows. It also enables I/O-bound threads to keep the I/O devices busy while permitting compute-bound threads to use spare CPU cycles in the background. In addition, the thread associated with the user's active GUI window receives a priority boost to enhance its response time.

Scheduling occurs when a thread enters the ready or wait state, when a thread terminates, or when an application changes a thread's priority or processor affinity. If a higher-priority thread becomes ready while a lower-priority thread is running, the lower-priority thread is preempted. This preemption gives the higher-priority thread preferential access to the CPU. Windows is not a hard real-time operating system, however, because it does not guarantee that a real-time thread will start to execute within a particular time limit; threads are blocked indefinitely while DPCs and **interrupt service routines (ISRs)** are running (as discussed further below).

Traditionally, operating-system schedulers used sampling to measure CPU utilization by threads. The system timer would fire periodically, and the timer interrupt handler would take note of what thread was currently scheduled and whether it was executing in user or kernel mode when the interrupt occurred. This sampling technique was necessary because either the CPU did not have a high-resolution clock or the clock was too expensive or unreliable to access frequently. Although efficient, sampling was inaccurate and led to anomalies such as incorporating interrupt servicing time as thread time and dispatching threads that had run for only a fraction of the quantum. Starting with Windows Vista, CPU time in Windows has been tracked using the hardware **timestamp counter (TSC)** included in recent processors. Using the TSC results in more accurate accounting of CPU usage, and the scheduler will not preempt threads before they have run for a full quantum.

### B.3.2.3 Implementation of Synchronization Primitives

Key operating-system data structures are managed as objects using common facilities for allocation, reference counting, and security. **Dispatcher objects** control dispatching and synchronization in the system. Examples of these objects include the following:

- The **event object** is used to record an event occurrence and to synchronize this occurrence with some action. Notification events signal all waiting threads, and synchronization events signal a single waiting thread.
- The **mutant** provides kernel-mode or user-mode mutual exclusion associated with the notion of ownership.
- The **mutex**, available only in kernel mode, provides deadlock-free mutual exclusion.
- The **semaphore object** acts as a counter or gate to control the number of threads that access a resource.
- The **thread object** is the entity that is scheduled by the kernel dispatcher. It is associated with a **process object**, which encapsulates a virtual address space. The thread object is signaled when the thread exits, and the process object, when the process exits.
- The **timer object** is used to keep track of time and to signal timeouts when operations take too long and need to be interrupted or when a periodic activity needs to be scheduled.

Many of the dispatcher objects are accessed from user mode via an open operation that returns a handle. The user-mode code polls or waits on handles to synchronize with other threads as well as with the operating system (see Section B.7.1).

### B.3.2.4 Software Interrupts: Asynchronous and Deferred Procedure Calls

The dispatcher implements two types of software interrupts: **asynchronous procedure calls (APCs)** and deferred procedure calls (DPCs, mentioned earlier). An asynchronous procedure call breaks into an executing thread and calls a procedure. APCs are used to begin execution of new threads, suspend or resume existing threads, terminate threads or processes, deliver notification that an asynchronous I/O has completed, and extract the contents of the CPU registers from a running thread. APCs are queued to specific threads and allow the system to execute both system and user code within a process's context. User-mode execution of an APC cannot occur at arbitrary times, but only when the thread is waiting in the kernel and marked *alertable*.

DPCs are used to postpone interrupt processing. After handling all urgent device-interrupt processing, the ISR schedules the remaining processing by queuing a DPC. The associated software interrupt will not occur until the CPU is next at a priority lower than the priority of all I/O device interrupts but higher than the priority at which threads run. Thus, DPCs do not block other device ISRs. In addition to deferring device-interrupt processing, the dispatcher uses

DPCs to process timer expirations and to preempt thread execution at the end of the scheduling quantum.

Execution of DPCs prevents threads from being scheduled on the current processor and also keeps APCs from signaling the completion of I/O. This is done so that completion of DPC routines does not take an extended amount of time. As an alternative, the dispatcher maintains a pool of worker threads. ISRs and DPCs may queue work items to the worker threads where they will be executed using normal thread scheduling. DPC routines are restricted so that they cannot take page faults (be paged out of memory), call system services, or take any other action that might result in an attempt to wait for a dispatcher object to be signaled. Unlike APCs, DPC routines make no assumptions about what process context the processor is executing.

### B.3.2.5 Exceptions and Interrupts

The kernel dispatcher also provides trap handling for exceptions and interrupts generated by hardware or software. Windows defines several architecture-independent exceptions, including:

- Memory-access violation
- Integer overflow
- Floating-point overflow or underflow
- Integer divide by zero
- Floating-point divide by zero
- Illegal instruction
- Data misalignment
- Privileged instruction
- Page-read error
- Access violation
- Paging file quota exceeded
- Debugger breakpoint
- Debugger single step

The trap handlers deal with simple exceptions. Elaborate exception handling is performed by the kernel's exception dispatcher. The **exception dispatcher** creates an exception record containing the reason for the exception and finds an exception handler to deal with it.

When an exception occurs in kernel mode, the exception dispatcher simply calls a routine to locate the exception handler. If no handler is found, a fatal system error occurs, and the user is left with the infamous “blue screen of death” that signifies system failure.

Exception handling is more complex for user-mode processes, because an environmental subsystem (such as the POSIX system) sets up a debugger port and an exception port for every process it creates. (For details on ports,

see Section B.3.3.4.) If a debugger port is registered, the exception handler sends the exception to the port. If the debugger port is not found or does not handle that exception, the dispatcher attempts to find an appropriate exception handler. If no handler is found, the debugger is called again to catch the error for debugging. If no debugger is running, a message is sent to the process's exception port to give the environmental subsystem a chance to translate the exception. For example, the POSIX environment translates Windows exception messages into POSIX signals before sending them to the thread that caused the exception. Finally, if nothing else works, the kernel simply terminates the process containing the thread that caused the exception.

When Windows fails to handle an exception, it may construct a description of the error that occurred and request permission from the user to send the information back to Microsoft for further analysis. In some cases, Microsoft's automated analysis may be able to recognize the error immediately and suggest a fix or workaround.

The interrupt dispatcher in the kernel handles interrupts by calling either an interrupt service routine (ISR) supplied by a device driver or a kernel trap-handler routine. The interrupt is represented by an **interrupt object** that contains all the information needed to handle the interrupt. Using an interrupt object makes it easy to associate interrupt-service routines with an interrupt without having to access the interrupt hardware directly.

Different processor architectures have different types and numbers of interrupts. For portability, the interrupt dispatcher maps the hardware interrupts into a standard set. The interrupts are prioritized and are serviced in priority order. There are 32 interrupt request levels (IRQLs) in Windows. Eight are reserved for use by the kernel; the remaining 24 represent hardware interrupts via the HAL (although most IA-32 systems use only 16). The Windows interrupts are defined in Figure B.2.

The kernel uses an **interrupt-dispatch table** to bind each interrupt level to a service routine. In a multiprocessor computer, Windows keeps a separate interrupt-dispatch table (IDT) for each processor, and each processor's IRQL can be set independently to mask out interrupts. All interrupts that occur at a level equal to or less than the IRQL of a processor are blocked until the IRQL is lowered

interrupt levels	types of interrupts
31	machine check or bus error
30	power fail
29	interprocessor notification (request another processor to act; e.g., dispatch a process or update the TLB)
28	clock (used to keep track of time)
27	profile
3–26	traditional PC IRQ hardware interrupts
2	dispatch and deferred procedure call (DPC) (kernel)
1	asynchronous procedure call (APC)
0	passive

**Figure B.2** Windows interrupt-request levels.

by a kernel-level thread or by an ISR returning from interrupt processing. Windows takes advantage of this property and uses software interrupts to deliver APCs and DPCs, to perform system functions such as synchronizing threads with I/O completion, to start thread execution, and to handle timers.

### B.3.2.6 Switching between User-Mode and Kernel-Mode Threads

What the programmer thinks of as a thread in traditional Windows is actually two threads: a **user-mode thread (UT)** and a **kernel-mode thread (KT)**. Each has its own stack, register values, and execution context. A UT requests a system service by executing an instruction that causes a trap to kernel mode. The kernel layer runs a trap handler that switches between the UT and the corresponding KT. When a KT has completed its kernel execution and is ready to switch back to the corresponding UT, the kernel layer is called to make the switch to the UT, which continues its execution in user mode.

Windows 7 modifies the behavior of the kernel layer to support user-mode scheduling of the UTs. User-mode schedulers in Windows 7 support cooperative scheduling. A UT can explicitly yield to another UT by calling the user-mode scheduler; it is not necessary to enter the kernel. User-mode scheduling is explained in more detail in Section B.7.3.7.

## B.3.3 Executive

The Windows executive provides a set of services that all environmental subsystems use. The services are grouped as follows: object manager, virtual memory manager, process manager, advanced local procedure call facility, I/O manager, cache manager, security reference monitor, plug-and-play and power managers, registry, and booting.

### B.3.3.1 Object Manager

For managing kernel-mode entities, Windows uses a generic set of interfaces that are manipulated by user-mode programs. Windows calls these entities *objects*, and the executive component that manipulates them is the **object manager**. Examples of objects are semaphores, mutexes, events, processes, and threads; all these are *dispatcher objects*. Threads can block in the kernel dispatcher waiting for any of these objects to be signaled. The process, thread, and virtual memory APIs use process and thread handles to identify the process or thread to be operated on. Other examples of objects include files, sections, ports, and various internal I/O objects. File objects are used to maintain the open state of files and devices. Sections are used to map files. Local-communication endpoints are implemented as port objects.

User-mode code accesses these objects using an opaque value called a **handle**, which is returned by many APIs. Each process has a **handle table** containing entries that track the objects used by the process. The **system process**, which contains the kernel, has its own handle table, which is protected from user code. The handle tables in Windows are represented by a tree structure, which can expand from holding 1,024 handles to holding over 16 million. Kernel-mode code can access an object by using either a handle or a **referenced pointer**.

A process gets a handle by creating an object, by opening an existing object, by receiving a duplicated handle from another process, or by inheriting a handle from the parent process. When a process exits, all its open handles are implicitly closed. Since the object manager is the only entity that generates object handles, it is the natural place to check security. The object manager checks whether a process has the right to access an object when the process tries to open the object. The object manager also enforces quotas, such as the maximum amount of memory a process may use, by charging a process for the memory occupied by all its referenced objects and refusing to allocate more memory when the accumulated charges exceed the process's quota.

The object manager keeps track of two counts for each object: the number of handles for the object and the number of referenced pointers. The handle count is the number of handles that refer to the object in the handle tables of all processes, including the system process that contains the kernel. The referenced pointer count is incremented whenever a new pointer is needed by the kernel and decremented when the kernel is done with the pointer. The purpose of these reference counts is to ensure that an object is not freed while it is still referenced by either a handle or an internal kernel pointer.

The object manager maintains the Windows internal name space. In contrast to UNIX, which roots the system name space in the file system, Windows uses an abstract name space and connects the file systems as devices. Whether a Windows object has a name is up to its creator. Processes and threads are created without names and referenced either by handle or through a separate numerical identifier. Synchronization events usually have names, so that they can be opened by unrelated processes. A name can be either permanent or temporary. A permanent name represents an entity, such as a disk drive, that remains even if no process is accessing it. A temporary name exists only while a process holds a handle to the object. The object manager supports directories and symbolic links in the name space. As an example, MS-DOS drive letters are implemented using symbolic links; \Global??\C: is a symbolic link to the device object \Device\HarddiskVolume2, representing a mounted file-system volume in the \Device directory.

Each object, as mentioned earlier, is an instance of an *object type*. The object type specifies how instances are to be allocated, how the data fields are to be defined, and how the standard set of virtual functions used for all objects are to be implemented. The standard functions implement operations such as mapping names to objects, closing and deleting, and applying security checks. Functions that are specific to a particular type of object are implemented by system services designed to operate on that particular object type, not by the methods specified in the object type.

The `parse()` function is the most interesting of the standard object functions. It allows the implementation of an object. The file systems, the registry configuration store, and GUI objects are the most notable users of `parse` functions to extend the Windows name space.

Returning to our Windows naming example, device objects used to represent file-system volumes provide a `parse` function. This allows a name like \Global??\C:\foo\bar.doc to be interpreted as the file \foo\bar.doc on the volume represented by the device object HarddiskVolume2. We can illustrate how naming, `parse` functions, objects, and handles work together by looking at the steps to open the file in Windows:

1. An application requests that a file named C:\foo\bar.doc be opened.
2. The object manager finds the device object HarddiskVolume2, looks up the parse procedure IopParseDevice from the object's type, and invokes it with the file's name relative to the root of the file system.
3. IopParseDevice() allocates a file object and passes it to the file system, which fills in the details of how to access C:\foo\bar.doc on the volume.
4. When the file system returns, IopParseDevice() allocates an entry for the file object in the handle table for the current process and returns the handle to the application.

If the file cannot successfully be opened, IopParseDevice() deletes the file object it allocated and returns an error indication to the application.

### B.3.3.2 Virtual Memory Manager

The executive component that manages the virtual address space, physical memory allocation, and paging is the **virtual memory (VM) manager**. The design of the VM manager assumes that the underlying hardware supports virtual-to-physical mapping, a paging mechanism, and transparent cache coherence on multiprocessor systems, as well as allowing multiple page-table entries to map to the same physical page frame. The VM manager in Windows uses a page-based management scheme with page sizes of 4 KB and 2 MB on AMD64 and IA-32-compatible processors and 8 KB on the IA64. Pages of data allocated to a process that are not in physical memory are either stored in the **paging file** on disk or mapped directly to a regular file on a local or remote file system. A page can also be marked zero-fill-on-demand, which initializes the page with zeros before it is allocated, thus erasing the previous contents.

On IA-32 processors, each process has a 4-GB virtual address space. The upper 2 GB are mostly identical for all processes and are used by Windows in kernel mode to access the operating-system code and data structures. For the AMD64 architecture, Windows provides a 8-TB virtual address space for user mode out of the 16 EB supported by existing hardware for each process.

Key areas of the kernel-mode region that are not identical for all processes are the self-map, hyperspace, and session space. The hardware references a process's page table using physical page-frame numbers, and the **page table self-map** makes the contents of the process's page table accessible using virtual addresses. **Hyperspace** maps the current process's working-set information into the kernel-mode address space. **Session space** is used to share an instance of the Win32 and other session-specific drivers among all the processes in the same terminal-server (TS) session. Different TS sessions share different instances of these drivers, yet they are mapped at the same virtual addresses. The lower, user-mode region of virtual address space is specific to each process and accessible by both user- and kernel-mode threads.

The Windows VM manager uses a two-step process to allocate virtual memory. The first step *reserves* one or more pages of virtual addresses in the process's virtual address space. The second step *commits* the allocation by assigning virtual memory space (physical memory or space in the paging files). Windows limits the amount of virtual memory space a process consumes by enforcing a quota on committed memory. A process decommits memory that it

is no longer using to free up virtual memory space for use by other processes. The APIs used to reserve virtual addresses and commit virtual memory take a handle on a process object as a parameter. This allows one process to control the virtual memory of another. Environmental subsystems manage the memory of their client processes in this way.

Windows implements shared memory by defining a **section object**. After getting a handle to a section object, a process maps the memory of the section to a range of addresses, called a **view**. A process can establish a view of the entire section or only the portion it needs. Windows allows sections to be mapped not just into the current process but into any process for which the caller has a handle.

Sections can be used in many ways. A section can be backed by disk space either in the system-paging file or in a regular file (a **memory-mapped file**). A section can be *based*, meaning that it appears at the same virtual address for all processes attempting to access it. Sections can also represent physical memory, allowing a 32-bit process to access more physical memory than can fit in its virtual address space. Finally, the memory protection of pages in the section can be set to read-only, read-write, read-write-execute, execute-only, no access, or copy-on-write.

Let's look more closely at the last two of these protection settings:

- A *no-access page* raises an exception if accessed. The exception can be used, for example, to check whether a faulty program iterates beyond the end of an array or simply to detect that the program attempted to access virtual addresses that are not committed to memory. User- and kernel-mode stacks use no-access pages as **guard pages** to detect stack overflows. Another use is to look for heap buffer overruns. Both the user-mode memory allocator and the special kernel allocator used by the device verifier can be configured to map each allocation onto the end of a page, followed by a no-access page to detect programming errors that access beyond the end of an allocation.
- The *copy-on-write mechanism* enables the VM manager to use physical memory more efficiently. When two processes want independent copies of data from the same section object, the VM manager places a single shared copy into virtual memory and activates the copy-on-write property for that region of memory. If one of the processes tries to modify data in a copy-on-write page, the VM manager makes a private copy of the page for the process.

The virtual address translation in Windows uses a multilevel page table. For IA-32 and AMD64 processors, each process has a **page directory** that contains 512 **page-directory entries (PDEs)** 8 bytes in size. Each PDE points to a **PTE table** that contains 512 **page-table entries (PTEs)** 8 bytes in size. Each PTE points to a 4-KB **page frame** in physical memory. For a variety of reasons, the hardware requires that the page directories or PTE tables at each level of a multilevel page table occupy a single page. Thus, the number of PDEs or PTEs that fit in a page determine how many virtual addresses are translated by that page. See Figure B.3 for a diagram of this structure.

The structure described so far can be used to represent only 1 GB of virtual address translation. For IA-32, a second page-directory level is needed, con-

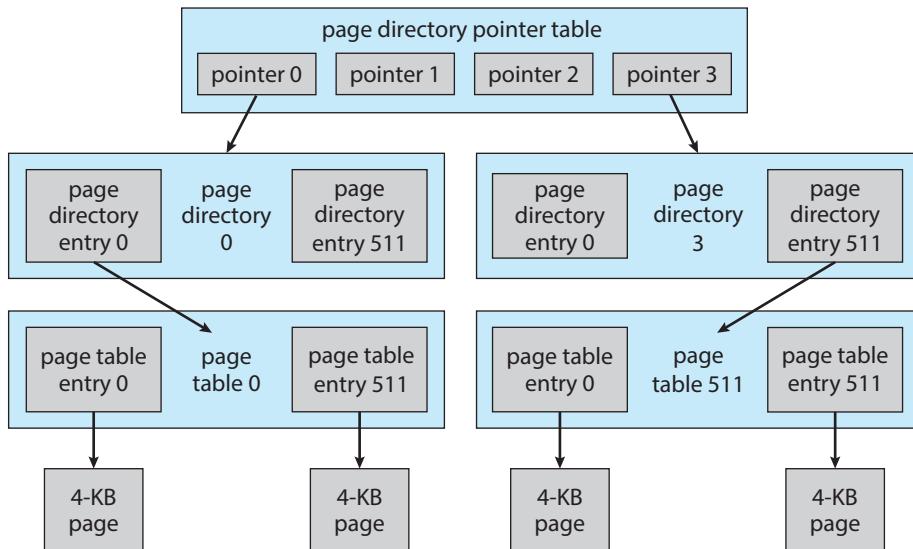


Figure B.3 Page-table layout.

taining only four entries, as shown in the diagram. On 64-bit processors, more levels are needed. For AMD64, Windows uses a total of four full levels. The total size of all page-table pages needed to fully represent even a 32-bit virtual address space for a process is 8 MB. The VM manager allocates pages of PDEs and PTEs as needed and moves page-table pages to disk when not in use. The page-table pages are faulted back into memory when referenced.

We next consider how virtual addresses are translated into physical addresses on IA-32-compatible processors. A 2-bit value can represent the values 0, 1, 2, 3. A 9-bit value can represent values from 0 to 511; a 12-bit value, values from 0 to 4,095. Thus, a 12-bit value can select any byte within a 4-KB page of memory. A 9-bit value can represent any of the 512 PDEs or PTEs in a page directory or PTE-table page. As shown in Figure B.4, translating a virtual address pointer to a byte address in physical memory involves breaking the 32-bit pointer into four values, starting from the most significant bits:

- Two bits are used to index into the four PDEs at the top level of the page table. The selected PDE will contain the physical page number for each of the four page-directory pages that map 1 GB of the address space.

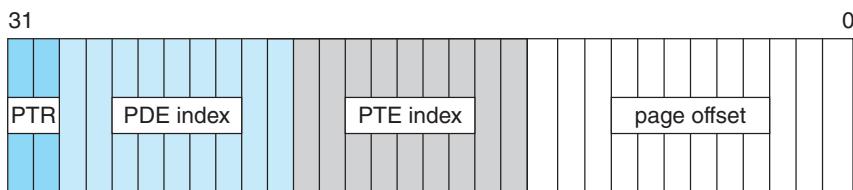


Figure B.4 Virtual-to-physical address translation on IA-32.

- Nine bits are used to select another PDE, this time from a second-level page directory. This PDE will contain the physical page numbers of up to 512 PTE-table pages.
- Nine bits are used to select one of 512 PTEs from the selected PTE-table page. The selected PTE will contain the physical page number for the byte we are accessing.
- Twelve bits are used as the byte offset into the page. The physical address of the byte we are accessing is constructed by appending the lowest 12 bits of the virtual address to the end of the physical page number we found in the selected PTE.

The number of bits in a physical address may be different from the number of bits in a virtual address. In the original IA-32 architecture, the PTE and PDE were 32-bit structures that had room for only 20 bits of physical page number, so the physical address size and the virtual address size were the same. Such systems could address only 4 GB of physical memory. Later, the IA-32 was extended to the larger 64-bit PTE size used today, and the hardware supported 24-bit physical addresses. These systems could support 64 GB and were used on server systems. Today, all Windows servers are based on either the AMD64 or the IA64 and support very, very large physical addresses—more than we can possibly use. (Of course, once upon a time 4 GB seemed optimistically large for physical memory.)

To improve performance, the VM manager maps the page-directory and PTE-table pages into the same contiguous region of virtual addresses in every process. This self-map allows the VM manager to use the same pointer to access the current PDE or PTE corresponding to a particular virtual address no matter what process is running. The self-map for the IA-32 takes a contiguous 8-MB region of kernel virtual address space; the AMD64 self-map occupies 512 GB. Although the self-map occupies significant address space, it does not require any additional virtual memory pages. It also allows the page table's pages to be automatically paged in and out of physical memory.

In the creation of a self-map, one of the PDEs in the top-level page directory refers to the page-directory page itself, forming a “loop” in the page-table translations. The virtual pages are accessed if the loop is not taken, the PTE-table pages are accessed if the loop is taken once, the lowest-level page-directory pages are accessed if the loop is taken twice, and so forth.

The additional levels of page directories used for 64-bit virtual memory are translated in the same way except that the virtual address pointer is broken up into even more values. For the AMD64, Windows uses four full levels, each of which maps 512 pages, or  $9+9+9+9+12 = 48$  bits of virtual address.

To avoid the overhead of translating every virtual address by looking up the PDE and PTE, processors use **translation look-aside buffer (TLB)** hardware, which contains an associative memory cache for mapping virtual pages to PTEs. The TLB is part of the **memory-management unit (MMU)** within each processor. The MMU needs to “walk” (navigate the data structures of) the page table in memory only when a needed translation is missing from the TLB.

The PDEs and PTEs contain more than just physical page numbers. They also have bits reserved for operating-system use and bits that control how the hardware uses memory, such as whether hardware caching should be used for

each page. In addition, the entries specify what kinds of access are allowed for both user and kernel modes.

A PDE can also be marked to say that it should function as a PTE rather than a PDE. On a IA-32, the first 11 bits of the virtual address pointer select a PDE in the first two levels of translation. If the selected PDE is marked to act as a PTE, then the remaining 21 bits of the pointer are used as the offset of the byte. This results in a 2-MB size for the page. Mixing and matching 4-KB and 2-MB page sizes within the page table is easy for the operating system and can significantly improve the performance of some programs by reducing how often the MMU needs to reload entries in the TLB, since one PDE mapping 2 MB replaces 512 PTEs each mapping 4 KB.

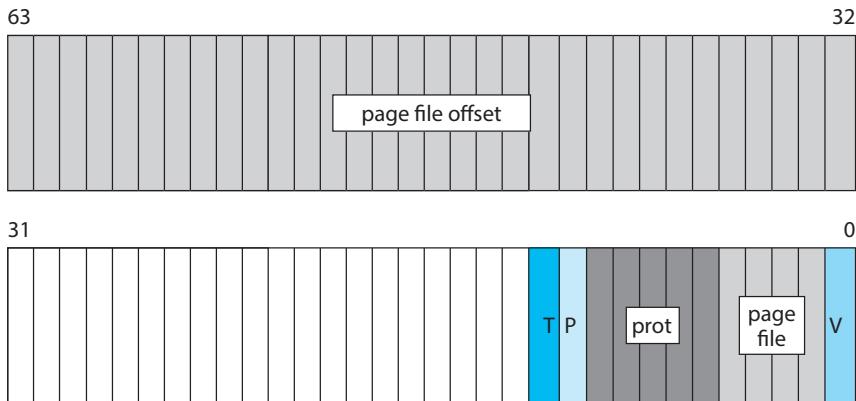
Managing physical memory so that 2-MB pages are available when needed is difficult, however, as they may continually be broken up into 4-KB pages, causing external fragmentation of memory. Also, the large pages can result in very significant internal fragmentation. Because of these problems, it is typically only Windows itself, along with large server applications, that use large pages to improve the performance of the TLB. They are better suited to do so because operating-system and server applications start running when the system boots, before memory has become fragmented.

Windows manages physical memory by associating each physical page with one of seven states: free, zeroed, modified, standby, bad, transition, or valid.

- A *free* page is a page that has no particular content.
- A *zeroed* page is a free page that has been zeroed out and is ready for immediate use to satisfy zero-on-demand faults.
- A *modified* page has been written by a process and must be sent to the disk before it is allocated for another process.
- A *standby* page is a copy of information already stored on disk. Standby pages may be pages that were not modified, modified pages that have already been written to the disk, or pages that were prefetched because they are expected to be used soon.
- A *bad* page is unusable because a hardware error has been detected.
- A *transition* page is on its way in from disk to a page frame allocated in physical memory.
- A *valid* page is part of the working set of one or more processes and is contained within these processes' page tables.

While valid pages are contained in processes' page tables, pages in other states are kept in separate lists according to state type. The lists are constructed by linking the corresponding entries in the **page frame number (PFN)** database, which includes an entry for each physical memory page. The PFN entries also include information such as reference counts, locks, and NUMA information. Note that the PFN database represents pages of physical memory, whereas the PTEs represent pages of virtual memory.

When the valid bit in a PTE is zero, hardware ignores all the other bits, and the VM manager can define them for its own use. Invalid pages can have a number of states represented by bits in the PTE. Page-file pages that have never



**Figure B.5** Page-file page-table entry. The valid bit is zero.

been faulted in are marked zero-on-demand. Pages mapped through section objects encode a pointer to the appropriate section object. PTEs for pages that have been written to the page file contain enough information to locate the page on disk, and so forth. The structure of the page-file PTE is shown in Figure B.5. The T, P, and V bits are all zero for this type of PTE. The PTE includes 5 bits for page protection, 32 bits for page-file offset, and 4 bits to select the paging file. There are also 20 bits reserved for additional bookkeeping.

Windows uses a per-working-set, least-recently-used (LRU) replacement policy to take pages from processes as appropriate. When a process is started, it is assigned a default minimum working-set size. The working set of each process is allowed to grow until the amount of remaining physical memory starts to run low, at which point the VM manager starts to track the age of the pages in each working set. Eventually, when the available memory runs critically low, the VM manager trims the working set to remove older pages.

How old a page is depends not on how long it has been in memory but on when it was last referenced. This is determined by periodically making a pass through the working set of each process and incrementing the age for pages that have not been marked in the PTE as referenced since the last pass. When it becomes necessary to trim the working sets, the VM manager uses heuristics to decide how much to trim from each process and then removes the oldest pages first.

A process can have its working set trimmed even when plenty of memory is available, if it was given a *hard limit* on how much physical memory it could use. In Windows 7, the VM manager will also trim processes that are growing rapidly, even if memory is plentiful. This policy change significantly improves the responsiveness of the system for other processes.

Windows tracks working sets not only for user-mode processes but also for the system process, which includes all the pageable data structures and code that run in kernel mode. Windows 7 created additional working sets for the system process and associated them with particular categories of kernel memory; the file cache, kernel heap, and kernel code now have their own working sets. The distinct working sets allow the VM manager to use different policies to trim the different categories of kernel memory.

The VM manager does not fault in only the page immediately needed. Research shows that the memory referencing of a thread tends to have a **locality** property. That is, when a page is used, it is likely that adjacent pages will be referenced in the near future. (Think of iterating over an array or fetching sequential instructions that form the executable code for a thread.) Because of locality, when the VM manager faults in a page, it also faults in a few adjacent pages. This prefetching tends to reduce the total number of page faults and allows reads to be clustered to improve I/O performance.

In addition to managing committed memory, the VM manager manages each process's reserved memory, or virtual address space. Each process has an associated tree that describes the ranges of virtual addresses in use and what the uses are. This allows the VM manager to fault in page-table pages as needed. If the PTE for a faulting address is uninitialized, the VM manager searches for the address in the process's tree of **virtual address descriptors (VADs)** and uses this information to fill in the PTE and retrieve the page. In some cases, a PTE-table page itself may not exist; such a page must be transparently allocated and initialized by the VM manager. In other cases, the page may be shared as part of a section object, and the VAD will contain a pointer to that section object. The section object contains information on how to find the shared virtual page so that the PTE can be initialized to point at it directly.

### B.3.3.3 Process Manager

The Windows process manager provides services for creating, deleting, and using processes, threads, and jobs. It has no knowledge about parent-child relationships or process hierarchies; those refinements are left to the particular environmental subsystem that owns the process. The process manager is also not involved in the scheduling of processes, other than setting the priorities and affinities in processes and threads when they are created. Thread scheduling takes place in the kernel dispatcher.

Each process contains one or more threads. Processes themselves can be collected into larger units called **job objects**. The use of job objects allows limits to be placed on CPU usage, working-set size, and processor affinities that control multiple processes at once. Job objects are used to manage large data-center machines.

An example of process creation in the Win32 environment is as follows:

1. A Win32 application calls `CreateProcess()`.
2. A message is sent to the Win32 subsystem to notify it that the process is being created.
3. `CreateProcess()` in the original process then calls an API in the process manager of the NT executive to actually create the process.
4. The process manager calls the object manager to create a process object and returns the object handle to Win32.
5. Win32 calls the process manager again to create a thread for the process and returns handles to the new process and thread.

The Windows APIs for manipulating virtual memory and threads and for duplicating handles take a process handle, so subsystems can perform

operations on behalf of a new process without having to execute directly in the new process's context. Once a new process is created, the initial thread is created, and an asynchronous procedure call is delivered to the thread to prompt the start of execution at the user-mode image loader. The loader is in `ntdll.dll`, which is a link library automatically mapped into every newly created process. Windows also supports a UNIX `fork()` style of process creation in order to support the POSIX environmental subsystem. Although the Win32 environment calls the process manager directly from the client process, POSIX uses the cross-process nature of the Windows APIs to create the new process from within the subsystem process.

The process manager relies on the asynchronous procedure calls (APCs) implemented by the kernel layer. APCs are used to initiate thread execution, suspend and resume threads, access thread registers, terminate threads and processes, and support debuggers.

The debugger support in the process manager includes the APIs to suspend and resume threads and to create threads that begin in suspended mode. There are also process-manager APIs that get and set a thread's register context and access another process's virtual memory. Threads can be created in the current process; they can also be injected into another process. The debugger makes use of thread injection to execute code within a process being debugged.

While running in the executive, a thread can temporarily attach to a different process. **Thread attach** is used by kernel worker threads that need to execute in the context of the process originating a work request. For example, the VM manager might use thread attach when it needs access to a process's working set or page tables, and the I/O manager might use it in updating the status variable in a process for asynchronous I/O operations.

The process manager also supports **impersonation**. Each thread has an associated **security token**. When the login process authenticates a user, the security token is attached to the user's process and inherited by its child processes. The token contains the **security identity (SID)** of the user, the SIDs of the groups the user belongs to, the privileges the user has, and the integrity level of the process. By default, all threads within a process share a common token, representing the user and the application that started the process. However, a thread running in a process with a security token belonging to one user can set a thread-specific token belonging to another user to impersonate that user.

The impersonation facility is fundamental to the client–server RPC model, where services must act on behalf of a variety of clients with different security IDs. The right to impersonate a user is most often delivered as part of an RPC connection from a client process to a server process. Impersonation allows the server to access system services as if it were the client in order to access or create objects and files on behalf of the client. The server process must be trustworthy and must be carefully written to be robust against attacks. Otherwise, one client could take over a server process and then impersonate any user who made a subsequent client request.

#### B.3.3.4 Facilities for Client–Server Computing

The implementation of Windows uses a client–server model throughout. The environmental subsystems are servers that implement particular operating-system personalities. Many other services, such as user authentication, net-

work facilities, printer spooling, web services, network file systems, and plug-and-play, are also implemented using this model. To reduce the memory footprint, multiple services are often collected into a few processes running the `svchost.exe` program. Each service is loaded as a dynamic-link library (DLL), which implements the service by relying on the user-mode thread-pool facilities to share threads and wait for messages (see Section B.3.3.3).

The normal implementation paradigm for client–server computing is to use RPCs to communicate requests. The Win32 API supports a standard RPC protocol, as described in Section B.6.2.7. RPC uses multiple transports (for example, named pipes and TCP/IP) and can be used to implement RPCs between systems. When an RPC always occurs between a client and server on the local system, the advanced local procedure call facility (ALPC) can be used as the transport. At the lowest level of the system, in the implementation of the environmental systems, and for services that must be available in the early stages of booting, RPC is not available. Instead, native Windows services use ALPC directly.

ALPC is a message-passing mechanism. The server process publishes a globally visible connection-port object. When a client wants services from a subsystem or service, it opens a handle to the server’s connection-port object and sends a connection request to the port. The server creates a channel and returns a handle to the client. The channel consists of a pair of private communication ports: one for client-to-server messages and the other for server-to-client messages. Communication channels support a callback mechanism, so the client and server can accept requests when they would normally be expecting a reply.

When an ALPC channel is created, one of three message-passing techniques is chosen.

1. The first technique is suitable for small to medium messages (up to 63 KB). In this case, the port’s message queue is used as intermediate storage, and the messages are copied from one process to the other.
2. The second technique is for larger messages. In this case, a shared-memory section object is created for the channel. Messages sent through the port’s message queue contain a pointer and size information referring to the section object. This avoids the need to copy large messages. The sender places data into the shared section, and the receiver views them directly.
3. The third technique uses APIs that read and write directly into a process’s address space. ALPC provides functions and synchronization so that a server can access the data in a client. This technique is normally used by RPC to achieve higher performance for specific scenarios.

The Win32 window manager uses its own form of message passing, which is independent of the executive ALPC facilities. When a client asks for a connection that uses window-manager messaging, the server sets up three objects: (1) a dedicated server thread to handle requests, (2) a 64-KB shared section object, and (3) an event-pair object. An *event-pair object* is a synchronization object used by the Win32 subsystem to provide notification when the client thread has copied a message to the Win32 server, or vice versa. The section object is used to pass the messages, and the event-pair object provides synchronization.

Window-manager messaging has several advantages:

- The section object eliminates message copying, since it represents a region of shared memory.
- The event-pair object eliminates the overhead of using the port object to pass messages containing pointers and lengths.
- The dedicated server thread eliminates the overhead of determining which client thread is calling the server, since there is one server thread per client thread.
- The kernel gives scheduling preference to these dedicated server threads to improve performance.

### B.3.3.5 I/O Manager

The **I/O manager** is responsible for managing file systems, device drivers, and network drivers. It keeps track of which device drivers, filter drivers, and file systems are loaded, and it also manages buffers for I/O requests. It works with the VM manager to provide memory-mapped file I/O and controls the Windows cache manager, which handles caching for the entire I/O system. The I/O manager is fundamentally asynchronous, providing synchronous I/O by explicitly waiting for an I/O operation to complete. The I/O manager provides several models of asynchronous I/O completion, including setting of events, updating of a status variable in the calling process, delivery of APCs to initiating threads, and use of I/O completion ports, which allow a single thread to process I/O completions from many other threads.

Device drivers are arranged in a list for each device (called a driver or I/O stack). A driver is represented in the system as a **driver object**. Because a single driver can operate on multiple devices, the drivers are represented in the I/O stack by a **device object**, which contains a link to the driver object. The I/O manager converts the requests it receives into a standard form called an **I/O request packet (IRP)**. It then forwards the IRP to the first driver in the targeted I/O stack for processing. After a driver processes the IRP, it calls the I/O manager either to forward the IRP to the next driver in the stack or, if all processing is finished, to complete the operation represented by the IRP.

The I/O request may be completed in a context different from the one in which it was made. For example, if a driver is performing its part of an I/O operation and is forced to block for an extended time, it may queue the IRP to a worker thread to continue processing in the system context. In the original thread, the driver returns a status indicating that the I/O request is pending so that the thread can continue executing in parallel with the I/O operation. An IRP may also be processed in interrupt-service routines and completed in an arbitrary process context. Because some final processing may need to take place in the context that initiated the I/O, the I/O manager uses an APC to do final I/O-completion processing in the process context of the originating thread.

The I/O stack model is very flexible. As a driver stack is built, various drivers have the opportunity to insert themselves into the stack as **filter drivers**. Filter drivers can examine and potentially modify each I/O operation. Mount management, partition management, and disk striping and mirroring are all examples of functionality implemented using filter drivers that execute

beneath the file system in the stack. File-system filter drivers execute above the file system and have been used to implement functionalities such as hierarchical storage management, single instancing of files for remote boot, and dynamic format conversion. Third parties also use file-system filter drivers to implement virus detection.

Device drivers for Windows are written to the Windows Driver Model (WDM) specification. This model lays out all the requirements for device drivers, including how to layer filter drivers, share common code for handling power and plug-and-play requests, build correct cancellation logic, and so forth.

Because of the richness of the WDM, writing a full WDM device driver for each new hardware device can involve a great deal of work. Fortunately, the port/miniport model makes it unnecessary to do this. Within a class of similar devices, such as audio drivers, SATA devices, or Ethernet controllers, each instance of a device shares a common driver for that class, called a **port driver**. The port driver implements the standard operations for the class and then calls device-specific routines in the device's **miniport driver** to implement device-specific functionality. The TCP/IP network stack is implemented in this way, with the `ndis.sys` class driver implementing much of the network driver functionality and calling out to the network miniport drivers for specific hardware.

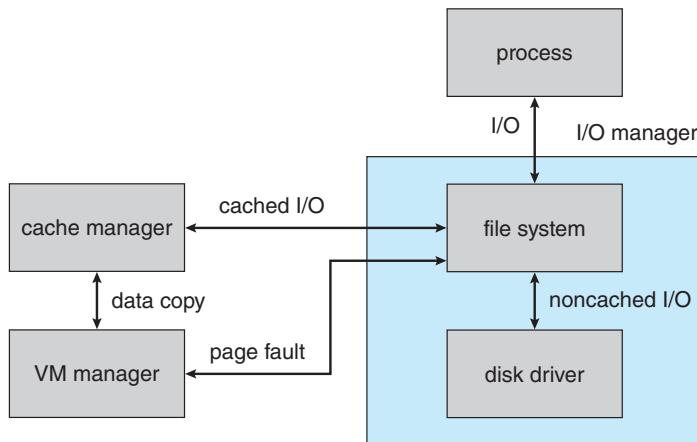
Recent versions of Windows, including Windows 7, provide additional simplifications for writing device drivers for hardware devices. Kernel-mode drivers can now be written using the **Kernel-Mode Driver Framework (KMDF)**, which provides a simplified programming model for drivers on top of WDM. Another option is the **User-Mode Driver Framework (UMDF)**. Many drivers do not need to operate in kernel mode, and it is easier to develop and deploy drivers in user mode. It also makes the system more reliable, because a failure in a user-mode driver does not cause a kernel-mode crash.

### B.3.3.6 Cache Manager

In many operating systems, caching is done by the file system. Instead, Windows provides a centralized caching facility. The **cache manager** works closely with the VM manager to provide cache services for all components under the control of the I/O manager. Caching in Windows is based on files rather than raw blocks. The size of the cache changes dynamically according to how much free memory is available in the system. The cache manager maintains a private working set rather than sharing the system process's working set. The cache manager memory-maps files into kernel memory and then uses special interfaces to the VM manager to fault pages into or trim them from this private working set.

The cache is divided into blocks of 256 KB. Each cache block can hold a view (that is, a memory-mapped region) of a file. Each cache block is described by a **virtual address control block (VACB)** that stores the virtual address and file offset for the view, as well as the number of processes using the view. The VACBs reside in a single array maintained by the cache manager.

When the I/O manager receives a file's user-level read request, the I/O manager sends an IRP to the I/O stack for the volume on which the file resides. For files that are marked as cacheable, the file system calls the cache manager



**Figure B.6** File I/O.

to look up the requested data in its cached file views. The cache manager calculates which entry of that file's VACB index array corresponds to the byte offset of the request. The entry either points to the view in the cache or is invalid. If it is invalid, the cache manager allocates a cache block (and the corresponding entry in the VACB array) and maps the view into the cache block. The cache manager then attempts to copy data from the mapped file to the caller's buffer. If the copy succeeds, the operation is completed.

If the copy fails, it does so because of a page fault, which causes the VM manager to send a noncached read request to the I/O manager. The I/O manager sends another request down the driver stack, this time requesting a paging operation, which bypasses the cache manager and reads the data from the file directly into the page allocated for the cache manager. Upon completion, the VACB is set to point at the page. The data, now in the cache, are copied to the caller's buffer, and the original I/O request is completed. Figure B.6 shows an overview of these operations.

A kernel-level read operation is similar, except that the data can be accessed directly from the cache rather than being copied to a buffer in user space. To use file-system metadata (data structures that describe the file system), the kernel uses the cache manager's mapping interface to read the metadata. To modify the metadata, the file system uses the cache manager's pinning interface. **Pinning** a page locks the page into a physical-memory page frame so that the VM manager cannot move the page or page it out. After updating the metadata, the file system asks the cache manager to unpin the page. A modified page is marked dirty, and so the VM manager flushes the page to disk.

To improve performance, the cache manager keeps a small history of read requests and from this history attempts to predict future requests. If the cache manager finds a pattern in the previous three requests, such as sequential access forward or backward, it prefetches data into the cache before the next request is submitted by the application. In this way, the application may find its data already cached and not need to wait for disk I/O.

The cache manager is also responsible for telling the VM manager to flush the contents of the cache. The cache manager's default behavior is write-back

caching: it accumulates writes for 4 to 5 seconds and then wakes up the cache-writer thread. When write-through caching is needed, a process can set a flag when opening the file, or the process can call an explicit cache-flush function.

A fast-writing process could potentially fill all the free cache pages before the cache-writer thread had a chance to wake up and flush the pages to disk. The cache writer prevents a process from flooding the system in the following way. When the amount of free cache memory becomes low, the cache manager temporarily blocks processes attempting to write data and wakes the cache-writer thread to flush pages to disk. If the fast-writing process is actually a network redirector for a network file system, blocking it for too long could cause network transfers to time out and be retransmitted. This retransmission would waste network bandwidth. To prevent such waste, network redirectors can instruct the cache manager to limit the backlog of writes in the cache.

Because a network file system needs to move data between a disk and the network interface, the cache manager also provides a DMA interface to move the data directly. Moving data directly avoids the need to copy data through an intermediate buffer.

#### B.3.3.7 Security Reference Monitor

Centralizing management of system entities in the object manager enables Windows to use a uniform mechanism to perform run-time access validation and audit checks for every user-accessible entity in the system. Whenever a process opens a handle to an object, the **security reference monitor (SRM)** checks the process's security token and the object's access-control list to see whether the process has the necessary access rights.

The SRM is also responsible for manipulating the privileges in security tokens. Special privileges are required for users to perform backup or restore operations on file systems, debug processes, and so forth. Tokens can also be marked as being restricted in their privileges so that they cannot access objects that are available to most users. Restricted tokens are used primarily to limit the damage that can be done by execution of untrusted code.

The integrity level of the code executing in a process is also represented by a token. Integrity levels are a type of capability mechanism, as mentioned earlier. A process cannot modify an object with an integrity level higher than that of the code executing in the process, whatever other permissions have been granted. Integrity levels were introduced to make it harder for code that successfully attacks outward-facing software, like Internet Explorer, to take over a system.

Another responsibility of the SRM is logging security audit events. The Department of Defense's **Common Criteria** (the 2005 successor to the Orange Book) requires that a secure system have the ability to detect and log all attempts to access system resources so that it can more easily trace attempts at unauthorized access. Because the SRM is responsible for making access checks, it generates most of the audit records in the security-event log.

#### B.3.3.8 Plug-and-Play Manager

The operating system uses the **plug-and-play (PnP)** manager to recognize and adapt to changes in the hardware configuration. PnP devices use standard protocols to identify themselves to the system. The PnP manager automatically recognizes installed devices and detects changes in devices as the system

operates. The manager also keeps track of hardware resources used by a device, as well as potential resources that could be used, and takes care of loading the appropriate drivers. This management of hardware resources—primarily interrupts and I/O memory ranges—has the goal of determining a hardware configuration in which all devices are able to operate successfully.

The PnP manager handles dynamic reconfiguration as follows. First, it gets a list of devices from each bus driver (for example, PCI or USB). It loads the installed driver (after finding one, if necessary) and sends an `add-device` request to the appropriate driver for each device. The PnP manager then figures out the optimal resource assignments and sends a `start-device` request to each driver specifying the resource assignments for the device. If a device needs to be reconfigured, the PnP manager sends a `query-stop` request, which asks the driver whether the device can be temporarily disabled. If the driver can disable the device, then all pending operations are completed, and new operations are prevented from starting. Finally, the PnP manager sends a `stop` request and can then reconfigure the device with a new `start-device` request.

The PnP manager also supports other requests. For example, `query-remove`, which operates similarly to `query-stop`, is employed when a user is getting ready to eject a removable device, such as a USB storage device. The `surprise-remove` request is used when a device fails or, more likely, when a user removes a device without telling the system to stop it first. Finally, the `remove` request tells the driver to stop using a device permanently.

Many programs in the system are interested in the addition or removal of devices, so the PnP manager supports notifications. Such a notification, for example, gives GUI file menus the information they need to update their list of disk volumes when a new storage device is attached or removed. Installing devices often results in adding new services to the `svchost.exe` processes in the system. These services frequently set themselves up to run whenever the system boots and continue to run even if the original device is never plugged into the system. Windows 7 introduced a **service-trigger** mechanism in the **service control manager (SCM)**, which manages the system services. With this mechanism, services can register themselves to start only when SCM receives a notification from the PnP manager that the device of interest has been added to the system.

### B.3.3.9 Power Manager

Windows works with the hardware to implement sophisticated strategies for energy efficiency, as described in Section B.2.8. The policies that drive these strategies are implemented by the **power manager**. The power manager detects current system conditions, such as the load on CPUs or I/O devices, and improves energy efficiency by reducing the performance and responsiveness of the system when need is low. The power manager can also put the entire system into a very efficient *sleep* mode and can even write all the contents of memory to disk and turn off the power to allow the system to go into *hibernation*.

The primary advantage of sleep is that the system can enter fairly quickly, perhaps just a few seconds after the lid closes on a laptop. The return from sleep is also fairly quick. The power is turned down low on the CPUs and I/O devices, but the memory continues to be powered enough that its contents are not lost.

Hibernation takes considerably longer because the entire contents of memory must be transferred to disk before the system is turned off. However, the fact that the system is, in fact, turned off is a significant advantage. If there is a loss of power to the system, as when the battery is swapped on a laptop or a desktop system is unplugged, the saved system data will not be lost. Unlike shutdown, hibernation saves the currently running system so a user can resume where she left off, and because hibernation does not require power, a system can remain in hibernation indefinitely.

Like the PnP manager, the power manager provides notifications to the rest of the system about changes in the power state. Some applications want to know when the system is about to be shut down so they can start saving their states to disk.

#### B.3.3.10 Registry

Windows keeps much of its configuration information in internal databases, called **hives**, that are managed by the Windows configuration manager, which is commonly known as the **registry**. There are separate hives for system information, default user preferences, software installation, security, and boot options. Because the information in the **system hive** is required to boot the system, the registry manager is implemented as a component of the executive.

The registry represents the configuration state in each hive as a hierarchical namespace of keys (directories), each of which can contain a set of typed values, such as UNICODE string, ANSI string, integer, or untyped binary data. In theory, new keys and values are created and initialized as new software is installed; then they are modified to reflect changes in the configuration of that software. In practice, the registry is often used as a general-purpose database, as an interprocess-communication mechanism, and for many other such inventive purposes.

Restarting applications, or even the system, every time a configuration change was made would be a nuisance. Instead, programs rely on various kinds of notifications, such as those provided by the PnP and power managers, to learn about changes in the system configuration. The registry also supplies notifications; it allows threads to register to be notified when changes are made to some part of the registry. The threads can thus detect and adapt to configuration changes recorded in the registry itself.

Whenever significant changes are made to the system, such as when updates to the operating system or drivers are installed, there is a danger that the configuration data may be corrupted (for example, if a working driver is replaced by a nonworking driver or an application fails to install correctly and leaves partial information in the registry). Windows creates a **system restore point** before making such changes. The restore point contains a copy of the hives before the change and can be used to return to this version of the hives and thereby get a corrupted system working again.

To improve the stability of the registry configuration, Windows added a transaction mechanism beginning with Windows Vista that can be used to prevent the registry from being partially updated with a collection of related configuration changes. Registry transactions can be part of more general transactions administered by the **kernel transaction manager (KTM)**, which can also

include file-system transactions. KTM transactions do not have the full semantics found in normal database transactions, and they have not supplanted the system restore facility for recovering from damage to the registry configuration caused by software installation.

#### B.3.3.11 Booting

The booting of a Windows PC begins when the hardware powers on and firmware begins executing from ROM. In older machines, this firmware was known as the BIOS, but more modern systems use UEFI (the Unified Extensible Firmware Interface), which is faster and more general and makes better use of the facilities in contemporary processors. The firmware runs **power-on self-test (POST)** diagnostics; identifies many of the devices attached to the system and initializes them to a clean, power-up state; and then builds the description used by the **advanced configuration and power interface (ACPI)**. Next, the firmware finds the system disk, loads the Windows bootmgr program, and begins executing it.

In a machine that has been hibernating, the `winresume` program is loaded next. It restores the running system from disk, and the system continues execution at the point it had reached right before hibernating. In a machine that has been shut down, the `bootmgr` performs further initialization of the system and then loads `winload`. This program loads `hal.dll`, the kernel (`ntoskrnl.exe`), any drivers needed in booting, and the system hive. `winload` then transfers execution to the kernel.

The kernel initializes itself and creates two processes. The **system process** contains all the internal kernel worker threads and never executes in user mode. The first user-mode process created is SMSS, for *session manager subsystem*, which is similar to the INIT (initialization) process in UNIX. SMSS performs further initialization of the system, including establishing the paging files, loading more device drivers, and managing the Windows sessions. Each session is used to represent a logged-on user, except for *session 0*, which is used to run system-wide background services, such as LSASS and SERVICES. A session is anchored by an instance of the CSRSS process. Each session other than 0 initially runs the WINLOGON process. This process logs on a user and then launches the EXPLORER process, which implements the Windows GUI experience. The following list itemizes some of these aspects of booting:

- SMSS completes system initialization and then starts up session 0 and the first login session.
- WININIT runs in session 0 to initialize user mode and start LSASS, SERVICES, and the local session manager, LSM.
- LSASS, the security subsystem, implements facilities such as authentication of users.
- SERVICES contains the service control manager, or SCM, which supervises all background activities in the system, including user-mode services. A number of services will have registered to start when the system boots. Others will be started only on demand or when triggered by an event such as the arrival of a device.

- CSRSS is the Win32 environmental subsystem process. It is started in every session—unlike the POSIX subsystem, which is started only on demand when a POSIX process is created.
- WINLOGON is run in each Windows session other than session 0 to log on a user.

The system optimizes the boot process by prepaging from files on disk based on previous boots of the system. Disk access patterns at boot are also used to lay out system files on disk to reduce the number of I/O operations required. The processes necessary to start the system are reduced by grouping services into fewer processes. All of these approaches contribute to a dramatic reduction in system boot time. Of course, system boot time is less important than it once was because of the sleep and hibernation capabilities of Windows.

## B.4 Terminal Services and Fast User Switching

Windows supports a GUI-based console that interfaces with the user via keyboard, mouse, and display. Most systems also support audio and video. Audio input is used by Windows voice-recognition software; voice recognition makes the system more convenient and increases its accessibility for users with disabilities. Windows 7 added support for **multi-touch hardware**, allowing users to input data by touching the screen and making gestures with one or more fingers. Eventually, the video-input capability, which is currently used for communication applications, is likely to be used for visually interpreting gestures, as Microsoft has demonstrated for its Xbox 360 Kinect product. Other future input experiences may evolve from Microsoft's **surface computer**. Most often installed at public venues, such as hotels and conference centers, the surface computer is a table surface with special cameras underneath. It can track the actions of multiple users at once and recognize objects that are placed on top.

The PC was, of course, envisioned as a **personal computer**—an inherently single-user machine. Modern Windows, however, supports the sharing of a PC among multiple users. Each user that is logged on using the GUI has a **session** created to represent the GUI environment he will be using and to contain all the processes created to run his applications. Windows allows multiple sessions to exist at the same time on a single machine. However, Windows only supports a single console, consisting of all the monitors, keyboards, and mice connected to the PC. Only one session can be connected to the console at a time. From the logon screen displayed on the console, users can create new sessions or attach to an existing session that was previously created. This allows multiple users to share a single PC without having to log off and on between users. Microsoft calls this use of sessions **fast user switching**.

Users can also create new sessions, or connect to existing sessions, on one PC from a session running on another Windows PC. The terminal server (TS) connects one of the GUI windows in a user's local session to the new or existing session, called a **remote desktop**, on the remote computer. The most common use of remote desktops is for users to connect to a session on their work PC from their home PC.

Many corporations use corporate terminal-server systems maintained in data centers to run all user sessions that access corporate resources, rather than

allowing users to access those resources from the PCs in each user's office. Each server computer may handle many dozens of remote-desktop sessions. This is a form of **thin-client** computing, in which individual computers rely on a server for many functions. Relying on data-center terminal servers improves reliability, manageability, and security of the corporate computing resources.

The TS is also used by Windows to implement **remote assistance**. A remote user can be invited to share a session with the user logged on to the session on the console. The remote user can watch the user's actions and even be given control of the desktop to help resolve computing problems.

## B.5 File System

The native file system in Windows is NTFS. It is used for all local volumes. However, associated USB thumb drives, flash memory on cameras, and external disks may be formatted with the 32-bit FAT file system for portability. FAT is a much older file-system format that is understood by many systems besides Windows, such as the software running on cameras. A disadvantage is that the FAT file system does not restrict file access to authorized users. The only solution for securing data with FAT is to run an application to encrypt the data before storing it on the file system.

In contrast, NTFS uses ACLs to control access to individual files and supports implicit encryption of individual files or entire volumes (using Windows BitLocker feature). NTFS implements many other features as well, including data recovery, fault tolerance, very large files and file systems, multiple data streams, UNICODE names, sparse files, journaling, volume shadow copies, and file compression.

### B.5.1 NTFS Internal Layout

The fundamental entity in NTFS is a volume. A volume is created by the Windows logical disk management utility and is based on a logical disk partition. A volume may occupy a portion of a disk or an entire disk, or may span several disks.

NTFS does not deal with individual sectors of a disk but instead uses clusters as the units of disk allocation. A **cluster** is a number of disk sectors that is a power of 2. The cluster size is configured when an NTFS file system is formatted. The default cluster size is based on the volume size—4 KB for volumes larger than 2 GB. Given the size of today's disks, it may make sense to use cluster sizes larger than the Windows defaults to achieve better performance, although these performance gains will come at the expense of more internal fragmentation.

NTFS uses **logical cluster numbers (LCNs)** as disk addresses. It assigns them by numbering clusters from the beginning of the disk to the end. Using this scheme, the system can calculate a physical disk offset (in bytes) by multiplying the LCN by the cluster size.

A file in NTFS is not a simple byte stream as it is in UNIX; rather, it is a structured object consisting of typed **attributes**. Each attribute of a file is an independent byte stream that can be created, deleted, read, and written. Some attribute types are standard for all files, including the file name (or names, if the file has aliases, such as an MS-DOS short name), the creation time, and the

security descriptor that specifies the access control list. User data are stored in *data attributes*.

Most traditional data files have an *unnamed* data attribute that contains all the file's data. However, additional data streams can be created with explicit names. For instance, in Macintosh files stored on a Windows server, the resource fork is a named data stream. The IProp interfaces of the Component Object Model (COM) use a named data stream to store properties on ordinary files, including thumbnails of images. In general, attributes may be added as necessary and are accessed using a *file-name:attribute* syntax. NTFS returns only the size of the unnamed attribute in response to file-query operations, such as when running the `dir` command.

Every file in NTFS is described by one or more records in an array stored in a special file called the master file table (MFT). The size of a record is determined when the file system is created; it ranges from 1 to 4 KB. Small attributes are stored in the MFT record itself and are called *resident attributes*. Large attributes, such as the unnamed bulk data, are called *nonresident attributes* and are stored in one or more contiguous *extents* on the disk. A pointer to each extent is stored in the MFT record. For a small file, even the data attribute may fit inside the MFT record. If a file has many attributes—or if it is highly fragmented, so that many pointers are needed to point to all the fragments—one record in the MFT might not be large enough. In this case, the file is described by a record called the *base fil record*, which contains pointers to overflow records that hold the additional pointers and attributes.

Each file in an NTFS volume has a unique ID called a *fil reference*. The file reference is a 64-bit quantity that consists of a 48-bit file number and a 16-bit sequence number. The file number is the record number (that is, the array slot) in the MFT that describes the file. The sequence number is incremented every time an MFT entry is reused. The sequence number enables NTFS to perform internal consistency checks, such as catching a stale reference to a deleted file after the MFT entry has been reused for a new file.

### B.5.1.1 NTFS B+ Tree

As in UNIX, the NTFS namespace is organized as a hierarchy of directories. Each directory uses a data structure called a *B+ tree* to store an index of the file names in that directory. In a B+ tree, the length of every path from the root of the tree to a leaf is the same, and the cost of reorganizing the tree is eliminated. The *index root* of a directory contains the top level of the B+ tree. For a large directory, this top level contains pointers to disk extents that hold the remainder of the tree. Each entry in the directory contains the name and file reference of the file, as well as a copy of the update timestamp and file size taken from the file's resident attributes in the MFT. Copies of this information are stored in the directory so that a directory listing can be efficiently generated. Because all the file names, sizes, and update times are available from the directory itself, there is no need to gather these attributes from the MFT entries for each of the files.

### B.5.1.2 NTFS Metadata

The NTFS volume's metadata are all stored in files. The first file is the MFT. The second file, which is used during recovery if the MFT is damaged, contains a

copy of the first 16 entries of the MFT. The next few files are also special in purpose. They include the files described below.

- The **log file** records all metadata updates to the file system.
- The **volume file** contains the name of the volume, the version of NTFS that formatted the volume, and a bit that tells whether the volume may have been corrupted and needs to be checked for consistency using the chkdsk program.
- The **attribute-definitio table** indicates which attribute types are used in the volume and what operations can be performed on each of them.
- The **root directory** is the top-level directory in the file-system hierarchy.
- The **bitmap fil** indicates which clusters on a volume are allocated to files and which are free.
- The **boot fil** contains the startup code for Windows and must be located at a particular disk address so that it can be found easily by a simple ROM bootstrap loader. The boot file also contains the physical address of the MFT.
- The **bad-cluster file** keeps track of any bad areas on the volume; NTFS uses this record for error recovery.

Keeping all the NTFS metadata in actual files has a useful property. As discussed in Section B.3.3.6, the cache manager caches file data. Since all the NTFS metadata reside in files, these data can be cached using the same mechanisms used for ordinary data.

### B.5.2 Recovery

In many simple file systems, a power failure at the wrong time can damage the file-system data structures so severely that the entire volume is scrambled. Many UNIX file systems, including UFS but not ZFS, store redundant metadata on the disk, and they recover from crashes by using the fsck program to check all the file-system data structures and restore them forcibly to a consistent state. Restoring them often involves deleting damaged files and freeing data clusters that had been written with user data but not properly recorded in the file system's metadata structures. This checking can be a slow process and can cause the loss of significant amounts of data.

NTFS takes a different approach to file-system robustness. In NTFS, all file-system data-structure updates are performed inside transactions. Before a data structure is altered, the transaction writes a log record that contains redo and undo information. After the data structure has been changed, the transaction writes a commit record to the log to signify that the transaction succeeded.

After a crash, the system can restore the file-system data structures to a consistent state by processing the log records, first redoing the operations for committed transactions and then undoing the operations for transactions that did not commit successfully before the crash. Periodically (usually every 5 seconds), a checkpoint record is written to the log. The system does not need log records prior to the checkpoint to recover from a crash. They can be

discarded, so the log file does not grow without bounds. The first time after system startup that an NTFS volume is accessed, NTFS automatically performs file-system recovery.

This scheme does not guarantee that all the user-file contents are correct after a crash. It ensures only that the file-system data structures (the metadata files) are undamaged and reflect some consistent state that existed prior to the crash. It would be possible to extend the transaction scheme to cover user files, and Microsoft took some steps to do this in Windows Vista.

The log is stored in the third metadata file at the beginning of the volume. It is created with a fixed maximum size when the file system is formatted. It has two sections: the *logging area*, which is a circular queue of log records, and the *restart area*, which holds context information, such as the position in the logging area where NTFS should start reading during a recovery. In fact, the restart area holds two copies of its information, so recovery is still possible if one copy is damaged during the crash.

The logging functionality is provided by the *log-file service*. In addition to writing the log records and performing recovery actions, the log-file service keeps track of the free space in the log file. If the free space gets too low, the log-file service queues pending transactions, and NTFS halts all new I/O operations. After the in-progress operations complete, NTFS calls the cache manager to flush all data and then resets the log file and performs the queued transactions.

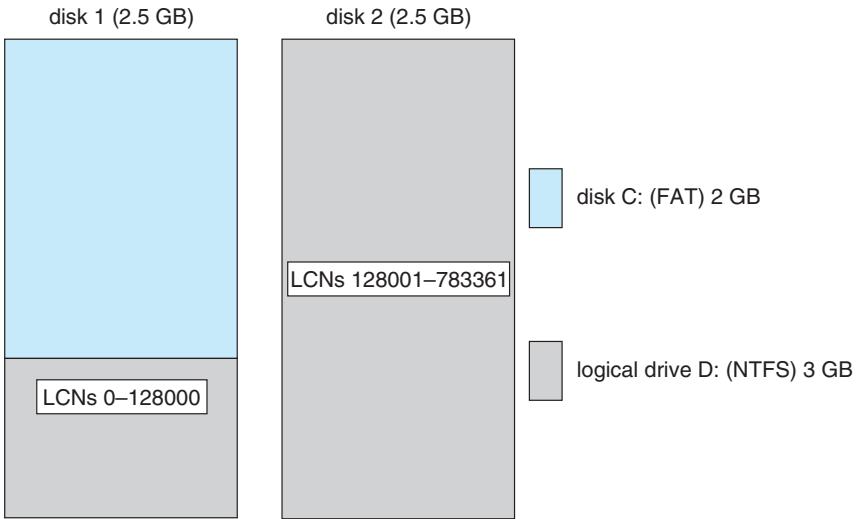
### B.5.3 Security

The security of an NTFS volume is derived from the Windows object model. Each NTFS file references a security descriptor, which specifies the owner of the file, and an access-control list, which contains the access permissions granted or denied to each user or group listed. Early versions of NTFS used a separate security descriptor as an attribute of each file. Beginning with Windows 2000, the security-descriptors attribute points to a shared copy, with a significant savings in disk and caching space; many, many files have identical security descriptors.

In normal operation, NTFS does not enforce permissions on traversal of directories in file path names. However, for compatibility with POSIX, these checks can be enabled. Traversal checks are inherently more expensive, since modern parsing of file path names uses prefix matching rather than directory-by-directory parsing of path names. Prefix matching is an algorithm that looks up strings in a cache and finds the entry with the longest match—for example, an entry for `\foo\bar\dir` would be a match for `\foo\bar\dir2\dir3\myfile`. The prefix-matching cache allows path-name traversal to begin much deeper in the tree, saving many steps. Enforcing traversal checks means that the user's access must be checked at each directory level. For instance, a user might lack permission to traverse `\foo\bar`, so starting at the access for `\foo\bar\dir` would be an error.

### B.5.4 Volume Management and Fault Tolerance

FtDisk is the fault-tolerant disk driver for Windows. When installed, it provides several ways to combine multiple disk drives into one logical volume so as to improve performance, capacity, or reliability.



**Figure B.7** Volume set on two drives.

#### B.5.4.1 Volume Sets and RAID Sets

One way to combine multiple disks is to concatenate them logically to form a large logical volume, as shown in Figure B.7. In Windows, this logical volume, called a **volume set**, can consist of up to 32 physical partitions. A volume set that contains an NTFS volume can be extended without disturbance of the data already stored in the file system. The bitmap metadata on the NTFS volume are simply extended to cover the newly added space. NTFS continues to use the same LCN mechanism that it uses for a single physical disk, and the FtDisk driver supplies the mapping from a logical-volume offset to the offset on one particular disk.

Another way to combine multiple physical partitions is to interleave their blocks in round-robin fashion to form a **stripe set**. This scheme is also called RAID level 0, or **disk striping**. (For more on RAID (redundant arrays of inexpensive disks), see Section 11.8.) FtDisk uses a stripe size of 64 KB. The first 64 KB of the logical volume are stored in the first physical partition, the second 64 KB in the second physical partition, and so on, until each partition has contributed 64 KB of space. Then, the allocation wraps around to the first disk, allocating the second 64-KB block. A stripe set forms one large logical volume, but the physical layout can improve the I/O bandwidth, because for a large I/O, all the disks can transfer data in parallel. Windows also supports RAID level 5, stripe set with parity, and RAID level 1, mirroring.

#### B.5.4.2 Sector Sparing and Cluster Remapping

To deal with disk sectors that go bad, FtDisk uses a hardware technique called sector sparing, and NTFS uses a software technique called cluster remapping. **Sector sparing** is a hardware capability provided by many disk drives. When a disk drive is formatted, it creates a map from logical block numbers to good sectors on the disk. It also leaves extra sectors unmapped, as spares. If a sector fails, FtDisk instructs the disk drive to substitute a spare. **Cluster remapping**

is a software technique performed by the file system. If a disk block goes bad, NTFS substitutes a different, unallocated block by changing any affected pointers in the MFT. NTFS also makes a note that the bad block should never be allocated to any file.

When a disk block goes bad, the usual outcome is a data loss. But sector sparing or cluster remapping can be combined with fault-tolerant volumes to mask the failure of a disk block. If a read fails, the system reconstructs the missing data by reading the mirror or by calculating the exclusive or parity in a stripe set with parity. The reconstructed data are stored in a new location that is obtained by sector sparing or cluster remapping.

### **B.5.5 Compression**

NTFS can perform data compression on individual files or on all data files in a directory. To compress a file, NTFS divides the file's data into **compression units**, which are blocks of 16 contiguous clusters. When a compression unit is written, a data-compression algorithm is applied. If the result fits into fewer than 16 clusters, the compressed version is stored. When reading, NTFS can determine whether data have been compressed: if they have been, the length of the stored compression unit is less than 16 clusters. To improve performance when reading contiguous compression units, NTFS prefetches and decompresses ahead of the application requests.

For sparse files or files that contain mostly zeros, NTFS uses another technique to save space. Clusters that contain only zeros because they have never been written are not actually allocated or stored on disk. Instead, gaps are left in the sequence of virtual-cluster numbers stored in the MFT entry for the file. When reading a file, if NTFS finds a gap in the virtual-cluster numbers, it just zero-fills that portion of the caller's buffer. This technique is also used by UNIX.

### **B.5.6 Mount Points, Symbolic Links, and Hard Links**

Mount points are a form of symbolic link specific to directories on NTFS that were introduced in Windows 2000. They provide a mechanism for organizing disk volumes that is more flexible than the use of global names (like drive letters). A mount point is implemented as a symbolic link with associated data that contains the true volume name. Ultimately, mount points will supplant drive letters completely, but there will be a long transition due to the dependence of many applications on the drive-letter scheme.

Windows Vista introduced support for a more general form of symbolic links, similar to those found in UNIX. The links can be absolute or relative, can point to objects that do not exist, and can point to both files and directories even across volumes. NTFS also supports **hard links**, where a single file has an entry in more than one directory of the same volume.

### **B.5.7 Change Journal**

NTFS keeps a journal describing all changes that have been made to the file system. User-mode services can receive notifications of changes to the journal and then identify what files have changed by reading from the journal. The search indexer service uses the change journal to identify files that need to be

re-indexed. The file-replication service uses it to identify files that need to be replicated across the network.

### B.5.8 Volume Shadow Copies

Windows implements the capability of bringing a volume to a known state and then creating a shadow copy that can be used to back up a consistent view of the volume. This technique is known as *snapshots* in some other file systems. Making a shadow copy of a volume is a form of copy-on-write, where blocks modified after the shadow copy is created are stored in their original form in the copy. To achieve a consistent state for the volume requires the cooperation of applications, since the system cannot know when the data used by the application are in a stable state from which the application could be safely restarted.

The server version of Windows uses shadow copies to efficiently maintain old versions of files stored on file servers. This allows users to see documents stored on file servers as they existed at earlier points in time. The user can use this feature to recover files that were accidentally deleted or simply to look at a previous version of the file, all without pulling out backup media.

## B.6 Networking

Windows supports both peer-to-peer and client–server networking. It also has facilities for network management. The networking components in Windows provide data transport, interprocess communication, file sharing across a network, and the ability to send print jobs to remote printers.

### B.6.1 Network Interfaces

To describe networking in Windows, we must first mention two of the internal networking interfaces: the [network device interface specification \(NDIS\)](#) and the [transport driver interface \(TDI\)](#). The NDIS interface was developed in 1989 by Microsoft and 3Com to separate network adapters from transport protocols so that either could be changed without affecting the other. NDIS resides at the interface between the data-link and network layers in the ISO model and enables many protocols to operate over many different network adapters. In terms of the ISO model, the TDI is the interface between the transport layer (layer 4) and the session layer (layer 5). This interface enables any session-layer component to use any available transport mechanism. (Similar reasoning led to the streams mechanism in UNIX.) The TDI supports both connection-based and connectionless transport and has functions to send any type of data.

### B.6.2 Protocols

Windows implements transport protocols as drivers. These drivers can be loaded and unloaded from the system dynamically, although in practice the system typically has to be rebooted after a change. Windows comes with several networking protocols. Next, we discuss a number of these protocols.

### B.6.2.1 Server-Message Block

The **server-message-block (SMB)** protocol was first introduced in MS-DOS 3.1. The system uses the protocol to send I/O requests over the network. The SMB protocol has four message types. Session control messages are commands that start and end a redirector connection to a shared resource at the server. A redirector uses File messages to access files at the server. Printer messages are used to send data to a remote print queue and to receive status information from the queue, and Message messages are used to communicate with another workstation. A version of the SMB protocol was published as the **common Internet fil system (CIFS)** and is supported on a number of operating systems.

### B.6.2.2 Transmission Control Protocol/Internet Protocol

The transmission control protocol/Internet protocol (TCP/IP) suite that is used on the Internet has become the de facto standard networking infrastructure. Windows uses TCP/IP to connect to a wide variety of operating systems and hardware platforms. The Windows TCP/IP package includes the simple network-management protocol (SNM), the dynamic host-configuration protocol (DHCP), and the older Windows Internet name service (WINS). Windows Vista introduced a new implementation of TCP/IP that supports both IPv4 and IPv6 in the same network stack. This new implementation also supports offloading of the network stack onto advanced hardware, to achieve very high performance for servers.

Windows provides a software firewall that limits the TCP ports that can be used by programs for network communication. Network firewalls are commonly implemented in routers and are a very important security measure. Having a firewall built into the operating system makes a hardware router unnecessary, and it also provides more integrated management and easier use.

### B.6.2.3 Point-to-Point Tunneling Protocol

The **point-to-point tunneling protocol (PPTP)** is a protocol provided by Windows to communicate between remote-access server modules running on Windows server machines and other client systems that are connected over the Internet. The remote-access servers can encrypt data sent over the connection, and they support multiprotocol **virtual private networks (VPNs)** over the Internet.

### B.6.2.4 HTTP Protocol

The HTTP protocol is used to get/put information using the World Wide Web. Windows implements HTTP using a kernel-mode driver, so web servers can operate with a low-overhead connection to the networking stack. HTTP is a fairly general protocol that Windows makes available as a transport option for implementing RPC.

### B.6.2.5 Web-Distributed Authoring and Versioning Protocol

Web-distributed authoring and versioning (WebDAV) is an HTTP-based protocol for collaborative authoring across a network. Windows builds a WebDAV

redirector into the file system. Being built directly into the file system enables WebDAV to work with other file-system features, such as encryption. Personal files can then be stored securely in a public place. Because WebDAV uses HTTP, which is a get/put protocol, Windows has to cache the files locally so programs can use read and write operations on parts of the files.

#### B.6.2.6 Named Pipes

**Named pipes** are a connection-oriented messaging mechanism. A process can use named pipes to communicate with other processes on the same machine. Since named pipes are accessed through the file-system interface, the security mechanisms used for file objects also apply to named pipes. The SMB protocol supports named pipes, so they can also be used for communication between processes on different systems.

The format of pipe names follows the **uniform naming convention (UNC)**. A UNC name looks like a typical remote file name. The format is \\server\_name\share\_name\x\y\z, where server\_name identifies a server on the network; share\_name identifies any resource that is made available to network users, such as directories, files, named pipes, and printers; and \x\y\z is a normal file path name.

#### B.6.2.7 Remote Procedure Calls

A remote procedure call (RPC) is a client–server mechanism that enables an application on one machine to make a procedure call to code on another machine. The client calls a local procedure—a stub routine—that packs its arguments into a message and sends them across the network to a particular server process. The client-side stub routine then blocks. Meanwhile, the server unpacks the message, calls the procedure, packs the return results into a message, and sends them back to the client stub. The client stub unblocks, receives the message, unpacks the results of the RPC, and returns them to the caller. This packing of arguments is sometimes called **marshaling**. The client stub code and the descriptors necessary to pack and unpack the arguments for an RPC are compiled from a specification written in the **Microsoft Interface Definition Language**.

The Windows RPC mechanism follows the widely used distributed-computing-environment standard for RPC messages, so programs written to use Windows RPCs are highly portable. The RPC standard is detailed. It hides many of the architectural differences among computers, such as the sizes of binary numbers and the order of bytes and bits in computer words, by specifying standard data formats for RPC messages.

#### B.6.2.8 Component Object Model

The **component object model (COM)** is a mechanism for interprocess communication that was developed for Windows. COM objects provide a well-defined interface to manipulate the data in the object. For instance, COM is the infrastructure used by Microsoft's **object linking and embedding (OLE)** technology for inserting spreadsheets into Microsoft Word documents. Many Windows services provide COM interfaces. Windows has a distributed extension called

**DCOM** that can be used over a network utilizing RPC to provide a transparent method of developing distributed applications.

### B.6.3 Redirectors and Servers

In Windows, an application can use the Windows I/O API to access files from a remote computer as though they were local, provided that the remote computer is running a CIFS server such as those provided by Windows. A **redirector** is the client-side object that forwards I/O requests to a remote system, where they are satisfied by a server. For performance and security, the redirectors and servers run in kernel mode.

In more detail, access to a remote file occurs as follows:

1. The application calls the I/O manager to request that a file be opened with a file name in the standard UNC format.
2. The I/O manager builds an I/O request packet, as described in Section B.3.3.5.
3. The I/O manager recognizes that the access is for a remote file and calls a driver called a **multiple universal-naming-convention provider (MUP)**.
4. The MUP sends the I/O request packet asynchronously to all registered redirectors.
5. A redirector that can satisfy the request responds to the MUP. To avoid asking all the redirectors the same question in the future, the MUP uses a cache to remember which redirector can handle this file.
6. The redirector sends the network request to the remote system.
7. The remote-system network drivers receive the request and pass it to the server driver.
8. The server driver hands the request to the proper local file-system driver.
9. The proper device driver is called to access the data.
10. The results are returned to the server driver, which sends the data back to the requesting redirector. The redirector then returns the data to the calling application via the I/O manager.

A similar process occurs for applications that use the Win32 network API, rather than the UNC services, except that a module called a **multi-provider router** is used instead of a MUP.

For portability, redirectors and servers use the TDI API for network transport. The requests themselves are expressed in a higher-level protocol, which by default is the SMB protocol described in Section B.6.2. The list of redirectors is maintained in the system hive of the registry.

#### B.6.3.1 Distributed File System

UNC names are not always convenient, because multiple file servers may be available to serve the same content and UNC names explicitly include the name

of the server. Windows supports a **distributed file-system (DFS)** protocol that allows a network administrator to serve up files from multiple servers using a single distributed name space.

### B.6.3.2 Folder Redirection and Client-Side Caching

To improve the PC experience for users who frequently switch among computers, Windows allows administrators to give users **roaming profile**, which keep users' preferences and other settings on servers. **Folder redirection** is then used to automatically store a user's documents and other files on a server.

This works well until one of the computers is no longer attached to the network, as when a user takes a laptop onto an airplane. To give users off-line access to their redirected files, Windows uses **client-side caching (CSC)**. CSC is also used when the computer is on-line to keep copies of the server files on the local machine for better performance. The files are pushed up to the server as they are changed. If the computer becomes disconnected, the files are still available, and the update of the server is deferred until the next time the computer is online.

## B.6.4 Domains

Many networked environments have natural groups of users, such as students in a computer laboratory at school or employees in one department in a business. Frequently, we want all the members of the group to be able to access shared resources on their various computers in the group. To manage the global access rights within such groups, Windows uses the concept of a domain. Previously, these domains had no relationship whatsoever to the domain-name system (DNS) that maps Internet host names to IP addresses. Now, however, they are closely related.

Specifically, a Windows domain is a group of Windows workstations and servers that share a common security policy and user database. Since Windows uses the Kerberos protocol for trust and authentication, a Windows domain is the same thing as a Kerberos realm. Windows uses a hierarchical approach for establishing trust relationships between related domains. The trust relationships are based on DNS and allow transitive trusts that can flow up and down the hierarchy. This approach reduces the number of trusts required for  $n$  domains from  $n * (n - 1)$  to  $O(n)$ . The workstations in the domain trust the domain controller to give correct information about the access rights of each user (loaded into the user's access token by LSASS). All users retain the ability to restrict access to their own workstations, however, no matter what any domain controller may say.

## B.6.5 Active Directory

**Active Directory** is the Windows implementation of **lightweight directory-access protocol (LDAP)** services. Active Directory stores the topology information about the domain, keeps the domain-based user and group accounts and passwords, and provides a domain-based store for Windows features that need it, such as **Windows group policy**. Administrators use group policies to establish uniform standards for desktop preferences and software. For many

corporate information-technology groups, this uniformity drastically reduces the cost of computing.

## B.7 Programmer Interface

The **Win32 API** is the fundamental interface to the capabilities of Windows. This section describes five main aspects of the Win32 API: access to kernel objects, sharing of objects between processes, process management, interprocess communication, and memory management.

### B.7.1 Access to Kernel Objects

The Windows kernel provides many services that application programs can use. Application programs obtain these services by manipulating kernel objects. A process gains access to a kernel object named XXX by calling the CreateXXX function to open a handle to an instance of XXX. This handle is unique to the process. Depending on which object is being opened, if the Create() function fails, it may return 0, or it may return a special constant named INVALID\_HANDLE\_VALUE. A process can close any handle by calling the CloseHandle() function, and the system may delete the object if the count of handles referencing the object in all processes drops to zero.

### B.7.2 Sharing Objects between Processes

Windows provides three ways to share objects between processes. The first way is for a child process to inherit a handle to the object. When the parent calls the CreateXXX function, the parent supplies a SECURITIES\_ATTRIBUTES structure with the bInheritHandle field set to TRUE. This field creates an inheritable handle. Next, the child process is created, passing a value of TRUE to the CreateProcess() function's bInheritHandle argument. Figure B.8 shows a code sample that creates a semaphore handle inherited by a child process.

---

```
SECURITY_ATTRIBUTES sa;
sa.nlength = sizeof(sa);
sa.lpSecurityDescriptor = NULL;
sa.bInheritHandle = TRUE;
Handle a_semaphore = CreateSemaphore(&sa, 1, 1, NULL);
char command_line[132];
ostringstream ostring(command_line, sizeof(command_line));
ostring << a_semaphore << ends;
CreateProcess("another_process.exe", command_line,
NULL, NULL, TRUE, . . .);
```

---

**Figure B.8** Code enabling a child to share an object by inheriting a handle.

```
// Process A
. . .
HANDLE a_semaphore = CreateSemaphore(NULL, 1, 1, "MySEM1");
. . .

// Process B
. . .
HANDLE b_semaphore = OpenSemaphore(SEMAPHORE_ALL_ACCESS,
    FALSE, "MySEM1");
. . .
```

---

**Figure B.9** Code for sharing an object by name lookup.

Assuming the child process knows which handles are shared, the parent and child can achieve interprocess communication through the shared objects. In the example in Figure B.8, the child process gets the value of the handle from the first command-line argument and then shares the semaphore with the parent process.

The second way to share objects is for one process to give the object a name when the object is created and for the second process to open the name. This method has two drawbacks: Windows does not provide a way to check whether an object with the chosen name already exists, and the object name space is global, without regard to the object type. For instance, two applications may create and share a single object named “foo” when two distinct objects—possibly of different types—were desired.

Named objects have the advantage that unrelated processes can readily share them. The first process calls one of the `CreateXXX` functions and supplies a name as a parameter. The second process gets a handle to share the object by calling `OpenXXX()` (or `CreateXXX`) with the same name, as shown in the example in Figure B.9.

The third way to share objects is via the `DuplicateHandle()` function. This method requires some other method of interprocess communication to pass the duplicated handle. Given a handle to a process and the value of a handle within that process, a second process can get a handle to the same object and thus share it. An example of this method is shown in Figure B.10.

### B.7.3 Process Management

In Windows, a **process** is a loaded instance of an application and a **thread** is an executable unit of code that can be scheduled by the kernel dispatcher. Thus, a process contains one or more threads. A process is created when a thread in some other process calls the `CreateProcess()` API. This routine loads any dynamic link libraries used by the process and creates an initial thread in the process. Additional threads can be created by the `CreateThread()` function. Each thread is created with its own stack, which defaults to 1 MB unless otherwise specified in an argument to `CreateThread()`.

```

// Process A wants to give Process B access to a semaphore

// Process A
HANDLE a_semaphore = CreateSemaphore(NULL, 1, 1, NULL);
// send the value of the semaphore to Process B
// using a message or shared memory object
. . .

// Process B
HANDLE process_a = OpenProcess(PROCESS_ALL_ACCESS, FALSE,
    process_id_of_A);
HANDLE b_semaphore;
DuplicateHandle(process_a, a_semaphore,
    GetCurrentProcess(), &b_semaphore,
    0, FALSE, DUPLICATE_SAME_ACCESS);
// use b_semaphore to access the semaphore
. . .

```

---

**Figure B.10** Code for sharing an object by passing a handle.

### B.7.3.1 Scheduling Rule

Priorities in the Win32 environment are based on the native kernel (NT) scheduling model, but not all priority values may be chosen. The Win32 API uses four priority classes:

1. IDLE\_PRIORITY\_CLASS (NT priority level 4)
2. NORMAL\_PRIORITY\_CLASS (NT priority level 8)
3. HIGH\_PRIORITY\_CLASS (NT priority level 13)
4. REALTIME\_PRIORITY\_CLASS (NT priority level 24)

Processes are typically members of the NORMAL\_PRIORITY\_CLASS unless the parent of the process was of the IDLE\_PRIORITY\_CLASS or another class was specified when CreateProcess was called. The priority class of a process is the default for all threads that execute in the process. It can be changed with the SetPriorityClass() function or by passing an argument to the START command. Only users with the *increase scheduling priority* privilege can move a process into the REALTIME\_PRIORITY\_CLASS. Administrators and power users have this privilege by default.

When a user is running an interactive process, the system needs to schedule the process's threads to provide good responsiveness. For this reason, Windows has a special scheduling rule for processes in the NORMAL\_PRIORITY\_CLASS. Windows distinguishes between the process associated with the foreground window on the screen and the other (background) processes. When a process moves into the foreground, Windows increases the scheduling quantum for all its threads by a factor of 3; CPU-bound threads

in the foreground process will run three times longer than similar threads in background processes.

### B.7.3.2 Thread Priorities

A thread starts with an initial priority determined by its class. The priority can be altered by the `SetThreadPriority()` function. This function takes an argument that specifies a priority relative to the base priority of its class:

- `THREAD_PRIORITY_LOWEST`: base – 2
- `THREAD_PRIORITY_BELOW_NORMAL`: base – 1
- `THREAD_PRIORITY_NORMAL`: base + 0
- `THREAD_PRIORITY_ABOVE_NORMAL`: base + 1
- `THREAD_PRIORITY_HIGHEST`: base + 2

Two other designations are also used to adjust the priority. Recall from Section B.3.2.2 that the kernel has two priority classes: 16–31 for the real-time class and 1–15 for the variable class. `THREAD_PRIORITY_IDLE` sets the priority to 16 for real-time threads and to 1 for variable-priority threads. `THREAD_PRIORITY_TIME_CRITICAL` sets the priority to 31 for real-time threads and to 15 for variable-priority threads.

As discussed in Section B.3.2.2, the kernel adjusts the priority of a variable class thread dynamically depending on whether the thread is I/O bound or CPU bound. The Win32 API provides a method to disable this adjustment via `SetProcessPriorityBoost()` and `SetThreadPriorityBoost()` functions.

### B.7.3.3 Thread Suspend and Resume

A thread can be created in a *suspended state* or can be placed in a suspended state later by use of the `SuspendThread()` function. Before a suspended thread can be scheduled by the kernel dispatcher, it must be moved out of the suspended state by use of the `ResumeThread()` function. Both functions set a counter so that if a thread is suspended twice, it must be resumed twice before it can run.

### B.7.3.4 Thread Synchronization

To synchronize concurrent access to shared objects by threads, the kernel provides synchronization objects, such as semaphores and mutexes. These are dispatcher objects, as discussed in Section B.3.2.2. Threads can also synchronize with kernel services operating on kernel objects—such as threads, processes, and files—because these are also dispatcher objects. Synchronization with kernel dispatcher objects can be achieved by use of the `WaitForSingleObject()` and `WaitForMultipleObjects()` functions; these functions wait for one or more dispatcher objects to be signaled.

Another method of synchronization is available to threads within the same process that want to execute code exclusively. The Win32 **critical section object** is a user-mode mutex object that can often be acquired and released without entering the kernel. On a multiprocessor, a Win32 critical section will attempt to spin while waiting for a critical section held by another thread to be released.

If the spinning takes too long, the acquiring thread will allocate a kernel mutex and yield its CPU. Critical sections are particularly efficient because the kernel mutex is allocated only when there is contention and then used only after attempting to spin. Most mutexes in programs are never actually contended, so the savings are significant.

Before using a critical section, some thread in the process must call `InitializeCriticalSection()`. Each thread that wants to acquire the mutex calls `EnterCriticalSection()` and then later calls `LeaveCriticalSection()` to release the mutex. There is also a `TryEnterCriticalSection()` function, which attempts to acquire the mutex without blocking.

For programs that want user-mode reader-writer locks rather than a mutex, Win32 supports **slim reader-writer (SRW) locks**. SRW locks have APIs similar to those for critical sections, such as `InitializeSRWLock`, `AcquireSRWLockXXX`, and `ReleaseSRWLockXXX`, where XXX is either Exclusive or Shared, depending on whether the thread wants write access or just read access to the object protected by the lock. The Win32 API also supports **condition variables**, which can be used with either critical sections or SRW locks.

#### B.7.3.5 Thread Pool

Repeatedly creating and deleting threads can be expensive for applications and services that perform small amounts of work in each instantiation. The Win32 thread pool provides user-mode programs with three services: a queue to which work requests may be submitted (via the `SubmitThreadpoolWork()` function), an API that can be used to bind callbacks to waitable handles (`RegisterWaitForSingleObject()`), and APIs to work with timers (`CreateThreadpoolTimer()` and `WaitForThreadpoolTimerCallbacks()`) and to bind callbacks to I/O completion queues (`BindIoCompletionCallback()`).

The goal of using a thread pool is to increase performance and reduce memory footprint. Threads are relatively expensive, and each processor can only be executing one thread at a time no matter how many threads are available. The thread pool attempts to reduce the number of runnable threads by slightly delaying work requests (reusing each thread for many requests) while providing enough threads to effectively utilize the machine's CPUs. The wait and I/O- and timer-callback APIs allow the thread pool to further reduce the number of threads in a process, using far fewer threads than would be necessary if a process were to devote separate threads to servicing each waitable handle, timer, or completion port.

#### B.7.3.6 Fibers

A **fibre** is user-mode code that is scheduled according to a user-defined scheduling algorithm. Fibers are completely a user-mode facility; the kernel is not aware that they exist. The fiber mechanism uses Windows threads as if they were CPUs to execute the fibers. Fibers are cooperatively scheduled, meaning that they are never preempted but must explicitly yield the thread on which they are running. When a fiber yields a thread, another fiber can be scheduled on it by the run-time system (the programming language run-time code).

The system creates a fiber by calling either `ConvertThreadToFiber()` or `CreateFiber()`. The primary difference between these functions is that

`CreateFiber()` does not begin executing the fiber that was created. To begin execution, the application must call `SwitchToFiber()`. The application can terminate a fiber by calling `DeleteFiber()`.

Fibers are not recommended for threads that use Win32 APIs rather than standard C-library functions because of potential incompatibilities. Win32 user-mode threads have a **thread-environment block (TEB)** that contains numerous per-thread fields used by the Win32 APIs. Fibers must share the TEB of the thread on which they are running. This can lead to problems when a Win32 interface puts state information into the TEB for one fiber and then the information is overwritten by a different fiber. Fibers are included in the Win32 API to facilitate the porting of legacy UNIX applications that were written for a user-mode thread model such as Pthreads.

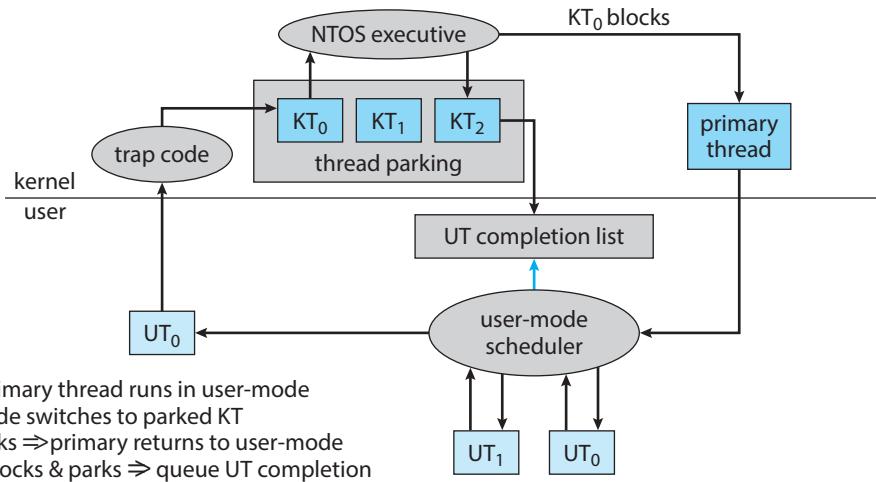
### B.7.3.7 User-Mode Scheduling (UMS) and ConcRT

A new mechanism in Windows 7, user-mode scheduling (UMS), addresses several limitations of fibers. First, recall that fibers are unreliable for executing Win32 APIs because they do not have their own TEBs. When a thread running a fiber blocks in the kernel, the user scheduler loses control of the CPU for a time as the kernel dispatcher takes over scheduling. Problems may result when fibers change the kernel state of a thread, such as the priority or impersonation token, or when they start asynchronous I/O.

UMS provides an alternative model by recognizing that each Windows thread is actually two threads: a kernel thread (KT) and a user thread (UT). Each type of thread has its own stack and its own set of saved registers. The KT and UT appear as a single thread to the programmer because UTs can never block but must always enter the kernel, where an implicit switch to the corresponding KT takes place. UMS uses each UT's TEB to uniquely identify the UT. When a UT enters the kernel, an explicit switch is made to the KT that corresponds to the UT identified by the current TEB. The reason the kernel does not know which UT is running is that UTs can invoke a user-mode scheduler, as fibers do. But in UMS, the scheduler switches UTs, including switching the TEBs.

When a UT enters the kernel, its KT may block. When this happens, the kernel switches to a scheduling thread, which UMS calls a *primary*, and uses this thread to reenter the user-mode scheduler so that it can pick another UT to run. Eventually, a blocked KT will complete its operation and be ready to return to user mode. Since UMS has already reentered the user-mode scheduler to run a different UT, UMS queues the UT corresponding to the completed KT to a completion list in user mode. When the user-mode scheduler is choosing a new UT to switch to, it can examine the completion list and treat any UT on the list as a candidate for scheduling.

Unlike fibers, UMS is not intended to be used directly by the programmer. The details of writing user-mode schedulers can be very challenging, and UMS does not include such a scheduler. Rather, the schedulers come from programming language libraries that build on top of UMS. Microsoft Visual Studio 2010 shipped with Concurrency Runtime (ConcRT), a concurrent programming framework for C++. ConcRT provides a user-mode scheduler together with facilities for decomposing programs into tasks, which can then be scheduled on the available CPUs. ConcRT provides support for `par_for` styles of con-



**Figure B.11** User-mode scheduling.

structs, as well as rudimentary resource management and task synchronization primitives. The key features of UMS are depicted in Figure B.11.

#### B.7.3.8 Winsock

**Winsock** is the Windows sockets API. Winsock is a session-layer interface that is largely compatible with UNIX sockets but has some added Windows extensions. It provides a standardized interface to many transport protocols that may have different addressing schemes, so that any Winsock application can run on any Winsock-compliant protocol stack. Winsock underwent a major update in Windows Vista to add tracing, IPv6 support, impersonation, new security APIs and many other features.

Winsock follows the Windows Open System Architecture (WOSA) model, which provides a standard service provider interface (SPI) between applications and networking protocols. Applications can load and unload *layered protocols* that build additional functionality, such as additional security, on top of the transport protocol layers. Winsock supports asynchronous operations and notifications, reliable multicasting, secure sockets, and kernel mode sockets. There is also support for simpler usage models, like the `WSAConnectByName()` function, which accepts the target as strings specifying the name or IP address of the server and the service or port number of the destination port.

#### B.7.4 IPC Using Windows Messaging

Win32 applications handle interprocess communication in several ways. One way is by using shared kernel objects. Another is by using the Windows messaging facility, an approach that is particularly popular for Win32 GUI applications. One thread can send a message to another thread or to a window by calling `PostMessage()`, `PostThreadMessage()`, `SendMessage()`, `SendThreadMessage()`, or `SendMessageCallback()`. *Posting* a message and *sending* a message differ in this way: the post routines are asynchronous, they return immediately, and the calling thread does not know when the message

```
// allocate 16 MB at the top of our address space
void *buf = VirtualAlloc(0, 0x1000000, MEM_RESERVE | MEM_TOP_DOWN,
    PAGE_READWRITE);
// commit the upper 8 MB of the allocated space
VirtualAlloc(buf + 0x800000, 0x800000, MEM_COMMIT, PAGE_READWRITE);
// do something with the memory
. . .
// now decommit the memory
VirtualFree(buf + 0x800000, 0x800000, MEM_DECOMMIT);
// release all of the allocated address space
VirtualFree(buf, 0, MEM_RELEASE);
```

---

**Figure B.12** Code fragments for allocating virtual memory.

is actually delivered. The send routines are synchronous: they block the caller until the message has been delivered and processed.

In addition to sending a message, a thread can send data with the message. Since processes have separate address spaces, the data must be copied. The system copies data by calling `SendMessage()` to send a message of type `WM_COPYDATA` with a `COPYDATASTRUCT` data structure that contains the length and address of the data to be transferred. When the message is sent, Windows copies the data to a new block of memory and gives the virtual address of the new block to the receiving process.

Every Win32 thread has its own input queue from which it receives messages. If a Win32 application does not call `GetMessage()` to handle events on its input queue, the queue fills up, and after about five seconds, the system marks the application as “Not Responding”.

## B.7.5 Memory Management

The Win32 API provides several ways for an application to use memory: virtual memory, memory-mapped files, heaps, and thread-local storage.

### B.7.5.1 Virtual Memory

An application calls `VirtualAlloc()` to reserve or commit virtual memory and `VirtualFree()` to decommit or release the memory. These functions enable the application to specify the virtual address at which the memory is allocated. They operate on multiples of the memory page size. Examples of these functions appear in Figure B.12.

A process may lock some of its committed pages into physical memory by calling `VirtualLock()`. The maximum number of pages a process can lock is 30, unless the process first calls `SetProcessWorkingSetSize()` to increase the maximum working-set size.

### B.7.5.2 Memory-Mapping Files

Another way for an application to use memory is by memory-mapping a file into its address space. Memory mapping is also a convenient way for two

```

// open the file or create it if it does not exist
HANDLE hfile = CreateFile("somefile", GENERIC_READ | GENERIC_WRITE,
    FILE_SHARE_READ | FILE_SHARE_WRITE, NULL,
    OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
// create the file mapping 8 MB in size
HANDLE hmap = CreateFileMapping(hfile, PAGE_READWRITE,
    SEC_COMMIT, 0, 0x800000, "SHM_1");
// now get a view of the space mapped
void *buf = MapViewOfFile(hmap, FILE_MAP_ALL_ACCESS,
    0, 0, 0x800000);
// do something with the mapped file
. . .
// now unmap the file
UnMapViewOfFile(buf);
CloseHandle(hmap);
CloseHandle(hfile);

```

---

**Figure B.13** Code fragments for memory mapping of a file.

processes to share memory: both processes map the same file into their virtual memory. Memory mapping is a multistage process, as you can see in the example in Figure B.13.

If a process wants to map some address space just to share a memory region with another process, no file is needed. The process calls `CreateFileMapping()` with a file handle of `0xffffffff` and a particular size. The resulting file-mapping object can be shared by inheritance, by name lookup, or by handle duplication.

### B.7.5.3 Heaps

Heaps provide a third way for applications to use memory, just as with `malloc()` and `free()` in standard C. A heap in the Win32 environment is a region of reserved address space. When a Win32 process is initialized, it is created with a **default heap**. Since most Win32 applications are multithreaded, access to the heap is synchronized to protect the heap's space-allocation data structures from being damaged by concurrent updates by multiple threads.

Win32 provides several heap-management functions so that a process can allocate and manage a private heap. These functions are `HeapCreate()`, `HeapAlloc()`, `HeapRealloc()`, `HeapSize()`, `HeapFree()`, and `HeapDestroy()`. The Win32 API also provides the `HeapLock()` and `HeapUnlock()` functions to enable a thread to gain exclusive access to a heap. Unlike `VirtualLock()`, these functions perform only synchronization; they do not lock pages into physical memory.

The original Win32 heap was optimized for efficient use of space. This led to significant problems with fragmentation of the address space for larger server programs that ran for long periods of time. A new **low-fragmentation heap (LFH)** design introduced in Windows XP greatly reduced the fragmen-

```
// reserve a slot for a variable  
DWORD var_index = T1sAlloc();  
// set it to the value 10  
T1sSetValue(var_index, 10);  
// get the value  
int var = T1sGetValue(var_index);  
// release the index  
T1sFree(var_index);
```

---

**Figure B.14** Code for dynamic thread-local storage.

tation problem. The Windows 7 heap manager automatically turns on LFH as appropriate.

#### B.7.5.4 Thread-Local Storage

A fourth way for applications to use memory is through a **thread-local storage (TLS)** mechanism. Functions that rely on global or static data typically fail to work properly in a multithreaded environment. For instance, the C run-time function `strtok()` uses a static variable to keep track of its current position while parsing a string. For two concurrent threads to execute `strtok()` correctly, they need separate `current_position` variables. TLS provides a way to maintain instances of variables that are global to the function being executed but not shared with any other thread.

TLS provides both dynamic and static methods of creating thread-local storage. The dynamic method is illustrated in Figure B.14. The TLS mechanism allocates global heap storage and attaches it to the thread environment block that Windows allocates to every user-mode thread. The TEB is readily accessible by each thread and is used not just for TLS but for all the per-thread state information in user mode.

To use a thread-local static variable, the application declares the variable as follows to ensure that every thread has its own private copy:

```
_declspec(thread) DWORD cur_pos = 0;
```

## B.8 Summary

Microsoft designed Windows to be an extensible, portable operating system—one able to take advantage of new techniques and hardware. Windows supports multiple operating environments and symmetric multiprocessing, including both 32-bit and 64-bit processors and NUMA computers. The use of kernel objects to provide basic services, along with support for client–server computing, enables Windows to support a wide variety of application environments. Windows provides virtual memory, integrated caching, and preemptive scheduling. It supports elaborate security mechanisms and includes internationalization features. Windows runs on a wide variety of computers, so users can choose and upgrade hardware to match their budgets and performance requirements without needing to alter the applications they run.

## Practice Exercises

- B.1** What type of operating system is Windows? Describe two of its major features.
- B.2** List the design goals of Windows. Describe two in detail.
- B.3** Describe the booting process for a Windows system.
- B.4** Describe the three main architectural layers of the Windows kernel.
- B.5** What is the job of the object manager?
- B.6** What types of services does the process manager provide?
- B.7** What is a local procedure call?
- B.8** What are the responsibilities of the I/O manager?
- B.9** What types of networking does Windows support? How does Windows implement transport protocols? Describe two networking protocols.
- B.10** How is the NTFS namespace organized?
- B.11** How does NTFS handle data structures? How does NTFS recover from a system crash? What is guaranteed after a recovery takes place?
- B.12** How does Windows allocate user memory?
- B.13** Describe some of the ways in which an application can use memory via the Win32 API.

## Further Reading

[Russinovich et al. (2017)] provides an overview of Windows 7 and considerable technical detail about system internals and components. [Brown (2000)] presents details of the security architecture of Windows.

The Microsoft Developer Network Library (<http://msdn.microsoft.com>) supplies a wealth of information on Windows and other Microsoft products, including documentation of all the published APIs.

[Iseminger (2000)] provides a good reference on the Windows Active Directory. Detailed discussions of writing programs that use the Win32 API appear in [Richter (1997)].

The source code for a 2005 WRK version of the Windows kernel, together with a collection of slides and other CRK curriculum materials, is available from [www.microsoft.com/WindowsAcademic](http://www.microsoft.com/WindowsAcademic) for use by universities.

## Bibliography

- [Brown (2000)]** K. Brown, *Programming Windows Security*, Addison-Wesley (2000).
- [Iseminger (2000)]** D. Iseminger, *Active Directory Services for Microsoft Windows 2000. Technical Reference*, Microsoft Press (2000).

[Richter (1997)] J. Richter, *Advanced Windows*, Microsoft Press (1997).

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



# BSD UNIX



This chapter was first written in 1991 and has been updated over time.

In Chapter 20, we presented an in-depth examination of the Linux operating system. In this chapter, we examine another popular UNIX version—UnixBSD. We start by presenting a brief history of the UNIX operating system. We then describe the system's user and programmer interfaces. Finally, we discuss the internal data structures and algorithms used by the FreeBSD kernel to support the user–programmer interface.

## C.1 UNIX History

The first version of UNIX was developed in 1969 by Ken Thompson of the Research Group at Bell Laboratories to use an otherwise idle PDP-7. Thompson was soon joined by Dennis Ritchie and they, with other members of the Research Group, produced the early versions of UNIX.

Ritchie had previously worked on the MULTICS project, and MULTICS had a strong influence on the newer operating system. Even the name *UNIX* is a pun on *MULTICS*. The basic organization of the file system, the idea of the command interpreter (or the shell) as a user process, the use of a separate process for each command, the original line-editing characters (# to erase the last character and @ to erase the entire line), and numerous other features came directly from MULTICS. Ideas from other operating systems, such as MIT's CTSS and the XDS-940 system, were also used.

Ritchie and Thompson worked quietly on UNIX for many years. They moved it to a PDP-11/20 for a second version; for a third version, they rewrote most of the operating system in the systems-programming language C, instead of the previously used assembly language. C was developed at Bell Laboratories to support UNIX. UNIX was also moved to larger PDP-11 models, such as the 11/45 and 11/70. Multiprogramming and other enhancements were added when it was rewritten in C and moved to systems (such as the 11/45) that had hardware support for multiprogramming.

As UNIX developed, it became widely used within Bell Laboratories and gradually spread to a few universities. The first version widely available out-

side Bell Laboratories was Version 6, released in 1976. (The version number for early UNIX systems corresponds to the edition number of the *UNIX Programmer's Manual* that was current when the distribution was made; the code and the manual were revised independently.)

In 1978, Version 7 was distributed. This UNIX system ran on the PDP-11/70 and the Interdata 8/32 and is the ancestor of most modern UNIX systems. In particular, it was soon ported to other PDP-11 models and to the VAX computer line. The version available on the VAX was known as 32V. Research has continued since then.

### **C.1.1 UNIX Support Group**

After the distribution of Version 7 in 1978, the UNIX Support Group (USG) assumed administrative control and responsibility from the Research Group for distributions of UNIX within AT&T, the parent organization for Bell Laboratories. UNIX was becoming a product, rather than simply a research tool. The Research Group continued to develop their own versions of UNIX, however, to support their internal computing. Version 8 included a facility called the **stream I/O system**, which allows flexible configuration of kernel IPC modules. It also contained RFS, a remote file system similar to Sun's NFS. The current version is Version 10, released in 1989 and available only within Bell Laboratories.

USG mainly provided support for UNIX within AT&T. The first external distribution from USG was System III, in 1982. System III incorporated features of Version 7 and 32V, as well as features of several UNIX systems developed by groups other than Research. For example, features of UNIX/RT, a real-time UNIX system, and numerous portions of the Programmer's Work Bench (PWB) software tools package were included in System III.

USG released System V in 1983; it is largely derived from System III. The divestiture of the various Bell operating companies from AT&T left AT&T in a position to market System V aggressively. USG was restructured as the UNIX System Development Laboratory (USDL), which released UNIX System V Release 2 (V.2) in 1984. UNIX System V Release 2, Version 4 (V.2.4) added a new implementation of virtual memory with copy-on-write paging and shared memory. USDL was in turn replaced by AT&T Information Systems (ATTIS), which distributed System V Release 3 (V.3) in 1987. V.3 adapts the V8 implementation of the stream I/O system and makes it available as **STREAMS**. It also includes RFS, the NFS-like remote file system mentioned earlier.

### **C.1.2 Berkeley Begins Development**

The small size, modularity, and clean design of early UNIX systems led to UNIX-based work at numerous other computer-science organizations, such as RAND, BBN, the University of Illinois, Harvard, Purdue, and DEC. The most influential UNIX development group outside of Bell Laboratories and AT&T, however, has been the University of California at Berkeley.

Bill Joy and Ozalp Babaoglu did the first Berkeley VAX UNIX work in 1978. They added virtual memory, demand paging, and page replacement to 32V to produce 3BSD UNIX. This version was the first to implement any of these facilities on a UNIX system. The large virtual memory space of 3BSD allowed the development of very large programs, such as Berkeley's own Franz LISP. The memory-management work convinced the Defense Advanced Research

Projects Agency (DARPA) to fund Berkeley for the development of a standard UNIX system for government use; 4BSD UNIX was the result.

The 4 BSD work for DARPA was guided by a steering committee that included many notable people from the UNIX and networking communities. One of the goals of this project was to provide support for the DARPA Internet networking protocols (TCP/IP). This support was provided in a general manner. It is possible in 4.2 BSD to communicate uniformly among diverse network facilities, including local-area networks (such as Ethernets and token rings) and wide-area networks (such as NSFNET). This implementation was the most important reason for the current popularity of these protocols. Many vendors of UNIX computer systems used it as the basis for their implementations, and it was even used in other operating systems. It permitted the Internet to grow from 60 connected networks in 1984 to more than 8,000 networks and an estimated 10 million users in 1993.

In addition, Berkeley adapted many features from contemporary operating systems to improve the design and implementation of UNIX. Many of the terminal line-editing functions of the TENEX (TOPS-20) operating system were provided by a new terminal driver. A new user interface (the C Shell), a new text editor (ex/vi), compilers for Pascal and LISP, and many new systems programs were written at Berkeley. For 4.2 BSD, certain efficiency improvements were inspired by the VMS operating system.

UNIX software from Berkeley was released in [Berkeley Software Distributions \(BSD\)](#). It is convenient to refer to the Berkeley VAX UNIX systems following 3 BSD as 4 BSD, but there were actually several specific releases, most notably 4.1 BSD and 4.2 BSD; 4.2 BSD, first distributed in 1983, was the culmination of the original Berkeley DARPA UNIX project. The equivalent version for PDP-11 systems was 2.9 BSD.

In 1986, 4.3 BSD was released. It was very similar to 4.2 BSD but included numerous internal changes, such as bug fixes and performance improvements. Some new facilities were also added, including support for the Xerox Network System protocols.

The next version was 4.3 BSD Tahoe, released in 1988. It included improved networking congestion control and TCP/IP performance. Disk configurations were separated from the device drivers and read off the disks themselves. Expanded time-zone support was also included. 4.3 BSD Tahoe was actually developed on and for the CCI Tahoe system (Computer Console, Inc., Power 6 computer), rather than for the usual VAX base. The corresponding PDP-11 release was 2.10.1BSD; it was distributed by the USENIX association, which also published the 4.3 BSD manuals. The 4.3.2 BSD Reno release saw the inclusion of an implementation of ISO/OSI networking.

The last Berkeley release, 4.4 BSD, was finalized in June of 1993. It included new X.25 networking support and POSIX standard compliance. It also had a radically new file system organization, with a new virtual file system interface and support for *stackable* file systems, allowing file systems to be layered on top of each other for easy inclusion of new features. An implementation of NFS was included in the release (Section 15.8), along with a new log-based file system (see Chapter 11). The 4.4 BSD virtual memory system was derived from Mach (described in Section A.13). Several other changes, such as enhanced security and improved kernel structure, were also included. With the release of version 4.4, Berkeley halted its research efforts.

### C.1.3 The Spread of UNIX

UNIX 4 BSD was the operating system of choice for the VAX from its initial release (in 1979) until the release of Ultrix, DEC's BSD implementation. Indeed, 4 BSD is still the best choice for many research and networking installations. The current set of UNIX operating systems is not limited to those from Bell Laboratories (which is currently owned by Lucent Technology) and Berkeley, however. Sun Microsystems helped popularize the BSD flavor of UNIX by shipping it on Sun workstations. As UNIX grew in popularity, it was moved to many computers and computer systems. A wide variety of UNIX and UNIX-like operating systems have been created. DEC supported its UNIX (Ultrix) on its workstations and is replacing Ultrix with another UNIX-derived operating system, OSF/1. Microsoft rewrote UNIX for the Intel 8088 family and called it XENIX, and its Windows NT operating system was heavily influenced by UNIX. IBM has UNIX (AIX) on its PCs, workstations, and mainframes. In fact, UNIX is available on almost all general-purpose computers. It runs on personal computers, workstations, minicomputers, mainframes, and supercomputers, from Apple Macintosh IIs to Cray IIs. Because of its wide availability, it is used in environments ranging from academic to military to manufacturing process control. Most of these systems are based on Version 7, System III, 4.2 BSD, or System V.

The wide popularity of UNIX with computer vendors has made UNIX the most portable of operating systems, and users can expect a UNIX environment independent of any specific computer manufacturer. But the large number of implementations of the system has led to remarkable variation in the programming and user interfaces distributed by the vendors. For true vendor independence, application-program developers need consistent interfaces. Such interfaces would allow all "UNIX" applications to run on all UNIX systems, which is certainly not the current situation. This issue has become important as UNIX has become the preferred program-development platform for applications ranging from databases to graphics and networking, and it has led to a strong market demand for UNIX standards.

Several standardization projects have been undertaken. The first was the */usr/group 1984 Standard*, sponsored by the UniForum industry user's group. Since then, many official standards bodies have continued the effort, including IEEE and ISO (the POSIX standard). The X/Open Group international consortium completed XPG3, a Common Application Environment, which subsumes the IEEE interface standard. Unfortunately, XPG3 is based on a draft of the ANSI C standard rather than the final specification, and therefore needed to be redone as XPG4. In 1989, the ANSI standards body standardized the C programming language, producing an ANSI C specification that vendors were quick to adopt.

As such projects continue, the flavors of UNIX will converge and lead to one programming interface to UNIX, allowing UNIX to become even more popular. In fact, two separate sets of powerful UNIX vendors are working on this problem: The AT&T-guided UNIX International (UI) and the Open Software Foundation (OSF) have both agreed to follow the POSIX standard. Recently, many of the vendors involved in those two groups have agreed on further standardization (the COSE agreement).

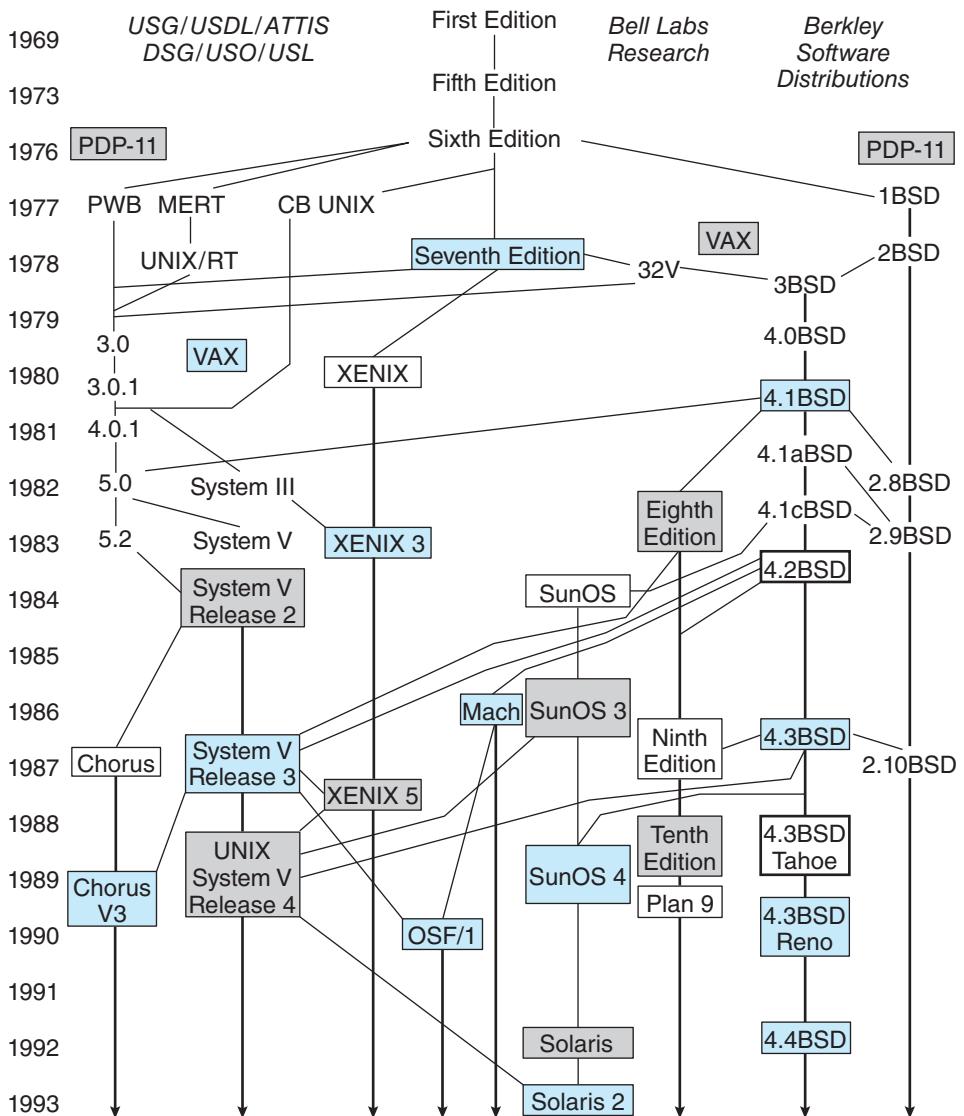


Figure C.1 History of UNIX versions up to 1993.

AT&T replaced its ATTIS group in 1989 with the UNIX Software Organization (USO), which shipped the first merged UNIX, System V Release 4. This system combines features from System V, 4.3 BSD, and Sun's SunOS, including long file names, the Berkeley file system, virtual memory management, symbolic links, multiple access groups, job control, and reliable signals; it also conforms to the published POSIX standard, POSIX.1. After USO produced SVR4, it became an independent AT&T subsidiary named Unix System Laboratories (USL); in 1993, it was purchased by Novell, Inc. Figure C.1 summarizes the relationships among the various versions of UNIX.

The UNIX system has grown from a personal project of two Bell Laboratories employees to an operating system defined by multinational standardization bodies. At the same time, UNIX is an excellent vehicle for academic study, and we believe it will remain an important part of operating-system theory and practice. For example, the Tunis operating system, the Xinu operating system, and the Minix operating system are based on the concepts of UNIX but were developed explicitly for classroom study. There is a plethora of ongoing UNIX-related research systems, including Mach, Chorus, Comandos, and Roisin. The original developers, Ritchie and Thompson, were honored in 1983 by the Association for Computing Machinery Turing Award for their work on UNIX.

#### **C.1.4    History of FreeBSD**

The specific UNIX version used in this chapter is the Intel version of FreeBSD. This system implements many interesting operating-system concepts, such as demand paging with clustering, as well as networking. The FreeBSD project began in early 1993 to produce a snapshot of 386 BSD to solve problems that could not be resolved using the existing patch mechanism. 386 BSD was derived from 4.3 BSD-Lite (Net/2) and was released in June 1992 by William Jolitz. FreeBSD (coined by David Greenman) 1.0 was released in December 1993, and FreeBSD 1.1 was released in May 1994. Both versions were based on 4.3 BSD-Lite. Legal issues between UCB and Novell required that 4.3 BSD-Lite code no longer be used, so the final 4.3 BSD-Lite release was made in July 1994 (FreeBSD 1.1.5.1).

FreeBSD was then reinvented based on 4.4BSD-Lite code, which was incomplete. FreeBSD 2.0 was released in November 1994. Later releases included 2.0.5 in June 1995, 2.1.5 in August 1996, 2.1.7.1 in February 1997, 2.2.1 in April 1997, 2.2.8 in November 1998, 3.0 in October 1998, 3.1 in February 1999, 3.2 in May 1999, 3.3 in September 1999, 3.4 in December 1999, 3.5 in June 2000, 4.0 in March 2000, 4.1 in July 2000, and 4.2 in November 2000.

The goal of the FreeBSD project is to provide software that can be used for any purpose with no strings attached. The idea is that the code will get the widest possible use and provide the most benefit. At present, it runs primarily on Intel platforms, although Alpha platforms are supported. Work is underway to port to other processor platforms as well.

## **C.2    Design Principles**

UNIX was designed to be a time-sharing system. The standard user interface (the shell) is simple and can be replaced by another, if desired. The file system is a multilevel tree, which allows users to create their own subdirectories. Each user data file is simply a sequence of bytes.

Disk files and I/O devices are treated as similarly as possible. Thus, device dependencies and peculiarities are kept in the kernel as much as possible. Even in the kernel, most of them are confined to the device drivers.

UNIX supports multiple processes. A process can easily create new processes. CPU scheduling is a simple priority algorithm. FreeBSD uses demand

paging as a mechanism to support memory-management and CPU-scheduling decisions. Swapping is used if a system is suffering from excess paging.

Because UNIX was originated by Thompson and Ritchie as a system for their own convenience, it was small enough to understand. Most of the algorithms were selected for *simplicity*, not for speed or sophistication. The intent was to have the kernel and libraries provide a small set of facilities that was sufficiently powerful to allow a person to build a more complex system if needed. UNIX's clean design has resulted in many imitations and modifications.

Although the designers of UNIX had a significant amount of knowledge about other operating systems, UNIX had no elaborate design spelled out before its implementation. This flexibility appears to have been one of the key factors in the development of the system. Some design principles were involved, however, even though they were not made explicit at the outset.

The UNIX system was designed by programmers for programmers. Thus, it has always been interactive, and facilities for program development have always been a high priority. Such facilities include the program *make* (which can be used to check which of a collection of source files for a program need to be compiled and then to do the compiling) and the *Source Code Control System (SCCS)* (which is used to keep successive versions of files available without having to store the entire contents of each step). The primary version-control system used by UNIX is the *Concurrent Versions System (CVS)* due to the large number of developers operating on and using the code.

The operating system is written mostly in C, which was developed to support UNIX, since neither Thompson nor Ritchie enjoyed programming in assembly language. The avoidance of assembly language was also necessary because of the uncertainty about the machines on which UNIX would be run. It has greatly simplified the problems of moving UNIX from one hardware system to another.

From the beginning, UNIX development systems have had all the UNIX sources available online, and the developers have used the systems under development as their primary systems. This pattern of development has greatly facilitated the discovery of deficiencies and their fixes, as well as of new possibilities and their implementations. It has also encouraged the plethora of UNIX variants existing today, but the benefits have outweighed the disadvantages. If something is broken, it can be fixed at a local site; there is no need to wait for the next release of the system. Such fixes, as well as new facilities, may be incorporated into later distributions.

The size constraints of the PDP-11 (and earlier computers used for UNIX) have forced a certain elegance. Where other systems have elaborate algorithms for dealing with pathological conditions, UNIX just does a controlled crash called *panic*. Instead of attempting to cure such conditions, UNIX tries to prevent them. Where other systems would use brute force or macro-expansion, UNIX mostly has had to develop more subtle, or at least simpler, approaches.

These early strengths of UNIX produced much of its popularity, which in turn produced new demands that challenged those strengths. UNIX was used for tasks such as networking, graphics, and real-time operation, which did not always fit into its original text-oriented model. Thus, changes were made to certain internal facilities, and new programming interfaces were added. Supporting these new facilities and others—particularly window interfaces

—required large amounts of code, radically increasing the size of the system. For instance, both networking and windowing doubled the size of the system. This pattern in turn pointed out the continued strength of UNIX—whenever a new development occurred in the industry, UNIX could usually absorb it but remain UNIX.

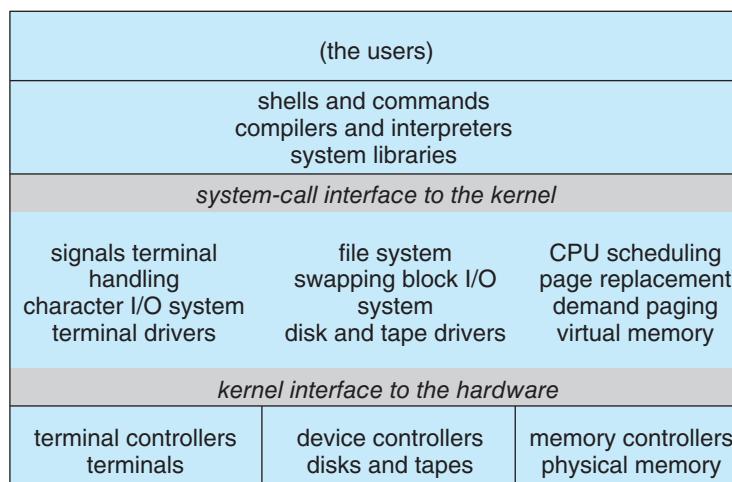
### C.3 Programmer Interface

Like most operating systems, UNIX consists of two separable parts: the kernel and the systems programs. We can view the UNIX operating system as being layered, as shown in Figure C.2. Everything below the system-call interface and above the physical hardware is the *kernel*. The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls. Systems programs use the kernel-supported system calls to provide useful functions, such as compilation and file manipulation.

System calls define the *programmer interface* to UNIX. The set of systems programs commonly available defines the *user interface*. The programmer and user interface define the context that the kernel must support.

Most systems programs are written in C, and the *UNIX Programmer's Manual* presents all system calls as C functions. A system program written in C for FreeBSD on the Pentium can generally be moved to an Alpha FreeBSD system and simply recompiled, even though the two systems are quite different. The details of system calls are known only to the compiler. This feature is a major reason for the portability of UNIX programs.

System calls for UNIX can be roughly grouped into three categories: file manipulation, process control, and information manipulation. In Chapter 2, we listed a fourth category, device manipulation, but since devices in UNIX are treated as (special) files, the same system calls support both files and devices (although there is an extra system call for setting device parameters).



**Figure C.2** 4.4BSD layer structure.

### C.3.1 File Manipulation

A *file* in UNIX is a sequence of bytes. Different programs expect various levels of structure, but the kernel does not impose a structure on files. For instance, the convention for text files is lines of ASCII characters separated by a single newline character (which is the linefeed character in ASCII), but the kernel knows nothing of this convention.

Files are organized in tree-structured *directories*. Directories are themselves files that contain information on how to find other files. A *path name* to a file is a text string that identifies a file by specifying a path through the directory structure to the file. Syntactically, it consists of individual file-name elements separated by the slash character. For example, in */usr/local/font*, the first slash indicates the root of the directory tree, called the *root directory*. The next element, *usr*, is a subdirectory of the root, *local* is a subdirectory of *usr*, and *font* is a file or directory in the directory *local*. Whether *font* is an ordinary file or a directory cannot be determined from the path-name syntax.

The UNIX file system has both *absolute path names* and *relative path names*. Absolute path names start at the root of the file system and are distinguished by a slash at the beginning of the path name; */usr/local/font* is an absolute path name. Relative path names start at the *current directory*, which is an attribute of the process accessing the path name. Thus, *local/font* indicates a file or directory named *font* in the directory *local* in the current directory, which might or might not be */usr*.

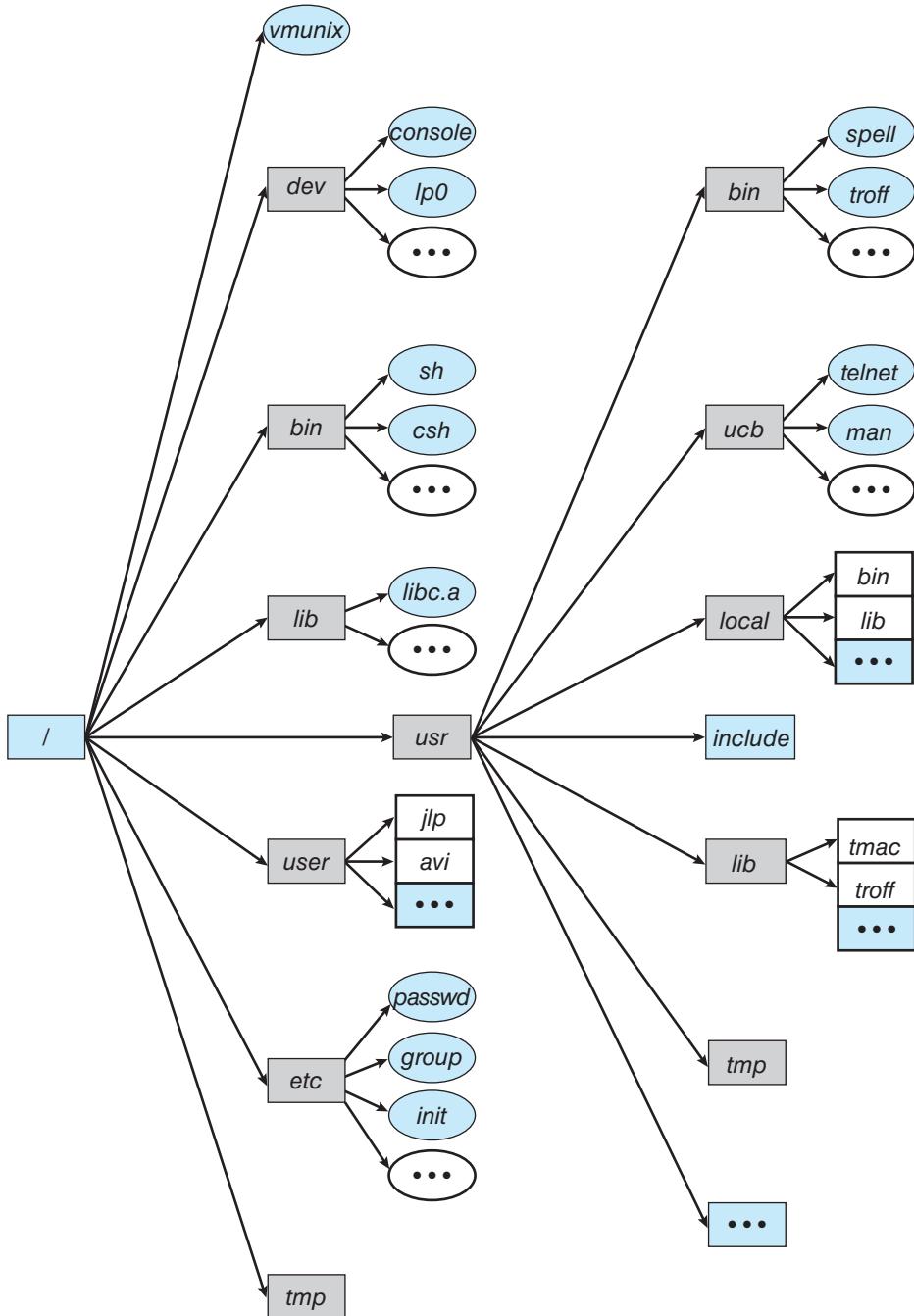
A file may be known by more than one name in one or more directories. Such multiple names are known as *links*, and all links are treated equally by the operating system. FreeBSD also supports *symbolic links*, which are files containing the path name of another file. The two kinds of links are also known as *hard links* and *soft links*. Soft (symbolic) links, unlike hard links, may point to directories and may cross file-system boundaries.

The file name “.” in a directory is a hard link to the directory itself. The file name “..” is a hard link to the parent directory. Thus, if the current directory is */user/jlp/programs*, then *../bin/wdf* refers to */user/jlp/bin/wdf*.

Hardware devices have names in the file system. These *device special files* or *special files* are known to the kernel as device interfaces, but they are nonetheless accessed by the user by much the same system calls as are other files.

Figure C.3 shows a typical UNIX file system. The root (/) normally contains a small number of directories as well as */kernel*, the binary boot image of the operating system; */dev* contains the device special files, such as */dev/console*, */dev/lp0*, */dev/mt0*, and so on; and */bin* contains the binaries of the essential UNIX systems programs. Other binaries may be in */usr/bin* (for applications systems programs, such as text formatters), */usr/compat* (for programs from other operating systems, such as Linux), or */usr/local/bin* (for systems programs written at the local site). Library files—such as the C, Pascal, and FORTRAN subroutine libraries—are kept in */lib* (or */usr/lib* or */usr/local/lib*).

The files of users themselves are stored in a separate directory for each user, typically in */usr*. Thus, the user directory for *carol* would normally be in */usr/carol*. For a large system, these directories may be further grouped to ease administration, creating a file structure with */usr/prof/avi* and */usr/staff/carol*. Administrative files and programs, such as the password file, are kept in */etc*.



**Figure C.3** Typical UNIX directory structure.

Temporary files can be put in `/tmp`, which is normally erased during system boot, or in `/usr/tmp`.

Each of these directories may have considerably more structure. For example, the font-description tables for the troff formatter for the Mergenthaler 202

typesetter are kept in `/usr/lib/troff/dev202`. All the conventions concerning the location of specific files and directories have been defined by programmers and their programs. The operating-system kernel needs only `/etc/init`, which is used to initialize terminal processes, to be operable.

System calls for basic file manipulation are `creat()`, `open()`, `read()`, `write()`, `close()`, `unlink()`, and `trunc()`. The `creat()` system call, given a path name, creates an empty file (or truncates an existing one). An existing file is opened by the `open()` system call, which takes a path name and a mode (such as read, write, or read-write) and returns a small integer, called a *file descriptor*. The file descriptor may then be passed to a `read()` or `write()` system call (along with a buffer address and the number of bytes to transfer) to perform data transfers to or from the file. A file is closed when its file descriptor is passed to the `close()` system call. The `trunc()` call reduces the length of a file to 0.

A file descriptor is an index into a small table of open files for this process. Descriptors start at 0 and seldom get higher than 6 or 7 for typical programs, depending on the maximum number of simultaneously open files.

Each `read()` or `write()` updates the current offset into the file, which is associated with the file-table entry and is used to determine the position in the file for the next `read()` or `write()`. The `lseek()` system call allows the position to be reset explicitly. It also allows the creation of sparse files (files with “holes” in them). The `dup()` and `dup2()` system calls can be used to produce a new file descriptor that is a copy of an existing one. The `fcntl()` system call can also do that and in addition can examine or set various parameters of an open file. For example, it can make each succeeding `write()` to an open file append to the end of that file. There is an additional system call, `ioctl()`, for manipulating device parameters. It can set the baud rate of a serial port or rewind a tape, for instance.

Information about the file (such as its size, protection modes, owner, and so on) can be obtained by the `stat()` system call. Several system calls allow some of this information to be changed: `rename()` (change file name), `chmod()` (change the protection mode), and `chown()` (change the owner and group). Many of these system calls have variants that apply to file descriptors instead of file names. The `link()` system call makes a hard link for an existing file, creating a new name for an existing file. A link is removed by the `unlink()` system call; if it is the last link, the file is deleted. The `symlink()` system call makes a symbolic link.

Directories are made by the `mkdir()` system call and are deleted by `rmdir()`. The current directory is changed by `cd()`.

Although the standard file calls (`open()` and others) can be used on directories, it is inadvisable to do so, since directories have an internal structure that must be preserved. Instead, another set of system calls is provided to open a directory, to step through each file entry within the directory, to close the directory, and to perform other functions; these are `opendir()`, `readdir()`, `closedir()`, and others.

### C.3.2 Process Control

A *process* is a program in execution. Processes are identified by their *process identifier*, which is an integer. A new process is created by the `fork()` system

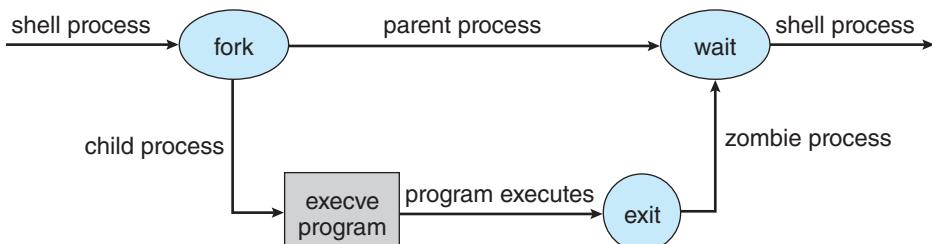
call. The new process consists of a copy of the address space of the original process (the same program and the same variables with the same values). Both processes (the parent and the child) continue execution at the instruction after the `fork()` with one difference: the return code for the `fork()` is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

Typically, the `execve()` system call is used after a `fork()` by one of the two processes to replace that process's virtual memory space with a new program. The `execve()` system call loads a binary file into memory (destroying the memory image of the program containing the `execve()` system call) and starts its execution.

A process may terminate by using the `exit()` system call, and its parent process may wait for that event by using the `wait()` system call. If the child process crashes, the system simulates the `exit()` call. The `wait()` system call provides the process ID of a terminated child so that the parent can tell which of possibly many children terminated. A second system call, `wait3()`, is similar to `wait()` but also allows the parent to collect performance statistics about the child. Between the time the child exits and the time the parent completes one of the `wait()` system calls, the child is *defunct*. A defunct process can do nothing but exists merely so that the parent can collect its status information. If the parent process of a defunct process exits before a child, the defunct process is inherited by the `init` process (which in turn waits on it) and becomes a *zombie* process. A typical use of these facilities is shown in Figure C.4.

The simplest form of communication between processes is by *pipes*. A pipe may be created before the `fork()`, and its endpoints are then set up between the `fork()` and the `execve()`. A pipe is essentially a queue of bytes between two processes. The pipe is accessed by a file descriptor, like an ordinary file. One process writes into the pipe, and the other reads from the pipe. The size of the original pipe system was fixed by the system. With FreeBSD pipes are implemented on top of the socket system, which has variable-sized buffers. Reading from an empty pipe or writing into a full pipe causes the process to be blocked until the state of the pipe changes. Special arrangements are needed for a pipe to be placed between a parent and child (so only one is reading and one is writing).

All user processes are descendants of one original process, called `init` (which has process identifier 1). Each terminal port available for interactive use has a `getty` process forked for it by `init`. The `getty` process initializes terminal line parameters and waits for a user's login name, which it passes through



**Figure C.4** A shell forks a subprocess to execute a program.

an `execve()` as an argument to a login process. The login process collects the user's password, encrypts it, and compares the result to an encrypted string taken from the file `/etc/passwd`. If the comparison is successful, the user is allowed to log in. The login process executes a *shell*, or command interpreter, after setting the numeric *user identifier* of the process to that of the user logging in. (The shell and the user identifier are found in `/etc/passwd` by the user's login name.) It is with this shell that the user ordinarily communicates for the rest of the login session. The shell itself forks subprocesses for the commands the user tells it to execute.

The user identifier is used by the kernel to determine the user's permissions for certain system calls, especially those involving file accesses. There is also a *group identifier*, which is used to provide similar privileges to a collection of users. In FreeBSD a process may be in several groups simultaneously. The login process puts the shell in all the groups permitted to the user by the files `/etc/passwd` and `/etc/group`.

Two user identifiers are used by the kernel: the effective user identifier and the real user identifier. The *effective user identifier* is used to determine file access permissions. If the file of a program being loaded by an `execve()` has the setuid bit set in its inode, the effective user identifier of the process is set to the user identifier of the owner of the file, whereas the *real user identifier* is left as it was. This scheme allows certain processes to have more than ordinary privileges while still being executable by ordinary users. The setuid idea was patented by Dennis Ritchie (U.S. Patent 4,135,240) and is one of the distinctive features of UNIX. A similar setgid bit exists for groups. A process may determine its real and effective user identifier with the `getuid()` and `geteuid()` calls, respectively. The `getgid()` and `getegid()` calls determine the process's real and effective group identifier, respectively. The rest of a process's groups may be found with the `getgroups()` system call.

### C.3.3 Signals

*Signals* are a facility for handling exceptional conditions similar to software interrupts. There are 20 different signals, each corresponding to a distinct condition. A signal may be generated by a keyboard interrupt, by an error in a process (such as a bad memory reference), or by a number of asynchronous events (such as timers or job-control signals from the shell). Almost any signal may also be generated by the `kill()` system call.

The interrupt signal, SIGINT, is used to stop a command before that command completes. It is usually produced by the ^C character (ASCII 3). As of 4.2BSD, the important keyboard characters are defined by a table for each terminal and can be redefined easily. The quit signal, SIGQUIT, is usually produced by the ^bs character (ASCII 28). The quit signal both stops the currently executing program and dumps its current memory image to a file named *core* in the current directory. The core file can be used by debuggers. SIGILL is produced by an illegal instruction and SIGSEGV by an attempt to address memory outside of the legal virtual memory space of a process.

Arrangements can be made either for most signals to be ignored (to have no effect) or for a routine in the user process (a signal handler) to be called. A signal handler may safely do one of two things before returning from catching a signal: call the `exit()` system call or modify a global variable. One signal

(the `kill` signal, number 9, `SIGKILL`) cannot be ignored or caught by a signal handler. `SIGKILL` is used, for example, to kill a runaway process that is ignoring other signals such as `SIGINT` and `SIGQUIT`.

Signals can be lost. If another signal of the same kind is sent before a previous signal has been accepted by the process to which it is directed, the first signal will be overwritten, and only the last signal will be seen by the process. In other words, a call to the signal handler tells a process that there has been at least one occurrence of the signal. Also, there is no relative priority among UNIX signals. If two different signals are sent to the same process at the same time, we cannot know which one the process will receive first.

Signals were originally intended to deal with exceptional events. As is true of most UNIX features, however, signal use has steadily expanded. 4.1BSD introduced job control, which uses signals to start and stop subprocesses on demand. This facility allows one shell to control multiple processes—starting, stopping, and backgrounding them as the user wishes. The `SIGWINCH` signal, invented by Sun Microsystems, is used for informing a process that the window in which output is being displayed has changed size. Signals are also used to deliver urgent data from network connections.

Users wanted more reliable signals and a bug fix in an inherent race condition in the old signal implementation. Thus, 4.2BSD brought with it a race-free, reliable, separately implemented signal capability. It allows individual signals to be blocked during critical sections, and it has a new system call to let a process sleep until interrupted. It is similar to hardware-interrupt functionality. This capability is now part of the POSIX standard.

### C.3.4 Process Groups

Groups of related processes frequently cooperate to accomplish a common task. For instance, processes may create, and communicate over, pipes. Such a set of processes is termed a *process group*, or a *job*. Signals may be sent to all processes in a group. A process usually inherits its process group from its parent, but the `setpgrp()` system call allows a process to change its group.

Process groups are used by the C shell to control the operation of multiple jobs. Only one process group may use a terminal device for I/O at any time. This *foreground* job has the attention of the user on that terminal, while all other nonattached jobs (*background* jobs) perform their functions without user interaction. Access to the terminal is controlled by process group signals. Each job has a *controlling terminal* (again, inherited from its parent). If the process group of the controlling terminal matches the group of a process, that process is in the foreground and is allowed to perform I/O. If a nonmatching (*background*) process attempts the same, a `SIGTTIN` or `SIGTTOU` signal is sent to its process group. This signal usually causes the process group to freeze until it is foregrounded by the user, at which point it receives a `SIGCONT` signal, indicating that the process can perform the I/O. Similarly, a `SIGSTOP` may be sent to the foreground process group to freeze it.

### C.3.5 Information Manipulation

System calls exist to set and return both an interval timer (`getitimer()` / `setitimer()`) and the current time (`gettimeofday()` / `settimeofday()`) in

microseconds. In addition, processes can ask for their process identifier (`getpid()`), their group identifier (`getgid()`), the name of the machine on which they are executing (`gethostname()`), and many other values.

### C.3.6 Library Routines

The system-call interface to UNIX is supported and augmented by a large collection of library routines and header files. The header files provide the definition of complex data structures used in system calls. In addition, a large library of functions provides additional program support.

For example, the UNIX I/O system calls provide for the reading and writing of blocks of bytes. Some applications may want to read and write only 1 byte at a time. Although possible, that would require a system call for each byte—a very high overhead. Instead, a set of standard library routines (the standard I/O package accessed through the header file `<stdio.h>`) provides another interface, which reads and writes several thousand bytes at a time using local buffers and transfers between these buffers (in user memory) when I/O is desired. Formatted I/O is also supported by the standard I/O package.

Additional library support is provided for mathematical functions, network access, data conversion, and so on. The FreeBSD kernel supports over 300 system calls; the C program library has over 300 library functions. The library functions eventually result in system calls where necessary (for example, the `getchar()` library routine will result in a `read()` system call if the file buffer is empty). However, the programmer generally does not need to distinguish between the basic set of kernel system calls and the additional functions provided by library functions.

## C.4 User Interface

Both the programmer and the user of a UNIX system deal mainly with the set of systems programs that have been written and are available for execution. These programs make the necessary system calls to support their function, but the system calls themselves are contained within the program and do not need to be obvious to the user.

The common systems programs can be grouped into several categories; most of them are file or directory oriented. For example, the systems programs to manipulate directories are `mkdir` to create a new directory, `rmdir` to remove a directory, `cd` to change the current directory to another, and `pwd` to print the absolute path name of the current (working) directory.

The `ls` program lists the names of the files in the current directory. Any of 28 options can ask that properties of the files be displayed also. For example, the `-l` option asks for a long listing showing the file name, owner, protection, date and time of creation, and size. The `cp` program creates a new file that is a copy of an existing file. The `mv` program moves a file from one place to another in the directory tree. In most cases, this move simply requires a renaming of the file. If necessary, however, the file is copied to the new location, and the old copy is deleted. A file is deleted by the `rm` program (which makes an `unlink()` system call).

To display a file on the terminal, a user can run `cat`. The `cat` program takes a list of files and concatenates them, copying the result to the standard output, commonly the terminal. On a high-speed cathode-ray tube (CRT) display, of course, the file may speed by too fast to be read. The `more` program displays the file one screen at a time, pausing until the user types a character to continue to the next screen. The `head` program displays just the first few lines of a file; `tail` shows the last few lines.

These are the basic systems programs widely used in UNIX. In addition, there are a number of editors (`ed`, `sed`, `emacs`, `vi`, and so on), compilers (C, `python`, FORTRAN, and so on), and text formatters (`troff`, `TEX`, `scribe`, and so on). There are also programs for sorting (`sort`) and comparing files (`cmp`, `diff`), looking for patterns (`grep`, `awk`), sending mail to other users (`mail`), and many other activities.

#### C.4.1 Shells and Commands

Both user-written and systems programs are normally executed by a command interpreter. The command interpreter in UNIX is a user process like any other. As noted earlier, it is called a shell—because it surrounds the kernel of the operating system. Users can write their own shells, and, in fact, several shells are in general use. The *Bourne shell*, written by Steve Bourne, is probably the most widely used—or, at least, the most widely available. The *C shell*, mostly the work of Bill Joy, a founder of Sun Microsystems, is the most popular on BSD systems. The Korn shell, by Dave Korn, has become popular because it combines the features of the Bourne shell and the C shell.

The common shells share much of their command-language syntax. UNIX is normally an interactive system. The shell indicates its readiness to accept another command by typing a prompt, and the user types a command on a single line. For instance, in the line

```
% ls -l
```

the percent sign is the usual C shell prompt, and the `ls -l` (typed by the user) is the (long) list-directory command. Commands can take arguments, which the user types after the command name on the same line, separated by white space (spaces or tabs).

Although a few commands are built into the shells (such as `cd`), a typical command is an executable binary object file. A list of several directories, the *search path*, is kept by the shell. For each command, each of the directories in the search path is searched, in order, for a file of the same name. If a file is found, it is loaded and executed. The search path can be set by the user. The directories `/bin` and `/usr/bin` are almost always in the search path, and a typical search path on a FreeBSD system might be

```
( . /usr/avi/bin /usr/local/bin /bin /usr/bin )
```

The `ls` command's object file is `/bin/ls`, and the shell itself is `/bin/sh` (the Bourne shell) or `/bin/csh` (the C shell).

Execution of a command is done by a `fork()` system call followed by an `execve()` of the object file. The shell usually then does a `wait()` to suspend

its own execution until the command completes (Figure C.4). There is a simple syntax (an ampersand [&] at the end of the command line) to indicate that the shell should *not* wait for the completion of the command. A command left running in this manner while the shell continues to interpret further commands is said to be a *background* command, or to be running in the background. Processes for which the shell *does* wait are said to run in the *foreground*.

The C shell in FreeBSD systems provides a facility called *job control* (partially implemented in the kernel), as mentioned previously. Job control allows processes to be moved between the foreground and the background. The processes can be stopped and restarted on various conditions, such as a background job wanting input from the user's terminal. This scheme allows most of the control of processes provided by windowing or layering interfaces but requires no special hardware. Job control is also useful in window systems, such as the X Window System developed at MIT. Each window is treated as a terminal, allowing multiple processes to be in the foreground (one per window) at any one time. Of course, background processes may exist on any of the windows. The Korn shell also supports job control, and job control (and process groups) will likely be standard in future versions of UNIX.

### C.4.2 Standard I/O

Processes can open files as they like, but most processes expect three file descriptors (numbers 0, 1, and 2) to be open when they start. These file descriptors are inherited across the `fork()` (and possibly the `execve()`) that created the process. They are known as *standard input* (0), *standard output* (1), and *standard error* (2). All three are frequently open to the user's terminal. Thus, the program can read what the user types by reading standard input, and the program can send output to the user's screen by writing to standard output. The standard-error file descriptor is also open for writing and is used for error output; standard output is used for ordinary output. Most programs can also accept a file (rather than a terminal) for standard input and standard output. The program does not care where its input is coming from and where its output is going. This is one of the elegant design features of UNIX.

The common shells have a simple syntax for changing what files are open for the standard I/O streams of a process. Changing a standard file is called *I/O redirection*. The syntax for I/O redirection is shown in Figure C.5. In this

command	meaning of command
<code>% ls &gt; filea</code>	direct output of <code>ls</code> to file <code>filea</code>
<code>% pr &lt; filea &gt; fileb</code>	input from <code>filea</code> and output to <code>fileb</code>
<code>% lpr &lt; fileb</code>	input from <code>fileb</code>
<code>% % make program &gt; &amp; errs</code>	save both standard output and standard error in a file

Figure C.5 Standard /io/ redirection.

example, the `ls` command produces a listing of the names of files in the current directory, the `pr` command formats that list into pages suitable for a printer, and the `lpr` command spools the formatted output to a printer, such as `/dev/lp0`. The subsequent command forces all output and all error messages to be redirected to a file. Without the ampersand, error messages appear on the terminal.

### C.4.3 Pipelines, Filters, and Shell Scripts

The first three commands of Figure C.5 could have been coalesced into the one command

```
% ls | pr | lpr
```

Each vertical bar tells the shell to arrange for the output of the preceding command to be passed as input to the following command. A **pipe** is used to carry the data from one process to the other. One process writes into one end of the pipe, and another process reads from the other end. In the example, the write end of one pipe would be set up by the shell to be the standard output of `ls`, and the read end of the pipe would be the standard input of `pr`. Another pipe would be between `pr` and `lpr`.

A command such as `pr` that passes its standard input to its standard output, performing some processing on it, is called a *filter*. Many UNIX commands can be used as filters. Complicated functions can be pieced together as pipelines of common commands. Also, common functions, such as output formatting, do not need to be built into numerous commands, because the output of almost any program can be piped through `pr` (or some other appropriate filter).

Both of the common UNIX shells are also programming languages, with shell variables and the usual higher-level programming-language control constructs (loops, conditionals). The execution of a command is analogous to a subroutine call. A file of shell commands, a *shell script*, can be executed like any other command, with the appropriate shell being invoked automatically to read it. *Shell programming* thus can be used to combine ordinary programs conveniently for sophisticated applications without the need for any programming in conventional languages.

This external user view is commonly thought of as the definition of UNIX, yet it is the most easily changed definition. Writing a new shell with a quite different syntax and semantics would greatly change the user view while not changing the kernel or even the programmer interface. Several menu-driven and iconic interfaces for UNIX exist, and the X Window System is rapidly becoming a standard. The heart of UNIX is, of course, the kernel. This kernel is much more difficult to change than is the user interface, because all programs depend on the system calls that it provides to remain consistent. Of course, new system calls can be added to increase functionality, but programs must then be modified to use the new calls.

## C.5 Process Management

A major design problem for operating systems is the representation of processes. One substantial difference between UNIX and many other systems is the ease with which multiple processes can be created and manipulated. These

processes are represented in UNIX by various control blocks. No system control blocks are accessible in the virtual address space of a user process; control blocks associated with a process are stored in the kernel. The kernel uses the information in these control blocks for process control and CPU scheduling.

### C.5.1 Process Control Blocks

The most basic data structure associated with processes is the *process structure*. A process structure contains everything that the system needs to know about a process when the process is swapped out, such as its unique process identifier, scheduling information (for example, the priority of the process), and pointers to other control blocks. There is an array of process structures whose length is defined at system-linking time. The process structures of ready processes are kept linked together by the scheduler in a doubly linked list (the ready queue), and there are pointers from each process structure to the process's parent, to its youngest living child, and to various other relatives of interest, such as a list of processes sharing the same program code (text).

The *virtual address space* of a user process is divided into text (program code), data, and stack segments. The data and stack segments are always in the same address space, but they may grow separately, and usually in opposite directions. Most frequently, the stack grows down as the data grow up toward it. The text segment is sometimes (as on an Intel 8086 with separate instruction and data space) in an address space different from the data and stack, and it is usually read-only. The debugger puts a text segment in read-write mode to allow insertion of breakpoints.

Every process with sharable text (almost all, under FreeBSD) has a pointer from its process structure to a *text structure*. The text structure records how many processes are using the text segment, including a pointer into a list of their process structures, and where the page table for the text segment can be found on disk when it is swapped. The text structure itself is always resident in main memory. An array of such structures is allocated at system link time. The text, data, and stack segments for the processes may be swapped. When the segments are swapped in, they are paged.

The *page tables* record information on the mapping from the process's virtual memory to physical memory. The process structure contains pointers to the page table, for use when the process is resident in main memory, or the address of the process on the swap device, when the process is swapped. There is no special separate page table for a shared text segment; every process sharing the text segment has entries for its pages in the process's page table.

Information about the process needed only when the process is resident (that is, not swapped out) is kept in the *user structure* (or *u structure*), rather than in the process structure. This structure is mapped read-only into user virtual address space, so user processes can read its contents. It is writable by the kernel. The user structure contains a copy of the process control block, or PCB, which is kept here for saving the process's general registers, stack pointer, program counter, and page-table base registers when the process is not running. There is space to keep system-call parameters and return values. All user and group identifiers associated with the process (not just the effective user identifier kept in the process structure) are kept here. Signals, timers, and quotas have data structures here. Of more obvious relevance to the ordinary

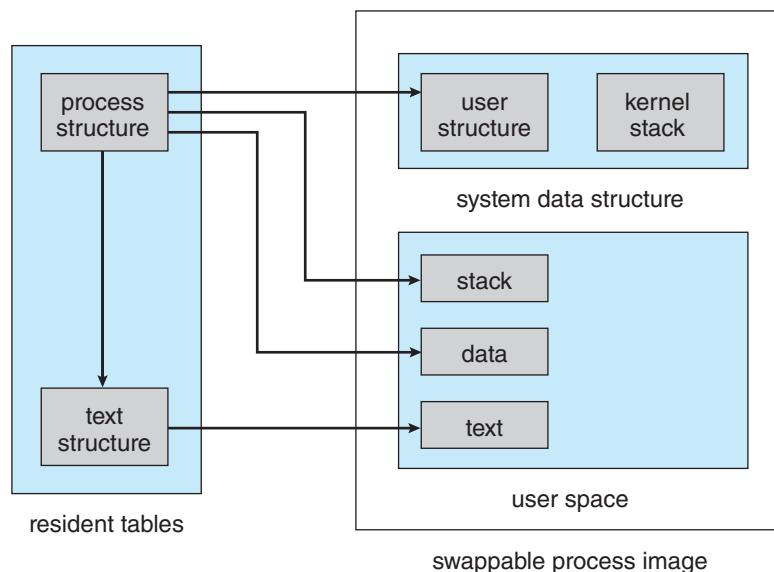
user, the current directory and the table of open files are maintained in the user structure.

Every process has both a user and a system mode. Most ordinary work is done in *user mode*, but when a system call is made, it is performed in *system mode*. The system and user phases of a process never execute simultaneously. When a process is executing in system mode, a *kernel stack* for that process is used, rather than the user stack belonging to that process. The kernel stack for the process immediately follows the user structure. The kernel stack and the user structure together compose the *system data segment* for the process. The kernel has its own stack for use when it is not doing work on behalf of a process (for instance, for interrupt handling).

Figure C.6 illustrates how the process structure is used to find the various parts of a process.

The `fork()` system call allocates a new process structure (with a new process identifier) for the child process and copies the user structure. There is ordinarily no need for a new text structure, as the processes share their text. The appropriate counters and lists are merely updated. A new page table is constructed, and new main memory is allocated for the data and stack segments of the child process. The copying of the user structure preserves open file descriptors, user and group identifiers, signal handling, and most similar properties of a process.

The `vfork()` system call does *not* copy the data and stack to the new process; rather, the new process simply shares the page table of the old one. A new user structure and a new process structure are still created. A common use of this system call occurs when a shell executes a command and waits for its completion. The parent process uses `vfork()` to produce the child process. Because the child process wishes to use an `execve()` immediately to change its virtual address space completely, there is no need for a complete copy of the



**Figure C.6** Finding parts of a process using the process structure.

parent process. Such data structures as are necessary for manipulating pipes may be kept in registers between the `vfork()` and the `execve()`. Files may be closed in one process without affecting the other process, since the kernel data structures involved depend on the user structure, which is not shared. The parent is suspended when it calls `vfork()` until the child either calls `execve()` or terminates, so that the parent will not change memory that the child needs.

When the parent process is large, `vfork()` can produce substantial savings in system CPU time. However, it is a fairly dangerous system call, since any memory change occurs in both processes until the `execve()` occurs. An alternative is to share all pages by duplicating the page table but to mark the entries of both page tables as *copy-on-write*. The hardware protection bits are set to trap any attempt to write in these shared pages. If such a trap occurs, a new frame is allocated, and the shared page is copied to the new frame. The page tables are adjusted to show that this page is no longer shared (and therefore no longer needs to be write-protected), and execution can resume.

An `execve()` system call creates no new process or user structure. Rather, the text and data of the process are replaced. Open files are preserved (although there is a way to specify that certain file descriptors are to be closed on an `execve()`). Most signal-handling properties are preserved, but arrangements to call a specific user routine on a signal are canceled, for obvious reasons. The process identifier and most other properties of the process are unchanged.

### C.5.2 CPU Scheduling

*CPU scheduling* in UNIX is designed to benefit interactive processes. Processes are given small CPU time slices by a priority algorithm that reduces to round-robin scheduling for CPU-bound jobs.

Every process has a *scheduling priority* associated with it; larger numbers indicate lower priority. Processes doing disk I/O or other important tasks have priorities less than “pzero” and cannot be killed by signals. Ordinary user processes have positive priorities and thus are less likely to be run than is any system process, although user processes can set precedence over one another through the `nice` command.

The more CPU time a process accumulates, the lower (more positive) its priority becomes, and vice versa. This negative feedback in CPU scheduling makes it difficult for a single process to take all the CPU time. Process aging is employed to prevent starvation.

Older UNIX systems used a 1-second quantum for the round-robin scheduling. FreeBSD reschedules processes every 0.1 second and recomputes priorities every second. The round-robin scheduling is accomplished by the *timeout* mechanism, which tells the clock interrupt driver to call a kernel subroutine after a specified interval. The subroutine to be called in this case causes the rescheduling and then resubmits a timeout to call itself again. The priority recomputation is also timed by a subroutine that resubmits a timeout for itself.

There is no preemption of one process by another in the kernel. A process may relinquish the CPU because it is waiting for I/O or because its time slice has expired. When a process chooses to relinquish the CPU, it goes to *sleep* on an *event*. The kernel primitive used for this purpose is called `sleep()` (not to be confused with the user-level library routine of the same name). `Sleep()` takes an argument that is, by convention, the address of a kernel data

structure related to an event for which a process is waiting. When the event occurs, the system process that knows about it calls `wakeup()` with the address corresponding to the event, and *all* processes that had done a sleep on the same address are put in the ready queue to be run.

For example, a process waiting for disk I/O to complete will sleep on the address of the buffer header corresponding to the data being transferred. When the interrupt routine for the disk driver notes that the transfer is complete, it calls `wakeup()` on the buffer header. The interrupt uses the kernel stack for whatever process happened to be running at the time, and the `wakeup()` is done from that system process.

The process that actually does run is chosen by the scheduler. `Sleep()` takes a second argument, which is the scheduling priority to be used for this purpose. This priority argument, if less than “pzero,” also prevents the process from being awakened prematurely by some exceptional event, such as a signal.

When a signal is generated, it is left pending until the system half of the affected process next runs. This event usually happens soon, since the signal normally causes the process to be awakened if the process has been waiting for some other condition.

No memory is associated with events. The caller of the routine that does a `sleep()` on an event must be prepared to deal with a premature return, including the possibility that the reason for waiting has vanished.

*Race conditions* are involved in the event mechanism. If a process decides (because of checking a flag in memory, for instance) to sleep on an event, and the event occurs before the process can execute the primitive that does the `sleep()` on the event, the process sleeping may then sleep forever. We prevent this situation by raising the hardware processor priority during the critical section so that no interrupts can occur. Thus, only the process desiring the event can run until it is sleeping. Hardware processor priority is used in this manner to protect critical regions throughout the kernel and is the greatest obstacle to porting UNIX to multiple-processor machines. However, this problem has not prevented such porting from being done repeatedly.

Many processes, such as text editors, are I/O bound and are, in general, scheduled mainly on the basis of waiting for I/O. Experience suggests that the UNIX scheduler performs best with I/O-bound jobs, as can be observed when several CPU-bound jobs, such as text formatters or language interpreters, are running.

What has been referred to here as *CPU scheduling* corresponds closely to the *short-term scheduling* of Chapter 3. However, the negative-feedback property of the priority scheme provides some long-term scheduling in that it largely determines the long-term *job mix*. Medium-term scheduling is done by the swapping mechanism described in Section C.6.

## C.6 **Memory Management**

Much of UNIX’s early development was done on a PDP-11. The PDP-11 has only eight segments in its virtual address space, and the size of each is at most 8,192 bytes. The larger machines, such as the PDP-11/70, allow separate instruction and address spaces, effectively doubling the address space and number of segments, but this address space is still relatively small. In addition, the kernel

was even more severely constrained due to dedication of one data segment to interrupt vectors, another to point at the per-process system data segment, and yet another for the UNIBUS (system I/O bus) registers. Further, on the smaller PDP-11s, total physical memory was limited to 256 KB. The total memory resources were insufficient to justify or support complex memory-management algorithms. Thus, UNIX swapped entire process memory images.

Berkeley introduced paging to UNIX with 3BSD. VAX 4.2BSD is a demand-paged virtual memory system. Paging eliminates external fragmentation of memory. (Internal fragmentation still occurs, but it is negligible with a reasonably small page size.) Because paging allows execution with only parts of each process in memory, more jobs can be kept in main memory, and swapping can be kept to a minimum. *Demand paging* is done in a straightforward manner. When a process needs a page and the page is not there, a page fault to the kernel occurs, a frame of main memory is allocated, and the proper disk page is read into the frame.

There are a few optimizations. If the page needed is still in the page table for the process but has been marked invalid by the page-replacement process, it can be marked valid and used without any I/O transfer. Pages can similarly be retrieved from the list of free frames. When most processes are started, many of their pages are prepaged and are put on the free list for recovery by this mechanism. Arrangements can also be made for a process to have no prepaging on startup. That is seldom done, however, because it results in more page-fault overhead, being closer to pure demand paging. FreeBSD implements page coloring with paging queues. The queues are arranged according to the size of the processor's L1 and L2 caches. When a new page needs to be allocated, FreeBSD tries to get one that is optimally aligned for the cache. If the page has to be fetched from disk, it must be locked in memory for the duration of the transfer. This locking ensures that the page will not be selected for page replacement. Once the page is fetched and mapped properly, it must remain locked if raw physical I/O is being done on it.

The page-replacement algorithm is more interesting. 4.2BSD uses a modification of the *second-chance (clock) algorithm* described in Section 10.4.5. The map of all nonkernel main memory (the *core map* or *cmap*) is swept linearly and repeatedly by a software *clock hand*. When the clock hand reaches a given frame, if the frame is marked as being in use by some software condition (for example, if physical I/O is in progress using it) or if the frame is already free, the frame is left untouched, and the clock hand sweeps to the next frame. Otherwise, the corresponding text or process page-table entry for this frame is located. If the entry is already invalid, the frame is added to the free list. Otherwise, the page-table entry is made invalid but reclaimable (that is, if it has not been paged out by the next time it is wanted, it can just be made valid again).

BSD Tahoe added support for systems that implement the reference bit. On such systems, one pass of the clock hand turns the reference bit off, and a second pass places those pages whose reference bits remain off onto the free list for replacement. Of course, if the page is dirty, it must first be written to disk before being added to the free list. Pageouts are done in clusters to improve performance.

There are checks to make sure that the number of valid data pages for a process does not fall too low and to keep the paging device from being flooded

with requests. There is also a mechanism by which a process can limit the amount of main memory it uses.

The LRU clock-hand scheme is implemented in the pagedaemon, which is process 2. (Remember that the swapper is process 0 and init is process 1.) This process spends most of its time sleeping, but a check is done several times per second (scheduled by a timeout) to see if action is necessary. If it is, process 2 is awakened. Whenever the number of free frames falls below a threshold, `lotsfree`, the pagedaemon is awakened. Thus, if there is always a large amount of free memory, the pagedaemon imposes no load on the system, because it never runs.

The sweep of the clock hand each time the pagedaemon process is awakened (that is, the number of frames scanned, which is usually more than the number paged out) is determined both by the number of frames lacking to reach `lotsfree` and by the number of frames that the scheduler has determined are needed for various reasons (the more frames needed, the longer the sweep). If the number of frames free rises to `lotsfree` before the expected sweep is completed, the hand stops, and the pagedaemon process sleeps. The parameters that control the range of the clock-hand sweep are determined at system startup according to the amount of main memory, such that pagedaemon does not use more than 10 percent of all CPU time.

If the scheduler decides that the paging system is overloaded, processes will be swapped out whole until the overload is relieved. This swapping usually happens only if several conditions are met: load average is high; free memory has fallen below a low limit, `minfree`; and the average memory available over recent time is less than a desirable amount, `desfree`, where  $\text{lotsfree} > \text{desfree} > \text{minfree}$ . In other words, only a chronic shortage of memory with several processes trying to run will cause swapping, and even then free memory has to be extremely low at the moment. (An excessive paging rate or a need for memory by the kernel itself may also enter into the calculations, in rare cases.) Processes may be swapped by the scheduler, of course, for other reasons (such as simply because they have not run for a long time).

The parameter `lotsfree` is usually one-quarter of the memory in the map that the clock hand sweeps, and `desfree` and `minfree` are usually the same across different systems but are limited to fractions of available memory. FreeBSD dynamically adjusts its paging queues so these virtual memory parameters will rarely need to be adjusted. Minfree pages must be kept free in order to supply any pages that might be needed at interrupt time.

Every process's text segment is, by default, shared and read-only. This scheme is practical with paging, because there is no external fragmentation, and the swap space gained by sharing more than offsets the negligible amount of overhead involved, as the kernel virtual space is large.

CPU scheduling, memory swapping, and paging interact. The lower the priority of a process, the more likely that its pages will be paged out and the more likely that it will be swapped in its entirety. The age preferences in choosing processes to swap guard against thrashing, but paging does so more effectively. Ideally, processes will not be swapped out unless they are idle, because each process will need only a small working set of pages in main memory at any one time, and the pagedaemon will reclaim unused pages for use by other processes.

The amount of memory the process will need is some fraction of that process's total virtual size—up to one-half if that process has been swapped out for a long time.

## C.7 File System

The UNIX file system supports two main objects: files and directories. Directories are just files with a special format, so the representation of a file is the basic UNIX concept.

### C.7.1 Blocks and Fragments

Most of the file system is taken up by *data blocks*, which contain whatever the users have put in their files. Let's consider how these data blocks are stored on the disk.

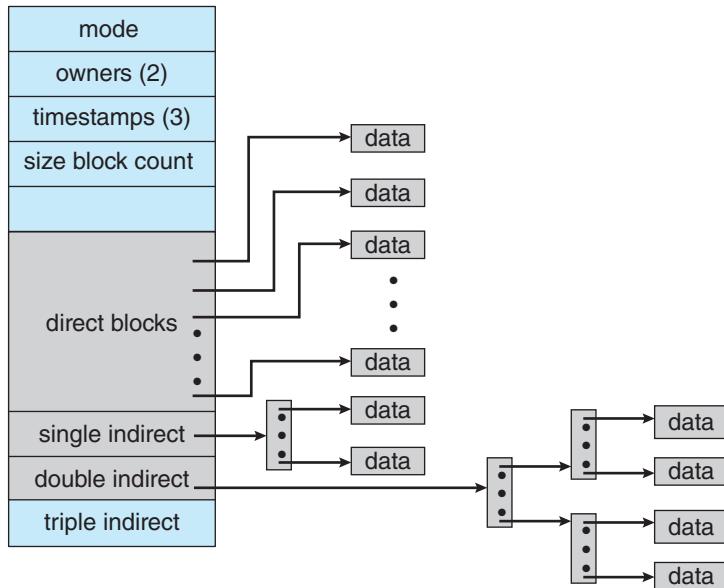
The hardware disk sector is usually 512 bytes. A block size larger than 512 bytes is desirable for speed. However, because UNIX file systems usually contain a very large number of small files, much larger blocks would cause excessive internal fragmentation. That is why the earlier 4.1BSD file system was limited to a 1,024-byte (1-KB) block. The 4.2BSD solution is to use *two* block sizes for files that have no indirect blocks. All the blocks of a file are of a large size (such as 8 KB) except the last. The last block is an appropriate multiple of a smaller fragment size (for example, 1,024 KB) to fill out the file. Thus, a file of size 18,000 bytes would have two 8-KB blocks and one 2-KB fragment (which would not be filled completely).

The block and fragment sizes are set during file-system creation according to the intended use of the file system. If many small files are expected, the fragment size should be small; if repeated transfers of large files are expected, the basic block size should be large. Implementation details force a maximum block-to-fragment ratio of 8:1 and a minimum block size of 4 KB, so typical choices are 4,096:512 for the former case and 8,192:1,024 for the latter.

Suppose data are written to a file in transfer sizes of 1-KB bytes, and the block and fragment sizes of the file system are 4 KB and 512 bytes. The file system will allocate a 1-KB fragment to contain the data from the first transfer. The next transfer will cause a new 2-KB fragment to be allocated. The data from the original fragment must be copied into this new fragment, followed by the second 1-KB transfer. The allocation routines attempt to find the required space on the disk immediately following the existing fragment so that no copying is necessary. If they cannot do so, up to seven copies may be required before the fragment becomes a block. Provisions have been made for programs to discover the block size for a file so that transfers of that size can be made, to avoid fragment recopying.

### C.7.2 Inodes

A file is represented by an *inode*, which is a record that stores most of the information about a specific file on the disk. (See Figure C.7.) The name *inode* (pronounced *EYE node*) is derived from “index node” and was originally spelled “i-node”; the hyphen fell out of use over the years. The term is sometimes spelled “I node.”



**Figure C.7** The UNIX inode.

The inode contains the user and group identifiers of the file, the times of the last file modification and access, a count of the number of hard links (directory entries) to the file, and the type of the file (plain file, directory, symbolic link, character device, block device, or socket). In addition, the inode contains 15 pointers to the disk blocks containing the data contents of the file. The first 12 of these pointers point to *direct blocks*. That is, they contain addresses of blocks that contain data of the file. Thus, the data for small files (no more than 12 blocks) can be referenced immediately, because a copy of the inode is kept in main memory while a file is open. If the block size is 4 KB, then up to 48 KB of data can be accessed directly from the inode.

The next three pointers in the inode point to *indirect blocks*. If the file is large enough to use indirect blocks, each of the indirect blocks is of the major block size; the fragment size applies only to data blocks. The first indirect block pointer is the address of a *single indirect block*. The single indirect block is an index block containing not data but the addresses of blocks that do contain data. Then, there is a *double-indirect-block pointer*, the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks. The last pointer would contain the address of a *triple indirect block*; however, there is no need for it.

The minimum block size for a file system in 4.2BSD is 4 KB, so files with as many as  $2^{32}$  bytes will use only double, not triple, indirection. That is, since each block pointer takes 4 bytes, we have 49,152 bytes accessible in direct blocks, 4,194,304 bytes accessible by a single indirection, and 4,294,967,296 bytes reachable through double indirection, for a total of 4,299,210,752 bytes, which is larger than  $2^{32}$  bytes. The number  $2^{32}$  is significant because the file offset in the file structure in main memory is kept in a 32-bit word. Files therefore cannot be larger than  $2^{32}$  bytes. Since file pointers are signed integers

(for seeking backward and forward in a file), the actual maximum file size is  $2^{32-1}$  bytes. Two gigabytes is large enough for most purposes.

### C.7.3 Directories

Plain files are not distinguished from directories at this level of implementation. Directory contents are kept in data blocks, and directories are represented by an inode in the same way as plain files. Only the inode type field distinguishes between plain files and directories. Plain files are not assumed to have a structure, whereas directories have a specific structure. In Version 7, file names were limited to 14 characters, so directories were a list of 16-byte entries: 2 bytes for an inode number and 14 bytes for a file name.

In FreeBSD file names are of variable length, up to 255 bytes, so directory entries are also of variable length. Each entry contains first the length of the entry, then the file name and the inode number. This variable-length entry makes the directory management and search routines more complex, but it allows users to choose much more meaningful names for their files and directories. The first two names in every directory are “.” and “..”. New directory entries are added to the directory in the first space available, generally after the existing files. A linear search is used.

The user refers to a file by a path name, whereas the file system uses the inode as its definition of a file. Thus, the kernel has to map the supplied user path name to an inode. The directories are used for this mapping.

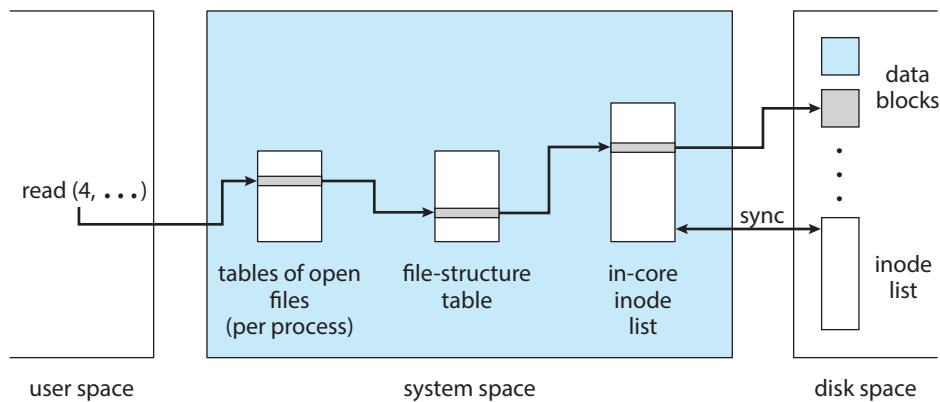
First, a starting directory is determined. As mentioned earlier, if the first character of the path name is “/,” the starting directory is the root directory. If the path name starts with any character other than a slash, the starting directory is the current directory of the current process. The starting directory is checked for proper file type and access permissions, and an error is returned if necessary. The inode of the starting directory is always available.

The next element of the path name, up to the next “/” or to the end of the path name, is a file name. The starting directory is searched for this name, and an error is returned if the name is not found. If the path name has yet another element, the current inode must refer to a directory; an error is returned if it does not or if access is denied. This directory is searched in the same way as the previous one. This process continues until the end of the path name is reached and the desired inode is returned. This step-by-step process is needed because at any directory a mount point (or symbolic link, as discussed below) may be encountered, causing the translation to move to a different directory structure for continuation.

Hard links are simply directory entries like any other. We handle symbolic links for the most part by starting the search over with the path name taken from the contents of the symbolic link. We prevent infinite loops by counting the number of symbolic links encountered during a path-name search and returning an error when a limit (eight) is exceeded.

Nondisk files (such as devices) do not have data blocks allocated on the disk. The kernel notices these file types (as indicated in the inode) and calls appropriate drivers to handle I/O for them.

Once the inode is found by, for instance, the `open()` system call, a *file structure* is allocated to point to the inode. The file descriptor given to the user refers to this file structure. FreeBSD has a *directory name cache* to hold



**Figure C.8** File-system control blocks.

recent directory-to-inode translations, which greatly increases file-system performance.

#### C.7.4 Mapping a File Descriptor to an Inode

A system call that refers to an open file indicates the file by passing a file descriptor as an argument. The file descriptor is used by the kernel to index a table of open files for the current process. Each entry in the table contains a pointer to a file structure. This file structure in turn points to the inode; see Figure C.8. The open file table has a fixed length, which is settable only at boot time. Therefore, there is a fixed limit on the number of concurrently open files in a system.

The `read()` and `write()` system calls do not take a position in the file as an argument. Rather, the kernel keeps a *file offset*, which is updated by an appropriate amount after each `read()` or `write()` according to the number of data actually transferred. The offset can be set directly by the `lseek()` system call. If the file descriptor indexed an array of inode pointers instead of file pointers, this offset would have to be kept in the inode. Because more than one process may open the same file, and each such process needs its own offset for the file, keeping the offset in the inode is inappropriate. Thus, the file structure is used to contain the offset. File structures are inherited by the child process after a `fork()`, so several processes may share the same offset location for a file.

The *inode structure* pointed to by the file structure is an in-core copy of the inode on the disk. The in-core inode has a few extra fields, such as a reference count of how many file structures are pointing at it, and the file structure has a similar reference count for how many file descriptors refer to it. When a count becomes zero, the entry is no longer needed and may be reclaimed and reused.

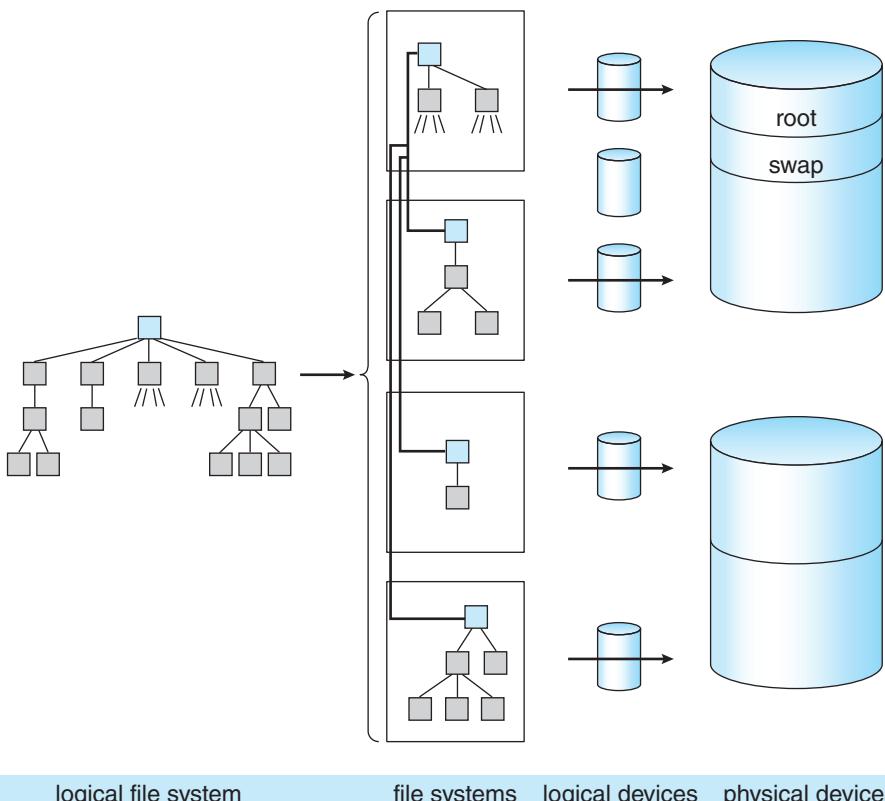
#### C.7.5 Disk Structures

The file system that the user sees is supported by data on a mass storage device—usually, a disk. The user ordinarily knows of only one file system, but this one logical file system may actually consist of several *physical* file systems, each on a different device. Because device characteristics differ, each separate

hardware device defines its own physical file system. In fact, we generally want to partition large physical devices, such as disks, into multiple *logical* devices. Each logical device defines a physical file system. Figure C.9 illustrates how a directory structure is partitioned into file systems, which are mapped onto logical devices, which are partitions of physical devices. The sizes and locations of these partitions were coded into device drivers in earlier systems, but they are maintained on the disk by FreeBSD.

Partitioning a physical device into multiple file systems has several benefits. Different file systems can support different uses. Although most partitions will be used by the file system, at least one will be needed as a swap area for the virtual memory software. Reliability is improved, because software damage is generally limited to only one file system. We can improve efficiency by varying the file-system parameters (such as the block and fragment sizes) for each partition. Also, having separate file systems prevents one program from using all available space for a large file, because files cannot be split across file systems. Finally, disk backups are done per partition, and it is faster to search a backup tape for a file if the partition is smaller. Restoring the full partition from tape is also faster.

The number of file systems on a drive varies according to the size of the disk and the purpose of the computer system as a whole. One file system, the



**Figure C.9** Mapping of a logical file system to physical devices.

*root file system*, is always available. Other file systems may be *mounted*—that is, integrated into the directory hierarchy of the root file system.

A bit in the inode structure indicates that the inode has a file system mounted on it. A reference to this file causes the *mount table* to be searched to find the device number of the mounted device. The device number is used to find the inode of the root directory of the mounted file system, and that inode is used. Conversely, if a path-name element is “..” and the directory being searched is the root directory of a file system that is mounted, the mount table is searched to find the inode it is mounted on, and that inode is used.

Each file system is a separate system resource and represents a set of files. The first sector on the logical device is the *boot block*, possibly containing a primary bootstrap program, which may be used to call a secondary bootstrap program residing in the next 7.5 KB. A system needs only one partition containing boot-block data, but the system manager may install duplicates via privileged programs, to allow booting when the primary copy is damaged. The *superblock* contains static parameters of the file system. These parameters include the total size of the file system, the block and fragment sizes of the data blocks, and assorted parameters that affect allocation policies.

### C.7.6 Implementations

The user interface to the file system is simple and well defined, allowing the implementation of the file system itself to be changed without significant effect on the user. The file system was changed between Version 6 and Version 7 of 3BSD, and again between Version 7 and 4BSD. For Version 7, the size of inodes doubled, the maximum file and file-system sizes increased, and the details of free-list handling and superblock information changed. At that time also, `seek()` (with a 16-bit offset) became `lseek()` (with a 32-bit offset), to allow specification of offsets in larger files, but few other changes were visible outside the kernel.

In 4.0BSD, the size of blocks used in the file system was increased from 512 bytes to 1,024 bytes. Although this increased size produced increased internal fragmentation on the disk, it doubled throughput, due mainly to the greater number of data accessed on each disk transfer. This idea was later adopted by System V, along with a number of other ideas, device drivers, and programs.

4.2BSD added the Berkeley Fast File System, which increased speed and was accompanied by new features. Symbolic links required new system calls. Long file names necessitated new directory system calls to traverse the now-complex internal directory structure. Finally, `truncate()` calls were added. The Fast File System was a success and is now found in most implementations of UNIX. Its performance is made possible by its layout and allocation policies, which we discuss next. In Section 14.4.4, we discussed changes made in SunOS to increase disk throughput further.

### C.7.7 Layout and Allocation Policies

The kernel uses a *<logical device number, inode number>* pair to identify a file. The logical device number defines the file system involved. The inodes in the file system are numbered in sequence. In the Version 7 file system, all inodes are in an array immediately following a single superblock at the beginning of

the logical device, with the data blocks following the inodes. The inode number is effectively just an index into this array.

With the Version 7 file system, a block of a file can be anywhere on the disk between the end of the inode array and the end of the file system. Free blocks are kept in a linked list in the superblock. Blocks are pushed onto the front of the free list and are removed from the front as needed to serve new files or to extend existing files. Thus, the blocks of a file may be arbitrarily far from both the inode and one another. Furthermore, the more a file system of this kind is used, the more disorganized the blocks in a file become. We can reverse this process only by reinitializing and restoring the entire file system, which is not a convenient task to perform. This process was described in Section 14.7.4.

Another difficulty is that the reliability of the file system is suspect. For speed, the superblock of each mounted file system is kept in memory. Keeping the superblock in memory allows the kernel to access a superblock quickly, especially for using the free list. Every 30 seconds, the superblock is written to the disk, to keep the in-core and disk copies synchronized (by the update program, using the `sync()` system call). However, system bugs or hardware failures may cause a system crash, which destroys the in-core superblock between updates to the disk. Then, the free list on disk does not accurately reflect the state of the disk. To reconstruct it, we must perform a lengthy examination of all blocks in the file system. (This problem remains in the new file system.)

The 4.2BSD file-system implementation is radically different from that of Version 7. This reimplemention was done primarily to improve efficiency and robustness, and most such changes are invisible outside the kernel. Other changes introduced at the same time are visible at both the system-call and the user levels; examples include symbolic links and long file names (up to 255 characters). Most of the changes required for these features were not in the kernel, however, but in the programs that use them.

Space allocation is especially different. The major new concept in FreeBSD is the *cylinder group*. The cylinder group was introduced to allow localization of the blocks in a file. Each cylinder group occupies one or more consecutive cylinders of the disk, so that disk accesses within the cylinder group require minimal disk head movement. Every cylinder group has a superblock, a cylinder block, an array of inodes, and some data blocks (Figure C.10).



Figure C.10 4.3 BSD cylinder group.

The superblocks in all cylinder groups are identical, so that a superblock can be recovered from any one of them in the event of disk corruption. The *cylinder block* contains dynamic parameters of the particular cylinder group. These include a bit map of free data blocks and fragments and a bitmap of free inodes. Statistics on recent progress of the allocation strategies are also kept here.

The header information in a cylinder group (the superblock, the cylinder block, and the inodes) is not always at the beginning of the group. If it were, the header information for every cylinder group might be on the same disk platter, and a single disk head crash could wipe out all of them. Therefore, each cylinder group has its header information at a different offset from the beginning of the group.

The directory-listing command `ls` commonly reads all the inodes of every file in a directory, making it desirable for all such inodes to be close together on the disk. For this reason, the inode for a file is usually allocated from the cylinder group containing the inode of the file's parent directory. Not everything can be localized, however, so an inode for a new directory is put in a *different* cylinder group from that of its parent directory. The cylinder group chosen for such a new directory inode is that with the greatest number of unused inodes.

To reduce disk head seeks involved in accessing the data blocks of a file, we allocate blocks from the same cylinder group as often as possible. Because a single file cannot be allowed to take up all the blocks in a cylinder group, a file exceeding a certain size (such as 2 MB) has further block allocation redirected to a different cylinder group; the new group is chosen from among those having more than average free space. If the file continues to grow, allocation is again redirected (at each megabyte) to yet another cylinder group. Thus, all the blocks of a small file are likely to be in the same cylinder group, and the number of long head seeks involved in accessing a large file is kept small.

There are two levels of disk-block-allocation routines. The global policy routines select a desired disk block according to the considerations already discussed. The local policy routines use the specific information recorded in the cylinder blocks to choose a block near the one requested. If the requested block is not in use, it is returned. Otherwise, the routine returns either the block rotationally closest to the one requested in the *same cylinder* or a block in a different cylinder but in the same cylinder group. If no more blocks are in the cylinder group, a quadratic rehash is done among all the other cylinder groups to find a block. If that fails, an exhaustive search is done. If enough free space (typically 10 percent) is left in the file system, blocks are usually found where desired, the quadratic rehash and exhaustive search are not used, and performance of the file system does not degrade with use.

Because of the increased efficiency of the Fast File System, typical disks are now utilized at 30 percent of their raw transfer capacity. This percentage is a marked improvement over that realized with the Version 7 file system, which used about 3 percent of the bandwidth.

BSD Tahoe introduced the Fat Fast File System, which allows the number of inodes per cylinder group, the number of cylinders per cylinder group, and the number of distinguished rotational positions to be set when the file system is created. FreeBSD previously set these parameters according to the disk hardware type.

## C.8 I/O System

One of the purposes of an operating system is to hide the peculiarities of specific hardware devices from the user. For example, the file system presents a simple, consistent storage facility (the file) independent of the underlying disk hardware. In UNIX, the peculiarities of I/O devices are also hidden from the bulk of the kernel itself by the *I/O system*. The I/O system consists of a buffer caching system, general device-driver code, and drivers for specific hardware devices. Only the device driver knows the peculiarities of a specific device. The major parts of the I/O system are diagrammed in Figure C.11.

There are three main kinds of I/O in FreeBSD: block devices, character devices, and the socket interface. The socket interface, together with its protocols and network interfaces, will be described in Section C.9.1.

*Block devices* include disks and tapes. Their distinguishing characteristic is that they are directly addressable in a fixed block size—usually 512 bytes. A block-device driver is required to isolate details of tracks, cylinders, and so on from the rest of the kernel. Block devices are accessible directly through appropriate device special files (such as `/dev/rp0`), but they are more commonly accessed indirectly through the file system. In either case, transfers are buffered through the *block buffer cache*, which has a profound effect on efficiency.

*Character devices* include terminals and line printers but also include almost everything else (except network interfaces) that does not use the block buffer cache. For instance, `/dev/mem` is an interface to physical main memory, and `/dev/null` is a bottomless sink for data and an endless source of end-of-file markers. Some devices, such as high-speed graphics interfaces, may have their own buffers or may always do I/O directly into the user's data space; because they do not use the block buffer cache, they are classed as character devices. Terminals and terminal-like devices use *C-lists*, which are buffers smaller than those of the block buffer cache.

Block devices and character devices are the two main device classes. Device drivers are accessed by one of two arrays of entry points. One array is for block devices; the other is for character devices. A device is distinguished by a class (block or character) and a *device number*. The device number consists of two parts. The *major device number* is used to index the array for character or block devices to find entries into the appropriate device driver. The *minor*

system-call interface to the kernel					
socket	plain file	cooked block interface	raw block interface	raw tty interface	cooked TTY
protocols	file system				line discipline
network interface	block-device driver			character-device driver	
the hardware					

Figure C.11 4.3 BSD kernel I/O structure.

*device number* is interpreted by the device driver as, for example, a logical disk partition or a terminal line.

A device driver is connected to the rest of the kernel only by the entry points recorded in the array for its class and by its use of common buffering systems. This segregation is important for portability and for system configuration.

### C.8.1 Block Buffer Cache

The block devices, as mentioned, use a block buffer cache. The buffer cache consists of a number of buffer headers, each of which can point to a piece of physical memory as well as to a device number and a block number on the device. The buffer headers for blocks not currently in use are kept in several linked lists, one for each of the following:

- Buffers recently used, linked in LRU order (the LRU list)
- Buffers not recently used or without valid contents (the AGE list)
- EMPTY buffers with no physical memory associated with them

The buffers in these lists are also hashed by device and block number for search efficiency.

When a block is wanted from a device (a read), the cache is searched. If the block is found, it is used, and no I/O transfer is necessary. If it is not found, a buffer is chosen from the AGE list or, if that list is empty, the LRU list. Then the device number and block number associated with it are updated, memory is found for it if necessary, and the new data are transferred into it from the device. If there are no empty buffers, the LRU buffer is written to its device (if it is modified), and the buffer is reused.

On a write, if the block in question is already in the buffer cache, the new data are put in the buffer (overwriting any previous data), the buffer header is marked to indicate that the buffer has been modified, and no I/O is immediately necessary. The data will be written when the buffer is needed for other data. If the block is not found in the buffer cache, an empty buffer is chosen (as with a read), and a transfer is done to this buffer. Writes are periodically forced for dirty buffer blocks to minimize potential file-system inconsistencies after a crash.

The number of data in a buffer in FreeBSD is variable, up to a maximum over all file systems, usually 8 KB. The minimum size is the paging-cluster size, usually 1,024 bytes. Buffers are page-cluster aligned, and any page cluster may be mapped into only one buffer at a time, just as any disk block may be mapped into only one buffer at a time. The EMPTY list holds buffer headers, which are used if a physical memory block of 8 KB is split to hold multiple, smaller blocks. Headers are needed for these blocks and are retrieved from EMPTY.

The number of data in a buffer may grow as a user process writes more data following those already in the buffer. When this increase occurs, a new buffer large enough to hold all the data is allocated, and the original data are copied into it, followed by the new data. If a buffer shrinks, a buffer is taken

off the empty queue, excess pages are put in it, and that buffer is released to be written to disk.

Some devices, such as magnetic tapes, require that blocks be written in a certain order. Facilities are therefore provided to force a synchronous write of buffers to these devices in the correct order. Directory blocks are also written synchronously, to forestall crash inconsistencies. Consider the chaos that could occur if many changes were made to a directory but the directory entries themselves were not updated!

The size of the buffer cache can have a profound effect on the performance of a system, because, if it is large enough, the percentage of cache hits can be high and the number of actual I/O transfers low. FreeBSD optimizes the buffer cache by continually adjusting the amount of memory used by programs and the disk cache.

Some interesting interactions occur among the buffer cache, the file system, and the disk drivers. When data are written to a disk file, they are buffered in the cache, and the disk driver sorts its output queue according to disk address. These two actions allow the disk driver to minimize disk head seeks and to write data at times optimized for disk rotation. Unless synchronous writes are required, a process writing to disk simply writes into the buffer cache, and the system asynchronously writes the data to disk when convenient. The user process sees very fast writes. When data are read from a disk file, the block I/O system does some read-ahead; however, writes are much nearer to asynchronous than are reads. Thus, output to the disk through the file system is often faster than is input for large transfers, counter to intuition.

### C.8.2 Raw Device Interfaces

Almost every block device also has a character interface, and these are called *raw device interfaces*. Such an interface differs from the *block interface* in that the block buffer cache is bypassed.

Each disk driver maintains a queue of pending transfers. Each record in the queue specifies whether it is a read or a write and gives a main memory address for the transfer (usually in 512-byte increments), a device address for the transfer (usually the address of a disk sector), and a transfer size (in sectors). It is simple to map the information from a block buffer to what is required for this queue.

It is almost as simple to map a piece of main memory corresponding to part of a user process's virtual address space. This mapping is what a raw disk interface, for instance, does. Unbuffered transfers directly to or from a user's virtual address space are thus allowed. The size of the transfer is limited by the physical devices, some of which require an even number of bytes.

The kernel accomplishes transfers for swapping and paging simply by putting the appropriate request on the queue for the appropriate device. No special swapping or paging device driver is needed.

The 4.2BSD file-system implementation was actually written and largely tested as a user process that used a raw disk interface, before the code was moved into the kernel. In an interesting about-face, the Mach operating system

has no file system per se. File systems can be implemented as user-level tasks (see Appendix D).

### C.8.3 C-Lists

As mentioned, terminals and terminal-like devices use a character-buffering system that keeps small blocks of characters (usually 28 bytes) in linked lists called C-lists. Although all free character buffers are kept in a single free list, most device drivers that use them limit the number of characters that may be queued at one time for any given terminal line.

There are routines to enqueue and dequeue characters for such lists. A `write()` system call to a terminal enqueues characters on a list for the device. An initial transfer is started, and interrupts cause dequeuing of characters and further transfers.

Input is similarly interrupt driven. Terminal drivers typically support *two* input queues, however, and conversion from the first (raw queue) to the other (canonical queue) is triggered when the interrupt routine puts an end-of-line character on the raw queue. The process doing a read on the device is then awakened, and its system phase does the conversion. The characters thus put on the canonical queue are then available to be returned to the user process by the read.

The device driver can bypass the canonical queue and return characters directly from the raw queue. This mode of operation is known as *raw mode*. Full-screen editors, as well as other programs that need to react to every keystroke, use this mode.

## C.9 Interprocess Communication

Although many tasks can be accomplished in isolated processes, many others require interprocess communication. Isolated computing systems have long served for many applications, but networking is increasingly important. Furthermore, with the increasing use of personal workstations, resource sharing is becoming more common. Interprocess communication has not traditionally been one of UNIX's strong points.

### C.9.1 Sockets

The pipe (discussed in Section C.4.3) is the IPC mechanism most characteristic of UNIX. A pipe permits a reliable unidirectional byte stream between two processes. It is traditionally implemented as an ordinary file, with a few exceptions. It has no name in the file system, being created instead by the `pipe()` system call. Its size is fixed, and when a process attempts to write to a full pipe, the process is suspended. Once all data previously written into the pipe have been read out, writing continues at the beginning of the file (pipes are not true circular buffers). One benefit of the small size of pipes (usually 4,096 bytes) is that pipe data are seldom actually written to disk; they usually are kept in memory by the normal block buffer cache.

In FreeBSD pipes are implemented as a special case of the *socket* mechanism. The socket mechanism provides a general interface not only to facilities such as pipes, which are local to one machine, but also to networking facilities.

Even on the same machine, a pipe can be used only by two processes related through use of the `fork()` system call. The socket mechanism can be used by unrelated processes.

A socket is an endpoint of communication. A socket in use usually has an *address* bound to it. The nature of the address depends on the *communication domain* of the socket. A characteristic property of a domain is that processes communicating in the same domain use the same *address format*. A single socket can communicate in only one domain.

The three domains currently implemented in FreeBSD are the UNIX domain (AF\_UNIX), the Internet domain (AF\_INET), and the XEROX Network Services (NS) domain (AF\_NS). The address format of the UNIX domain is that of an ordinary file-system path name, such as `/alpha/beta/gamma`. Processes communicating in the Internet domain use DARPA Internet communications protocols (such as TCP/IP) and Internet addresses, which consist of a 32-bit host number and a 32-bit port number (representing a rendezvous point on the host).

There are several **socket types**, which represent classes of services. Each type may or may not be implemented in any communication domain. If a type is implemented in a given domain, it may be implemented by one or more protocols, which may be selected by the user:

- **Stream sockets.** These sockets provide reliable, duplex, sequenced data streams. No data are lost or duplicated in delivery, and there are no record boundaries. This type is supported in the Internet domain by TCP. In the UNIX domain, pipes are implemented as a pair of communicating stream sockets.
- **Sequenced packet sockets.** These sockets provide data streams like those of stream sockets, except that record boundaries are provided. This type is used in the XEROX AF\_NS protocol.
- **Datagram sockets.** These sockets transfer messages of variable size in either direction. There is no guarantee that such messages will arrive in the same order they were sent, or that they will be unduplicated, or that they will arrive at all, but the original message (or record) size is preserved in any datagram that does arrive. This type is supported in the Internet domain by UDP.
- **Reliably delivered message sockets.** These sockets transfer messages that are guaranteed to arrive and that otherwise are like the messages transferred using datagram sockets. This type is currently unsupported.
- **Raw sockets.** These sockets allow direct access by processes to the protocols that support the other socket types. The protocols accessible include not only the uppermost ones but also lower-level protocols. For example, in the Internet domain, it is possible to reach TCP, IP beneath that, or an Ethernet protocol beneath that. This capability is useful for developing new protocols.

A set of system calls is specific to the socket facility. The `socket()` system call creates a socket. It takes as arguments specifications of the communication domain, the socket type, and the protocol to be used to support that type. The value returned by the call is a small integer called a **socket descriptor**, which

occupies the same name space as file descriptors. The socket descriptor indexes the array of open files in the *u* structure in the kernel and has a file structure allocated for it. The FreeBSD file structure may point to a socket structure instead of to an inode. In this case, certain socket information (such as the socket's type, its message count, and the data in its input and output queues) is kept directly in the socket structure.

For another process to address a socket, the socket must have a name. A name is bound to a socket by the `bind()` system call, which takes the socket descriptor, a pointer to the name, and the length of the name as a byte string. The contents and length of the byte string depend on the address format. The `connect()` system call is used to initiate a connection. The arguments are syntactically the same as those for `bind()`; the socket descriptor represents the local socket, and the address is that of the foreign socket to which the attempt to connect is made.

Many processes that communicate using the socket IPC follow the client-server model. In this model, the *server* process provides a *service* to the *client* process. When the service is available, the server process listens on a well-known address, and the client process uses `connect()` to reach the server.

A server process uses `socket()` to create a socket and `bind()` to bind the well-known address of its service to that socket. Then, it uses the `listen()` system call to tell the kernel that it is ready to accept connections from clients and to specify how many pending connections the kernel should queue until the server can service them. Finally, the server uses the `accept()` system call to accept individual connections. Both `listen()` and `accept()` take as an argument the socket descriptor of the original socket. The system call `accept()` returns a new socket descriptor corresponding to the new connection; the original socket descriptor is still open for further connections. The server usually uses `fork()` to produce a new process after the `accept()` to service the client while the original server process continues to listen for more connections. There are also system calls for setting parameters of a connection and for returning the address of the foreign socket after an `accept()`.

When a connection for a socket type, such as a stream socket, is established, the addresses of both endpoints are known, and no further addressing information is needed to transfer data. The ordinary `read()` and `write()` system calls may then be used to transfer data.

The simplest way to terminate a connection, and to destroy the associated socket, is to use the `close()` system call on its socket descriptor. We may also wish to terminate only one direction of communication of a duplex connection; the `shutdown()` system call can be used for this purpose.

Some socket types, such as datagram sockets, do not support connections. Instead, their sockets exchange datagrams that must be addressed individually. The system calls `sendto()` and `recvfrom()` are used for such connections. Both take as arguments a socket descriptor, a buffer pointer and length, and an address-buffer pointer and length. The address buffer contains the appropriate address for `sendto()` and is filled in with the address of the datagram just received by `recvfrom()`. The number of data actually transferred is returned by both system calls.

The `select()` system call can be used to multiplex data transfers on several file descriptors and/or socket descriptors. It can even be used to allow one server process to listen for client connections for many services and to `fork()`

a process for each connection as the connection is made. The server does a `socket()`, `bind()`, and `listen()` for each service and then does a `select()` on all the socket descriptors. When `select()` indicates activity on a descriptor, the server does an `accept()` on it and forks a process on the new descriptor returned by `accept()`, leaving the parent process to do a `select()` again.

### C.9.2 Network Support

Almost all current UNIX systems support the UUCP network facilities, which are mostly used over dial-up telephone lines to support the UUCP mail network and the USENET news network. These are, however, rudimentary networking facilities; they do not support even remote login, much less remote procedure calls or distributed file systems. These facilities are almost completely implemented as user processes and are not part of the operating system itself.

FreeBSD supports the DARPA Internet protocols UDP, TCP, IP, and ICMP on a wide range of Ethernet, token-ring, and ARPANET interfaces. The framework in the kernel to support these protocols is intended to facilitate the implementation of further protocols, and all protocols are accessible via the socket interface. Rob Gurwitz of BBN wrote the first version of the code as an add-on package for 4.1BSD.

The International Standards Organization's (ISO) Open System Interconnection (OSI) Reference Model for networking prescribes seven layers of network protocols and strict methods of communication between them. An implementation of a protocol may communicate only with a peer entity speaking the same protocol at the same layer or with the protocol–protocol interface of a protocol in the layer immediately above or below in the same system. The ISO networking model is implemented in FreeBSD Reno and 4.4BSD.

The FreeBSD networking implementation, and to a certain extent the socket facility, is more oriented toward the ARPANET Reference Model (ARM). The ARPANET in its original form served as a proof of concept for many networking ideas, such as packet switching and protocol layering. The ARPANET was retired in 1988 because the hardware that supported it was no longer state of the art. Its successors, such as the NSFNET and the Internet, are even larger and serve as communications utilities for researchers and test-beds for Internet gateway research. The ARM predates the ISO model; the ISO model was in large part inspired by the ARPANET research.

Although the ISO model is often interpreted as setting a limit of one protocol communicating per layer, the ARM allows several protocols in the same layer. There are only four protocol layers in the ARM:

- **Process/applications.** This layer subsumes the application, presentation, and session layers of the ISO model. Such user-level programs as the file-transfer protocol (FTP) and Telnet (remote login) exist at this level.
- **Host–host.** This layer corresponds to ISO's transport and the top part of its network layers. Both the Transmission Control Protocol (TCP) and the Internet Protocol (IP) are in this layer, with TCP on top of IP. TCP corresponds to an ISO transport protocol, and IP performs the addressing functions of the ISO network layer.
- **Network interface.** This layer spans the lower part of the ISO network layer and the entire data-link layer. The protocols involved here depend on the

physical network type. The ARPANET uses the IMP-Host protocols, whereas an Ethernet uses Ethernet protocols.

- **Network hardware.** The ARM is primarily concerned with software, so there is no explicit network hardware layer. However, any actual network will have hardware corresponding to the ISO physical layer.

The networking framework in FreeBSD is more generalized than either the ISO model or the ARM, although it is most closely related to the ARM (Figure C.12).

User processes communicate with network protocols (and thus with other processes on other machines) via the socket facility. This facility corresponds to the ISO session layer, as it is responsible for setting up and controlling communications.

Sockets are supported by protocols—possibly by several, layered one on another. A protocol may provide services such as reliable delivery, suppression of duplicate transmissions, flow control, and addressing, depending on the socket type being supported and the services required by any higher protocols.

A protocol may communicate with another protocol or with the network interface that is appropriate for the network hardware. There is little restriction in the general framework on what protocols may communicate with what other protocols or on how many protocols may be layered on top of one another. The user process may, by means of the raw socket type, directly access any layer of protocol from the uppermost used to support one of the other socket types, such as streams, down to a raw network interface. This capability is used by routing processes and also for new protocol development.

Most often, there is one **network-interface driver** per network controller type. The network interface is responsible for handling characteristics specific to the local network being addressed. This arrangement ensures that the protocols using the interface do not need to be concerned with these characteristics.

The functions of the network interface depend largely on the **network hardware**, which is whatever is necessary for the network. Some networks may

ISO reference model	ARPANET reference model	4.2BSD layers	example layering
application	process applications	user programs and libraries	telnet
presentation		sockets	sock_stream
session transport		host-host	TCP
network data link	network interface		IP
hardware	network interfaces	Ethernet driver	
	network hardware	network hardware	interlan controller

**Figure C.12** Network reference models and layering.

support reliable transmission at this level, but most do not. Some networks provide broadcast addressing, but many do not.

The socket facility and the networking framework use a common set of memory buffers, or *mbufs*. These are intermediate in size between the large buffers used by the block I/O system and the C-lists used by character devices. An *mbuf* is 128 bytes long; 112 bytes may be used for data, and the rest is used for pointers to link the *mbuf* into queues and for indicators of how much of the data area is actually in use.

Data are ordinarily passed between layers—socket–protocol, protocol–protocol, or protocol–network interface—in *mbufs*. The ability to pass the buffers containing the data eliminates some data copying, but there is still frequently a need to remove or add protocol headers. It is also convenient and efficient for many purposes to be able to hold data that occupy an area the size of the memory-management page. Thus, the data of an *mbuf* may reside not in the *mbuf* itself but elsewhere in memory. There is an *mbuf* page table for this purpose, as well as a pool of pages dedicated to *mbuf* use.

## C.10 Summary

The early advantages of UNIX were that it was written in a high-level language, was distributed in source form, and provided powerful operating-system primitives on an inexpensive platform. These advantages led to UNIX's popularity at educational, research, and government institutions and eventually in the commercial world. This popularity produced many strains of UNIX with varying and improved facilities.

UNIX provides a file system with tree-structured directories. The kernel supports files as unstructured sequences of bytes. Direct access and sequential access are supported through system calls and library routines.

Files are stored as an array of fixed-size data blocks with perhaps a trailing fragment. The data blocks are found by pointers in the inode. Directory entries point to inodes. Disk space is allocated from cylinder groups to minimize head movement and to improve performance.

UNIX is a multiprogrammed system. Processes can easily create new processes with the `fork()` system call. Processes can communicate with pipes or, more generally, sockets. They may be grouped into jobs that may be controlled with signals.

Processes are represented by two structures: the process structure and the user structure. CPU scheduling is a priority algorithm with dynamically computed priorities that reduces to round-robin scheduling in the extreme case.

FreeBSD memory management uses swapping supported by paging. A pagedaemon process uses a modified second-chance page-replacement algorithm to keep enough free frames to support the executing processes.

Page and file I/O uses a block buffer cache to minimize the amount of actual I/O. Terminal devices use a separate character-buffering system.

Networking support is one of the most important features in FreeBSD. The socket concept provides the programming mechanism to access other processes, even across a network. Sockets provide an interface to several sets of protocols.

## Further Reading

[McKusick et al. (2015)] provides a good general discussion of FreeBSD. A modern scheduler for FreeBSD is described in [Roberson (2003)]. Locking in the Multithreaded FreeBSD Kernel is described in [Baldwin (2002)].

FreeBSD is described in *The FreeBSD Handbook*, which can be downloaded from <http://www.freebsd.org>.

## Bibliography

**[Baldwin (2002)]** J. Baldwin, “Locking in the Multithreaded FreeBSD Kernel”, *USENIX BSD* (2002).

**[McKusick et al. (2015)]** M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson, *The Design and Implementation of the FreeBSD UNIX Operating System—Second Edition*, Pearson (2015).

**[Roberson (2003)]** J. Roberson, “ULE: A Modern Scheduler For FreeBSD”, *Proceedings of the USENIX BSDCon Conference* (2003), pages 17–28.

# The Mach System



This chapter was first written in 1991 and has been updated over time but is no longer modified

In this appendix we examine the Mach operating system. Mach is designed to incorporate the many recent innovations in operating-system research to produce a fully functional, technically advanced system. Unlike UNIX, which was developed without regard for multiprocessing, Mach incorporates multiprocessing support throughout. This support is exceedingly flexible, accommodating shared-memory systems as well as systems with no memory shared between processors. Mach is designed to run on computer systems ranging from one processor to thousands of processors. In addition, it is easily ported to many varied computer architectures. A key goal of Mach is to be a distributed system capable of functioning on heterogeneous hardware.

Although many experimental operating systems are being designed, built, and used, Mach satisfies the needs of most users better than the others because it offers full compatibility with UNIX 4.3 BSD. This compatibility also gives us a unique opportunity to compare two functionally similar, but internally dissimilar, operating systems. Mach and UNIX differ in their emphases, so our Mach discussion does not exactly parallel our UNIX discussion. In addition, we do not include a section on the user interface, because that component is similar to the user interface in 4.3 BSD. As you will see, Mach provides the ability to layer emulation of other operating systems as well; other operating systems can even run concurrently with Mach.

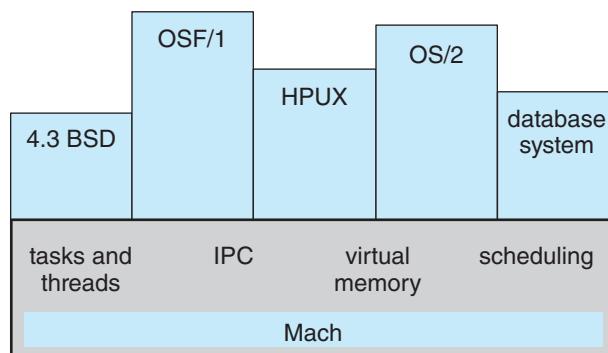
## D.1 History of the Mach System

Mach traces its ancestry to the Accent operating system developed at Carnegie Mellon University (CMU). Although Accent pioneered a number of novel operating-system concepts, its utility was limited by its inability to execute UNIX applications and its strong ties to a single hardware architecture, which made it difficult to port. Mach's communication system and philosophy are derived from Accent, but many other significant portions of the system (for example, the virtual memory system and the management of tasks and threads) were developed from scratch. An important goal of the Mach effort was support for multiprocessors.

Mach's development followed an evolutionary path from BSD UNIX systems. Mach code was initially developed inside the 4.2BSD kernel, with BSD kernel components replaced by Mach components as the Mach components were completed. The BSD components were updated to 4.3 BSD when that became available. By 1986, the virtual memory and communication subsystems were running on the DEC VAX computer family, including multiprocessor versions of the VAX. Versions for the IBM RT/PC and for Sun 3 workstations followed shortly; 1987 saw the completion of the Encore Multimax and Sequent Balance multiprocessor versions, including task and thread support, as well as the first official releases of the system, Release 0 and Release 1.

Through Release 2, Mach provided compatibility with the corresponding BSD systems by including much of BSD's code in the kernel. The new features and capabilities of Mach made the kernels in these releases larger than the corresponding BSD kernels. Mach 3 (Figure D.1) moved the BSD code outside of the kernel, leaving a much smaller microkernel. This system implements only basic Mach features in the kernel; all UNIX-specific code has been evicted to run in user-mode servers. Excluding UNIX-specific code from the kernel allows replacement of BSD with another operating system or the simultaneous execution of multiple operating-system interfaces on top of the microkernel. In addition to BSD, user-mode implementations have been developed for DOS, the Macintosh operating system, and OSF/1. This approach has similarities to the virtual machine concept, but the virtual machine is defined by software (the Mach kernel interface), rather than by hardware. With Release 3.0, Mach became available on a wide variety of systems, including single-processor Sun, Intel, IBM, and DEC machines and multiprocessor DEC, Sequent, and Encore systems.

Mach was propelled to the forefront of industry attention when the Open Software Foundation (OSF) announced in 1989 that it would use Mach 2.5 as the basis for its new operating system, OSF/1. The release of OSF/1 occurred a year later, and it now competes with UNIX System V, Release 4, the operating system of choice among *UNIX International (UI)* members. OSF members include key technological companies such as IBM, DEC, and HP. Mach 2.5 is also the basis for the operating system on the NeXT workstation, the brainchild of Steve Jobs, of Apple Computer fame. OSF is evaluating Mach 3 as the basis for a future



**Figure D.1** Mach 3 structure.

operating-system release, and research on Mach continues at CMU, OSF, and elsewhere.

## D.2 Design Principles

The Mach operating system was designed to provide basic mechanisms that most current operating systems lack. The goal is to design an operating system that is BSD-compatible and, in addition, excels in the following areas:

- Support for diverse architectures, including multiprocessors with varying degrees of shared memory access: uniform memory access (UMA), non-uniform memory access (NUMA), and no remote memory access (NORMA)
- Ability to function with varying intercomputer network speeds, from wide-area networks to high-speed local-area networks and tightly coupled multiprocessors
- Simplified kernel structure, with a small number of abstractions (in turn, these abstractions are sufficiently general to allow other operating systems to be implemented on top of Mach.)
- Distributed operation, providing network transparency to clients and an object-oriented organization both internally and externally
- Integrated memory management and interprocess communication, to provide efficient communication of large numbers of data as well as communication-based memory management
- Heterogeneous system support, to make Mach widely available and interoperable among computer systems from multiple vendors

The designers of Mach have been heavily influenced by BSD (and by UNIX in general), whose benefits include

- A simple programmer interface, with a good set of primitives and a consistent set of interfaces to system facilities
- Easy portability to a wide class of single processors
- An extensive library of utilities and applications
- The ability to combine utilities easily via pipes

Of course, the designers also wanted to redress what they saw as the drawbacks of BSD:

- A kernel that has become the repository of many redundant features—and that consequently is difficult to manage and modify
- Original design goals that made it difficult to provide support for multiprocessors, distributed systems, and shared program libraries (for instance, because the kernel was designed for single processors, it has no provisions for locking code or data that other processors might be using.)

- Too many fundamental abstractions, providing too many similar, competing means with which to accomplish the same tasks

The development of Mach continues to be a huge undertaking. The benefits of such a system are equally large, however. The operating system runs on many existing single-processor and multiprocessor architectures, and it can be easily ported to future ones. It makes research easier, because computer scientists can add features via user-level code, instead of having to write their own tailor-made operating system. Areas of experimentation include operating systems, databases, reliable distributed systems, multiprocessor languages, security, and distributed artificial intelligence. In its current version, the Mach system is usually as efficient as other major versions of UNIX when performing similar tasks.

### D.3 System Components

To achieve the design goals of Mach, the developers reduced the operating-system functionality to a small set of basic abstractions, out of which all other functionality can be derived. The Mach approach is to place as little as possible within the kernel but to make what is there powerful enough that all other features can be implemented at the user level.

Mach's design philosophy is to have a simple, extensible kernel, concentrating on communication facilities. For instance, all requests to the kernel, and all data movement among processes, are handled through one communication mechanism. Mach is therefore able to provide system-wide protection to its users by protecting the communication mechanism. Optimizing this one communication path can result in significant performance gains, and it is simpler than trying to optimize several paths. Mach is extensible, because many traditionally kernel-based functions can be implemented as user-level servers. For instance, all pagers (including the default pager) can be implemented externally and called by the kernel for the user.

Mach is an example of an object-oriented system where the data and the operations that manipulate that data are encapsulated into an abstract object. Only the operations of the object are able to act on the entities defined in it. The details of how these operations are implemented are hidden, as are the internal data structures. Thus, a programmer can use an object only by invoking its defined, exported operations. A programmer can change the internal operations without changing the interface definition, so changes and optimizations do not affect other aspects of system operation. The object-oriented approach supported by Mach allows objects to reside anywhere in a network of Mach systems, transparent to the user. The port mechanism, discussed later in this section, makes all of this possible.

Mach's primitive abstractions are the heart of the system and are as follows:

- A **task** is an execution environment that provides the basic unit of resource allocation. It consists of a virtual address space and protected access to system resources via ports, and it may contain one or more threads.
- A **thread** is the basic unit of execution and must run in the context of a task (which provides the address space). All threads within a task share the

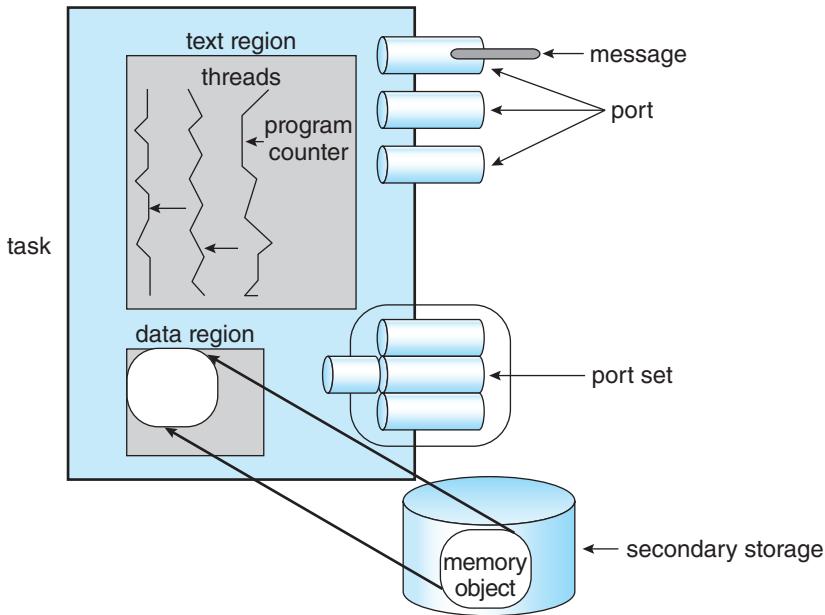
task's resources (ports, memory, and so on). There is no notion of a *process* in Mach. Rather, a traditional process is implemented as a task with a single thread of control.

- A **port** is the basic object-reference mechanism in Mach and is implemented as a kernel-protected communication channel. Communication is accomplished by sending messages to ports; messages are queued at the destination port if no thread is immediately ready to receive them. Ports are protected by kernel-managed capabilities, or **port rights**. A task must have a port right to send a message to a port. The programmer invokes an operation on an object by *sending* a message to a port associated with that object. The object being represented by a port *receives* the messages.
- A **port set** is a group of ports sharing a common message queue. A thread can receive messages for a port set and thus service multiple ports. Each received message identifies the individual port (within the set) from which it was received. The receiver can use this to identify the object referred to by the message.
- A **message** is the basic method of communication between threads in Mach. It is a typed collection of data objects. For each object, it may contain the actual data or a pointer to out-of-line data. Port rights are passed in messages; this is the only way to move them among tasks. (Passing a port right in shared memory does not work, because the Mach kernel will not permit the new task to use a right obtained in this manner.)
- A **memory object** is a source of memory. Tasks can access it by mapping portions of an object (or the entire object) into their address spaces. The object can be managed by a user-mode external memory manager. One example is a file managed by a file server; however, a memory object can be any object for which memory-mapped access makes sense. A mapped buffer implementation of a UNIX pipe is another example.

Figure D.2 illustrates these abstractions, which we explain further in the remainder of this chapter.

An unusual feature of Mach, and a key to the system's efficiency, is the blending of memory and interprocess-communication (IPC) features. Whereas some other distributed systems (such as Solaris, with its NFS features) have special-purpose extensions to the file system to extend it over a network, Mach provides a general-purpose, extensible merger of memory and messages at the heart of its kernel. This feature not only allows Mach to be used for distributed and parallel programming but also helps in the implementation of the kernel itself.

Mach connects memory management and IPC by allowing each to be used in the implementation of the other. Memory management is based on the use of memory objects. A memory object is represented by a port (or ports), and IPC messages are sent to this port to request operations (for example, pagein, pageout) on the object. Because IPC is used, memory objects can reside on remote systems and be accessed transparently. The kernel caches the contents of memory objects in local memory. Conversely, memory-management techniques are used in the implementation of message passing. Where possible,



**Figure D.2** Mach's basic abstractions.

Mach passes messages by moving pointers to shared memory objects, rather than by copying the objects themselves.

IPC tends to involve considerable system overhead. For intrasystem messages, it is generally less efficient than communication accomplished through shared memory. Because Mach is a message-based kernel, message handling must be carried out efficiently. Most of the inefficiency of message handling in traditional operating systems is due to either the copying of messages from one task to another (for intracomputer messages) or low network-transfer speed (for intercomputer messages). To solve these problems, Mach uses virtual memory remapping to transfer the contents of large messages. In other words, the message transfer modifies the receiving task's address space to include a copy of the message contents. Virtual copy (or copy-on-write) techniques are used to avoid or delay the actual copying of the data. This approach has several advantages:

- Increased flexibility in memory management for user programs
- Greater generality, allowing the virtual copy approach to be used in tightly and loosely coupled computers
- Improved performance over UNIX message passing
- Easier task migration (because ports are location independent, a task and all its ports can be moved from one machine to another. All tasks that previously communicated with the moved task can continue to do so because they reference the task only by its ports and communicate via messages to these ports.)

In the sections that follow, we detail the operation of process management, IPC, and memory management. Then, we discuss Mach’s chameleonlike ability to support multiple operating-system interfaces.

## D.4 Process Management

A task can be thought of as a traditional process that does not have an instruction pointer or a register set. A task contains a virtual address space, a set of port rights, and accounting information. A task is a passive entity that does nothing unless it has one or more threads executing in it.

### D.4.1 Basic Structure

A task containing one thread is similar to a UNIX process. Just as a `fork()` system call produces a new UNIX process, Mach creates a new task by using `fork()`. The new task’s memory is a duplicate of the parent’s address space, as dictated by the **inheritance attributes** of the parent’s memory. The new task contains one thread, which is started at the same point as the creating `fork()` call in the parent. Threads and tasks can also be suspended and resumed.

Threads are especially useful in server applications, which are common in UNIX, since one task can have multiple threads to service multiple requests to the task. Threads also allow efficient use of parallel computing resources. Rather than having one process on each processor, with the corresponding performance penalty and operating-system overhead, a task can have its threads spread among parallel processors. Threads add efficiency to user-level programs as well. For instance, in UNIX, an entire process must wait when a page fault occurs or when a system call is executed. In a task with multiple threads, only the thread that causes the page fault or executes a system call is delayed; all other threads continue executing. Because Mach has kernel-supported threads (Chapter 4), the threads have some cost associated with them. They must have supporting data structures in the kernel, and more complex kernel-scheduling algorithms must be provided. These algorithms and thread states are discussed in Chapter 4.

At the user level, threads may be in one of two states:

- **Running.** The thread is either executing or waiting to be allocated a processor. A thread is considered to be running even if it is blocked within the kernel (waiting for a page fault to be satisfied, for instance).
- **Suspended.** The thread is neither executing on a processor nor waiting to be allocated a processor. A thread can resume its execution only if it is returned to the running state.

These two states are also associated with a task. An operation on a task affects all threads in a task, so suspending a task involves suspending all the threads in it. Task and thread suspensions are separate, independent mechanisms, however, so resuming a thread in a suspended task does not resume the task.

Mach provides primitives from which thread-synchronization tools can be built. This practice is consistent with Mach’s philosophy of providing mini-

mum yet sufficient functionality in the kernel. The Mach IPC facility can be used for synchronization, with processes exchanging messages at rendezvous points. Thread-level synchronization is provided by calls to start and stop threads at appropriate times. A *suspend count* is kept for each thread. This count allows multiple `suspend()` calls to be executed on a thread, and only when an equal number of `resume()` calls occur is the thread resumed. Unfortunately, this feature has its own limitation. Because it is an error for a `start()` call to be executed before a `stop()` call (the suspend count would become negative), these routines cannot be used to synchronize shared data access. However, `wait()` and `signal()` operations associated with semaphores, and used for synchronization, can be implemented via the IPC calls. We discuss this method in Section D.5.

### D.4.2 The C Threads Package

Mach provides low-level but flexible routines instead of polished, large, and more restrictive functions. Rather than making programmers work at this low level, Mach provides many higher-level interfaces for programming in C and other languages. For instance, the C threads package provides multiple threads of control, shared variables, mutual exclusion for critical sections, and condition variables for synchronization. In fact, C threads is one of the major influences on the POSIX Pthreads standard, which many operating systems support. As a result, there are strong similarities between the C threads and Pthreads programming interfaces. The thread-control routines include calls to perform these tasks:

- Create a new thread within a task, given a function to execute and parameters as input. The thread then executes concurrently with the creating thread, which receives a thread identifier when the call returns.
- Destroy the calling thread, and return a value to the creating thread.
- Wait for a specific thread to terminate before allowing the calling thread to continue. This call is a synchronization tool, much like the UNIX `wait()` system calls.
- Yield use of a processor, signaling that the scheduler can run another thread at this point. This call is also useful in the presence of a preemptive scheduler, as it can be used to relinquish the CPU voluntarily before the time quantum (or scheduling interval) expires if a thread has no use for the CPU.

Mutual exclusion is achieved through the use of spinlocks, as discussed in Chapter 6. The routines associated with mutual exclusion are these:

- The routine `mutex_alloc()` dynamically creates a mutex variable.
- The routine `mutex_free()` deallocates a dynamically created mutex variable.
- The routine `mutex_lock()` locks a mutex variable. The executing thread loops in a spinlock until the lock is attained. A deadlock results if a thread with a lock tries to lock the same mutex variable. Bounded waiting is

not guaranteed by the C threads package. Rather, it is dependent on the hardware instructions used to implement the mutex routines.

- The routine `mutex_unlock()` unlocks a mutex variable, much like the typical `signal()` operation of a semaphore.

General synchronization without busy waiting can be achieved through the use of condition variables, which can be used to implement a monitor, as described in Chapter 6. A condition variable is associated with a mutex variable and reflects a Boolean state of that variable. The routines associated with general synchronization are these:

- The routine `condition_alloc()` dynamically allocates a condition variable.
- The routine `condition_free()` deletes a dynamically created condition variable allocated as a result of `condition_alloc()`.
- The routine `condition_wait()` unlocks the associated mutex variable and blocks the thread until a `condition_signal()` is executed on the condition variable, indicating that the event being waited for may have occurred. The mutex variable is then locked, and the thread continues. A `condition_signal()` does not guarantee that the condition still holds when the unblocked thread finally returns from its `condition_wait()` call, so the awakened thread must loop, executing the `condition_wait()` routine until it is unblocked and the condition holds.

As an example of the C threads routines, consider the bounded-buffer synchronization problem of Section 7.1.1. The producer and consumer are represented as threads that access the common bounded-buffer pool. We use a mutex variable to protect the buffer while it is being updated. Once we have exclusive access to the buffer, we use condition variables to block the producer thread if the buffer is full and to block the consumer thread if the buffer is empty. As in Chapter 6, we assume that the buffer consists of  $n$  slots, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The `empty` and `full` semaphores count the number of empty and full buffers, respectively. The semaphore `empty` is initialized to the value  $n$ ; the semaphore `full` is initialized to the value 0. The condition variable `nonempty` is true while the buffer has items in it, and `nonfull` is true if the buffer has an empty slot. The first step includes the allocation of the mutex and condition variables:

```
mutex_alloc(mutex);
condition_alloc(nonempty, nonfull);
```

The code for the producer thread is shown in Figure D.3, and the code for the consumer thread is shown in Figure D.4. When the program terminates, the mutex and condition variables need to be deallocated:

```
mutex_free(mutex);
condition_free(nonempty, nonfull);
```

```

do {
    . . .
    // produce an item into nextp
    . . .
    mutex_lock(mutex);
    while(full)
        condition_wait(nonfull, mutex);
    . . .
    // add nextp to buffer
    . . .
    condition_signal(nonempty);
    mutex_unlock(mutex);
} while(TRUE);

```

---

**Figure D.3** The structure of the producer process.

#### D.4.3 The CPU Scheduler

The CPU scheduler for a thread-based multiprocessor operating system is more complex than its process-based relatives. There are generally more threads in a multithreaded system than there are processes in a multitasking system. Keeping track of multiple processors is also difficult and is a relatively new area of research. Mach uses a simple policy to keep the scheduler manageable. Only threads are scheduled, so no knowledge of tasks is needed in the scheduler. All threads compete equally for resources, including time quanta.

Each thread has an associated priority number ranging from 0 through 127, which is based on the exponential average of its usage of the CPU. That is, a thread that recently used the CPU for a large amount of time has the lowest

---

```

do {
    mutex_lock(mutex);
    while(empty)
        condition_wait(nonempty, mutex);
    . . .
    // remove an item from the buffer to nextc
    . . .
    condition_signal(nonfull);
    mutex_unlock(mutex);
    . . .
    // consume the item in nextc
    . . .
} until(FALSE);

```

---

**Figure D.4** The structure of the consumer process.

priority. Mach uses the priority to place the thread in one of 32 global run queues. These queues are searched in priority order for waiting threads when a processor becomes idle. Mach also keeps per-processor, or local, run queues. A local run queue is used for threads that are bound to an individual processor. For instance, a device driver for a device connected to an individual CPU must run on only that CPU.

Instead of a central dispatcher that assigns threads to processors, each processor consults the local and global run queues to select the appropriate next thread to run. Threads in the local run queue have absolute priority over those in the global queues, because it is assumed that they are performing some chore for the kernel. The run queues—like most other objects in Mach—are locked when they are modified to avoid simultaneous changes by multiple processors. To speed dispatching of threads on the global run queue, Mach maintains a list of idle processors.

Additional scheduling difficulties arise from the multiprocessor nature of Mach. A fixed time quantum is not appropriate because, for instance, there may be fewer runnable threads than there are available processors. It would be wasteful to interrupt a thread with a context switch to the kernel when that thread's quantum runs out, only to have the thread be placed right back in the running state. Thus, instead of using a fixed-length quantum, Mach varies the size of the time quantum inversely with the total number of threads in the system. It keeps the time quantum over the entire system constant, however. For example, in a system with 10 processors, 11 threads, and a 100-millisecond quantum, a context switch needs to occur on each processor only once per second to maintain the desired quantum.

Of course, complications still exist. Even relinquishing the CPU while waiting for a resource is more difficult than it is on traditional operating systems. First, a thread must issue a call to alert the scheduler that the thread is about to block. This alert avoids race conditions and deadlocks, which could occur when the execution takes place in a multiprocessor environment. A second call actually causes the thread to be moved off the run queue until the appropriate event occurs. The scheduler uses many other internal thread states to control thread execution.

#### D.4.4 Exception Handling

Mach was designed to provide a single, simple, consistent exception-handling system, with support for standard as well as user-defined exceptions. To avoid redundancy in the kernel, Mach uses kernel primitives whenever possible. For instance, an exception handler is just another thread in the task in which the exception occurs. Remote procedure call (RPC) messages are used to synchronize the execution of the thread causing the exception (the *victim*) and that of the handler and to communicate information about the exception between the victim and handler. Mach exceptions are also used to emulate the BSD signal package.

Disruptions to normal program execution come in two varieties: internally generated exceptions and external interrupts. Interrupts are asynchronously generated disruptions of a thread or task, whereas exceptions are caused by the occurrence of unusual conditions during a thread's execution. Mach's general-purpose exception facility is used for error detection and debugger support.

This facility is also useful for other functions, such as taking a core dump of a bad task, allowing tasks to handle their own errors (mostly arithmetic), and emulating instructions not implemented in hardware.

Mach supports two different granularities of exception handling. Error handling is supported by per-thread exception handling, whereas debuggers use per-task handling. It makes little sense to try to debug only one thread or to have exceptions from multiple threads invoke multiple debuggers. Aside from this distinction, the only difference between the two types of exceptions lies in their inheritance from a parent task. Task-wide exception-handling facilities are passed from the parent to child tasks, so debuggers are able to manipulate an entire tree of tasks. Error handlers are not inherited and default to no handler at thread- and task-creation time. Finally, error handlers take precedence over debuggers if the exceptions occur simultaneously. The reason for this approach is that error handlers are normally part of the task and therefore should execute normally even in the presence of a debugger.

Exception handling proceeds as follows:

- The victim thread causes notification of an exception's occurrence via a `raise()` RPC message sent to the handler.
- The victim then calls a routine to wait until the exception is handled.
- The handler receives notification of the exception, usually including information about the exception, the thread, and the task causing the exception.
- The handler performs its function according to the type of exception. The handler's action involves *clearing* the exception, causing the victim to *resume*, or *terminating* the victim thread.

To support the execution of BSD programs, Mach needs to support BSD-style signals. Signals provide software-generated interrupts and exceptions. Unfortunately, signals are of limited functionality in multithreaded operating systems. The first problem is that, in UNIX, a signal's handler must be a routine in the process receiving the signal. If the signal is caused by a problem in the process itself (for example, a division by 0), the problem cannot be remedied, because a process has limited access to its own context. A second, more troublesome aspect of signals is that they were designed for only single-threaded programs. For instance, it makes no sense for all threads in a task to get a signal, but how can a signal be seen by only one thread?

Because the signal system must work correctly with multithreaded applications for Mach to run 4.3 BSD programs, signals could not be abandoned. Producing a functionally correct signal package required several rewrites of the code, however. A final problem with UNIX signals is that they can be lost. This loss occurs when another signal of the same type occurs before the first is handled. Mach exceptions are queued as a result of their RPC implementation.

Externally generated signals, including those sent from one BSD process to another, are processed by the BSD server section of the Mach 2.5 kernel. Their behavior is therefore the same as it is under BSD. Hardware exceptions are a different matter, because BSD programs expect to receive hardware exceptions as signals. Therefore, a hardware exception caused by a thread must arrive at the thread as a signal. So that this result is produced, hardware exceptions are

converted to exception RPCs. For tasks and threads that do not make explicit use of the Mach exception-handling facility, the destination of this RPC defaults to an in-kernel task. This task has only one purpose: Its thread runs in a continuous loop, receiving the exception RPCs. For each RPC, it converts the exception into the appropriate signal, which is sent to the thread that caused the hardware exception. It then completes the RPC, clearing the original exception condition. With the completion of the RPC, the initiating thread reenters the run state. It immediately sees the signal and executes its signal-handling code. In this manner, all hardware exceptions begin in a uniform way—as exception RPCs. Threads not designed to handle such exceptions, however, receive the exceptions as they would on a standard BSD system—as signals. In Mach 3.0, the signal-handling code is moved entirely into a server, but the overall structure and flow of control is similar to those of Mach 2.5.

## D.5 Interprocess Communication

Most commercial operating systems, such as UNIX, provide communication between processes and between hosts with fixed, global names (or Internet addresses). There is no location independence of facilities, because any remote system needing to use a facility must know the name of the system providing that facility. Usually, data in the messages are untyped streams of bytes. Mach simplifies this picture by sending messages between location-independent ports. The messages contain typed data for ease of interpretation. All BSD communication methods can be implemented with this simplified system.

The two components of Mach IPC are ports and messages. Almost everything in Mach is an object, and all objects are addressed via their communication ports. Messages are sent to these ports to initiate operations on the objects by the routines that implement the objects. By depending on only ports and messages for all communication, Mach delivers location independence of objects and security of communication. Data independence is provided by the NetMsgServer task (Section D.5.3).

Mach ensures security by requiring that message senders and receivers have *rights*. A right consists of a port name and a capability—send or receive—on that port, and is much like a capability in object-oriented systems. Only one task may have receive rights to any given port, but many tasks may have send rights. When an object is created, its creator also allocates a port to represent the object and obtains the access rights to that port. Rights can be given out by the creator of the object, including the kernel, and are passed in messages. If the holder of a receive right sends that right in a message, the receiver of the message gains the right, and the sender loses it. A task may allocate ports to allow access to any objects it owns or for communication. The destruction of either a port or the holder of the receive right causes the revocation of all rights to that port, and the tasks holding send rights can be notified if desired.

### D.5.1 Ports

A port is implemented as a protected, bounded queue within the kernel of the system on which the object resides. If a queue is full, a sender may abort the

send, wait for a slot to become available in the queue, or have the kernel deliver the message.

Several system calls provide the port with the following functionalities:

- Allocate a new port in a specified task and give the caller's task all access rights to the new port. The port name is returned.
- Deallocate a task's access rights to a port. If the task holds the receive right, the port is destroyed, and all other tasks with send rights are, potentially, notified.
- Get the current status of a task's port.
- Create a backup port, which is given the receive right for a port if the task containing the receive right requests its deallocation or terminates.

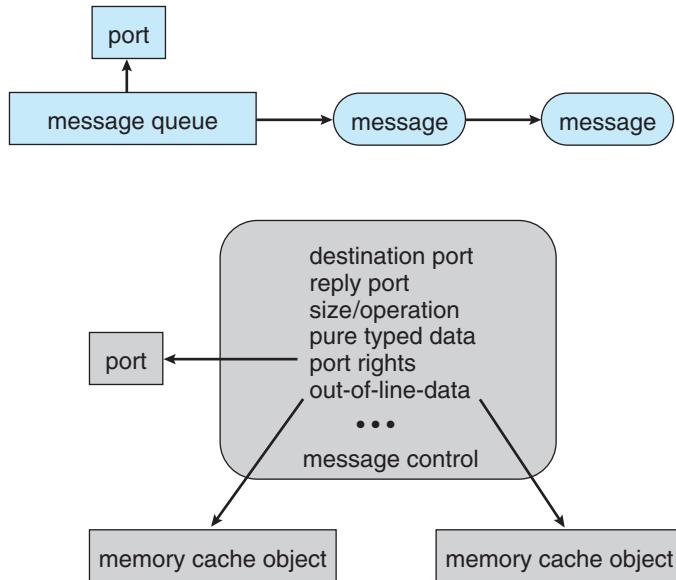
When a task is created, the kernel creates several ports for it. The function `task_self()` returns the name of the port that represents the task in calls to the kernel. For instance, to allocate a new port, a task calls `port_allocate()` with `task_self()` as the name of the task that will own the port. Thread creation results in a similar `thread_self()` thread kernel port. This scheme is similar to the standard process-ID concept found in UNIX. Another port is returned by `task_notify()`; this is the port to which the kernel will send event-notification messages (such as notifications of port terminations).

Ports can also be collected into *port sets*. This facility is useful if one thread is to service requests coming in on multiple ports—for example, for multiple objects. A port may be a member of no more than one port set at a time. Furthermore, if a port is in a set, it may not be used directly to receive messages. Instead, messages will be routed to the port set's queue. A port set may not be passed in messages, unlike a port. Port sets are objects that serve a purpose similar to the 4.3 BSD `select()` system call, but they are more efficient.

### D.5.2 Messages

A message consists of a fixed-length header and a variable number of typed data objects. The header contains the destination's port name, the name of the reply port to which return messages should be sent, and the length of the message (Figure D.5). The data in the message (or in-line data) were limited to less than 8 KB in Mach 2.5 systems, but Mach 3.0 has no limit. Each data section may be a simple type (numbers or characters), port rights, or pointers to out-of-line data. Each section has an associated type, so that the receiver can unpack the data correctly even if it uses a byte ordering different from that used by the sender. The kernel also inspects the message for certain types of data. For instance, the kernel must process port information within a message, either by translating the port name into an internal port data structure address or by forwarding it for processing to the NetMsgServer (Section D.5.3).

The use of pointers in a message provides the means to transfer the entire address space of a task in one single message. The kernel also must process pointers to out-of-line data, since a pointer to data in the sender's address space would be invalid in the receiver's—especially if the sender and receiver reside on different systems. Generally, systems send messages by copying the data from the sender to the receiver. Because this technique can be inefficient, especially for large messages, Mach takes a different approach. The data refer-



**Figure D.5** Mach messages.

enced by a pointer in a message being sent to a port on the same system are not copied between the sender and the receiver. Instead, the address map of the receiving task is modified to include a copy-on-write copy of the pages of the message. This operation is *much* faster than a data copy and makes message passing more efficient. In essence, message passing is implemented via virtual memory management.

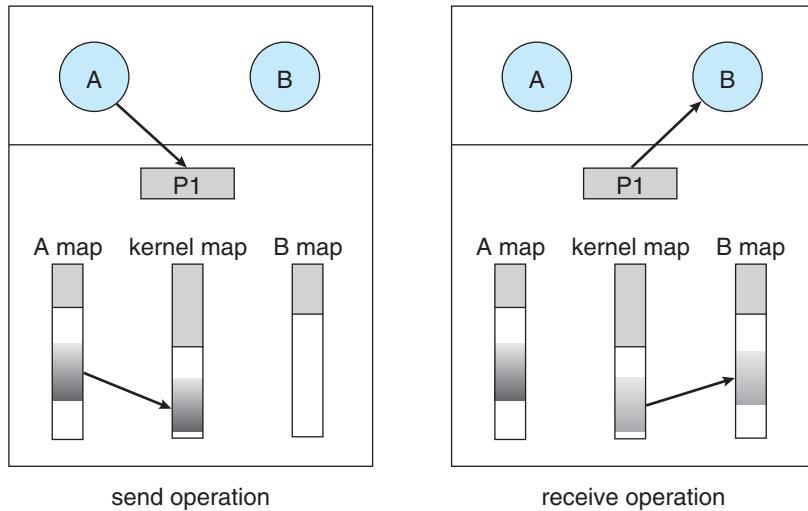
In Version 2.5, this operation was implemented in two phases. A pointer to a region of memory caused the kernel to map that region into its own space temporarily, setting the sender's memory map to copy-on-write mode to ensure that any modifications did not affect the original version of the data. When a message was received at its destination, the kernel moved its mapping to the receiver's address space, using a newly allocated region of virtual memory within that task.

In Version 3, this process was simplified. The kernel creates a data structure that would be a copy of the region if it were part of an address map. On receipt, this data structure is added to the receiver's map and becomes a copy accessible to the receiver.

The newly allocated regions in a task do not need to be contiguous with previous allocations, so Mach virtual memory is said to be *sparse*, consisting of regions of data separated by unallocated addresses. A full message transfer is shown in Figure D.6.

### D.5.3 The NetMsgServer

For a message to be sent between computers, the message's destination must be located, and the message must be transmitted to the destination. UNIX traditionally leaves these mechanisms to the low-level network protocols, which require the use of statically assigned communication endpoints (for example,



**Figure D.6** Mach message transfer.

the port number for services based on TCP or UDP). One of Mach’s tenets is that all objects within the system are location independent and that the location is transparent to the user. This tenet requires Mach to provide location-transparent naming and transport to extend IPC across multiple computers.

This naming and transport are performed by the [Network Message Server \(NetMsgServer\)](#), a user-level, capability-based networking daemon that forwards messages between hosts. It also provides a primitive network-wide name service that allows tasks to register ports for lookup by tasks on any other computer in the network. Mach ports can be transferred only in messages, and messages must be sent to ports. The primitive name service solves the problem of transferring the first port. Subsequent IPC interactions are fully transparent, because the NetMsgServer tracks all rights and out-of-line memory passed in intercomputer messages and arranges for the appropriate transfers. The NetMsgServers maintain among themselves a distributed database of port rights that have been transferred between computers and of the ports to which these rights correspond.

The kernel uses the NetMsgServer when a message needs to be sent to a port that is not on the kernel’s computer. Mach’s kernel IPC is used to transfer the message to the local NetMsgServer. The NetMsgServer then uses whatever network protocols are appropriate to transfer the message to its peer on the other computer. The notion of a NetMsgServer is protocol independent, and NetMsgServers have been built to use various protocols. Of course, the NetMsgServers involved in a transfer must agree on the protocol used. Finally, the NetMsgServer on the destination computer uses that kernel’s IPC to send the message to the correct destination task.

The ability to extend local IPC transparently across nodes is supported by the use of proxy ports. When a send right is transferred from one computer to another, the NetMsgServer on the destination computer creates a new port, or proxy, to represent the original port at the destination. Messages sent to this proxy are received by the NetMsgServer and are forwarded transparently

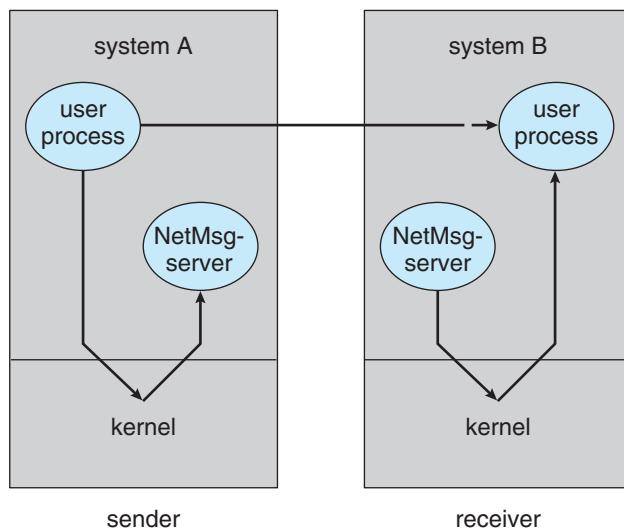
to the original port. This procedure is one example of how NetMsgServers cooperate to make a proxy indistinguishable from the original port.

Because Mach is designed to function in a network of heterogeneous systems, it must provide a way for systems to send data formatted in a way that is understandable by both the sender and the receiver. Unfortunately, computers differ in the formats they use to store various types of data. For instance, an integer on one system might take 2 bytes to store, and the most significant byte might be stored before the least significant one. Another system might reverse this ordering. The NetMsgServer therefore uses the type information stored in a message to translate the data from the sender's to the receiver's format. In this way, all data are represented correctly when they reach their destination.

The NetMsgServer on a given computer accepts RPCs that add, look up, and remove network ports from the NetMsgServer's name service. As a security precaution, a port value provided in an add request for a port must match that in the remove request for a thread to ask for a port name to be removed from the database.

As an example of the NetMsgServer's operation, consider a thread on node A sending a message to a port that happens to be in a task on node B. The program simply sends a message to a port to which it has a send right. The message is first passed to the kernel, which delivers it to its first recipient, the NetMsgServer on node A. The NetMsgServer then contacts (through its database information) the NetMsgServer on node B and sends the message. The NetMsgServer on node B presents the message to the kernel with the appropriate local port for node B. The kernel finally provides the message to the receiving task when a thread in that task executes a `msg_receive()` call. This sequence of events is shown in Figure D.7.

Mach 3.0 provides an alternative to the NetMsgServer as part of its improved support for NORMA multiprocessors. The NORMA IPC subsystem of Mach 3.0 implements functionality similar to the NetMsgServer directly in the



**Figure D.7** Network IPC forwarding by NetMsgServer.

Mach kernel, providing much more efficient internode IPC for multicomputers with fast interconnection hardware. For example, the time-consuming copying of messages between the NetMsgServer and the kernel is eliminated. Use of the NORMA IPC does not preclude use of the NetMsgServer; the NetMsgServer can still be used to provide Mach IPC service over networks that link a NORMA multiprocessor to other computers. In addition to the NORMA IPC, Mach 3.0 also provides support for memory management across a NORMA system and enables a task in such a system to create child tasks on nodes other than its own. These features support the implementation of a single-system-image operating system on a NORMA multiprocessor. The multiprocessor behaves like one large system rather than an assemblage of smaller systems (for both users and applications).

#### **D.5.4 Synchronization Through IPC**

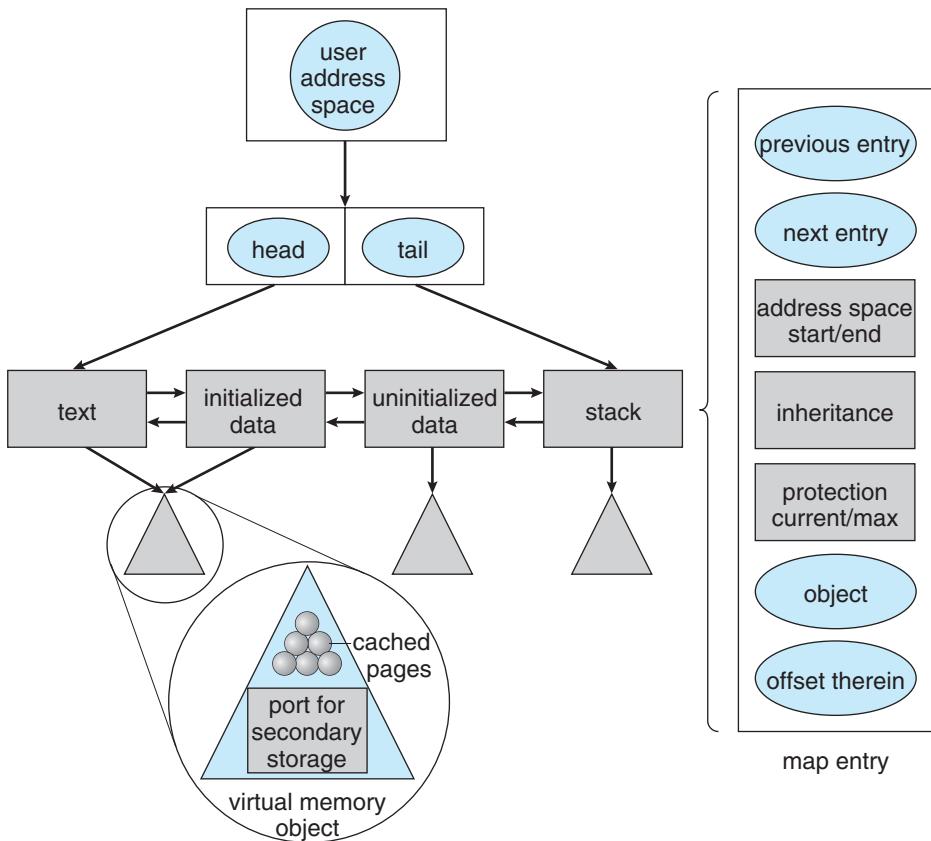
The IPC mechanism is extremely flexible and is used throughout Mach. For example, it may be used for thread synchronization. A port may be used as a synchronization variable and may have  $n$  messages sent to it for  $n$  resources. Any thread wishing to use a resource executes a receive call on that port. The thread will receive a message if the resource is available. Otherwise, it will wait on the port until a message is available there. To return a resource after use, the thread can send a message to the port. In this regard, receiving is equivalent to the semaphore operation `wait()`, and sending is equivalent to `signal()`. This method can be used for synchronizing semaphore operations among threads in the same task, but it cannot be used for synchronization among tasks, because only one task may have receive rights to a port. For more general-purpose semaphores, a simple daemon can be written to implement the same method.

### **D.6 Memory Management**

Given the object-oriented nature of Mach, it is not surprising that a principal abstraction in Mach is the memory object. Memory objects are used to manage secondary storage and generally represent files, pipes, or other data that are mapped into virtual memory for reading and writing (Figure D.8). Memory objects may be backed by user-level memory managers, which take the place of the more traditional kernel-incorporated virtual memory pager found in other operating systems. In contrast to the traditional approach of having the kernel manage secondary storage, Mach treats secondary-storage objects (usually files) as it does all other objects in the system. Each object has a port associated with it and may be manipulated by messages sent to its port. Memory objects—unlike the memory-management routines in monolithic, traditional kernels—allow easy experimentation with new memory-manipulation algorithms.

#### **D.6.1 Basic Structure**

The virtual address space of a task is generally sparse, consisting of many holes of unallocated space. For instance, a memory-mapped file is placed in some set of addresses. Large messages are also transferred as shared memory segments. For each of these segments, a section of virtual memory address is used to provide the threads with access to the message. As new items are mapped or



**Figure D.8** Mach virtual memory task address map.

removed from the address space, holes of unallocated memory appear in the address space.

Mach makes no attempt to compress the address space, although a task may fail (or crash) if it has no room for a requested region in its address space. Given that address spaces are 4 GB or more, this limitation is not currently a problem. However, maintaining a regular page table for a 4-GB address space for each task, especially one with holes in it, would use excessive amounts of memory (1 MB or more). The key to sparse address spaces is that page-table space is used only for currently allocated regions. When a page fault occurs, the kernel must check to see whether the page is in a valid region, rather than simply indexing into the page table and checking the entry. Although the resulting lookup is more complex, the benefits of reduced memory-storage requirements and simpler address-space maintenance make the approach worthwhile.

Mach also has system calls to support standard virtual memory functionality, including the allocation, deallocation, and copying of virtual memory. When allocating a new virtual memory object, the thread may provide an address for the object or may let the kernel choose the address. Physical memory is not allocated until pages in this object are accessed. The object's backing store is managed by the default pager (Section D.6.2). Virtual memory objects

are also allocated automatically when a task receives a message containing out-of-line data.

Associated system calls return information about a memory object in a task's address space, change the access protection of the object, and specify how an object is to be passed to child tasks at the time of their creation (shared, copy-on-write, or not present).

### D.6.2 User-Level Memory Managers

A secondary-storage object is usually mapped into the virtual address space of a task. Mach maintains a cache of memory-resident pages of all mapped objects, as in other virtual memory implementations. However, a page fault occurring when a thread accesses a nonresident page is executed as a message to the object's port. The concept that a memory object can be created and serviced by nonkernel tasks (unlike threads, for instance, which are created and maintained only by the kernel) is important. The end result is that, in the traditional sense, memory can be paged by user-written memory managers. When the object is destroyed, it is up to the memory manager to write back any changed pages to secondary storage. No assumptions are made by Mach about the content or importance of memory objects, so the memory objects are independent of the kernel.

In several circumstances, user-level memory managers are insufficient. For instance, a task allocating a new region of virtual memory might not have a memory manager assigned to that region, since it does not represent a secondary-storage object (but must be paged), or a memory manager might fail to perform pageout. Mach itself also needs a memory manager to take care of its memory needs. For these cases, Mach provides a default memory manager. The Mach 2.5 default memory manager uses the standard file system to store data that must be written to disk, rather than requiring a separate swap space, as in 4.3 BSD. In Mach 3.0 (and OSF/1), the default memory manager is capable of using either files in a standard file system or dedicated disk partitions. The default memory manager has an interface similar to that of the user-level ones, but with some extensions to support its role as the memory manager that can be relied on to perform pageout when user-level managers fail to do so.

Pageout policy is implemented by an internal kernel thread, the *pageout daemon*. A paging algorithm based on FIFO with second chance (Section 10.4.5) is used to select pages for replacement. The selected pages are sent to the appropriate manager (either user level or default) for actual pageout. A user-level manager may be more intelligent than the default manager, and it may implement a different paging algorithm suitable to the object it is backing (that is, by selecting some other page and forcibly paging it out). If a user-level manager fails to reduce the resident set of pages when asked to do so by the kernel, the default memory manager is invoked, and it pages out the user-level manager to reduce the user-level manager's resident set size. Should the user-level manager recover from the problem that prevented it from performing its own pageouts, it will touch these pages (causing the kernel to page them in again) and can then page them out as it sees fit.

If a thread needs access to data in a memory object (for instance, a file), it invokes the `vm_map()` system call. Included in this system call is a port that identifies the object and the memory manager that is responsible for the

region. The kernel executes calls on this port when data are to be read or written in that region. An added complexity is that the kernel makes these calls asynchronously, since it would not be reasonable for the kernel to be waiting on a user-level thread. Unlike the situation with pageout, the kernel has no recourse if its request is not satisfied by the external memory manager. The kernel has no knowledge of the contents of an object or of how that object must be manipulated.

Memory managers are responsible for the consistency of the contents of a memory object mapped by tasks on different machines. (Tasks on a single machine share a single copy of a mapped memory object.) Consider a situation in which tasks on two different machines attempt to modify the same page of an object at the same time. It is up to the manager to decide whether these modifications must be serialized. A conservative manager implementing strict memory consistency would force the modifications to be serialized by granting write access to only one kernel at a time. A more sophisticated manager could allow both accesses to proceed concurrently (for example, if the manager knew that the two tasks were modifying distinct areas within the page and that it could merge the modifications successfully at some future time). Most external memory managers written for Mach (for example, those implementing mapped files) do not implement logic for dealing with multiple kernels, due to the complexity of such logic.

When the first `vm_map()` call is made on a memory object, the kernel sends a message to the memory manager port passed in the call, invoking the `memory_manager_init()` routine, which the memory manager must provide as part of its support of a memory object. The two ports passed to the memory manager are a **control port** and a **name port**. The control port is used by the memory manager to provide data to the kernel—for example, pages to be made resident. Name ports are used throughout Mach. They do not receive messages but are used simply as points of reference and comparison. Finally, the memory object must respond to a `memory_manager_init()` call with a `memory_object_set_attributes()` call to indicate that it is ready to accept requests. When all tasks with send rights to a memory object relinquish those rights, the kernel deallocates the object's ports, thus freeing the memory manager and memory object for destruction.

Several kernel calls are needed to support external memory managers. The `vm_map()` call was just discussed. In addition, some commands get and set attributes and provide page-level locking when it is required (for instance, after a page fault has occurred but before the memory manager has returned the appropriate data). Another call is used by the memory manager to pass a page (or multiple pages, if read-ahead is being used) to the kernel in response to a page fault. This call is necessary since the kernel invokes the memory manager asynchronously. Finally, several calls allow the memory manager to report errors to the kernel.

The memory manager itself must provide support for several calls so that it can support an object. We have already discussed `memory_object_init()` and others. When a thread causes a page fault on a memory object's page, the kernel sends a `memory_object_data_request()` to the memory object's port on behalf of the faulting thread. The thread is placed in a wait state until the memory manager either returns the page in a `memory_object_data_provided()` call or returns an appropriate error to the kernel. Any of the pages that have

been modified, or any “precious pages” that the kernel needs to remove from resident memory (due to page aging, for instance), are sent to the memory object via `memory_object_data_write()`. *Precious pages* are pages that may not have been modified but that cannot be discarded as they otherwise would be because the memory manager no longer retains a copy. The memory manager declares these pages to be precious and expects the kernel to return them when they are removed from memory. Precious pages save unnecessary duplication and copying of memory.

In the current version, Mach does not allow external memory managers to affect the page-replacement algorithm directly. Mach does not export the memory-access information that would be needed for an external task to select the least recently used page, for instance. Methods of providing such information are currently under investigation. An external memory manager is still useful for a variety of reasons, however:

- It may reject the kernel’s replacement victim if it knows of a better candidate (for instance, MRU page replacement).
- It may monitor the memory object it is backing and request pages to be paged out before the memory usage invokes Mach’s pageout daemon.
- It is especially important in maintaining consistency of secondary storage for threads on multiple processors, as we show in Section D.6.3.
- It can control the order of operations on secondary storage to enforce consistency constraints demanded by database management systems. For example, in transaction logging, transactions must be written to a log file on disk before they modify the database data.
- It can control mapped file access.

### D.6.3 Shared Memory

Mach uses shared memory to reduce the complexity of various system facilities, as well as to provide these features in an efficient manner. Shared memory generally provides extremely fast interprocess communication, reduces overhead in file management, and helps to support multiprocessing and database management. Mach does not use shared memory for all these traditional shared-memory roles, however. For instance, all threads in a task share that task’s memory, so no formal shared-memory facility is needed within a task. However, Mach must still provide traditional shared memory to support other operating-system constructs, such as the UNIX `fork()` system call.

It is obviously difficult for tasks on multiple machines to share memory and to maintain data consistency. Mach does not try to solve this problem directly; rather, it provides facilities to allow the problem to be solved. Mach supports consistent shared memory only when the memory is shared by tasks running on processors that share memory. A parent task is able to declare which regions of memory are to be *inherited* by its children and which are to be readable –writable. This scheme is different from copy-on-write inheritance, in which each task maintains its own copy of any changed pages. A writable object is addressed from each task’s address map, and all changes are made to the same copy. The threads within the tasks are responsible for coordinating changes to memory so that they do not interfere with one another (by writing to the

same location concurrently). This coordination can be done through normal synchronization methods: critical sections or mutual-exclusion locks.

For the case of memory shared among separate machines, Mach allows the use of external memory managers. If a set of unrelated tasks wishes to share a section of memory, the tasks can use the same external memory manager and access the same secondary-storage areas through it. The implementor of this system would need to write the tasks and the external pager. This pager could be as simple or as complicated as needed. A simple implementation would allow no readers while a page was being written to. Any write attempt would cause the pager to invalidate the page in all tasks currently accessing it. The pager would then allow the write and would revalidate the readers with the new version of the page. The readers would simply wait on a page fault until the page again became available. Mach provides such a memory manager: the Network Memory Server (NetMemServer). For multicomputers, the NORMA configuration of Mach 3.0 provides similar support as a standard part of the kernel. This XMM subsystem allows multicomputer systems to use external memory managers that do not incorporate logic for dealing with multiple kernels. The XMM subsystem is responsible for maintaining data consistency among multiple kernels that share memory and makes these kernels appear to be a single kernel to the memory manager. The XMM subsystem also implements virtual copy logic for the mapped objects that it manages. This virtual copy logic includes both copy-on-reference among multicomputer kernels and sophisticated copy-on-write optimizations.

## D.7 Programmer Interface

A programmer can work at several levels within Mach. There is, of course, the system-call level, which, in Mach 2.5, is equivalent to the 4.3 BSD system-call interface. Version 2.5 includes most of 4.3 BSD as one thread in the kernel. A BSD system call traps to the kernel and is serviced by this thread on behalf of the caller, much as standard BSD would handle it. The emulation is not multithreaded, so it has limited efficiency.

Mach 3.0 has moved from the single-server model to support of multiple servers. It has therefore become a true microkernel without the full features normally found in a kernel. Rather, full functionality can be provided via emulation libraries, servers, or a combination of the two. In keeping with the definition of a microkernel, the emulation libraries and servers run outside the kernel at user level. In this way, multiple operating systems can run concurrently on one Mach 3.0 kernel.

An emulation library is a set of routines that lives in a read-only part of a program's address space. Any operating-system calls the program makes are translated into subroutine calls to the library. Single-user operating systems, such as MS-DOS and the Macintosh operating system, have been implemented solely as emulation libraries. For efficiency reasons, the emulation library lives in the address space of the program needing its functionality; in theory, however, it could be a separate task.

More complex operating systems are emulated through the use of libraries and one or more servers. System calls that cannot be implemented in the library are redirected to the appropriate server. Servers can be multithreaded for

improved efficiency; BSD and OSF/1 are implemented as single multithreaded servers. Systems can be decomposed into multiple servers for greater modularity.

Functionally, a system call starts in a task and passes through the kernel before being redirected, if appropriate, to the library in the task's address space or to a server. Although this extra transfer of control decreases the efficiency of Mach, this decrease is balanced to some extent by the ability of multiple threads to execute BSD-like code concurrently.

At the next higher programming level is the C threads package. This package is a run-time library that provides a C language interface to the basic Mach threads primitives. It provides convenient access to these primitives, including routines for the forking and joining of threads, mutual exclusion through mutex variables (Section D.4.2), and synchronization through use of condition variables. Unfortunately, it is not appropriate for the C threads package to be used between systems that share no memory (NORMA systems), since it depends on shared memory to implement its constructs. There is currently no equivalent of C threads for NORMA systems. Other run-time libraries have been written for Mach, including threads support for other languages.

Although the use of primitives makes Mach flexible, it also makes many programming tasks repetitive. For instance, significant amounts of code are associated with sending and receiving messages in each task that uses messages (which, in Mach, is most tasks). The designers of Mach therefore provide an interface generator (or stub generator) called *MIG*. MIG is essentially a compiler that takes as input a definition of the interface to be used (declarations of variables, types, and procedures) and generates the RPC interface code needed to send and receive the messages fitting this definition and to connect the messages to the sending and receiving threads.

## D.8 Summary

The Mach operating system is designed to incorporate the many recent innovations in operating-system research to produce a fully functional, technically advanced operating system.

The Mach operating system was designed with three critical goals in mind:

- Emulate 4.3 BSD UNIX so that the executable files from a UNIX system can run correctly under Mach.
- Have a modern operating system that supports many memory models and parallel and distributed computing.
- Design a kernel that is simpler and easier to modify than is 4.3 BSD.

As we have shown, Mach is well on its way to achieving these goals.

Mach 2.5 includes 4.3 BSD in its kernel, which provides the emulation needed but enlarges the kernel. This 4.3 BSD code has been rewritten to provide the same 4.3 functionality but to use the Mach primitives. This change allows the 4.3 BSD support code to run in user space on a Mach 3.0 system.

Mach uses lightweight processes, in the form of multiple threads of execution within one task (or address space), to support multiprocessing and parallel computation. Its extensive use of messages as the only communication method ensures that protection mechanisms are complete and efficient. By integrating messages with the virtual memory system, Mach also ensures that messages can be handled efficiently. Finally, by having the virtual memory system use messages to communicate with the daemons managing the backing store, Mach provides great flexibility in the design and implementation of these memory-object-managing tasks.

By providing low-level, or primitive, system calls from which more complex functions can be built, Mach reduces the size of the kernel while permitting operating-system emulation at the user level, much like IBM's virtual machine systems.

## Further Reading

The Accent operating system was described by [Rashid and Robertson (1981)]. A historical overview of the progression from an even earlier system, RIG, through Accent to Mach was given by [Rashid (1986)]. General discussions concerning the Mach model were offered by [Tevanian et al. (1989)].

[Accetta et al. (1986)] presented an overview of the original design of Mach. The Mach scheduler was described in detail by [Tevanian et al. (1987a)] and [Black (1990)]. An early version of the Mach shared memory and memory-mapping system was presented [Tevanian et al. (1987b)].

## Bibliography

**[Accetta et al. (1986)]** M. Accetta, R. Baron, W. Bolosky, D. B. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development", *Proceedings of the Summer USENIX Conference* (1986), pages 93–112.

**[Black (1990)]** D. L. Black, "Scheduling Support for Concurrency and Parallelism in the Mach Operating System", *IEEE Computer*, Volume 23, Number 5 (1990), pages 35–43.

**[Rashid (1986)]** R. F. Rashid, "From RIG to Accent to Mach: The Evolution of a Network Operating System", *Proceedings of the ACM/IEEE Computer Society, Fall Joint Computer Conference* (1986), pages 1128–1137.

**[Rashid and Robertson (1981)]** R. Rashid and G. Robertson, "Accent: A Communication-Oriented Network Operating System Kernel", *Proceedings of the ACM Symposium on Operating System Principles* (1981), pages 64–75.

**[Tevanian et al. (1987a)]** A. Tevanian, Jr., R. F. Rashid, D. B. Golub, D. L. Black, E. Cooper, and M. W. Young, "Mach Threads and the Unix Kernel: The Battle for Control", *Proceedings of the Summer USENIX Conference* (1987).

**[Tevanian et al. (1987b)]** A. Tevanian, Jr., R. F. Rashid, M. W. Young, D. B. Golub, M. R. Thompson, W. Bolosky, and R. Sanzi, "A UNIX Interface for Shared

Memory and Memory Mapped Files Under Mach”, Technical report, Carnegie-Mellon University (1987).

[**Tevanian et al. (1989)**] A. Tevanian, Jr., and B. Smith, “Mach: The Model for Future Unix”, *Byte* (1989).

---

## Credits

- Figure 1.14: From Hennesy and Patterson, *Computer Architecture: A Quantitative Approach, Third Edition*, ©2002, Morgan Kaufmann Publishers, Figure 5.3, p. 394. Reprinted with permission of the publisher.
- Figure 5.19: From Khanna/Sebree/Zolnowsky, “Realtime Scheduling in SunOS 5.0,” Proceedings of Winter USENIX, January 1992, San Francisco, California. Derived with permission of the authors.
- Figure 5.30 adapted with permission from Sun Microsystems, Inc.
- Figure 10.20: From *IBM Systems Journal*, Vol. 10, No. 3, ©1971, International Business Machines Corporation. Reprinted by permission of IBM Corporation.
- Figure 12.5: Based on a table from *Pentium Processor User’s Manual: Architecture and Programming Manual*, Volume 3, ©1993.
- Figure 14.8: From Leffler/McKusick/Karels/Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, ©1989 by Addison-Wesley Publishing Co., Inc., Reading, Massachusetts. Figure 7.6, p. 196. Reprinted with permission of the publisher.
- Figures 19.5, 19.6, and 19.8: From Halsall, *Data Communications, Computer Networks, and Open Systems, Third Edition*, ©1992, Addison-Wesley Publishing Co., Inc., Reading, Massachusetts. Figure 1.9, p. 14, Figure 1.10, p. 15, and Figure 1.11, p. 18. Reprinted with permission of the publisher.



---

# Index

4-byte pages, 363, 364  
32-byte memory, 363, 364  
50-percent rule, 359  
64-bit computing, 383

## A

ABI (application binary interface), 78-79  
aborting processes, 342  
absolute code, 352  
absolute path names, 546  
abstract data type (ADT), 277-278  
access, 539-541  
    anonymous, 605  
    controlling, 552-554  
    direct (relative), 539-541  
    effective access time, 397-398  
    kernel object, 884-885  
    lightweight directory-access protocol, 607, 884  
    memory, 15, 18, 19, 418-419, 498-500  
    process migration for, 753  
    and protection, 551  
    random-access devices, 502  
    random-access time, 450  
    read, 292  
    relative, 539-540  
    Remote Access Tool, 625  
    remote file, 764-767  
    security access tokens, 662  
    sequential, 539, 541  
    wireless access points, 736  
    write, 292  
access control:  
    discretionary, 684  
    in Linux, 816-818  
    MAC address, 745

    mandatory, 684-685  
    role-based, 683-684  
access-control lists (ACLs), 552, 555, 826  
accessed bits, 437  
access mask, 849  
access matrix, 675-685  
    defined, 675  
    implementation of, 679-682  
    and mandatory access control, 684-685  
    and revocation of access rights, 682-683  
    and role-based access control, 683-684  
access rights, 534, 673, 680, 682-683  
accounting, 110, 659, 788  
ACG (Arbitrary Code Guard), 827  
acknowledgment packet, 748  
ACLs, see access-control lists  
ACPI (advanced configuration and power interface), 516  
activation record, 107  
active directory, 607, 884  
acyclic graphs, 547  
acyclic-graph directories, 547-549  
additional-reference-bits algorithm, 409-410  
additional sense code, 512  
additional sense-code qualifie , 512  
address(es):  
    defined, 496  
    linear, 380, 382  
    logical, 353, 379  
    MAC, 745  
    physical, 354, 379  
    trusted, 638  
    virtual, 354  
address binding, 352-353  
address mapping, 456-457  
address resolution protocol (ARP), 745

- address space:**  
 logical vs. physical, 353-355  
 virtual, 390, 391, 799-800
- address-space identifier (ASIDs),** 366
- address-space layout randomization (ASLR),** 656, 827
- Address Window Extension (AWE) memory,** 894-895
- admission-control algorithms,** 230
- ADT (abstract data type),** 277-278
- advanced configuration and power interface (ACPI),** 516
- advanced encryption standard (AES),** 640
- advanced local procedure call (ALPC),** 138, 834
- advanced technology attachment (ATA) buses,** 456
- advisory file-locking mechanisms,** 535
- AES (advanced encryption standard),** 640
- affinity, processor,** 225-226
- age, page,** 800
- aging,** 213
- ahead-of-time (AOT) compilation,** 89, 90
- alertable threads,** 846
- allocation:**  
 buddy-system, 427, 428  
 committing, 852  
 contiguous, 356-360, 570-573  
 equal, 414  
 frame, 413-419  
 free frames before and after, 364  
 global, 415-418  
 indexed, 575-577  
 kernel memory, 426-430  
 linked, 573-575  
 local, 415-418  
 over-, 401  
 proportional, 414-415  
 resource, 57  
 of secondary storage, 570-578  
 slab, 427-430, 797-798
- Allocation (data structure),** 335, 336, 339
- allocation problem,** 358, 540, 571
- ALPC (advanced local procedure call),** 138, 834
- altitudes,** 863
- AMD64 architecture,** 382
- Amdahl's Law,** 164
- AMD virtualization technology (AMD-V),** 710-711
- amplification write,** 462
- analytic evaluation,** 245
- Andrew file system (OpenAFS),** 759
- Android operating system,** 89-91
- process hierarchy, 122-123
- protection domain, 675
- RPC, 151-153
- thread pools, 178
- TrustZone, 670, 671
- anomaly detection,** 656
- anonymous access,** 605
- anonymous memory,** 399, 469
- anonymous pipes,** 141-145
- AOT (ahead-of-time) compilation,** 89, 90
- APCs (asynchronous procedure calls),** 189-190, 846
- APFS (Apple File System),** 592
- API (application program interface),** 63-66. See also specific types
- Appending files** 551
- Apple File System (APFS),** 592
- application binary interface (ABI),** 78-79
- application component,** 151-152
- Application Container,** 868
- application containment,** 703, 718-719
- application frameworks layer (macOS and iOS),** 87
- application interface (I/O systems),** 500-508
- block and character devices, 503-504
- clocks and timers, 505-506
- network devices, 504-505
- nonblocking and asynchronous I/O, 506-507
- application layer (OSI model),** 742
- application programs (apps),** 4, 75, 823
- compatibility of, 830-831
- disinfection of, 658
- packaged, 859
- security of, 624
- specificity of, 77-79
- system services, 75
- user IDs for, 675
- application program interface (API),** 63-66. See also specific types
- application proxy firewalls,** 660
- application state,** 378
- Aqua interface,** 59, 87
- Arbitrary Code Guard (ACG),** 827
- architecture(s),** 15-21
- AMD64, 382
  - ARMv8, 383-384, 671, 672
  - big.LITTLE, 226-227
  - clustered systems, 19-21
  - IA-32, 379-382
  - IA-64, 382
  - multiprocessing, 124
  - multiprocessor systems, 16-19

- NFS, 614  
single-processor systems, 15-16  
von Neumann, 12  
x86-64, 382
- Arduino**, 70
- argument vector, 787  
armored viruses, 634  
ARMv8 architecture, 383-384, 671, 672  
ARP (address resolution protocol), 745  
arrays:  
  redundant, see RAID [redundant arrays of inexpensive disks]  
  storage, 472-473, 481
- ASICs**, 46
- ASIDs (address-space identifiers) 366
- ASLR** (address-space layout randomization), 656, 827
- assignment edge, 323
- asymmetric clustering, 19
- asymmetric encryption, 641, 645
- asymmetric encryption algorithm, 641
- asymmetric multiprocessing, 220
- asymmetry, in addressing, 129
- asynchronous cancellation, 190
- asynchronous devices, 502, 506-507
- asynchronous message passing, 130
- asynchronous procedure calls (APCs), 189-190, 846
- asynchronous threading, 169
- asynchronous writes, 585
- ATA buses, 456
- "at most once" functionality, 150
- atomic instructions, 266, 269
- atomic safe-save, 592
- atomic variables, 269-270
- attacks, 622  
  buffer-overflow, 628-631  
  code-injection, 628-631  
  code reuse, 827  
  denial-of-service, 622, 636  
  information leak, 827  
  man-in-the-middle, 623, 635, 645  
  replay, 622  
  with tunneling, 659-660  
  zero-day, 656
- attackers, 622
- attack surface, 624
- attributes, 551, 826, 875-876
- attribute-definitio table, 877
- auditing, 659
- audit trail, 669
- augmented-reality applications, 42
- authentication:  
  breaching of, 622  
  and encryption, 641-644  
  in Linux, 816  
  multifactor, 653  
  two-factor, 652  
  user, 648-653
- automatic working-set trimming, 438
- automount feature, 763
- autoprobes, 785
- availability, breach of, 622
- Available (data structure), 334, 336, 338
- AWE memory, 894-895
- B**
- back door, 503, 626, 627, 638
- background class, 186
- background processes, 74-75, 115, 123, 215, 241
- backing store, 376
- back-pointers, 682
- backups, 588-589
- bad blocks, 466-467
- bad-cluster file, 877
- bad page, 856
- balance, in multicore programming, 163
- balanced binary search trees, 38
- balloon memory manager, 721
- bandwidth, 457
- banker's algorithm, 333-337
- barriers, memory, 265-266
- based sections, 852
- base fil record, 876
- base register, 351-352
- bash (bourne-again shell), 58, 783
- basic file systems, 564, 565
- Bayes' theorem, 657
- BCC (BPF Compiler Collection), 98-100
- Belady's anomaly, 406
- best-fi strategy, 358, 359
- BGP (Border Gateway Protocol), 745
- big cores, 226-227
- big data, 22
- big-endian, 150
- big.LITTLE architecture, 226-227
- binary format, 785
- binary general tree, 38
- binary search trees, 38, 39
- binary semaphore, 273
- binary translation, 708-710
- binary trees, 38, 39
- binders, 151
- binding, 352
- biometrics, 652-653
- Bionic standard, 90

- BIOS**, 94
- bit(s)**:
  - accessed, 437
  - additional-reference-bits algorithm, 409-410
  - contiguous, 432-433
  - defined, 12
  - mode, 24
  - modify (dirty), 402
  - reference, 409
  - setuid, 674-675
  - 64-bit computing, 383
  - valid-invalid, 368-369
- bit-level striping**, 475
- BitLocker**, 863
- bitmaps (bit vectors)**, 38-39, 579, 877
- BKL**, running on, 794
- blade servers**, 18-19
- block(s)**, 186
  - bad, 466-467
  - boot, 94, 464-466, 566
  - boot control, 566
  - defined, 564
  - direct, 576
  - disk, 40
  - file-control, 565, 567
  - index, 575-577
  - indirect, 576, 577
  - logical, 456
  - process control, 109-110
  - thread building, 186-188
  - thread environment, 889-890
  - TRIMing unused, 581-582
  - virtual address control, 865
  - volume control, 566
- block ciphers**, 639
- block devices**, 502-504, 810-811
- block device interface**, 503
- block groups**, 806
- blocking**, indefinite 213
- blocking I/O**, 506
- blocking (synchronous) message passing**, 130
- block-interleaved distributed parity**, 477-478
- block-level striping**, 475
- block number, relative**, 540
- block started by symbol (bss) field** 108
- block synchronization**, 305
- body (value)**, 187
- boot block**, 94, 465-466, 566
- boot control block**, 566
- boot disk (system disk)**, 465
- boot file** 877
- booting**, 86, 94-95, 863-864, 872-874
- boot loaders**, see **bootstrap programs**
- boot partition**, 465
- boot sector**, 466
- bootstrap port**, 136
- bootstrap programs (boot loaders, bootstrap loaders)**, 11, 70, 94, 465, 601
- bootstrap server**, 136
- boot viruses**, 632, 633
- Border Gateway Protocol (BGP)**, 745
- bottlenecks**, 95
- bottom half (interrupt service routines)**, 793-794
- bounded buffer**, 126
- bounded-buffer problem**, 290, 304
- bounded capacity (of queue)**, 131-132
- bounded waiting**, 261
- bourne-again shell (bash)**, 58, 783
- BPF Compiler Collection (BCC)**, 98-100
- breach of availability**, 622
- breach of confidentialit** , 622
- breach of integrity**, 622
- bridging**, 723
- broadcasting**, 745
- brokers**, 837
- browser process**, 124
- BSD UNIX**, 49-50
- bss (block started by symbol) field**, 108
- B+ tree (NTFS)**, 876
- buddies**, 427
- buddy heap (Linux)**, 796
- buddy system (Linux)**, 796
- buddy-system allocation**, 427, 428
- buffers**:
  - bounded and unbounded, 126
  - bounded-buffer problem, 290, 304
  - circular, 587, 716-717
  - defined, 509
  - translation look-aside, 365-368, 376, 384, 855
- buffer cache**, 583-585
- buffering**, 131-132, 412, 499, 509-510
- buffer-overflo** attacks, 628-631
- bugs**, 66
- bug bounty programs**, 826
- bus(es)**, 7, 456
  - advanced technology attachment, 456
  - defined, 490-491
  - eSATA, 456
  - expansion, 490
  - fibre channel, 456
  - I/O, 456
  - PCIe, 490

serial ATA, 456  
 serial-attached SCSI, 456, 490  
 universal serial, 456  
**busy waiting**, 272, 493-494  
**byte**, 11  
**bytecode**, 727  
**byte stream**, 748

**C**

**caches**, 583-586  
 buffer, 583-584  
 defined, 510  
 in Linux, 797, 798  
 location of, 765-766  
 as memory buffer, 350  
 page, 583, 798  
 and performance improvement, 583-586  
 policy for updating, 766-767  
 slabs in, 427, 428  
 unified buffer, 583-585  
**cache coherency**, 32  
**cache-consistency problem**, 765  
**cache management**, 30-31  
**cache manager** (Windows 10), 864-866  
**caching**, 30-31, 510-511  
 basic scheme, 764-765  
 client-side, 883  
 double, 584  
 write-back, 766  
**cancellation, thread**, 190-192  
**cancellation points**, 191  
**Canonical**, 779  
**capability(-ies)**, 680, 682-683, 685-686, 697  
**capability-based protection systems**,  
 685-687  
**capability lists**, 680-681  
**capability systems**, 826  
**capacity**, of queue, 131-132  
**cascading termination**, 121  
**catching interrupts**, 9, 494  
**CAV (constant angular velocity)**, 457  
**cd command**, 751  
**central processing units**, 18, 318. See also  
 entries beginning CPU  
**Ceph**, 484  
**certificat authorities**, 644  
**CET (Control-flo Enforcement**  
 Technology), 828  
**CFG (Control-Flow Guard)**, 827  
**CFQ scheduler**, 461, 811  
**CFS**, see Completely Fair Scheduler  
**CFS (clustered file system)**, 768  
**challenging passwords**, 652

**change journal (Windows 10)**, 879  
**character devices (Linux)**, 810-812  
**character-stream devices**, 502, 504  
**character-stream interface**, 504  
**checksums**, 462, 746  
**children**, 38, 111  
**chip multithreading (CMT)**, 222, 223  
**chipsets**, 835  
**Chrome**, 124  
**CIFS (common Internet file system)**, 607,  
 880  
**ciphers**, 639, 640  
**circular buffers**, 587, 716-717  
**circularly linked lists**, 37, 38  
**circular SCAN (C-SCAN) scheduling**  
 algorithm, 460  
**circular-wait condition (deadlocks)**, 321,  
 328-330  
**claim edge**, 333  
**classes (Java)**, 694  
**class loader**, 727  
**cleanup handler**, 191  
**clearing interrupts**, 9, 494  
**CLI (command-line interface)**, 56  
**C library**, see **libc**  
**client(s)**, 73  
 in client-server model, 606  
 defined, 757  
 diskless, 762  
 in distributed systems, 734  
 thin, 40  
**client-initiated approach to verifying**  
 cached data, 767  
**client interface**, 757  
**client-server DFS model**, 758-759  
**client-server distributed system**, 734  
**client-server model**, 606, 758-759, 861-862  
**client-server systems**, 42-43, 734  
**client-side caching (CSC)**, 883  
**client systems**, 42  
**clocks**, 505-506  
**clock algorithm**, 410-411  
**clock owner**, 837  
**clones**, 591, 705  
**clone()** system call, 195-196  
**closed-source operating systems**, 46  
**closures**, 174, 186  
**cloud computing**, 44-45, 706  
**cloud storage**, 471, 751  
**clusters**, 19-21, 464, 574, 875  
**cluster-based DFS model**, 758-760  
**clustered file system (CFS)**, 768  
**clustered page tables**, 374  
**clustered systems**, 19-21

- clustering, 19, 20, 438
- CLV (constant linear velocity), 456-457
- CMT (chip multithreading), 222, 223
- coarse-grained multithreading, 222
- coaxial cables, 736
- Cocoa framework, 87
- Cocoa Touch, 87
- code:**
  - absolute, 352
  - additional sense, 512
  - byte-, 727
  - error-correction, 462-463
  - injection of, 628-631
  - kernel, 261
  - message-authentication, 643
  - position-independent, 803
  - reentrant (pure), 370
  - relocatable, 353
- code-injection attack, 628-631
- code integrity module (Windows 10), 828
- code reuse attacks, 827
- code review, 627, 628
- code signing, 644, 690
- codewords, 697
- COM (Component Object Model), 882
- com (top-level domain), 739
- combined scheme index block, 576
- command interpreter, 58-59
- command-line interface (CLI), 56
- committing allocations, 852
- Common Criteria, 869
- common Internet file system (CIFS), 607, 880
- common name, 647
- communication(s):**
  - direct, 128
  - indirect, 129
  - inter-computer, 522
  - interprocess, see **interprocess communication [IPC]**
  - network, 738-749
    - communication protocols, 741-745
    - and naming/name resolution, 738-741
    - TCP/IP example, 745-746
    - UDP and TCP transport protocols, 746-749
  - as operating system service, 57
  - secure, with symmetric encryption, 639, 640
  - systems programs for, 74
- communication links**, 128
- communication ports**, 138
- communication protocols**, 741-745
- communication system calls**, 72-73
- compaction, 360, 572
- compare\_and\_swap() instruction, 267-269
- compartmentalization, 669
- compiler-based enforcement, 691-693
- compile time, 352
- Completely Fair Queuing (CFQ) scheduler, 461, 811
- Completely Fair Scheduler (CFS), 236, 237, 790
- complex messages, 136
- Component Object Model (COM), 882
- compression, 425-426, 757, 858, 878-879
- compression ratio, 426
- compression units, 878
- computational kernels, 833
- computation migration, 752
- computation speedup, 123, 735, 753
- computer programs**, see **application programs**
- computer system(s):**
  - architecture of, 15-21
  - clustered systems, 19-21
  - multiprocessor systems, 16-19
  - single-processor systems, 15-16
  - distributed systems, 35-36
  - firewalling to protect, 659-660
  - I/O structure in, 14-15
  - operating system viewed by, 5
  - organization, 7-15
    - interrupts, 8-11
    - I/O structure, 14-15
    - storage structure, 11-14
  - process management in, 27-28
  - protection in, 33-34
  - real-time embedded systems, 45-46
  - secure, 622
  - security in, 33-34
  - storage in, 11-14
  - storage management in, 30, 32
  - threats to, 634-637
- compute-servers system**, 42-43
- computing:**
  - 64-bit, 383
  - cloud, 44-45, 706
  - high-performance, 20
  - mobile, 41-42
  - peer-to-peer, 43-44
  - safe, 658
  - thin-client, 874-875
  - traditional, 40-41
- computing environments**, 40-46
  - client-server computing, 42-43
  - cloud computing, 44-45
  - mobile computing, 41-42

- peer-to-peer computing, 43-44
- real-time embedded systems, 45-46
- traditional, 40-41
- virtualization, 34-35
- concurrency, 163**
- Concurrency Runtime (ConcRT), 241-242, 890**
- concurrent dispatch queue, 185
- conditional-wait construct, 281
- condition variables, 278, 279, 302-303, 309-311, 889
- confidentiality, breach of, 622
- confinement problem, 678
- conflict phase (of dispatch latency), 229
- conflict-resolution mechanism (Linux), 784, 785
- congestion control, 749
- Connected Standby, 837**
- connectionless protocols, 747
- connectionless (UDP) sockets, 147
- connection-oriented protocols, 748
- connection-oriented (TCP) sockets, 147
- connection ports, 138
- consistency, of distributed file systems, 767
- consistency checker, 586-587
- consistency checking, 586-587
- consistency semantics, 608-609
- consolidation, 706
- constant angular velocity (CAV), 457
- constant linear velocity (CLV), 456-457
- consumer process, 126-127, 290, 291, 559-560
- containers, 592, 718, 719
- container objects (Windows 10), 664
- containment, application, 703, 718-719
- contaminants, 344
- contented locks, 271
- content-addressable storage, 484
- contention scope, 217-218
- context (of process), 114
- context (of thread), 194
- context switches, 114-115, 204
- contiguous allocation, 356-360, 570-573
- contiguous bit, 432-433
- Control-flow Enforcement Technology (CET), 828**
- Control-Flow Guard (CFG), 827**
- controlled access, 552-554
- controller(s), 456**
  - defined, 491
  - device, 456
  - direct-memory-access, 498
  - fibre channel bus, 491
  - host, 456
- control partitions, 714**
- control programs, 5**
- control register, 492**
- convenience, 1**
- convoy effect, 207**
- cooperating processes, 123, 257**
- cooperative scheduling, 202**
- coordination, among processes, 260**
- copy-on-write technique, 399-401, 853**
- copy rights, access matrix with, 677**
- copy semantics, 510**
- core(s), 15-16, 18**
  - big and little, 226-227
  - dual-core design, 17, 18
  - multicore processors, 221-224
  - multicore programming, 162-166
  - multicore systems, 16-18
  - scheduling processes to run on, 199
- core dump, 95-96**
- core frameworks (macOS and iOS), 87**
- CoreUI, 825**
- counts, 533, 534**
- counters, 96-97**
  - LRU page replacement with, 408
  - program, 27, 106, 109
  - timestamp, 845
- counting, 580**
- counting-based page replacement algorithm, 411-412**
- counting semaphore, 273**
- C program, memory layout in, 108**
- CPUs (central processing units), 18, 318**
- CPU-bound processes, 112**
- CPU burst, 201**
- CPU-I/O burst cycle, 201**
- CPU registers, 110**
- CPU scheduler, 113-114, 201**
- CPU scheduling, 24, 199-251**
  - about, 201
  - algorithms for, 205-217
    - evaluation of, 244-249
    - first-come, first-served scheduling of, 206-207
  - multilevel feedback-queue scheduling of, 216-217
  - multilevel queue scheduling of, 214-216
  - priority scheduling of, 211-214
  - round-robin scheduling of, 209-211
  - shortest-job-first scheduling of, 207-209
- criteria, 204-205**
- dispatcher, role of, 203-204**

- and I/O-CPU burst cycle, 201  
 multi-processor scheduling, 220-227  
   approaches to, 220-221  
   heterogeneous multiprocessing, 226-227  
   and load balancing, 224-225  
   and multicore processors, 221-224  
   and processor affinity, 225-226  
 operating-system examples, 234-244  
   Linux scheduling, 234-239  
   Solaris scheduling, 242-244  
   Windows scheduling, 239-242  
 preemptive vs. nonpreemptive  
   scheduling, 202-203  
 real-time, 227-234  
   earliest-deadline-first scheduling, 232-233  
   and minimizing latency, 227-229  
   POSIX real-time scheduling, 233-234  
   priority-based scheduling, 229-230  
   proportional share scheduling, 233  
   rate-monotonic scheduling, 230-232  
   thread scheduling, 217-219  
   virtual machines, 720  
**CPU-scheduling information (PCBs)**, 110  
**CPU utilization**, 204  
**crashes**, 96  
**crash dumps**, 96  
**CRCs (cyclic redundancy checks)**, 462  
**creation:**  
   of files, 532, 542  
   of processes, 116-121  
**credentials**, 787  
**critical sections**, 260  
**critical-section object**, 297, 888  
**critical-section problem**, 260-270  
   Peterson's solution to, 262-265  
   and semaphores, 272-276  
   and synchronization hardware, 265-270  
**cryptography**, 637-648  
   defined, 638  
   and encryption, 638-645  
     asymmetric encryption, 641  
     authentication, 641-644  
     key distribution, 644-645  
     symmetric encryption, 639-640  
   implementation of, 645-646  
   TLS example of, 646-648  
**CSC (client-side caching)**, 883  
**C-SCAN scheduling**, 460  
**C-SCAN scheduling algorithm**, 460  
**C shell**, 58  
**ctfs (file system)**, 598  
**cumulative ACK**, 748  
**current directory**, 546  
**current-file-position pointer**, 532  
**cycle stealing**, 499  
**cyclic redundancy checks (CRCs)**, 462  
**cylinder (hard disk drive)**, 450  
**cylinder groups**, 806
- D**
- d (page offset)**, 360  
**DAC (discretionary access control)**, 684  
**daemons**, 22, 781  
**daemon processes**, 690  
**daisy chain**, 490  
**DAM (Desktop Activity Moderator)**, 837  
**dark web**, 634  
**Darwin operating system**, 85, 88, 687-688  
**data attributes**, 875  
**databases**, 341, 842, 856  
**data dependency**, 164  
**data-encryption standard (DES)**, 639  
**Data Execution Prevention (DEP)**, 827  
**datagrams**, 743  
**data-in register**, 492  
**data-link layer**, 742  
**data-link layer protocol**, 645  
**data loss, mean time of**, 474  
**data migration**, 751-752  
**data-out register**, 492  
**data parallelism**, 165, 166  
**data passing, between processes**, 813  
**data section (of process)**, 106  
**data splitting**, 164  
**data striping**, 475  
**data view attribute**, 862  
**DCOM**, 882  
**DDoS attacks**, 636  
**deadline scheduler**, 460, 461  
**deadlock(s)**, 283-284, 317-343  
   avoidance of, 326, 330-337  
     with banker's algorithm, 333-337  
     with resource-allocation-graph  
       algorithm, 333  
     with safe-state algorithm, 331-333  
   characterization, 321-326  
   defined, 317  
   detection of, 337-341  
   methods for handling, 326-327  
   in multithreaded applications, 319-321  
   necessary conditions for, 321-323  
   prevention of, 326-330  
   recovery from, 341-343  
   system model for, 318-319

- system resource-allocation graphs for describing, 321-323
- Debian**, 779
- debuggers, 66
- debugging, 95-100, 165
- dedicated devices, 502
- deduplication, 757
- default access rights, 680
- default heap, 893
- default signal handlers, 189
- defense in depth, 653, 669
- deferred cancellation, 190
- deferred procedure calls (DPCs), 841, 847
- degree of multiprogramming, 112
- delayed revocation, 682
- delayed-write policy, 766
- deleting files, 532, 542, 551
- demand paging**, 392-399, 430-436
  - basic mechanism, 393-396
  - defined, 393
  - free-frame list, 396-397
  - with inverted page tables, 433
  - and I/O interlock, 434-436
  - and page size, 431-432
  - and performance, 397-399
  - and prepaging, 430-431
  - and program structure, 433-434
  - pure, 395
  - and TLB reach, 432-433
- demand-zero memory**, 799
- demilitarized zone (DMZ)**, 659
- denial-of-service (DOS) attacks**, 622, 636
- dentry objects**, 605, 804, 805
- DEP (Data Execution Prevention)**, 827
- DES (data-encryption standard)**, 639
- design of operating systems**, 79-80
  - distributed systems, 753-757
  - Linux, 780-783
  - Windows 10, 826-838
    - application compatibility, 830-831
    - dynamic device support, 837-838
    - energy efficiency, 836-837
    - extensibility, 833-834
    - international support, 835-836
    - performance, 831-833
    - portability, 838-839
    - reliability, 828-829
    - security, 826-828
- desktop**, 59
- Desktop Activity Moderator (DAM)**, 837
- Desktop Window Manager (DWM)**, 825
- detection-algorithm usage (deadlock)**, 340-341
- deterministic modeling**, 245-247
- development kernels (Linux)**, 777
- device controllers**, 456. See also **I/O**
- device directory**, see **directory(-ies)**
- device drivers**, 7, 490, 785
- Device Guard**, 828
- device-management system calls**, 71-72
- device objects**, 863
- device reservation**, 511
- device stacks**, 862
- device-status table**, 508-509
- DFSs**, see **distributed file systems**
- digital certificates** 644
- digital signatures**, 643, 828
- digital-signature algorithm**, 643
- dining-philosophers problem**, 293-295
- dir command**, 751
- direct access (files)** 539-541
- direct blocks**, 576
- direct communication**, 128
- Direct-Compute**, 825
- direct I/O**, 504
- direct memory access (DMA)**, 15, 498-500
- direct-memory-access (DMA) controller**, 498
- directory(-ies)**, 541-550
  - active, 607, 884
  - acyclic-graph, 547-549
  - current, 546
  - fast sizing of, 592
  - file-system interface, 541-550
  - general graph, 549-550
  - implementation of, 568-570
  - lightweight directory-access protocol, 607
  - listing, 542
  - master file, 543
  - page, 381, 853
  - protecting, 554
  - root, 877
  - single-level, 542-543
  - tree-structured, 545-547
  - two-level, 543-545
  - user file, 543
- directory object**, 850
- direct virtual memory access (DVMA)**, 500
- dirty bits (modify bits)**, 402
- discretionary access control (DAC)**, 684
- disinfection, program**, 658
- disk(s)**. See also **mass-storage structure**;
  - RAID (redundant arrays of inexpensive disks)**
- boot (system)**, 465
- mini-**, 704

- raw, 413, 464, 601
- disk arm**, 450
- disk blocks**, 40
- disk image**, 723-724
- diskless clients**, 762
- Disk Management tool**, 465
- disk-scheduling algorithms**, 460-461
- dispatched process**, 112
- dispatchers**, 203-204, 239, 840-841
- dispatcher database**, 842
- dispatcher objects**, 297, 845-846
- dispatching interrupts**, 9, 494
- dispatch latency**, 203, 228, 229
- dispatch queue**, 185
- distinguished name**, 647
- Distributed Denial-of-Service (DDoS)**
  - attacks, 636
- distributed file systems (DFSs)**, 605, 757-768
  - client-server model, 758-759
  - cluster-based model, 759-761
  - defined, 757
  - implementation techniques, 763-764
  - naming in, 761-764
  - remote file access in, 764-767
  - trends in, 767-768
  - Windows 10, 883
- distributed information systems**
  - (distributed naming services), 607
- distributed lock manager (DLM)**, 21
- distributed operating systems**, 749-753
- distributed systems**, 35-36
  - advantages of, 733-735
  - defined, 733
  - design issues in, 753-757
  - distributed file systems, 757-768
    - client-server model, 758-759
    - cluster-based model, 759-761
    - defined, 757
    - implementation techniques, 763-764
    - naming in, 761-764
    - remote file access in, 764-767
    - trends in, 767-768
  - distributed operating systems, 749-753
- distributions (GNU/Linux)**, 48
- DLLs (dynamically linked libraries)**, 76, 355-356
- DLM (distributed lock manager)**, 21
- DMA (direct memory access)**, 15, 498-500
- DMA-acknowledge**, 499
- DMA controller**, 498
- DMA-request**, 499
- DMZ (demilitarized zone)**, 659
- DNS (domain-name system)**, 607, 739-740
- dockers**, 719
- document(s)**
  - File System Hierarchy Standard, 778-779
  - living, 653
- domains**
  - capability lists for, 680-681
  - protection, 671-675, 711
  - public domain software, 779-780
  - scheduling, 238
  - security, 659
  - Windows 10, 884
- domain-name system (DNS)**, 607, 739-740
- domain switching**, 673, 674
- DOS attacks**, 622, 636
- double buffering**, 499, 509
- double caching**, 584
- double indirect blocks**, 576
- doubly linked lists**, 37
- down time**, 572
- DPCs (deferred procedure calls)**, 841, 847
- DRAM (dynamic random-access memory)**, 11
- drive formatting**, 463, 464
- drive mirroring**, 476
- driver end (STREAM)**, 519
- driver objects**, 863
- driver-registration system (Linux)**, 784, 785
- Drive Writes Per Day (DWPD)**, 453
- dropped packets**
  - TCP transfer with, 748, 749
  - UDP transfer with, 747-748
- dual-booted systems**, 601
- dual-core design**, 17, 18
- dual-mode operation**, 24-25
- DVMA (direct virtual memory access)**, 500
- DWM (Desktop Window Manager)**, 825
- DWPD (Drive Writes Per Day)**, 453
- dynamically linked libraries (DLLs)**, 76, 355-356
- dynamic device support (Windows 10)**, 837-838
- dynamic linking**, 803
- dynamic loading**, 355
- dynamic protection**, 673
- dynamic random-access memory (DRAM)**, 11
- dynamic storage-allocation problem**, 358, 571
- dynamic tick**, 836-837

earliest-deadline-firs (EDF) scheduling, 232-233  
 ease of use, 4, 822  
 easily remembered passwords, 651  
 eBPF tracing tool, 99, 100  
 ec2, 44  
 ECC (error-correction code), 462-463  
 economic benefits of multithreaded processes, 162  
 EDF (earliest-deadline-first scheduling, 232-233  
 edu (top-level domain), 739  
 effective access time, 397-398  
 effective capabilities, 685  
 effective memory-access time, 367  
 effective transfer rates, 451, 486  
 effective UID, 34  
 efficienc , 1, 582-583, 692, 836-837  
 electrical storage systems, 14  
 elevator algorithm, see SCAN scheduling algorithm  
 ELF (Executable and Linkable Format), 76-77, 801, 802  
 embedded computers, 5  
 empty processes, 123  
 empty slabs, 429, 798  
 emulation, 34, 717-718  
 emulators, 703  
 encapsulation (Java), 696  
 encrypted viruses, 633  
 encryption, 638-645  
     asymmetric, 641, 645  
     authentication, 641-644  
     defined, 638  
     key distribution, 644-645  
     public-key, 641  
     symmetric, 639-640  
 energy efficienc , 836-837  
 enhanced second-chance page-replacement algorithm, 410-411  
 entitlements (Darwin), 686-687  
 entry section, 260  
 entry set, 303, 304, 307  
 environment:  
     computing, 40-46  
     kernel, 88  
     operating system as, 4  
     programming, 703, 717  
     run-time, 64-65  
     thread environment blocks, 889-890  
 environment vector, 787  
 equal allocation, 414  
 errors, 462-463, 467, 511-512  
 error-correcting organization, 476-477  
 error-correction code (ECC), 462-463  
 error detection, 57, 462  
 error handling, 511-512  
 eSATA buses, 456  
 escalate privileges, 34  
 escape (operating systems), 503  
 Ethernet packets, 745-746  
 events, 297  
 event latency, 227-228  
 event objects (Windows 10), 845  
 event tracing, 822  
 event-vector table, 11  
 eVM, 729  
 "exactly once" functionality, 150  
 exceptions, 22, 497, 847-848  
 exception dispatcher, 847  
 exclusive locks, 534  
 exec() system call, 188, 786-789  
 Executable and Linkable Format, see ELF  
 executable files 75, 107, 530  
 executing files, 551  
 execution of user programs, 801-803  
 execution time, 353  
 executive (Windows 10), 848-874  
     booting, 872-874  
     cache manager, 864-866  
     facilities for client-server computing, 861-862  
     I/O manager, 862-864  
     object manager, 849-851  
     plug-and-play manager, 869-870  
     power manager, 870-871  
     process manager, 858-860  
     registry, 871-872  
     security reference monitor, 866-869  
     virtual memory manager, 851-858  
 exit section, 260  
 exit() system call, 121-122  
 expansion bus, 490  
 exponential average, 208  
 export list, 612  
 ext2 (second extended file system), 805  
 ext3 (third extended file system), 805-807  
 ext4 (fourth extended file system), 805  
 extended file attributes, 531  
 extended file system (extfs), 566, 805  
 extensibility, of Windows 10, 833-834  
 extent (contiguous space), 572  
 external data representation (XDR), 150  
 external fragmentation, 359-360, 571-572  
 extfs (extended file system), 566, 805

- failure(s), 473, 474, 754-756
- failure analysis, 95-96
- failure modes (remote fil systems), 607-608
- fairness parameter, 307
- fair scheduling, 791
- false negatives, 656
- false positives, 656
- fast directory sizing, 592
- fast I/O mechanism, 865-866
- fast-user switching, 825, 874-875
- FAT (file-allocatio table), 574-575
- fault, page, 394-395
- fault tolerance, 19, 754
- fault-tolerant systems, 754
- FC (fibe channel), 470
- FCB (file-contro block), 565, 567
- FC buses, 456
- FC bus controller, 491
- FCFS scheduling, 458, 459
- FCFS scheduling algorithm, 206-207, 458
- fd (fil descriptor), 568, 788
- fences, memory, 266
- fibers 241, 889-890
- fibe optic cables, 736
- fibr channel (FC), 470
- fibr channel (FC) buses, 456
- fibr channel (FC) bus controller, 491
- fidelit , 704
- FIFO, 38
- FIFO page replacement algorithm, 404-406
- 50-percent rule, 359
- file(s) 29-30, 529-530. See also
  - directory(-ies); specific types
  - appending, 551
  - attributes of, 530-531
  - defined, 255, 527, 529
  - deleting, 532, 542, 551
  - executing, 551
  - internal structure of, 537-539
  - locking, 534-536
  - opening, 532
  - operations on, 532-536
  - paging, 851
  - reading, 532, 551
  - renaming, 542
  - searches for, 541, 542
  - truncating, 532
  - writing, 532, 551
- file-allocatio table (FAT), 574-575
- file-contro block (FCB), 565, 567
- fil descriptor (fd), 568, 788
- fil handle, 568
- fil info window (macOS), 531
- fil management, 74
- file-managemen system calls, 71
- fil mapping, 555, 557
- fil migration, 761-762
- fil modification 74
- fil objects, 605, 804-805, 862
- file-ope count, 534
- file-organizatio module, 565
- fil pointers, 534
- fil reference, 876
- fil replication, 761
- file-serve system, 43
- fil sessions, 608
- fil sharing 602-603
- fil systems, 597-616
  - Andrew, 759
  - Apple, 592
  - basic, 564, 565
  - clustered, 768
  - common Internet, 607, 880
  - consistency semantics in, 608-609
  - defined, 564
  - distributed, see **distributed fil systems** (DFSs)
  - extended, 566, 805
  - file sharing, 602-603
  - Google, 759-761
  - Hadoop, 484
  - Linux, 803-810
    - ext3 file system, 805-807
    - journaling, 808
    - /proc file system, 808-810
    - virtual, 804-805
  - log-based transaction-oriented, 587-588
  - logical, 565
  - mounting of, 598-602
  - network, 610-615
  - network file, 610-615, 759
  - operations, 566-568
  - parallel, 768
  - partitions in, 601-602
  - registration of, 785
  - remote, 605-608
  - Solaris, 482-484, 597, 599
  - special-purpose, 597-598
  - structure, 564-566
  - traversing, 542
  - traversing of, 542
  - UNIX, 565-566, 598
  - usage, 568
  - virtual, 603-605, 804-805
  - Windows 10, 875-879
  - write-anywhere file layout, 589-593

- ZFS, 482-484, 581, 588, 598
- file-syste context**, 788
- File System Hierarchy Standard document**, 778-779
- fil system implementation**, 563-593
  - allocation methods, 570-578
  - contiguous allocation, 570-573
  - indexed allocation, 575-577
  - linked allocation, 573-575
  - performance, 577-578
  - directory implementation, 568-570
  - efficiency, 582-583
  - file-system operations, 566-568
  - file-system structure, 564-566
  - free-space management, 578-582
  - performance, 583-586
  - recovery, 586-589
  - WAFL example, 589-593
- file-syste interface**, 529-560
  - access methods, 539-541, 551-554
  - directory structure, 541-550
    - acyclic-graph directories, 547-549
    - general graph directory, 549-550
    - single-level directory, 542-543
    - tree-structured directories, 545-547
    - two-level directory, 543-545
  - file attributes, 530-531
  - and file concept, 529-530
  - file operations, 532-536
  - file structure, 537-539
  - file types, 536-537
  - memory-mapped files, 555-560
    - protection, 550-555
- file-syste management**, 29-30
- file-syste manipulation (operating system service)**, 56-57
- fil table**, 788
- fil transfer**, 750-751
- fil viruses**, 632
- filte drivers**, 863
- filtering system-call**, 688
- filte management**, 863
- fine-graine multithreading**, 222
- Finish (data structure)**, 335, 339
- fire alls**, 41, 659-660
- fire all chains**, 815
- fire all management**, 815
- firm are**, 11, 12
- first-come first-serve (FCFS) scheduling algorithm**, 206-207, 458
- first-fit strategy**, 358, 359
- first-leve interrupt handler (FLIH)**, 496
- firs readers**, 291
- flas translation layer (FTL)**, 453-454
- flexibilit , of compiler-based enforcement**, 692
- FLIH (first-leve interrupt handler)**, 496
- flo control**, 519, 748, 749
- flushing** 366
- folder redirection**, 883
- folders**, 59
- foreground priority separation boost**, 843
- foreground processes**, 115, 122, 215, 241
- fork() and exec() process model (Linux)**, 786-789
- fork-join model**, 180-184
- fork0system call**, 118-119, 188, 786-789
- formatting**, 463, 464
- forwarding**, 466
- forward-mapped page tables**, 371
- 4-byte pages**, 363, 364
- four-layered model of security**, 623-625
- fourth extended file system (ext4)**, 805
- fragments, packet**, 815
- fragmentation**, 359-360, 571-572
- frame(s)**, 360
  - in data-link layer, 742
  - free, 364, 396-397, 425-426
  - minimum number of, 413-414
  - page, 853
  - page faults vs., 404, 405
  - victim, 402
- frame allocation**, 413-419
  - allocation algorithms, 403, 414-415
  - equal, 414
  - global vs. local, 415-418
  - minimum number of frames, 413-414
  - non-uniform memory access, 418-419
  - proportional, 414-415
- frame-allocation algorithm**, 403, 414-415
- frame table**, 365
- free-behind technique**, 585
- FreeBSD**, 70, 71
- free frames, allocation and**, 364
- free-frame list**, 396-397, 425-426
- free page**, 856
- Free Software Foundation (FSF)**, 48
- free-space list**, 578-579
- free-space management**, 578-582
- fresh value**, 647
- front-end processors**, 522
- FSF (Free Software Foundation)**, 48
- fsgid property**, 818
- fsuid property**, 818
- FTL (flas translation layer)**, 453-454
- full backup**, 589
- full slabs**, 429, 798

functional programming languages, 313-314

## G

Galois field math, 478  
 Gantt chart, 206  
 garbage collection, 454, 549, 550, 727  
 GB (gigabyte), 11  
 gcc (GNU C compiler), 778  
 GCD (Grand Central Dispatch), 185-186  
 GDT (global descriptor table), 379  
 general graph directories, 549-550  
 general revocation, 682  
 gestures, 60  
 get command, 751  
 GFS (Google file system), 759-761  
 gigabyte (GB), 11  
 git, 50  
 global allocation, 415-418  
 global descriptor table (GDT), 379  
 global dispatch queues, 185  
 global replacement, 415-418  
 global table, 679  
 GNOME desktop, 60  
 GNU C compiler (gcc), 778  
 GNU General Public License (GPL), 48  
 GNU/Linux, 48  
 Google Android, 42  
 Google file system (GFS), 759-761  
 GPFS, 768  
 GPL (GNU General Public License), 48  
 GPU (graphics processing unit), 735  
 graceful degradation, 19  
 Grand Central Dispatch (GCD), 185-186  
 granularity, minimum, 791  
 graphs, acyclic, 547  
 graphical user interfaces (GUIs), 56, 59-61  
 graphics processing unit (GPU), 735  
 green threads, 167  
 group (user class), 551  
 group identifiers 33-34  
 grouping, 580  
 group policies, 884  
 group rights (Linux), 817  
 GRUB, 94  
 guard pages, 852  
 guest (operating system), 34  
 guest processes, 702  
 GUIs (graphical user interfaces), 56, 59-61

## H

hackers, 622

Hadoop, 22  
 Hadoop distributed file system (HDFS), 759, 760  
 Hadoop file system (HDFS), 484  
 HAL (hardware-abstraction layer), 835, 840  
 handles, 849  
 handle tables, 849  
 handling, signal, 188-189  
 handshake, three-way, 748  
 hard affinity, 225  
 hard-coding techniques, 129  
 hard disk drives (HDDs), 13  
     components of, 450-451  
     defined, 449  
     scheduling, 457-461  
 hard errors, 463, 467  
 hard limits, 438, 857  
 hard links, 532, 549, 879  
 hard page faults, 416  
 hard real-time systems, 227  
 hardware, 4  
     instructions, 266-269  
     I/O system, 490-500  
         direct memory access, 498-500  
         interrupts, 494-498  
         polling, 493-494  
     and main memory, 350-352  
     and memory management, 28  
     process migration and, 753  
     for relocation and limit registers, 357  
     for storing page tables, 365-368  
     synchronization, 265-270  
     transformation of requests to operations  
         by, 516-519  
     virtual machine, 710-713  
 hardware-abstraction layer (HAL), 835, 840  
 hardware objects, 672  
 hardware threads, 222  
 hardware transactional memory (HTM), 312  
 hard working-set limits, 438  
 hash collision, 39  
 hashed page tables, 373-374  
 hash functions, 38-39, 643  
 hash map, 39  
 hash tables, 570  
 hash value (message digest), 643  
 HBA (host bus adapter), 491  
 HDDs, see hard disk drives  
 HDFS (Hadoop distributed file system), 759, 760  
 HDFS (Hadoop file system), 484

**head crash**, 451  
**heaps**, 893-894  
**heap section (of process)**, 107  
**heartbeat procedure**, 754-755  
**heterogeneous multiprocessing**, 226-227  
**hibernation**, 870  
**hierarchical paging**, 371-373  
**high-availability service**, 19  
**high contention**, 271, 286  
**high memory**, 795  
**high-performance computing**, 20  
**high-performance event timer (HPET)**, 505  
**high priority**, 212  
**hijacking, session**, 623  
**hit ratio**, 367, 432  
**hives**, 871  
**hold-and-wait condition (deadlocks)**, 321, 327-328  
**holes**, 358  
**HoloLens**, 874  
**honeypot**, 655-656  
**horizontal scalability**, 484  
**host(s):**  
 distributed system, 734  
 operating system, 35, 177  
 virtual machine, 702  
**host-attached storage**, 470  
**host bus adapter (HBA)**, 491  
**host controller**, 456  
**host-id**, 739  
**host name**, 73, 738  
**hot spare (drive)**, 480  
**hot-standby mode**, 19, 20  
**HPET (high-performance event timer)**, 505  
**HTM (hardware transactional memory)**, 312  
**HTTP protocol**, 881  
**huge pages**, 363, 432  
**Hybrid Boot**, 873-874  
**hybrid cloud**, 44  
**hybrid operating systems**, 86-91  
**hypcalls**, 717  
**hypercall interface**, 839  
**hyper-threading**, 222, 832  
**Hyper-V for Client**, 831  
**hyper-V hypervisor**, 839  
**hypervisors**, 670, 702  
 separation, 729  
 type 0, 702, 713-714  
 type 1, 703, 714-715  
 type 2, 703, 715-716

**I**

**IA-32 architecture**, 379-382  
**IA-64 architecture**, 382  
**IaaS (infrastructure as a service)**, 44  
**IB (InfiniBand)** 473  
**icons**, 59  
**ideal processors**, 242, 842  
**idempotent**, 759  
**identifiers**  
 address-space, 366  
 file, 530  
 group, 33-34  
 host names vs., 738  
 location-independent file, 764  
 process, 116  
 spoofed, 606  
 user, 33  
**idle process**, 872  
**idle threads**, 239, 842  
**IKE protocol**, 646  
**image, disk**, 723-724  
**immediate revocation**, 682  
**immutable shared files**, 609  
**immutable-shared-fil semantics**, 609  
**imperative languages**, 313  
**impersonation**, 867  
**implementation:**  
 CPU scheduling algorithm, 249  
 cryptography, 645-646  
 directory, 568-570  
 file system, see **fil system**  
**implementation**  
 monitor, 280-281  
 of naming techniques, 763-764  
 of operating systems, 80-81  
 Pthread, 169, 170  
 of security defenses, 653-662  
 and accounting, 659  
 and auditing, 659  
 and firewalls, 659-660  
 and intrusion prevention, 655-657  
 levels of defenses, 661-662  
 and logging, 659  
 and security policy, 653  
 and virus protection, 657-659  
 and vulnerability assessment, 653-655  
**semaphore**, 274-276  
**synchronization primitive**, 845-846  
**virtual machine**, 713-719  
 application containment, 718-719  
 emulation, 717-718  
 paravirtualization, 716-717

- programming-environment
  - virtualization, 717
- type 0 hypervisors, 713-714
- type 1 hypervisors, 714-715
- type 2 hypervisors, 715-716
- and virtual machine life cycle, 713
- implicit threading, 176-188**
  - fork join, 180-183
  - Grand Central Dispatch, 185-186
  - Intel thread building blocks, 186-188
  - OpenMP and, 183-185
  - thread pools and, 177-180
- importance hierarchy (Android), 122**
- include file 40**
- increase scheduling priority privilege, 887**
- incremental backup, 589**
- indefinit blocking (starvation), 213, 343**
- independence, location, 762, 769**
- independent processes, 123**
- indexes, 540, 542, 576**
- index blocks, 575-577**
- indexed allocation, 575-577**
- index root, 876**
- indirect blocks, 576, 577**
- indirect communication, 129**
- indirection, 683, 703**
- InfiniBand (IB), 473**
- information leak attacks, 827**
- information-maintenance system calls, 72**
- information sharing, 123**
- infrastructure as a service (IaaS), 44**
- inheritable capabilities, 685**
- init process, 117**
- in-memory file-system structures, 568, 569**
- inode, 482, 565, 577**
- inode objects, 605, 804**
- input/output, see I/O**
- input/output operations per second (IOPS), 461**
- insert() method, 305-307**
- InServ storage array, 481**
- instruction register, 12**
- integrity, 622, 687-688**
- integrity label (Windows 10), 663**
- integrity levels, 826**
- Intel processors, 379-382**
  - event-vector table, 496
  - IA-32 architecture, 379-382
  - IA-64 architecture, 382
  - thread building blocks, 186-188
- inter-computer communications, 522**
- interface(s). See also specific types**
  - choice of, 60-62
- defined, 501**
- speeds of, 510**
- interlock, I/O, 434-436**
- intermachine interface, 757**
- internal fragmentation, 359**
- international support (Windows 10), 835-836**
- Internet, 737**
- Internet Key Exchange (IKE), 646**
- Internet model, 742**
- Internet Protocol (IP), 743-746. See also Transmission Control Protocol/Internet Protocol (TCP/IP)**
- Internet Service Providers (ISPs), 737**
- interpreted languages, 717**
- interprocess communication (IPC), 123-153**
  - in client-server systems, 145-153
  - remote procedure calls, 149-153
  - sockets, 146-149
- in Linux, 777, 812-813**
- Mach example of, 135-138**
- in message-passing systems, 127-132**
- pipes in, 139-145**
- POSIX shared-memory example of, 132-135**
- in shared-memory systems, 125-127**
- Windows example of, 138-139**
- interrupt(s), 8-11, 494-498**
  - defined, 494
  - in Linux, 794
  - maskable, 10, 495-496
  - nonmaskable, 10, 495
  - software (traps), 497
  - in Windows 10, 846-848
- interrupt chaining, 10, 496**
- interrupt-controller hardware, 9, 495**
- interrupt-dispatch table (Windows 10), 848**
- interrupt-handler routine, 9, 494-495**
- interrupt latency, 228**
- interrupt objects, 848**
- interrupt priority levels, 10-11, 497**
- interrupt request levels (IRQLs), 841, 846**
- interrupt-request line, 9, 494**
- interrupt service routines (ISRs), 844**
- interrupt vector, 9, 496**
- intruders, 622**
- intrusion prevention, 655-657**
- intrusion-prevention systems (IPSs), 656**
- inverted page tables, 374-375, 433**
- involuntary context switches, 204**
- I/O (input/output):**
  - fast mechanism for, 865-866

- raw, 464, 465
  - structure of, 14-15
  - in virtual machines, 722-723
  - I/O-bound processes**, 112
  - I/O burst**, 201
  - I/O bus**, 456
  - I/O channel**, 522
  - I/O control level (file system)**, 564
  - I/O interlock**, 434-436
  - I/O manager**, 862-864
  - I/O operations**, 56
  - IOPS (input/output operations per second)**, 461
  - I/O request packet (IRP)**, 863
  - iOS operating system**, 42, 87-89
  - I/O status information (PCBs)**, 110
  - I/O subsystem(s)**, 32-33
    - kernels in, 508-516
    - procedures supervised by, 516
  - I/O system(s)**, 489-525
    - application interface, 500-508
    - block and character devices, 503-504
    - clocks and timers, 505-506
    - network devices, 504-505
    - nondblocking and asynchronous I/O, 506-507
    - vectorized I/O, 507-508
  - hardware, 490-500
    - direct memory access, 498-500
    - interrupts, 494-498
    - for memory-mapped I/O, 491-493
    - polling, 493-494
    - summary, 500
  - kernel subsystem, 508-516
    - buffering, 509-510
    - caching, 510-511
    - data structures, 512-514
    - error handling, 511-512
    - I/O scheduling, 508-509
    - power management, 514-516
    - procedures supervised by, 516
    - protection, 512
    - spooling and device reservation, 511
  - Linux, 810-812
    - overview, 489-490
  - STREAMS mechanism, 519-521
  - and system performance, 521-524
    - transformation of requests to hardware operations, 516-519
  - IP (Internet Protocol)**, 743-746. See also
    - Transmission Control Protocol/Internet Protocol (TCP/IP)**
  - IPC**, see **interprocess communication**
  - iPhone**, 60
  - IPIs (Windows 10)**, 847-848
  - IPSec**, 646
  - IPSSs (intrusion-prevention systems)**, 656
  - IRP (I/O request packet)**, 863
  - IRQs (interrupt request levels)**, 841, 846
  - iSCSI**, 471
  - ISPs (Internet Service Providers)**, 737
  - ISRs (interrupt service routines)**, 844
  - Itanium**, 382
  - iteration space**, 187
- J**
- Java**:
    - DNS lookup in, 740
    - file locking in, 534, 535
    - fork-join in, 180-184
    - Lambda expressions in, 174
    - language-based protection in, 694-696
    - synchronization, 303-311
      - condition variables, 309-311
      - monitors, 303-307
      - reentrant locks, 307-308
      - semaphores, 308-309
      - thread dumps in, 339
      - thread pools in, 179-180
  - Java Executor interface**, 175-176
  - Java threads**, 173-176
  - Java Virtual Machine (JVM)**, 177, 717, 727-728
  - JBOD (Just a Bunch of Disks)**, 472
  - JIT compilers**, 728
  - jobs, processes vs.**, 106
  - job objects**, 859
  - job scheduling**, 106
  - journaling**, 587-588, 808
  - Just a Bunch of Disks (JBOD)**, 472
  - just-in-time (JIT) compilers**, 728
  - JVM**, see **Java Virtual Machine**
- K**
- KB (kilobyte)**, 11
  - K Desktop Environment (KDE)**, 60
  - Kerberos network authentication protocol**, 607
  - kernel(s)**, 6, 7, 501, 508-516
    - buffering, 509-510
    - caching, 510-511
    - computational, 833
    - data structures, 36-40, 512-514
    - error handling, 511-512
    - I/O scheduling, 508-509
    - and I/O subsystems, 516

- Linux, 776-778, 781
  - nonpreemptive, 262
  - power management, 514-516
  - preemptive, 262
  - protection, 512
  - secure, 839-840
  - spooling and device reservation, 511
  - synchronization of, 295-299, 792-794
  - uni-, 728
  - Windows 10, 839-848
- kernel abstractions**, 89
- kernel code**, 261
- kernel data structures**, 36-40, 512-514
- kernel environment**, 88
- kernel extensions (kexts)**, 89
- kernel memory allocation**, 426-430
- kernel mode**, 24, 25, 782
  - Kernel-Mode Driver Framework (KMDF)**, 864
  - kernel-mode threads (KT)**, 841
  - kernel modules**, 86, 783-786
  - kernel module management**, 784
  - kernel object access (Windows 10)**, 884-885
  - kernel threads**, 166, 217, 234
  - kernel virtual memory**, 801
- Kernighan's Law**, 98
- kexts (kernel extensions)**, 89
- keys**:
  - for capabilities, 683
  - defined, 638
  - Internet Key Exchange, 646
  - in lock-key schemes, 681
  - master, 683
  - private, 641
  - public, 641
  - sense, 512
  - session, 647
- key distribution**, 644-645
- key ring**, 644
- keystreams**, 640
- keystroke logger**, 634
- kilobyte (KB)**, 11
- KMDF (Kernel-Mode Driver Framework)**, 864
- Korn shell**, 58
- KT (kernel-mode threads)**, 841
- Kubernetes**, 719
  
- L**
  
- labels**, for mandatory access control, 685
- Lambda expressions**, 174
- languages**, 313-314, 717
- language-based protection systems**, 690-696
  - compiler-based enforcement, 691-693
  - in Java, 694-696
- LANs (local-area networks)**, 36, 735-737
- large objects**, 430
- latency**:
  - dispatch, 203, 228, 229
  - event, 227-228
  - interrupt, 228
  - in real-time systems, 227-229
  - rotational, 451
  - target, 236, 791
- latency command**, 494
- layers (of network protocols)**, 645
- layered approach (operating system structure)**, 83-84
- layered protocols**, 891
- LBA (logical block address)**, 456
- LCNs (logical cluster numbers)**, 875
- LDAP (lightweight directory-access protocol)**, 607, 884
- LDT (local descriptor table)**, 379
- least-frequently used (LFU)**
  - page-replacement algorithm, 411-412
- least privilege, principle of**, 626, 627, 668-669
- least-recently-used (LRU) algorithm**, 407-408
- left child**, 38
- LFH design**, 894
- LFU page-replacement algorithm**, 411-412
- lgroups**, 419
- libc (C library)**, 63, 69, 370, 781
- libraries**:
  - C, 63, 69, 370, 781
  - Linux system, 781
  - shared, 356, 392
  - thread, 168-176
    - about, 168-169
    - Java, 173-176
    - Pthreads, 169-171
    - Windows, 171-173
- library operating systems**, 728
- licensing, Linux**, 779-780
- life cycle**:
  - I/O request, 518-519
  - virtual machine, 713
- lifetime, virtual address space**, 799-800
- LIFO**, 37-38
- lightweight directory-access protocol (LDAP)**, 607, 884
- lightweight process (LWP)**, 193

- limit register**, 351-352
- linear addresses**, 380, 382
- linear lists (files)** 569-570
- line discipline**, 811-812
- link(s):**
  - communication, 128
  - defined, 548
  - hard, 532, 549, 879
  - resolving, 548
  - symbolic, 879
- linked allocation**, 573-575
- linked lists**, 37, 38, 579-580
- linked scheme index block**, 576
- linkers**, 75, 76
- linking**, 355-356, 803, 882
- Linux**, 48, 775-819
  - capabilities in, 685-686
  - design principles for, 780-783
  - file systems, 803-810
    - ext3 file system, 805-807
    - journaling, 808
    - /proc file system, 808-810
    - virtual, 804-805
  - history of, 775-780
  - input and output, 810-812
  - interprocess communication, 812-813
  - kernel modules, 783-786
  - lockdep tool, 330
  - memory management, 795-803
    - execution and loading of user programs, 801-803
    - physical memory, 795-798
      - virtual memory, 436-437, 798-801
  - network structure, 813-815
  - process management, 786-790
  - process representation in, 111
  - scheduling in, 234-239, 790-794
  - security model, 816-818
  - swap-space management in, 468-470
  - synchronization in, 296-298
  - system structure, 83
  - threads example, 195-196
  - tree of processes, 116
  - Windows subsystem for, 91
- Linux distributions**, 776, 779
- Linux instance**, 91
- Linux kernel**, 776-778, 781
- Linux kernel data structures**, 40
- Linux system(s)**, 776
  - components of, 781-783
  - history of, 778-779
  - obtaining page size on, 364
- Linux timers**, 27
- lists**, 37
  - access, 679-680
  - access-control, 552, 555, 826
  - capability, 680-681
  - export, 612
  - free-frame, 396-397, 425-426
  - free-space, 578-579
  - linear, 569-570
  - linked, 37, 38, 579-580
  - user control, 561
- listing directories**, 542
- listing file names and attributes**, 551
- little cores**, 227
- little-endian**, 150
- Little's formula**, 247
- live CD**, 48
- live DVD**, 48
- livelock**, 320-322
- live migration (virtual machines)**, 706, 724-726
- liveness**, 283-284
- living documents**, 653
- loadable kernel modules (LKMs)**, 86
- load balancing**, 224-225, 735, 753
- loaders**, 75-77, 695, 727, 783. See also **bootstrap programs**
- loading**, 355, 801-803
- load sharing**, 220
- load time**, 353
- local allocation**, 415-418
- local-area networks (LANs)**, 36, 735-737
- local descriptor table (LDT)**, 379
- locality model**, 421
- locality of reference**, 395
- locality property**, 857
- local-name**, 763
- local replacement**, 415-418
- local replacement algorithm**, 420-421
- location**, file 530, 534
- location independence**, 761, 762
- location-independent fil identifiers** 764
- location transparency**, 761
- locks**, 681. See also **deadlock(s)**
  - advisory, 535
  - exclusive, 534
  - in Java API, 534, 535
  - mandatory, 535
  - mutex, 270-272, 299-300
  - nonrecursive, 299
  - Pushlocks, 831
  - reader-writer, 292-293
  - reentrant, 307-308
  - scope of, 305
  - shared, 534
  - for shared data, 70

- lock-free algorithms, 284
- locking, page, 434-436
- locking files, 534-536
- lock-key scheme, 681
- lofs (fil system), 598
- log-based transaction-oriented fil systems**, 587-588
- log files, 95, 876
- log-fil service, 878
- logging, 57, 659
- logging area, 878
- logical address, 353, 379
- logical address space, 353-355
- logical blocks, 456
- logical block address (LBA), 456
- logical cluster numbers (LCNs), 875
- logical file system, 565
- logical formatting, 464
- logical memory**, 24, 362. See also **virtual memory**
- logical processors, 832
- logical records, 539
- logic bomb, 627
- login, remote, 750
- loopback, 148
- loosely coupled system, 83
- loosely-coupled systems, 19
- love bug virus, 658
- low contention, 271
- low-fragmentation heap (LFH) design, 894
- low-level formatting (disks), 463
- low priority, 212
- LRU algorithm, 407-408
- LRU-approximation page replacement algorithm, 409-411
- LRU page replacement, 407-409
- ls command, 751
- Lustre, 768
- LWP (lightweight process), 193
- LXC containers, 718, 719
  
- M**
- MAC (mandatory access control), 684-685
- MAC (message-authentication code), 643
- MAC address, 745
- Mach-O format, 77
- Mach operating system, 84, 135-138
- macOS operating system:
  - GUI, 61
  - as hybrid system, 87-89
  - latency command, 494
  - sandboxing in, 690-691
- macro viruses, 632
- magic number (files) 537
- magnetic tapes, 455
- mailboxes, 129-130
- main memory**, 349-385
  - and address binding, 352-353
  - ARMv8 architecture, 383-384
  - contiguous allocation of, 356-360
  - and dynamic linking, 355-356
  - and dynamic loading, 355
  - and hardware, 350-352
  - Intel 32 and 64-bit architectures, 379-382
  - and logical vs. physical address space, 353-355
- paging for management of, 360-376
  - basic method, 360-365
  - hardware, 365-368
  - hashed page tables, 373-374
  - hierarchical paging, 371-373
  - inverted page tables, 374-375
  - and Oracle SPARC Solaris, 375-376
  - protection, 368-369
  - and shared pages, 369-371
  - swapping with, 377
- shared libraries, 356
  - and swapping, 376-378
- main queue**, 185
- main TLB**, 384
- major page faults**, 416
- malware, 625-628
- MANs (metropolitan-area networks), 36
- mandatory access control (MAC), 684-685
- mandatory file-lockin mechanisms, 535
- mandatory policy, 826
- man-in-the-middle attack, 623, 635, 645
- many-to-many multithreading model, 167-168
- many-to-one multithreading model, 166-167
- mapping, 39
  - address, 456-457
  - file, 555, 557
  - memory, 802-803
- MapReduce system, 22, 761
- marshaling, 150, 882
- Mars Pathfinde , 285
- maskable interrupts, 10, 495-496
- masquerading, 622, 623, 635
- mass-storage management, 30
- mass-storage structure, 449-486
  - address mapping, 456-457
  - attachment of storage, 469-473
  - device management, 463-467
  - error detection and correction, 462-463

- hard disk drives, 450-451, 457-461  
 nonvolatile memory devices, 452-454, 461-462  
 overview, 449-450  
**RAID**, 473-485
  - extensions, 481-482
  - for object storage, 483-485
  - performance improvement, 475
  - problems with, 482-483
  - RAID levels, 475-481
  - reliability improvement, 473-475
 scheduling, 457-462  
 secondary storage connection methods, 456  
 swap-space management, 467-469  
 volatile memory, 454-455
- master book record (MBR)**, 465, 466
- master fil directory (MFD)**, 543
- master fil table**, 566
- master key**, 683
- master secret (TLS)**, 647
- matchmakers**, 151
- Max (data structure)**, 335
- maximum number of resources**, declaration of, 330
- MB (megabyte)**, 11
- MBR (master book record)**, 465, 466
- MD5 message digest**, 643
- mean time between failures (MTBF)**, 473, 474
- mean time of data loss**, 474
- mean time to repair**, 474
- mechanical storage systems**, 14
- mechanisms**, 80, 668. See also specific mechanisms
- medium access control (MAC) address**, 745
- medium objects**, 430
- megabyte (MB)**, 11
- memory**:
  - Address Window Extension, 894-895
  - anonymous, 399, 469
  - defined, 14
  - demand-zero, 799
  - direct memory access, 14, 498-500
  - direct virtual memory access, 500
  - high, 795
  - in-memory file-system structures, 568, 569
  - layout of, in C program, 108
  - layout of process in, 106
  - logical, 24, 362
  - main, see **main memory**
  - network virtual memory, 765
- over-allocation of, 401  
 physical, 24, 362, 390, 391, 795-798  
 secondary, 395  
 semiconductor, 14  
 shared, 57, 73, 123, 125, 556-560  
 software transactional, 312  
 32-byte, 363, 364  
 transactional, 311-312  
 virtual, see **virtual memory**  
 volatile, 454-455
- memory access**:
  - direct, 15, 498-500
  - direct virtual, 500
  - effective memory-access time, 367
  - non-uniform, 18, 19, 418-419
- memory-address register**, 354
- memory allocation**, 358-359, 426-430
- memory barriers**, 265-266
- memory compression**, 425-426, 858
- memory devices**:
  - management of, 463-467
  - nonvolatile, 452-454
  - defined, 449
  - NAND flash controller algorithms for, 453-454
  - overview, 452-453
  - scheduling, 461-462
- memory fences**, 266
- memory management**, 28-29
  - in Linux, 795-803
    - execution and loading of user programs, 801-803
    - physical memory, 795-798
    - virtual memory, 798-801
  - with virtual machines, 721-722
  - in Windows 10, 892-895
- memory-management information (PCBs)**, 110
- memory-management unit (MMU)**, 354, 855
- memory manager (MM)**, 851
- memory-mapped files**, 555-560, 892-893
- memory-mapped I/O**, 491-493
- memory mapping (Linux)**, 802-803
- memory model**, 265-266
- memory protection**, 357, 368-369
- memory stall**, 221-222
- memory-style error-correcting organization**, 476-477
- memory transactions**, 311
- messages**, 135
  - complex, 136
  - in distributed systems, 734
  - OSI network, 742, 744

- in Win32 API, 891
- message-authentication code (MAC)**, 643
- message digest (hash value)**, 643
- message modification** 622-623
- message passing**, 123, 125, 130
- message-passing model**, 57, 72-73, 127-132
  - buffering, 131-132
  - Mach example, 135-138
  - naming, 128-130
  - synchronization, 130-131
- metadata**, 607, 876
- metaslabs**, 581
- methods (Java)**, 694
- Metro**, 823
- metropolitan-area networks (MANs)**, 36
- MFD (master fil directory)**, 543
- MFU page-replacement algorithm**, 412
- microkernels**, 84-86
- Microsoft Interface Definition Language (MIDL)**, 150, 882
- Microsoft Windows**, see **Windows operating system (generally)**
- micro TLBs**, 384
- middleware**, 6, 7
- IDL (Microsoft Interface Definition Language)**, 150, 882
- migration:**
  - computation, 752
  - data, 751-752
  - file, 761-762
  - process, 752-753
  - push and pull, 224
    - with virtual machines, 706, 724-726
- minidisks**, 704
- minifilters** 863
- minimum granularity**, 791
- miniport driver**, 864
- minor page faults**, 416
- mirrored volume**, 474
- mirroring**, 474, 476
- MM (memory manager)**, 851
- MMU (memory-management unit)**, 354, 855
- mobile computing**, 41-42
- mobile systems**, 115, 377-378
- mode bits**, 24
- moderate contention**, 285
- Modern**, 823
- modified page**, 856
- modify bits (dirty bits)**, 402
- modularity**, 123
- modules:**
  - file-organization, 565
  - kernel, 86, 783-786
- pluggable authentication**, 816
- stream**, 519
- module loader**, 783
- module-management system**, 783, 784
- module unloader**, 783
- monitors**, 276-282
  - dining-philosophers solution with, 295,296
  - implementation of, using semaphores, 280-281
- in Java**, 303-307
- resumption of processes within**, 281-282
- security reference**, 866-869
- usage of**, 277-280
- monitor calls**, see **system calls**
- monitor type**, 277
- monoculture**, 634
- monolithic operating systems**, 82-83
- Moore's Law**, 5
- most-frequently used (MFU)**
  - page-replacement algorithm, 412
- motherboard**, 20
- motivation, for multithreading**, 160-161
- mounting**, 464, 598-602
- mount points**, 598, 879
- mount protocol (NFS)**, 612
- mount table**, 517, 567
- MTBF (mean time between failures)**, 473, 474
- MUI support**, 840
- multicore processors**, 221-224
- multicore programming**, 162-166
- multicore systems**, 16-18
- multidimensional RAID level 6**, 478
- multifactor authentication**, 653
- multilevel feedback-queue scheduling algorithm**, 216-217
- multilevel index**, 576
- multilevel queue scheduling algorithm**, 214-216
- multimode operation**, 25-26
- multipartite viruses**, 634
- Multiple UNC Provider (MUP)**, 882-883
- multiple user interface (MUI) support**, 840
- multiprocessing**, 16, 220, 226-227, 794
- multiprocessors**, 18, 220
- multi-processor scheduling**, 220-227
  - approaches to, 220-221
  - examples, 234-242
    - Linux, 234-239
    - Solaris, 242-244
    - Windows, 239-242
- heterogeneous multiprocessing**, 226-227

and load balancing, 224-225  
 and multicore processors, 221-224  
     and processor affinity, 225-226  
**multiprocessor systems**, 16-19  
**multiprogramming**, 23, 112, 420  
**multi-provider router**, 883  
**multitasking**, 23, 115, 790  
**multithreaded processes**, 160  
     benefits of, 162  
     deadlocks in, 319-321  
     and exec() system call, 188  
     and fork() system call, 188  
     models of, 166-168  
     motivation for, 160-161  
     and signal handling, 188-189  
**multithreading**:  
     chip, 222, 223  
     coarse-grained, 222  
     fine-grained, 222  
     many-to-many, 167-168  
     many-to-one, 166-167  
     one-to-one, 167  
     simultaneous, 222  
**multi-touch hardware**, 874  
**MUP (Multiple UNC Provider)**, 882-883  
**mutex locks**, 270-272, 299-300  
**mutual exclusion (mutex)**, 260, 267, 268,  
     845  
**mutual-exclusion condition (deadlocks)**,  
     321, 327

## N

**names**:  
     common and distinguished, 647  
     host, 73, 738  
     resolution of, 738-741  
**named condition variables**, 309  
**named pipes**, 143-145, 881-882  
**named semaphores**, 300-301  
**named shared-memory object**, 559  
**name server**, 739  
**namespaces**, 787  
**naming**, 128-130  
     defined, 761  
     in distributed file systems, 761-764  
     distributed naming services, 607  
     domain name system, 607  
     file, 530  
     and network communication, 738-741  
     port, 135-136  
**naming schemes**, 763  
**naming structures (DFS)**, 761-763

**NAND flash controller algorithms**,  
     453-454  
**NAS (network-attached storage)**, 470-471  
**NAT (network address translation)**, 723  
**national-language-support (NLS) API**,  
     835  
**NDIS (Network Device Interface**  
     specification) 880  
**Need (data structure)**, 335, 336  
**need-to-know principle**, 672  
**nested page tables (NPTs)**, 710, 712  
**network(s)**:  
     communication structure, 738-749  
     communication protocols, 741-745  
     and naming/name resolution, 738-741  
     TCP/IP example, 745-746  
     UDP and TCP transport protocols,  
         746-749  
     defined, 36  
     firewalling to protect, 659-660  
     in Linux, 813-815  
     local-area, 36, 735-737  
     metropolitan-area, 36  
     network operating systems, 749-751  
     personal-area, 36  
     as resource types, 318  
     security in, 623  
     storage-area, 21, 470, 472  
     structure of, 735-738  
     threats to, 634-637  
     virtual private, 646, 881  
     wide-area, 36, 735, 737-738  
     wireless, 41, 736-737  
**network address translation (NAT)**, 723  
**network-attached storage (NAS)**, 470-471  
**network computers**, 40  
**network devices**, 504-505, 810  
**Network Device Interface specification**  
     (NDIS), 880  
**network file system (NFS)**, 610-615, 759  
**network information service (NIS)**, 607  
**networking**, 880-884  
**network interfaces (Windows 10)**, 880  
**network layer**, 742  
**network-layer protocol**, 645  
**network operating systems**, 36, 749-751  
**network protocols, registration of**, 785  
**network time protocol (NTP)**, 505  
**network virtual memory**, 765  
**new state**, 108  
**NFS (network file system)**, 610-615, 759  
**NFS protocol**, 612-614  
**nice value**, 236, 790  
**NIS (network information service)**, 607

- NLS API, 835
- no-access page**, 852
- nonblocking I/O, 506-507
- nonblocking message passing, 130
- noncontainer objects (Windows 10), 664
- nonmaskable interrupts, 10, 495
- nonpreemptive kernels, 262
- nonpreemptive scheduling, 202
- nonrecursive locks, 299
- nonrepudiation, 644
- nonresident attributes, 876
- nonsignaled state, 297
- non-uniform memory access (NUMA), 18, 19, 418-419
- nonvolatile memory (NVM) devices**, 13, 14, 452-454
  - defined, 449
  - NAND flash controller algorithms for, 453-454
  - overview, 452-453
  - scheduling, 461-462
- nonvolatile storage (NVS), 14, 449
- NOOP scheduler, 461
- no-preemption condition (deadlocks), 321, 328
- Normal World, 838
- notify() method, 305-307
- Notify port, 135
- NPTs (nested page tables), 710, 712
- NTFS, 875-877
- NTP (network time protocol), 505
- NUMA, see non-uniform memory access
- NUMA-aware algorithms, 225-226
- NUMA mode, 238-239
- NVM devices, see nonvolatile memory devices
- NVM express (NVMe), 456
- NVS (nonvolatile storage), 14, 449
  
- O**
  
- objects:**
  - access lists for, 679-680
  - in cache, 428
  - container, 664
  - critical-section, 297, 888
  - defined, 672
  - dentry, 605, 804, 805
  - device, 863
  - directory, 850
  - dispatcher, 297, 845-846
  - driver, 863
  - event, 845
  - file, 605, 804-805, 862
  - hardware, 672
  - inode, 605, 804
  - interrupt, 848
  - job, 859
  - in Linux, 797, 804
  - named shared-memory, 559
  - noncontainer, 664
  - section, 139, 852
  - semaphore, 845
  - sharing, 885-886
  - small, medium, and large, 430
  - software, 672
  - superblock, 605, 804, 805
  - timer, 845
  - in Windows 10, 664, 845-846, 848, 849
- Object Linking and Embedding (OLE)**, 882
- object manager (Windows 10)**, 849-851
- object storage**, 483-485
- object types**, 849, 850
- objfs (fil system)**, 598
- off-line compaction of space**, 572
- OLE (Object Linking and Embedding)**, 882
- one-time passwords**, 652
- one-to-one multithreading model**, 167
- on-line compaction of space**, 572
- OOM (out-of-memory) killers**, 418
- OpenAFS (Andrew fil system)**, 759
- open count**, 533
- open-fil table**, 533, 567
- opening files**, 532
- OpenMP**, 183-185, 312-313
- open operating systems**, 634
- open-source operating systems**, 46-51
- Open Systems Interconnection (OSI) model**, 741-744
- operating system(s)**:
  - application specificity to, 77-79
  - booting, 94-95
  - building, 92-93
  - closed-source, 46
  - computing environments, 40-46
  - CPU scheduling in, 234-244
    - Linux scheduling, 234-239
    - Solaris scheduling, 242-244
    - Windows scheduling, 239-242
  - debugging, 95-100
  - defined, 1, 3, 5-7
  - design goals for, 79-80
  - features of, 3
  - functioning of, 4-7
  - implementation of, 80-81
  - kernel data structures, 36-40

linkers and loaders, 75-77  
 network, 36  
 open-source, 46-51  
 operations, 21-27
 

- dual-mode and multimode, 24-26
- multiprogramming and multitasking, 23-24
- and timer, 26-27

 reasons for studying, 6  
 as resource allocator, 5  
 resource management by, 27-32  
 security in, 623-624  
 services provided by, 55-58  
 structure, 81-91
 

- hybrid systems, 86-91
- layered approach, 83-84
- microkernels, 84-86
- modules, 86
- monolithic, 82-83

 study of, 50  
 system calls, 62-74
 

- and API, 63-66
- functioning of, 62-63
- types of calls, 66-74

 system services, 74-75  
 system's view of, 5  
 threads in, 194-196  
 user interface with, 4-5, 58-62  
 virtualization components, 719-726
 

- CPU scheduling, 720
- I/O devices, 722-723
- live migration, 724-726
  - memory management, 721-722

**optimal page replacement**, 406-407  
**optimal page replacement algorithm**, 406-407  
**optimistic approach**, 285  
**Oracle SPARC Solaris**, 375-376  
**Orange Book**, 826  
**ordinary pipes**, 140-143  
**org (top-level domain)**, 739  
**orphan processes**, 122  
**OS/2 operating system**, 821-822  
**OSI model**, 741-744  
**OSI network model**, 741-744  
**OSI protocol stack**, 742-744  
**other users (class)**, 551  
**out-of-band key delivery**, 644  
**out-of-memory (OOM) killers**, 418  
**over-allocation of memory**, 401  
**overcommitment**, 720  
**over-provisioning**, 454  
**owners**:
 

- clock, 837

file, 603  
 as user class, 551  
**owner rights**, 678, 817

## P

**p (page number)**, 360  
**PaaS (platform as a service)**, 44  
**packaged applications**, 859  
**package systems**, 823  
**PAE (page address extension)**, 381  
**pages**. See also specific types
 

- defined, 360
- locking, 434-436
- obtaining size of, 364

**page address extension (PAE)**, 381  
**page allocator (Linux)**, 796  
**page-buffering algorithms**, 412  
**page cache**, 583, 798  
**page directory**, 381, 853  
**page-directory entries (PDEs)**, 853  
**page directory pointer table**, 381  
**page faults**, 394-395, 405, 416  
**page-fault-frequency (PFF)**, 424-425  
**page-fault rate**, 398, 423  
**page frames**, 853  
**page-frame number (PFN) database**, 856  
**page in**, 377  
**page locking**, 434-436  
**page number (p)**, 360  
**page offset (d)**, 360  
**page out**, 377  
**pageout policy (Linux)**, 800  
**pageout process (Solaris)**, 439  
**page replacement**, 401-413. See also **frame allocation**

- and application performance, 412-413
- basic mechanism, 401-404
- counting-based page replacement, 411-412
- defined, 401
- FIFO page replacement, 404-406
- global vs. local, 415-418
- LRU-approximation page replacement, 409-411
- LRU page replacement, 407-409
- optimal page replacement, 406-407
  - and page-buffering algorithms, 412

**page replacement algorithm**, 403  
**page size**, 363, 364, 431-432  
**page slots**, 469  
**page table(s)**, 361-378, 393
 

- clustered, 374
- defined, 361

- for demand paging, 395
- forward-mapped, 371
- hardware for storing, 365-368
- hashed, 373-374
- for hierarchical paging, 371-373
- inverted, 374-375, 433
- nested, 710, 712
- Oracle SPARC Solaris, 375-376
- page-table base register (PTBR), 365**
- page-table entries (PTEs), 853**
- page-table length register (PTLR), 369**
- paging, 360-376**
  - basic method, 360-365
  - demand, 392-399, 430-436
    - basic mechanism, 393-396
    - defined, 393
    - free-frame list, 396-397
    - with inverted page tables, 433
    - and I/O interlock, 434-436
    - and page size, 431-432
    - and performance, 397-399
    - and prepaging, 430-431
    - and program structure, 433-434
    - pure, 395
    - and TLB reach, 432-433
  - hardware, 365-368
  - for hashed page tables, 373-374
  - hierarchical, 371-373
  - IA-32, 380-381
  - inverted, 374-375
  - in Linux, 800
    - and memory protection, 368-369
    - and Oracle SPARC Solaris, 375-376
    - priority, 440
    - and shared pages, 369-371
    - swapping with, 377, 378
  - paging files 851
  - paired passwords, 652
  - PAM (pluggable authentication modules), 816
  - PAN (personal-area network), 36
  - parallel file system (PFS), 768
  - parallelism, 163, 165-166, 475
  - parallelization, 20
  - parallel regions, 183-184
  - paravirtualization, 703, 716-717
  - parent-child relationship, 140
  - parent process, 111
  - partial revocation, 682
  - partial slabs, 429, 798
  - partitions:
    - boot, 465
    - control, 714
    - file-system, 601-602
  - raw, 468
  - root, 601
  - storage device, 463-465
  - variable-partition schemes, 358
- partition boot sector, 566**
- partitioning, device, 463-464**
- passphrases, 651
- passwords, 554, 649-652
- path names, 544, 546
- path-name translation (NFS), 614-615
- PB (petabyte), 12
- PCBs (process control blocks), 109-110
- PCIe bus, 490
- PC motherboard, 20
- PCS (process-contention scope), 217-218
- PC systems, 874
- PDEs (page-directory entries), 853
- peer-to-peer computing, 43-44
- peer-to-peer distributed systems, 734
- PE (Portable Executable) format, 77
- penetration test, 654
- performance:**
  - and allocation of secondary storage, 578-579
  - and demand paging, 397-399
  - and file system implementation, 583-586
  - and I/O system, 521-524
  - and page replacement, 412-413
  - RAID structure to improve, 475
  - under swapping, 378
  - virtualization requirement related to, 704
  - of Windows 10, 831-833
- performance monitoring, 96-97**
- performance tuning, 95-97**
- periodic processes, 230
- periodic task rate, 230
- permanent revocation, 682
- permissions, 553, 669
- permitted capabilities, 685
- per-process open-fil table, 567
- per-process tools, 96, 97
- personal-area network (PAN), 36
- personal computer (PC) systems, 874
- personal fire alls, 660
- personal identificatio number (PIN), 652
- personalities, 87, 787
- pessimistic approach, 285
- petabyte (PB), 12
- Peterson's solution, 262-265
- PFF (page-fault-frequency), 424-425
- PFN database, 856
- PFS (parallel file system), 768
- phishing, 624

- PHY (ports), 490
- physical address, 354, 379
- physical address space, 353-355
- physical formatting, 463
- physical layer, 741, 742
- physical memory, 24, 362, 390, 391, 795-798
- physical security, 623
- physical-to-virtual (P-to-V) conversion, 724
- PIC (position-independent code), 803
- Pico process, 91
- Pico Providers, 823
- pid (process identifier) 116, 787
- PIN (personal identification number), 652
- pinning, 436, 866
- PIO (programmed I/O), 498
- pipes, 139-145
  - anonymous, 141-145
  - implementation considerations, 139-140
  - named, 143-145, 881-882
  - ordinary, 140-143
  - use of, 146
- pipe mechanism, 813
- platform as a service (PaaS), 44
- platter (disks), 450
- PLM (Process Lifetime Manager), 837
- plug-and-play and (PnP) managers, 869-870
- pluggable authentication modules (PAM), 816
- plug-in process, 124
- PnP managers, 869-870
- PoFX (power framework), 870
- Point-to-Point Tunneling Protocol (PPTP), 881
- policy(-ies), 80
  - cache updating, 766-767
  - delayed-write, 766
  - group, 884
  - mandatory, 826
  - mechanisms vs., 668
  - pageout, 800
  - security, 653
  - write-on-close, 766-767
  - write-through, 766
- policy algorithm (Linux), 800
- polling, 493-494
- polymorphic viruses, 633
- pools, 177-180, 483, 889
- pop, 66
- ports, 78, 129, 490
  - connection and communication, 138
  - naming of, 135-136
  - in remote procedure calls, 150
- well-known, 146
- portability, 834-835
- Portable Executable (PE) format, 77
- portals, 40
- port driver, 864
- port number, 746-747
- port rights, 135
- port scanning, 637
- position-independent code (PIC), 803
- positioning time (disks), 450
- POSIX:
  - interprocess communication example, 132-135
  - real-time scheduling, 232-234
  - synchronization examples, 299-303
- POSIX 1e, 685, 686
- possession (of capability), 680
- POST (power-on self-test), 872
- posting messages, 891
- power framework (PoFX), 870
- power management, 514-516
  - power manager (Windows 10), 870-871
  - power-of-2 allocator, 427
  - power-on self-test (POST), 872
  - power users, 60-61
- PPTP (Point-to-Point Tunneling Protocol), 881
- P + Q redundancy scheme, 478
- preemptive kernels, 262
- preemptive multitasking, 790
- preemptive scheduling, 202-203
- premaster secret (TLS), 647
- prepaging, 430-431
- presentation layer, 742
- primary thread, 890
- principle of least privilege, 626, 627, 668-669
- priority (field) 243
- priority-based scheduling, 229-230
- priority-inheritance protocol, 284
- priority inversion, 284, 285
- priority number, 281
- priority paging, 440
- priority replacement algorithm, 420-421
- priority scheduling algorithm, 211-214
- private cloud, 44
- private dispatch queues, 185
- private keys, 641
- privileged instructions, 25
- privileged mode, see kernel mode
- privilege escalation, 623
- privilege separation, 669
- procedural languages, 313
- procedures, as domains, 674

- process(es), 23, 105-154**  
     aborting, 342  
     background, 74-75, 115, 123, 215, 241  
     browser, 124  
     communication between, see  
         **interprocess communication**  
     components of, 106-107  
     consumer, 126-127, 290, 291, 559-560  
     context of, 788-789  
     cooperating, 123, 257  
     coordination among, 260  
     daemon, 690  
     defined, 103, 105  
     dispatched, 112  
     as domains, 674  
     empty, 123  
     environment of, 787-788  
     foreground, 115, 122, 215, 241  
     guest, 702  
     idle, 872  
     independent, 123  
     init, 117  
     I/O- vs. CPU-bound, 112  
     job vs., 106  
     lightweight, 193  
     in Linux, 789-790  
     multithreaded, see **multithreaded processes; multithreading**  
     operations on, 116-123  
         creation, 116-121  
         termination, 121-123  
     orphan, 122  
     parent, 111  
     passing data between, 813  
     periodic, 230  
     Pico, 91  
     plug-in, 124  
     producer, 126-127, 290, 558-559  
     renderer, 124  
     service, 123  
     sibling, 111  
     single-threaded, 160  
     state of, 107-109  
     system, 872  
     systemd, 117  
     threads performed by, 110  
     visible, 122  
     in Windows 10, 886  
     zombie, 122
- process-contention scope (PCS), 217-218**
- process control blocks (PCBs), 109-110**
- process-control system calls, 66-71**
- process identifie (pid), 116, 787**
- process identity (Linux), 787**
- Process Lifetime Manager (PLM), 837**
- process management:**  
     about, 27-28  
     in Linux, 786-790  
     Windows 10, 886-891
- process manager (Windows 10), 858-860**
- process migration, 752-753**
- process name, 73**
- processors, 18**  
     distributed system, 734  
     front-end, 522  
     ideal, 242, 842  
     Intel, 379-382  
         event-vector table, 496  
         IA-32 architecture, 379-382  
         IA-64 architecture, 382  
         thread building blocks, 186-188  
     logical, 832  
     multi-, 18, 220  
     multicore, 221-224
- processor affinit , 225-226**
- processor groups, 832**
- process reflection 860**
- process representation (Linux), 111**
- process scheduler, 110-112**
- process scheduling, 110-115, 199, 234**
- process synchronization, 260. See also synchronization tools**
- process termination, deadlock recovery by, 342**
- /proc file system (Linux), 808-810**
- procfs (fil system), 598**
- producer process, 126-127, 290, 558-559**
- production kernels (Linux), 777**
- program counters, 27, 106, 109**
- program execution (operating system service), 56**
- program loading and execution, 74**
- programmable interval timer, 505**
- programmed I/O (PIO), 498**
- programmer interface (Windows 10), 884-895**  
     IPC with Windows messaging, 891-892  
     kernel object access, 884-885  
     memory management, 892-895  
     process management, 886-891  
     sharing objects between processes, 885-886
- programming:**  
     multi-, 23, 112, 420  
     multicore, 162-166
- programming-environment**  
     virtualization, 703, 717
- programming languages, 313-314**

- programming-language support**, 74  
**program structure**, for demand paging, 433-434  
**program threats**, 625-634  
  code injection, 628-631  
  malware, 625-628  
  viruses, 631-634  
  worms, 631, 632  
**progress (requirement)**, 260  
**projects**, 244  
**proportional allocation**, 414-415  
**proportional share scheduling**, 233  
**proprietary software**, 46-47  
**protection**, 667-698  
  access matrix model, 675-685  
  implementation, 679-682  
  mandatory access control, 684-685  
  and revocation of access rights, 682-683  
  role-based access control, 683-684  
  capability-based systems, 685-687  
  code signing, 690  
  in computer systems, 33-34  
  with contiguous memory allocation, 357  
  domain of, 671-675  
  file, 531  
  file-system interface, 550-555  
  goals of, 667-668  
  I/O, 512  
  language-based systems, 690-696  
    compiler-based enforcement, 691-693  
    in Java, 694-696  
  as operating system service, 57-58  
  in paged environment, 368-369  
  and principle of least privilege, 668-669  
  rings of, 669-671  
  sandboxing, 689-690  
  static vs. dynamic, 673  
  system-call filtering, 688  
  system integrity, 687-688  
    from viruses, 657-659  
**protection domains**, 671-675, 711  
**protection mask (Linux)**, 817  
**protection rings**, 25, 669-671  
**protection system calls**, 73-74  
**pseudo-device driver**, 721-722  
**PTBR (page-table base register)**, 365  
**PTEs (page-table entries)**, 853  
**PTE tables**, 853  
**Pthreads**, 169-171, 218-219  
**PTLR (page-table length register)**, 369  
**P-to-V conversion**, 724  
**public cloud**, 44  
**public domain software**, 779-780  
**public keys**, 641  
**public-key encryption**, 641  
**pull migration**, 224  
**pure code (reentrant)**, 370  
**pure demand paging**, 395  
**pushing**, 66, 519  
**Pushlocks**, 831  
**push migration**, 224  
**put command**, 751
- Q**
- Quest-V**, 729  
**queue(s)**, 38  
  dispatch, 185  
  main, 185  
  ready, 112, 221, 843  
  scheduling, 112-113  
  wait, 112  
**queuing diagram**, 112, 113  
**queuing-network analysis**, 247
- R**
- race condition**, 259, 261  
**RAID (redundant arrays of inexpensive disks)**, 473-485  
  extensions, 481-482  
  levels of, 475-481  
  for object storage, 483-485  
  performance improvement, 475  
  problems with, 482-483  
  reliability improvement, 473-475  
  structuring, 474  
**RAID level 0**, 476  
**RAID level 0 + 1**, 478-479  
**RAID level 1**, 476  
**RAID level 1 + 0**, 478-479  
**RAID level 4**, 476-477  
**RAID level 5**, 477-478  
**RAID level 6**, 478  
**RAID levels**, 475-481  
  common, 475-478  
  selecting, 480-481  
  variations in, 478-480  
**raising interrupts**, 9, 494  
**RAM (random-access memory)**, 11  
**RAM drives**, 454, 455  
**random-access devices**, 502  
**random-access memory (RAM)**, 11  
**random-access time (disks)**, 450  
**range (value)**, 187  
**ransomware**, 626  
**RAT (Remote Access Tool)**, 625

- rate-monotonic scheduling, 230-232
- rate-monotonic scheduling algorithm, 230-232
- raw disk, 413, 464, 601
- raw I/O, 464, 465, 503-504
- raw partitions, 468
- RBAC (role-based access control), 683-684
- RDP, 707
- reacquisition, of capabilities, 682
- read access, locks with, 292
- read-ahead technique, 585
- read-end (of pipe), 140
- readers, 291, 292
- readers-writers problem, 290-293
- reader-writer locks, 292-293
- reading files, 532, 551
- read-modify-write cycle, 477
- read only devices, 502
- read pointer, 532
- read-write devices, 502
- ready queues, 112, 221, 843
- ready state, 108, 109
- real-time class, 239
- real-time CPU scheduling, 227-234
  - earliest-deadline-first scheduling, 232-233
    - and minimizing latency, 227-229
  - POSIX real-time scheduling, 233-234
  - priority-based scheduling, 229-230
  - proportional share scheduling, 233
  - rate-monotonic scheduling, 230-232
- real-time embedded systems, 45-46
- real-time operating systems, 46
- real-time range (Linux schedulers), 790
- real-time scheduling (Linux), 792
- reapers, 417-418
- receives, blocking vs. nonblocking, 130
- reconfiguration 755
- records:
  - activation, 107
  - base file, 876
  - logical, 539
  - master boot, 465, 466
- recovery:
  - from deadlock, 341-343
  - from failure, 755-756
  - and file system implementation, 586-589
  - Windows 10, 877-878
- recovery mode, 95
- red-black trees, 38, 40
- Red Hat, 779
- redirectors, 882-883
- redundancy, 473-475
- redundant arrays of inexpensive disks, see RAID
- reentrant code (pure code), 370
- reentrant locks, 307-308
- reference, locality of, 395
- reference bits, 409
- referenced pointer, 849
- reference string, 404, 406
- reflection process, 860
- reflecto , 864
- regions, 383
- register(s):
  - base, 351-352
  - control, 492
  - CPU, 110
  - data-in, 492
  - data-out, 492
  - instruction, 12
  - limit, 351-352
  - memory-address, 354
  - page-table base, 365
  - page-table length, 369
  - relocation, 354
  - status, 492
  - translation table base, 383
- registry, 74, 871-872
- regression testing, 249
- relative access, 539-540
- relative block number, 540
- relative path names, 546
- release, of resources, 318
- reliability:
  - of distributed systems, 735
  - RAID for improving, 473-475
  - of TCP, 748
  - of UDP, 747
  - of Windows 10, 828-829
- relocatable code, 353
- relocatable object file, 75
- relocation, 75, 76
- relocation register, 354
- remainder section, 260
- Remote Access Tool (RAT), 625
- remote desktop, 874
- remote fil access, 764-767
- remote file-systems 605-608
- remote fil transfer, 750-751
- remote login, 750
- remote operations (NFS), 615
- remote procedure calls (RPCs), 149-153, 834
- remote-service mechanism, 764
- removable storage media, 451
- remove() method, 305-307

- renaming files 542  
 renderer processes, 124  
 rendezvous, 131  
 repair, mean time to, 474  
 replacement, page, see page replacement  
 replay attacks, 622  
 replication, 480, 592-593  
 reply port, 135  
 repositioning (in files) 532  
**Request (data structure)**, 335-336, 339, 340  
 requests, for resources, 318  
 request consumer (circular buffer), 716  
 request edge, 323  
 request manager, 811  
 request producer (circular buffer), 716  
 resident attributes, 876  
 resolution:  
     address resolution protocol, 745  
     conflict, 784, 785  
     of links, 548  
     name, 738-741  
     and page size, 431-432  
**resource allocation (operating system service)**, 57  
**resource-allocation graph**, 323-326, 334, 338  
**resource-allocation-graph algorithm**, 333  
**resource allocator**, operating system as, 5  
**resource arbiters**, 869  
**resource management**, 27-32  
**resource preemption**, deadlock recovery by, 342-343  
**resource-request algorithm**, 335  
**resource sharing**, 162, 734-735  
**resource utilization**, 4  
**responses (password)**, 652  
**response consumer (circular buffer)**, 716  
**response producer (circular buffer)**, 716  
**response time**, 23, 205  
**responsibility**, for run-time-based enforcement, 694  
**responsiveness**, multithreaded process, 162  
**restart area**, 878  
**restore**, state, 114  
**restore point**, system, 871  
**restoring data**, 588-589  
**resuming**, 717, 888  
**return from sleep**, 243  
**reverse engineering**, 47  
**revocation of access rights**, 682-683  
**RHEL 7**, 461  
**rich text format (RTF)**, 658  
**right child**, 38  
**rights**:  
     access, 534, 673, 680, 682-683  
     copy, 677  
     group, 817  
     owner, 678, 817  
     port, 135  
     user, 817  
     world, 817  
**rings, protection**, 669-671  
**risk assessment**, 653-654  
**roaming profiles** 883  
**robustness, distributed system**, 754-756  
**roles**, 683  
**role-based access control (RBAC)**, 683-684  
**rollback**, 343  
**root directory**, 877  
**rootkit viruses**, 632  
**root partition**, 601  
**rotational latency (disks)**, 451  
**rotations per minute (RPM)**, 450  
**round robin**, 130  
**round-robin (RR) scheduling algorithm**, 209-211  
**routers**, 736  
**RPCs (remote procedure calls)**, 149-153, 834  
**RPM (rotations per minute)**, 450  
**RR scheduling algorithm**, 209-211  
**RSA algorithm**, 641, 642  
**RTE (run-time environment)**, 64-65  
**RTF (rich text format)**, 658  
**running state**, 108, 109  
**running system**, 94  
**run time, virtual**, 236  
**run-time-based enforcement**, 694-696  
**run-time environment (RTE)**, 64-65
- S**
- SaaS (software as a service)**, 44  
**safe computing**, 658  
**safe sequence**, 331  
**safe state**, 331-333  
**safety**, as virtualization requirement, 704  
**safety algorithm**, 335  
**sandbox**, 124, 658  
**sandboxing**, 689-690  
**SANs**, see storage-area networks  
**SAS buses**, 456, 490  
**SATA buses**, 456  
**save**, 114, 592  
**scalability**, 162, 484, 756-757  
**SCAN scheduling**, 458-459

- SCAN (elevator) scheduling algorithm,** 458-459
- scatter-gather method,** 498, 508
- schedulers:**
- CFQ, 461, 811
  - Completely Fair, 236, 237, 790
  - CPU, 113-114, 201
  - deadline, 460, 461
  - Linux, 790
  - NOOP, 461
  - process, 110-112
- scheduler activation,** 192-194
- scheduling:**
- cooperative, 202
  - CPU, see **CPU scheduling**
  - C-SCAN, 460
  - earliest-deadline-first, 232-233
  - fair, 791
  - FCFS, 458, 459
  - HDD, 457-461
  - I/O, 508-509
  - job, 106
  - in Linux, 790-794
  - multi-processor, see **multi-processor scheduling**
  - nonpreemptive, 202
  - NVM, 461-462
  - preemptive, 202-203
  - priority-based, 229-230
  - process, 110-115, 199, 234
  - proportional share, 233
  - rate-monotonic, 230-232
  - real-time, 792
  - SCAN, 458-459
  - selecting disk-scheduling algorithm, 460-461
  - shortest-remaining-time-first, 209
  - thread, 199, 790-791, 844-845
  - user-mode, 241, 833, 890-891
  - in Windows 10, 887
- scheduling classes,** 236
- scheduling context,** 788
- scheduling domain,** 238
- scheduling information, CPU,** 110
- scheduling rules,** 887
- SCM (service control manager),** 870
- scope:**
- contention, 217-218
  - of lock, 305
- script kiddies,** 631
- scripts, shell,** 61, 536
- SCS (system-contention scope),** 218
- searching, for files,** 541, 542
- search path,** 545
- secondary memory,** 395
- secondary storage,** 13. See also **disk(s)**
- allocation of, 570-578
  - contiguous allocation, 570-573
  - indexed allocation, 575-577
  - linked allocation, 573-575
  - and performance, 578-579
  - connection methods for, 456
- second-chance page-replacement algorithm (clock algorithm),** 410-411
- second extended file system (ext2),** 805
- second-level interrupt handler (SLIH),** 496
- second readers,** 291
- section objects,** 139, 852
- sectors,** 450, 466, 566
- sector slipping,** 467
- sector sparing,** 466
- Secure Boot,** 872
- secure by default,** 634
- secure kernel,** 839-840
- Secure Monitor Call (SMC),** 670
- secure shell,** 116
- secure system process,** 872
- secure systems,** 622
- Secure World,** 838
- security,** 621-665. See also **protection**
- of compiler-based enforcement, 692
  - in computer systems, 33-34
  - cryptography for, 637-648
    - and encryption, 638-645
    - implementation, 645-646
    - TLS example, 646-648
  - implementation of, 653-662
    - and accounting, 659
    - and auditing, 659
    - and firewalls, 659-660
    - and intrusion prevention, 655-657
    - levels of defenses, 661-662
    - and logging, 659
    - and security policy, 653
    - and virus protection, 657-659
    - and vulnerability assessment, 653-655
  - in Linux, 816-818
  - as operating system service, 57-58
  - as problem, 621-625
  - and program threats, 625-634
    - code injection, 628-631
    - malware, 625-628
    - viruses, 631-634
    - worms, 631, 632
  - and system/network threats, 634-637
  - user authentication for, 648-653

- in Windows 10, 662-664, 826-828, 878
- security access tokens (Windows 10), 662**
- security context (Windows 10), 662-663**
- security descriptor (Windows 10), 663**
- security domains, 659**
- security ID (SID), 33, 867**
- security policy, 653**
- security reference monitor (SRM), 866-869**
- security-through-obscurity approach, 655**
- security tokens, 867**
- seek, file, 532**
- seek time (disks), 450**
- segmentation, IA-32, 379-380**
- selective revocation, 682**
- semantics, 510, 608-609**
- semaphore(s), 272-276**
  - binary, 273
  - counting, 273
  - defined, 272
  - dining-philosophers solution with, 294-295
  - implementation, 274-276
  - in Java, 308-309
  - monitors using, 280-281
  - named, 300-301
  - POSIX examples, 300-302
  - unnamed, 300-302
  - usage of, 273-274
- semaphore objects (Windows 10), 845**
- semiconductor memory, 14**
- sends, blocking vs. nonblocking, 130**
- sending messages, 891**
- sense key, 512**
- separation hypervisors, 729**
- sequence numbers, 748**
- sequential access (files) 539, 541**
- sequential devices, 502**
- serial ATA (SATA) buses, 456**
- serial-attached SCSI (SAS) buses, 456, 490**
- serial dispatch queue, 185**
- server(s), 73**
  - blade, 18-19
  - bootstrap, 136
  - in client-server model, 606, 758-759, 861-862
  - defined, 757
  - in distributed systems, 734
  - file-server systems, 43
  - name, 739
  - and redirectors, 882-883
- server-initiated approach to verifying cached data, 767**
- Server Message Block (SMB), 880**
- server subject (Windows 10), 663**
- server systems, 42-43, 734, 874-875**
- service(s):**
  - defined, 757
  - denial of, 622, 636
  - distributed naming, 607
  - high-availability, 19
  - infrastructure as, 44
  - log-file, 878
  - network information, 607
  - operating system, 55-58, 74-75, 115, 152
  - platform as, 44
  - software as, 44
  - theft of, 622
- service control manager (SCM), 870**
- service processes, 123**
- service-trigger mechanism, 870**
- session(s), 751, 874**
- session 0, 873**
- session hijacking, 623**
- session key, 647**
- session layer, 742**
- session manager subsystem (SMSS), 872-873**
- session semantics, 609**
- sets:**
  - entry, 303, 304, 307
  - hard working-set limits, 438
  - of holes, 358
  - SMT, 242
  - wait, 304, 307
  - working, 422-424, 438
- setuid attribute, 34**
- setuid bit, 674-675**
- SHA-1 message digest, 643**
- shadow copies, 863**
- shareable devices, 502**
- shares, 244**
- shared directories, 547**
- shared files 609**
- shared libraries, 356, 392**
- shared lock, 534**
- shared memory, 123, 125, 556-560**
- shared-memory model, 57, 73, 125-127, 132-136**
- shared ready queue, 843**
- shared system interconnect, 18**
- sharing:**
  - file, 602-603
  - information, 123
  - load, 220
  - and paging, 369-371
  - resource, 162, 734-735
  - space, 592
- sharing objects, 885-886**

- shells, 58, 116, 783
  - shell scripts, 61, 536
  - short duration locks, 272
  - shortest-job-firs (SJF) scheduling algorithm, 207-209
  - shortest-next-CPU-burst algorithm, 207
  - shortest-remaining-time-firs scheduling, 209
  - shoulder surfing 649
  - sibling process, 111
  - SID (security ID), 33, 867
  - Siemens Jailhouse project, 729
  - signals, 188-189, 812-813
  - signal-and-continue method, 279
  - signal-and-wait method, 279
  - signaled state, 297
  - signal handlers, 188-189
  - signal-handler table, 789
  - signatures, 633, 643, 656, 828
  - signature-based detection, 656
  - silos, 859
  - SIMD, 833
  - simple messages, 136
  - simple subject (Windows 10), 662
  - simulations, 248-249
  - simultaneous multithreading, 222
  - single indirect blocks, 576
  - single-level directories, 542-543
  - single-processor systems, 15-16
  - single step (mode), 72
  - single-threaded processes, 160
  - single-user mode, 95
  - singly linked lists, 37
  - SIP (System Integrity Protection), 687-688
  - Siri, 5
  - sites, distributed system, 734
  - 64-bit computing, 383
  - SJF scheduling algorithm, 207-209
  - sketch, 70
  - slabs, 427-429, 797-798
  - slab allocation, 427-430, 797-798
  - Slackware, 779
  - sleep, return from, 243
  - SLIH (second-level interrupt handler), 496
  - slim reader-writer (SRW) locks, 889
  - SLOB allocator, 430
  - SLUB allocator, 430
  - small objects, 430
  - SMB (Server Message Block), 880
  - SMC (Secure Monitor Call), 670
  - SMP, see symmetric multiprocessing
  - SMSS (session manager subsystem), 872-873
  - SMT sets, 242
  - snapshots, 480, 588, 705, 879
  - sniffing 635-636, 649-650
  - social engineering, 624
  - sockets, 146-149
  - socket interface, 504
  - soft affinit , 225
  - soft errors, 463
  - soft page faults, 416
  - soft real-time systems, 227
  - software:
    - process migration and, 753
    - proprietary, 46-47
    - public domain, 779-780
  - software as a service (SaaS), 44
  - software engineering, 80
  - software interrupts (traps), 497
  - software objects, 672
  - software transactional memory (STM), 312
- Solaris**, 51
- file systems in, 482-484, 597, 599
  - Oracle SPARC, 375-376
  - scheduling example, 242-244
  - virtual memory in, 438-440
  - ZFS file system, 482-484, 581, 588, 598
- Solaris 10**:
- role-based access control in, 683, 684
  - zones in, 718, 719
- solid-state disks (SSDs)**, 452
- source-code viruses**, 633
- source files** 530
- space maps**, 581
- space sharing**, 592
- SPARC**, 375-376
- sparseness**, 374, 391
- special instructions**, 709
- special-purpose fil systems**, 597-598
- specifications thread behavior**, 169
- speed of operations (I/O devices)**, 502
- spinlocks**, 272
- split-screen**, 115
- spoofed identifiers** 606
- spoofing** 636
- spools**, 511
- spooling**, 511
- Springboard interface**, 60, 87
- spyware**, 626
- SRM (security reference monitor)**, 866-869
- SRW (slim reader-writer) locks**, 889
- SSDs (solid-state disks)**, 452
- stacks**, 37-38, 66
  - device, 862
  - LRU page replacement with, 408

- OSI protocol, 742-744  
**stack algorithms**, 408-409  
stack inspection, 694, 695  
stack section (of process), 107  
stalling, 350  
standard swapping, 377  
standby page, 856  
starvation (indefinit blocking), 213, 343  
**states**:  
    application, 378  
    new, 108  
    nonsignaled vs. nonsignaled, 297  
    of processes, 107-109  
    ready, 108, 109  
    running, 108, 109  
    safe, 331-333  
    suspended, 705, 888  
    terminated, 109  
    unsafe, 332-334  
    waiting, 108, 109  
**state information**, 608  
stateless DFS, 608  
state restore, 114  
state save, 114  
static linking, 355-356, 803  
static protection, 673  
status information, 74  
status register, 492  
stealth viruses, 633  
**STM (software transactional memory)**, 312  
**storage**, 11-14. See also **mass-storage structure**  
    cloud, 471, 751  
    content-addressable, 484  
    definitions and notations, 12  
    host-attached, 470  
    network-attached, 470-471  
    nonvolatile, 14, 449  
    object, 483-485  
    secondary, 13, 456, 570-578. See also  
        disk[s]  
    tertiary, 13  
    thread-local, 192, 894, 895  
    utility, 481  
    volatile, 11  
**storage-area networks (SANs)**, 21, 470, 472  
**storage array**, 472-473, 481  
**storage attachment**, 469-473  
**storage devices, organization of**, 597, 598  
**storage device management**, 463-467  
**storage management**, 30, 32, 723  
**stream ciphers**, 640  
**stream head**, 519  
**streaming transfer rate**, 486  
**stream modules**, 519  
**STREAMS mechanism**, 519-521  
**string, reference**, 404, 406  
**strongly ordered model**, 265  
**strong passwords**, 651  
**stubs**, 150  
**subjects (Windows 10)**, 662-663  
**subsets, stack algorithm**, 408  
**subsystems**, 75  
**SunOS**, 51  
**superblock**, 566  
**superblock objects**, 605, 804, 805  
**supervisor mode**, see **kernel mode**  
**SuSE**, 779  
**suspended state**, 705, 888  
**swap map**, 469  
**swapping**, 113-114, 376-378  
    in Linux, 800  
    on mobile systems, 377-378  
    with paging, 377, 378  
    standard, 377  
    system performance under, 378  
**swap space**, 395, 468-469  
**swap-space management**, 467-469  
**SwitchBranch mechanism**, 830  
**switches, context**, 114-115, 204  
**switching**:  
    domain, 673, 674  
    fast-user, 825, 874-875  
**symbolic links**, 879  
**symmetric clustering**, 20  
**symmetric encryption**, 639-640  
**symmetric encryption algorithm**, 639  
**symmetric multiprocessing (SMP)**, 16, 220, 794  
**symmetry, in addressing**, 129  
**synchronization**, 130-131, 289-314  
    alternative approaches to, 311-314  
    block, 305  
    bounded-buffer problem, 290  
    dining-philosophers problem, 293-295  
    for interprocess communication, 812-813  
    in Java, 303-311  
        condition variables, 309-311  
        monitors, 303-307  
    reentrant locks, 307-308  
    semaphores, 308-309  
    kernel, 295-299, 792-794  
    in Linux, 130-131, 812-813  
    in message-passing model, 130-131  
    in POSIX, 299-303  
    process, 260. See also **synchronization tools**

- readers-writers problem, 290-293
  - thread, 888-889
  - synchronization primitives**, 845-846
  - synchronization tools**, 257-287
    - about, 257-260
    - critical-section problem, 260-270
      - hardware solution to, 265-270
      - Peterson's solution to, 262-265
    - evaluation of, 284-286
    - and liveness, 283-284
    - monitors for, 276-282
      - resumption of processes within, 281-282
    - semaphores, implementation using, 280-281
    - usage, 277-280
  - mutex locks, 270-272
    - semaphores for, 272-276
  - synchronous devices**, 502, 506, 507
  - synchronous message passing**, 130
  - synchronous threading**, 169
  - synchronous writes**, 585
  - system administrators**, 60
  - system build**, 92
  - system calls (monitor calls)**, 22, 62-74
    - and API, 63-66
    - clone(), 195-196
    - for communication, 72-73
    - for device management, 71-72
    - exec(), 188
    - for file management, 71
    - fork(), 188
    - functioning of, 62-63
    - for information maintenance, 72
    - for I/O, 512, 513
    - for process control, 66-71
      - for protection, 73-74
  - system-call filtering**, 688
  - system-call firewalls**, 660
  - system-call interface**, 65
  - system components (Windows 10)**, 838-874
    - executive, 848-874
    - hardware-abstraction layer, 840
    - hyper-V hypervisor, 839
    - kernel, 840-848
      - secure kernel, 839-840
  - system-contention scope (SCS)**, 218
  - system daemons**, 22, 781
  - system-development time**, 705
  - system disk**, 465
  - systemd process**, 117
  - system goals**, 79
  - System Integrity Protection (SIP)**, 687-688
  - system libraries (Linux)**, 781
  - system mode**, see **kernel mode**
  - system model**, for deadlocks, 318-319
  - system processes**, 872
  - system programs**, 6
  - system resource-allocation graph**, 323-326
  - system restore point**, 871
  - system utilities**, 74-75, 781
  - System V init**, 117
  - system-wide open-file table**, 567
  - system-wide tools**, 96, 97
- T**
- table(s)**. See also **page table(s)**
    - attribute-definition, 877
    - device-status, 508-509
    - event-vector, 11
    - file, 788
    - file-allocation, 574-575
    - frame, 365
    - global, 679
    - global descriptor, 379
    - handle, 849
    - hash, 570
    - master file, 566
    - mount, 517, 567
    - open-file, 533, 567
    - page directory pointer, 381
    - per-process open-file, 567
    - PTE, 853
    - signal-handler, 789
    - system-wide open-file, 567
  - tags**, 680
  - tapes, magnetic**, 455
  - target latency**, 236, 791
  - target thread**, 190
  - tasks**, 106, 135, 195, 234. See also **user programs (user tasks)**
  - task control blocks**, see **process control blocks**
  - task identification for multicore programming**, 163
  - task parallelism**, 165, 166
  - Task Self port**, 135
  - TB (terabyte)**, 12
  - TBBs (thread building blocks)**, 186-188
  - TCP (transmission control protocol)**, 743-749
  - TCP/IP**, see **Transmission Control Protocol/Internet Protocol**
  - TCP sockets**, 147
  - TDI (Transport Driver Interface)**, 880

- TEBs (thread environment blocks),** 889-890
- templating,** 706
- temporary revocation,** 682
- terabyte (TB),** 12
- terminal concentrators,** 522
- terminal server systems,** 874-875
- terminated state,** 109
- termination,** 121-123, 342
- tertiary storage devices,** 13
- testing, multicore programming in,** 165
- text files** 530
- text section (of process),** 106
- theft of service,** 622
- thin-client computing,** 874-875
- thin clients,** 40
- third extended file system (ext3),** 805-807
- 32-byte memory,** 363, 364
- thrashing,** 419-425
  - cause of, 419-422
  - current practice, 425
  - and page-fault-frequency strategy, 424-425
  - and working-set model, 422-424
- threads, 159-197.** See also **threading**
  - alertable, 846
  - green, 167
  - hardware, 222
  - idle, 239, 842
  - Java, 173-176
  - kernel, 166, 217, 234
  - kernel-mode, 841
  - in Linux, 789-790
  - and multicore programming, 162-166
  - in operating systems, 194-196
  - and process model, 110
  - Pthreads, 169-171, 218-219
  - scheduling of, 199
  - target, 190
  - user, 166, 217
  - user-mode, 841
  - in Windows 10, 841-845, 886-889, 894
- Thread attach,** 860
- thread building blocks (TBBs),** 186-188
- thread cancellation,** 190-192
- thread dumps,** 339
- thread environment blocks (TEBs),** 889-890
- threading:**
  - asynchronous, 169
  - hyper-, 222, 832
  - implicit, 176-188
    - fork join, 180-183
  - Grand Central Dispatch, 185-186
- Intel thread building blocks,** 186-188
- OpenMP and,** 183-185
- thread pools and,** 177-180
- issues:**
  - fork() and excel() system calls, 188
  - scheduler activations, 192-194
  - signal handling, 188-190
  - thread cancellation, 190-192
  - thread-local storage, 192
  - multi-, 166-168, 222, 223
  - synchronous, 169
- thread libraries,** 168-176
  - about, 168-169
  - Java, 173-176
  - Pthreads, 169-171
  - Windows, 171-173
- thread-local storage (TLS),** 192, 894, 895
- thread pools,** 177-180, 889
- thread scheduling,** 199
  - in Linux, 790-791
  - in Windows 10, 844-845
- threats,** 622
  - program, 625-634
    - code injection, 628-631
    - malware, 625-628
    - viruses, 631-634
    - worms, 631, 632
  - system/network, 634-637
- three-way handshake,** 748
- throughput,** 204-205
- thunking,** 830
- tightly coupled systems,** 83
- time:**
  - compile, 352
  - down, 572
  - effective access, 397-398
  - effective memory-access, 367
  - execution, 353
  - load, 353
  - mean time between failures, 473, 474
  - mean time of data loss, 474
  - mean time to repair, 474
  - positioning, 450
  - random-access, 450
  - response, 23, 205
  - seek, 450
  - system-development, 705
  - turnaround, 205
  - virtual run, 236
  - waiting, 205
- time quantum,** 209-211, 243
- time quantum expired,** 243
- timers,** 26-27, 505-506
- timer objects,** 845

- time slice, 209-211, 790-791  
 timestamps, 531  
 timestamp counters (TSCs), 845  
 TLB, see translation look-aside buffer  
 TLB miss, 366  
 TLB reach, 432-433  
 TLB walk, 376  
 TLS (thread-local storage), 192, 894, 895  
 TLS (Transport Layer Security), 646-648  
 tmpfs (file system), 598  
 top half (interrupt service routines), 793-794  
 total revocation, 682  
 touch screens, 5  
 touch-screen interface, 56, 60  
 trace files, 248  
 tracing tools, 97-98  
 tracks, disk, 450  
 traditional computing, 40-41  
 traffic network, 635-636  
 transactions, 311, 587, 808  
 transactional memory, 311-312  
 transfer rates, 450, 451, 486  
 transition page, 856  
 translation:  
     binary, 708-710  
     flash translation layer, 453-454  
     network address, 723  
     path-name, 614-615  
 translation granules, 383  
 translation look-aside buffer (TLB), 365-368, 376, 384, 855  
 translation table base register, 383  
 transmission control protocol (TCP), 743-749  
 Transmission Control Protocol/Internet Protocol (TCP/IP), 36, 743-746, 880-881  
 transparency, 756, 761  
 Transport Driver Interface (TDI), 880  
 transport layer, 742  
 transport-layer protocol (TCP), 645  
 Transport Layer Security (TLS), 646-648  
 traps, 22, 89, 497, 847  
 trap-and-emulate method, 707-708  
 trap doors, 626, 627  
 traversing file system, 542  
 trees, 38, 39, 116  
 tree-structured directories, 545-547  
 TRIMming unused blocks, 581-582  
 trimming, automatic working-set, 438  
 triple DES, 639  
 triple indirect blocks, 576, 577  
 Trojan horses, 625-626  
 truncating files, 532  
 trusted addresses, 638  
 Trustlets, 838  
 TrustZone (TZ), 670, 671  
 TSCs (timestamp counters), 845  
 tunneling, attacks with, 659-660  
 turnaround time, 205  
 twisted pair cables, 736  
 two-factor authentication, 652  
 two-level directories, 543-545  
 two-level model, 168  
 two-level page-table scheme, 372-373  
 type 0 hypervisors, 702, 713-714  
 type 1 hypervisors, 703, 714-715  
 type 2 hypervisors, 703, 715-716  
 type safety (Java), 696  
 TZ (TrustZone), 670, 671
- U**
- UDP (user datagram protocol), 743, 746-748  
 UDP sockets, 147  
 UEFI (Unified Extensible Firmware Interface), 94  
 UFD (user file directory), 543  
 UFS (UNIX file system), 565-566, 598  
 UI, see user interface  
 UMDF (User-Mode Driver Framework), 864  
 UMS, see user-mode scheduling  
 unbounded buffer, 126  
 unbounded capacity (of queue), 132  
 UNC (Uniform Naming Convention), 881  
 uncontended loads, 285  
 uncontended locks, 271  
 unifie buffer cache, 583-585  
 Unifie Extensible Firmware Interface (UEFI), 94  
 unifie virtual memory, 583  
 Uniform Naming Convention (UNC), 881  
 unikernels, 728  
 universal serial buses (USBs), 456  
 Universal Windows Platform (UWP), 426  
 UNIX file system (UFS), 565-566, 598  
 UNIX operating system:  
     consistency semantics, 609  
     inode, 577  
     I/O kernel structure in, 513, 514  
     permissions in, 553  
     protection domain in, 674-675  
     system calls, 68  
     system structure, 82  
 unloader, module, 783

- unnamed data, 875
- unnamed semaphores, 300-302
- unsafe state, 332-334
- unstructured data, 484
- untrusted applet protection, 695
- upcalls, 193
- upcall handler, 193
- updating policy, cache, 766-767
- urgency value, 223
- URL loader, 695
- USB drive, 452
- USBs (universal serial buses), 456
- use, of resources, 318
- users, 4-5, 603
  - as domains, 674
  - multiple, file sharing between, 602-603
  - other users (class), 551
  - power, 60-61
- user accounts, 662
- user authentication, 648-653
- user control list, 561
- user datagram protocol (UDP), 743, 746-748
- user-defined signal handlers, 189
- user experience layer (macOS and iOS), 87
- user file directory (UFD), 543
- user goals, 79
- user IDs, 33, 531, 675
- user-initiated class, 185-186
- user-interactive class, 185
- user interface (UI), 4-5, 56, 58-62
- user mode, 24, 25, 782
- User-Mode Driver Framework (UMDF), 864
- user-mode scheduling (UMS), 241, 833, 890-891
- user-mode threads (UT), 841
- user programs (user tasks), 106, 353, 801-803
- user rights (Linux), 817
- user threads, 166, 217
- UT (user-mode threads), 841
- utility class, 186
- utility storage, 481
- UWP (Universal Windows Platform), 426
  
- V**
  
- VACB (virtual address control block), 865
- VADs (virtual address descriptors), 857
- valid-invalid bit, 368-369
- valid page, 856
- variables:
  - atomic, 269-270
  - condition, 278, 279, 302-303, 309-311, 889
  - variable class, 239
  - variable-partition schemes, 358
  - variable timer, 26
  - VCPU (virtual CPU), 707
  - vectored I/O, 507-508
  - verifie , 98
  - version control system, 49
  - vfork() (virtual memory fork), 400
  - VFS (virtual file system), 804-805
  - VFS layer, 601
  - victim, for resource preemption, 343
  - victim frames, 402
  - views, 557, 852
  - virtual address, 354
  - virtual address control block (VACB), 865
  - virtual address descriptors (VADs), 857
  - virtual address space, 390, 391, 799-800
  - VirtualBox project, 704
  - virtual CPU (VCPU), 707
  - virtual file-systems, 603-605
  - virtual file system (VFS), 804-805
  - virtual file system (VFS) layer, 601
  - virtualization, 34-35
    - defined, 701
    - operating-system components for, 719-726
      - CPU scheduling, 720
      - I/O devices, 722-723
      - live migration, 724-726
      - memory management, 721-722
      - storage management, 723
    - para-, 703, 716-717
    - programming-environment, 703, 717
    - research, 728-729
  - virtual machines, 34, 701-730. See also **virtualization**
    - benefits of, 704-707
    - building blocks, 707-713
      - binary translation, 708-710
      - hardware assistance, 710-713
      - trap-and-emulate method, 707-708
    - examples, 726-728
    - features of, 704-707
    - history of, 703-704
    - implementations, 713-719
      - application containment, 718-719
      - emulation, 717-718
    - paravirtualization, 716-717
    - programming-environment virtualization, 717
    - type 0 hypervisors, 713-714
    - type 1 hypervisors, 714-715

- type 2 hypervisors, 715-716
  - and virtual machine life cycle, 713
  - life cycle, 713
- virtual machine control structures (VMCSs), 711**
- virtual machine managers (VMMs), 25-26, 35, 702**
- virtual machine sprawl, 713**
- virtual memory, 24, 389-441**
  - background on, 389-392
  - and copy-on-write technique, 399-401
  - demand paging for conserving, 392-399, 430-436
    - basic mechanism, 393-396
    - free-frame list, 396-397
    - with inverted page tables, 433
    - and I/O interlock, 434-436
    - and page size, 431-432
    - and performance, 397-399
    - and prepaging, 430-431
    - and program structure, 433-434
    - and TLB reach, 432-433
  - direct virtual memory access, 500
  - and frame allocation, 413-419
    - allocation algorithms, 414-415
    - global vs. local allocation, 415-418
    - minimum number of frames, 413-414
    - non-uniform memory access, 418-419
  - kernel, 801
    - and kernel memory allocation, 426-430
  - in Linux, 798-801
  - and memory compression, 425-426
  - network, 765
  - operating-system examples, 436-440
  - page replacement for conserving, 401-413
    - and application performance, 412-413
    - basic mechanism, 401-404
    - counting-based page replacement, 411-412
    - FIFO page replacement, 404-406
    - LRU-approximation page replacement, 409-411
    - LRU page replacement, 407-409
    - optimal page replacement, 406-407
    - and page-buffering algorithms, 412
  - and thrashing, 419-425
    - cause, 419-422
    - current practice, 425
    - page-fault-frequency strategy, 424-425
    - working-set model, 422-424
  - unified, 583
  - in Win32 API, 892, 893
- virtual memory context, 789**
- virtual memory fork, 400**
- virtual memory (VM) manager, 851-858**
- virtual memory regions, 799**
- virtual private networks (VPNs), 646, 881**
- virtual run time, 236**
- virtual to physical (V-to-P) conversion, 724**
- Virtual Trust Levels (VTLs), 838**
- virus dropper, 632**
- viruses, 631-634, 657-659**
- virus signatures, 633**
- visible processes, 122**
- VMCSs (virtual machine control structures), 711**
- VM manager, 851-858**
- VMMs, see virtual machine managers**
- VMware, 704, 726-727**
- vnode, 604**
- voice over IP (VoIP), 44**
- voice recognition, 5**
- volatile memory, 454-455**
- volatile storage, 11**
- volume, 464-465, 474**
- volume control block, 566**
- volume file, 876-877**
- volume shadow copies, 879**
- voluntary context switches, 204**
- von Neumann architecture, 12**
- VPNs (virtual private networks), 646, 881**
- VSM Enclaves, 840**
- VTLs (Virtual Trust Levels), 838**
- V-to-P conversion, 724**
- VT-x instructions, 710**
- vulnerability assessment, 653-655**

## W

- WAFL fil system, 589-593**
- wait-for graph, 337, 338**
- waiting, busy, 272**
- waiting state, 108, 109**
- waiting time, 205**
- wait() method, 305-307**
- wait queue, 112**
- wait set, 304, 307**
- wait() system call, 119, 121-122**
- WANs, see wide-area networks**
- weakly ordered model, 265**
- wear leveling, 454**
- Web-distributed authoring and versioning (WebDAV), 881**
- well-known ports, 146**
- well-known port numbers, 747**

- wide-area networks (WANs)**, 36, 735, 737-738
- WiFi (wireless) networks**, 41, 736-737
- Win32 API**, 884-895
  - creating process, 119-120
  - IPC with Windows messaging, 891-892
  - kernel object access, 884-885
  - memory management, 892-895
  - process management, 886-891
  - shared memory, 556-560
  - sharing objects between processes, 885-886
- Windows operating system (generally)**:
  - anonymous pipes, 141, 145
  - booting from storage device, 466
  - interprocess communication example, 138-139
  - scheduling example, 239-242
  - synchronization within kernels, 296-298
  - system calls, 68
  - threads, 194-195
- Windows 7**, 465, 822
- Windows 8**, 823
- Windows 10**, 821-896
  - access-control list management in, 555
  - design principles, 826-838
    - application compatibility, 830-831
    - dynamic device support, 837-838
    - energy efficiency, 836-837
    - extensibility, 833-834
    - international support, 835-836
    - performance, 831-833
    - portability of, 834-835
    - reliability, 828-829
    - security, 826-828
  - developments, 823-825
  - fast-user switching with, 874-875
  - file system, 875-879
  - history of, 821-825
  - networking, 880-884
  - programmer interface, 884-895
    - IPC with Windows messaging, 891-892
    - kernel object access, 884-885
    - memory management, 892-895
    - process management, 886-891
    - sharing objects between processes, 885-886
  - security in, 662-664
  - system components, 838-874
    - executive, 848-874
    - hardware-abstraction layer, 840
    - hyper-V hypervisor, 839
    - kernel, 840-848
- secure kernel, 839-840
- terminal services, 874-875
- virtual memory in, 437-438
- Windows Desktop Bridge**, 823
- Windows Driver Foundation**, 864
- Windows executive**, 848-874
  - booting, 872-874
  - cache manager, 864-866
  - client-server computing, 861-862
  - I/O manager, 862-864
  - object manager, 849-851
  - plug-and-play manager, 869-870
  - power manager, 870-871
  - process manager, 858-860
  - registry, 871-872
  - security reference monitor, 866-869
  - virtual memory manager, 851-858
- Windows group policy**, 884
- Windows messaging**, 891-892
- Windows Store**, 823
- Windows subsystem for Linux (WSL)**, 91
- Windows Task Manager**, 97
- Windows thread library**, 171-173
- Windows Vista**, 822
- Windows XP**, 822
- WinRT**, 823
- Winsock**, 891
- wired down entries**, 366
- wireless access points**, 736
- wireless (WiFi) networks**, 41, 736-737
- word**, 11
- Work (data structure)**, 335, 339, 340
- working sets**, 422-424, 438
- working-set maximum**, 438
- working-set minimum**, 438
- working-set model**, 422-424
- working-set window**, 422
- Workstation (VMWare)**, 726-727
- work stealing algorithm**, 182
- world rights (Linux)**, 817
- World Wide Web**, 605, 737
- worms**, 631, 632
- worst-fi strategy**, 358, 359
- writes, synchronous vs. asynchronous**, 585
- write access, locks with**, 292
- write amplification** 462
- write-anywhere file layout (WAFL) file system**, 589-593
- write-back caching**, 766
- write-end (of pipe)**, 140
- write once devices**, 502
- write-on-close policy**, 766-767
- write pointer**, 532

writers, 291  
write-through policy, 766  
writing files, 532, 551  
WSL (Windows subsystem for Linux), 91

**X**

x86-64 architecture, 382  
XDR (external data representation), 150  
Xen, 704, 716-717  
XML firewalls, 660  
Xtratum, 729

**Y**

yellow pages, 607

**Z**

zero capacity (of queue), 131  
zero-day attacks, 656  
zeroed page, 856  
zero-fill-on-demand technique, 397  
ZFS file system, 482-484, 581, 588, 598  
zombie process, 122  
zombie systems, 634, 635  
zones, 718, 719, 795

---

# Glossary

**50-percent rule** A statistical finding that fragmentation may result in the loss of 50 percent of space.

**absolute code** Code with bindings to absolute memory addresses.

**absolute path name** A path name starting at the top of the file system hierarchy.

**abstract data type (ADT)** A programming construct that encapsulates data with a set of functions to operate on that data that are independent of any specific implementation of the ADT.

**access matrix** An abstract model of protection in which each row represents a domain, each column an object, and each entry a set of access rights.

**access right** The ability to execute an operation on an object.

**access-control list** A list of user names allowed to access a file.

**acknowledgment packet** In networking, a packet sent in response to the successful receipt of a message or packet.

**activation record** A record created when a function or subroutine is called; added to the stack by the call and removed when the call returns. Contains function parameters, local variables, and the return address.

**active directory (AD)** The Windows distributed information system, which is the Windows implementation of LDAP.

**acyclic graph** In directory structure implementation, a structure that contains no cycles (loops).

**adaptive mutex** A Solaris scheduling feature that starts as a standard spinlock and, if the object is locked and the lock is not held by a thread running on a CPU, blocks and sleeps until the lock is released.

**address space layout randomization (ASLR)** An operating system technique to avoid code-injection attacks that place memory objects like the stack and heap at unpredictable locations.

**address windowing extension (AWE)** A Windows mechanism for memory allocation that allows developers to directly request free pages of

RAM from the memory manager and later commit virtual memory on top of those pages.

**address-space identifier** A part of a TLB entry that identifies the process associated with that entry and, if the requesting process doesn't match the ID, causes a TLB miss for address-space protection.

**address-space layout randomization (ASRL)** A Windows 7 feature that randomizes process memory addresses to avoid viruses that jump to specific code locations to gain privileges.

**admission control** In real-time scheduling, a practice whereby the scheduler may not allow a process to start if it cannot guarantee that the task will be serviced by its deadline.

**advanced configuration and power interface (ACPI)** Firmware common to PCs and servers that manages certain aspects of hardware, including power and device information.

**advanced encryption standard (AES)** The NIST cipher designed to replace DES and triple DES.

**advanced local procedure call (ALPC)** In Windows OS, a method used for communication between two processes on the same machine.

**advanced technology attachment (ATA)** An older-generation I/O bus.

**advisory file-lock mechanism** A file-locking system in which the operating system does not enforce locking and file access, leaving it to processes to implement the details.

**AFS (OpenAFS)** A network file system designed at Carnegie Mellon University with the goal of enabling servers to support as many clients as possible.

**aging** A solution to scheduling starvation that involves gradually increasing the priority of threads as they wait for CPU time.

**ahead-of-time (AOT) compilation** A feature of the Android RunTime (ART) virtual machine environment in which Java applications are compiled to native machine code when they are installed on a system (rather than just in time, when they are executed).

## G-2      Glossary

**allocation problem** The determination by the operating system of where to store the blocks of a file.

**Amazon Elastic Compute Cloud (ec2)** An instance of cloud computing implemented by Amazon.

**AMD 64** A 64-bit CPU designed by Advanced Micro Devices; part of a class of CPUs collectively known as x86-64.

**AMD-V** AMD CPU virtualization technology.

**analytic evaluation** A means of comparing scheduling-algorithm effectiveness by analyzing an algorithm against a workload and assigning it a score.

**anomaly detection** In intrusion detection, the use of various techniques to detect anomalous behavior that could be a sign of an attack.

**anonymous access** Remote access that allows a user to transfer files without having an account on the remote system.

**anonymous memory** Memory not associated with a file. Pages not associated with a file, if dirty and paged out, must not lose their contents and are stored in swap space as anonymous memory.

**Apple file system (APFS)** The 2017 file system from Apple that is the default on all modern Apple devices; features a rich feature set, space sharing, clones, snapshots, and copy-on-write design.

**Apple iOS** The mobile operating system created by Apple Inc.

**application component** In Android, a basic building block that provides utility to an Android app.

**application containment** A virtualization-like operating system feature that segregates applications from the operating system (examples include Solaris Zones, BSD Jails, and IBM WPARs).

**application frameworks layer** In the layered macOS and iOS operating system design, the layer that includes Cocoa and Cocoa Touch frameworks, providing an API for Objective-C and Swift programming languages.

**application programming interface (API)** A set of commands, functions, and other tools that can be used by a programmer in developing a program.

**application program** A program designed for end-user execution, such as a word processor, spreadsheet, compiler, or Web browser.

**application proxy firewall** A firewall that understands protocols spoken by applications across a network, accepts connections to a target, and creates connections to that target, limiting and fixing what is sent from the originator.

**application state** A software construct used for data storage.

**arbitrary code guard (ACG)** A Windows 7 exploit-mitigation feature.

**argument vector** In Linux and UNIX, a list containing the command-line arguments used to invoke a process (and available to the process).

**ASIC** An application-specific integrated circuit (hardware chip) that performs its tasks without an operating system.

**assignment edge** In a system resource-allocation graph, an edge (arrow) indicating a resource assignment.

**asymmetric clustering** A configuration in which one machine in a cluster is in hot-standby mode while the other is running applications.

**asymmetric encryption algorithm** A cipher algorithm in which different keys are used for encryption and decryption.

**asymmetric multiprocessing** A simple multiprocessor scheduling algorithm in which only one processor accesses the system data structures and others run user threads, reducing the need for data sharing. A boss processor controls the system; the other processors either look to the boss for instruction or have predefined tasks.

**asynchronous** In I/O, a request that executes while the caller continues execution.

**asynchronous procedure call (APC)** A facility that enables a user thread to specify a function that is to be called when the user thread receives notification of a particular event.

**asynchronous threading** Threading in which a parent creates a child thread and then resumes execution, so that the parent and child execute concurrently and independently of one another.

**asynchronous write** A write that is buffered and written in arbitrary order, with the requesting thread continuing execution after the write is requested.

**atomic safe-save** In APFS, a feature primitive that performs file, bundle of files, and directory renaming in single atomic operations.

**atomic variable** A programming language construct that provides atomic operations on basic data types such as integers and booleans.

**atomically** A computer activity (such as a CPU instruction) that operates as one uninterruptable unit.

**attack** An attempt to break a computer system's security.

**attack surface** The sum of the methods available to attack a system (e.g., all of the network ports that are open, plus physical access).

**attacker** Someone attempting to breach a computer system's security.

**attribute** In the Windows NTFS file system, one of the elements making up a file. Each file is seen as a structured object consisting of typed attributes, with each attribute an independent byte stream that can be created, deleted, read, and written.

**audit trail** The collection of activities in a log for monitoring or review.

**authentication** The process of correctly identifying a person or device. In cryptography, constraining the set of potential senders of a message.

**automatic working-set trimming** In Windows, a process whereby, if a threshold of minimum free memory is reached, the number of working-set frames is decreased for every process.

**automount** A network/distributed file system feature in which file systems from remote servers are automatically located and mounted as needed.

**autoprobe** In Linux, a device driver probe that auto-detects device configuration.

**B+ tree** A tree data structure in which every path from the root of the tree to the leaf is the same length.

**back door** A daemon left behind after a successful attack to allow continued access by the attacker. In cryptography, a method of gaining access to encrypted information without first having the secret keys. More generally, a method of passing arbitrary commands or information when an interface does not provide a standard method.

**background** Describes a process or thread that is not currently interactive (has no interactive input directed to it), such as one not currently being used by a user. In the Grand Central Dispatch Apple OS scheduler, the scheduling class representing tasks that are not time sensitive and are not visible to the user.

**backing store** The secondary storage area used for process swapping.

**backup** In file systems, a copy or copies of the file system or changes to the file system that can be used to restore the file system if it is damaged or destroyed.

**bad block** An unusable sector on an HDD.

**balanced binary search tree** A tree containing  $n$  items that has, at most,  $\lg n$  levels, thus ensuring worst-case performance of  $O(\lg n)$ .

**bandwidth** The total amount of data transferred divided by the total time between the first request for service and the completion of the last transfer.

**banker's algorithm** A deadlock avoidance algorithm, less efficient than the resource-allocation graph

scheme but able to deal with multiple instances of each resource type.

**base file record** In the Windows NTFS file system, a descriptor of a large file containing pointers to overflow records that hold additional pointers and attributes.

**base register** A CPU register containing the starting address of an address space. Together with the limit register, it defines the logical address space.

**basic file system** A logical layer of the operating system responsible for issuing generic commands to the I/O control layer, such as "read block x," and also buffering and caching I/O.

**batch interface** A method for giving commands to a computer in which commands are entered into files, and the files are executed, without any human interaction.

**Bayes' theorem** A formula for determining conditional probability—e.g., the probability of an intrusion record detecting a real intrusion.

**Belady's anomaly** An anomaly in frame-allocation algorithms in which a page-fault rate may increase as the number of allocated frames increases.

**best-fit** In memory allocation, selecting the smallest hole large enough to satisfy the memory request.

**big data** Extremely large sets of data; distributed systems are well suited to working with big data.

**big.LITTLE** ARM processor implementation of HMP in which high-performance big cores are combined with energy efficient LITTLE cores.

**big-endian** A system architecture in which the most significant byte in a sequence of bytes is stored first.

**binary search tree** A type of binary tree data structure that requires an ordering between the parent's two children in which left child  $\leq$  right child.

**binary semaphore** A semaphore of values 0 and 1 that limits access to one resource (acting similarly to a mutex lock).

**binary translation** A virtualization method in which, when a guest is running in virtual kernel mode, every instruction is inspected by the virtual machine manager, and instructions that would behave differently in real kernel mode are translated into a set of new instructions that perform the equivalent task; used to implement virtualization on systems lacking hardware support for virtualization.

**binary tree** A tree data structure in which a parent may have at most two children.

## G-4      Glossary

**binder** In Android RPC, a framework (system component) for developing object-oriented OS services and allowing them to communicate.

**bind** Tie together. For example, a compiler binds a symbolic address to a relocatable address so the routine or variable can be found during execution.

**Bionic** A type of standard C library used by Android; it has a smaller memory footprint than glibc and is more efficient on slower (mobile) CPUs.

**BIOS** Code stored in firmware and run at boot time to start system operation.

**bit** The basic unit of computer storage. A bit can contain one of two values, 0 or 1.

**bit vector** A string of n binary digits that can be used to represent the status of n items. The availability of each item is indicated by the value of a binary digit: 0 means that the resource is available, while 1 indicates that it is unavailable (or vice-versa).

**bit-level striping** The splitting of data at the bit level, with each bit in a byte or word stored on a separate device.

**bitmap** A string of n binary digits that can be used to represent the status of n items. The availability of each item is indicated by the value of a binary digit: 0 means that the resource is available, while 1 indicates that it is unavailable (or vice-versa).

**BKL** In Linux kernel version 2.2, the “big kernel lock” spinlock, which protected all kernel data structures.

**blade server** A computer with multiple processor boards, I/O boards, and networking boards placed in the same chassis. The difference between these and traditional multiprocessor systems is that each blade-processor board boots independently and runs its own operating system.

**block** A self-contained unit of work. The smallest physical storage device storage unit, typically 512B or 4KB. In the Grand Central Dispatch Apple OS scheduler, a language extension that allows designation of a section of code that can be submitted to dispatch queues.

**block cipher** A cipher that works on blocks of data (rather than bits).

**block device** An I/O device that is randomly accessible, with block-size chunks being the smallest I/O unit.

**block-device interface** The interface for I/O to block devices.

**blocking** In interprocess communication, a mode of communication in which the sending process is

blocked until the message is received by the receiving process or by a mailbox and the receiver blocks until a message is available. In I/O, a request that does not return until the I/O completes.

**block-level striping** The splitting of data at the block level, with each block stored on a separate device.

**boot block** A block of code stored in a specific location on disk with the instructions to boot the kernel stored on that disk. The UFS boot control block.

**boot control block** A storage block of data containing information needed by the system to boot from the volume containing the block.

**boot disk** A disk that has a boot partition and a kernel to load to boot the system. A device that has a boot partition and can store an operating system for booting the computer.

**boot partition** A storage device partition containing an executable operating system.

**boot sector** The first sector of a Windows boot device, containing the bootstrap code.

**booting** The procedure of starting a computer by loading the kernel.

**bootstrap** The set of steps taken at computer power-on to bring the system to full operation.

**bootstrap loader** The small program that loads the kernel as part of the bootstrap procedure.

**bootstrap port** In Mach message passing, a pre-defined port that allows a task to register a port it has created.

**bootstrap program** The program that allows the computer to start running by initializing hardware and loading the kernel.

**bootstrap server** In Mach message passing, a system-wide service for registering ports.

**Border Gateway Protocol** A network protocol for determining network routes so that packets can move from source to destination across a WAN.

**bottleneck** A performance-limiting aspect of computing (e.g., poorly written code or a hardware component that is not as fast as others in the system).

**bounded buffer** A buffer with a fixed size.

**bounded waiting** A practice that places limits on the time a thread or process is forced to wait for something.

**Bourne-Again shell** A common shell, or command interpreter.

**BPF compiler collection (BCC)** A rich toolkit for tracing system activity on Linux for debugging and performance-tuning purposes.

**bridging** In networking, the connection of two devices—e.g., a virtual-machine guest and the network to which the host computer is connected.

**broadcast** In networking, the sending of one or more packets to all devices on the local network.

**browser** A process that accepts input in the form of a URL (Uniform Resource Locator), or web address, and displays its contents on a screen.

**buddies** Pairs of equal size, used in the buddy method of memory allocation.

**buddy heap** In Linux, the use of a power-of-two allocator for the kernel heap.

**buffer** A memory area that stores data being transferred (e.g., between two devices or between a device and a process).

**buffer cache** In file I/O, a cache of blocks used to decrease device I/O.

**bug** An error in computer software or hardware.

**bus** A communication system; e.g., within a computer, a bus connects various components, such as the CPU and I/O devices, allowing them to transfer data and commands.

**busy waiting** A practice that allows a thread or process to use CPU time continuously while waiting for something. An I/O loop in which an I/O thread continuously reads status information while waiting for I/O to complete.

**byte** Eight bits.

**bytecode** A computer object code resulting from compiling a program in a language (such as Java) that runs on a virtual machine.

**C library (libc)** The standard UNIX/Linux system API for programs written in the C programming language.

**cache** A temporary copy of data stored in a reserved memory area to improve performance. In the slab allocator, a cache consists of two or more slabs.

**cache coherency** The coordination of the contents of caches such that an update to a value stored in one cache is immediately reflected in all other caches that hold that value.

**cache management** The management of a cache's contents.

**cache-consistency problem** A challenge in caching in which the cached copies of data must be kept consistent with the master copy of the data.

**caching** The use of temporary data storage areas to improve performance.

**cancellation point** With deferred thread cancellation, a point in the code at which it is safe to terminate the thread.

**Canonical** A popular Linux distribution.

**capability** In protection, the representation of an object in a capability list.

**capability list** In protection, a list of objects together with the operations allowed on those objects.

**capability-based protection** A protection facility in which the powers of root are divided into specific abilities, each represented by a bit in a bit mask that is used to allow or deny operations.

**cascading termination** A technique in which, when a process is ended, all of its children are ended as well.

**Ceph** A brand of object storage management software.

**certificate authority** A trusted signer of digital certificates.

**character device** An I/O device that has a character (byte) as its smallest I/O unit.

**character-stream interface** The interface for I/O to character devices (like keyboards).

**checksum** The general term for an error detection and correction code.

**children** In a tree data structure, nodes connected below another node.

**chip multithreading (CMT)** A CPU with multiple cores, where each core supports multiple hardware threads.

**chipset** The CPU and support chips that create a computer and define its architecture.

**circular buffer** A buffer that uses a fixed amount of storage and wraps writes from the end of the buffer to the beginning when the buffer fills (so that the buffer acts as if it's circular in shape).

**Circular SCAN (CSCAN) scheduling** An HDD I/O scheduling algorithm in which the disk head moves from one end of the disk to the other performing I/O as the head passes the desired cylinders; the head then reverses direction and continues.

**claim edge** In the deadlock resource-allocation-graph algorithm, an edge indicating that a process might claim a resource in the future.

**class loader** In Java, a helper component that loads .class files for execution by the Java virtual machine.

**class** In Java, a program component that is a collection of data fields and functions (methods) that operate on those fields.

**clean-up handler** A function that allows any resources a thread has acquired to be released before the thread is terminated.

## G-6      Glossary

**client** A computer that uses services from other computers (such as a web client). The source of a communication.

**client interface** In distributed computing, a set of services provided to a caller of services.

**client system** A computer that uses services from other computers (such as a web client).

**client-server model** A mode of computing in which a server provides services to one or more clients. In distributed computing, a model in which a computer acts as a resource server to other computers that are clients of those resources.

**client-side caching (CSC)** In Windows, a caching method used to allow remote users to work off-line and then consolidate their changes once they are online.

**clock** In the second-chance page-replacement algorithm, a circular queue that contains possible victim frames. A frame is replaced only if it has not been recently referenced.

**clone** In file systems, a snapshot that is read-write and modifiable. In virtualization, a copy of a guest that enables another instance of the guest to run in a separate virtual machine.

**CLOOK scheduling** An HDD I/O scheduling algorithm that modifies CSCAN by stopping the head after the final request is completed (rather than at the innermost or outermost cylinder).

**closed-source** An operating system or other program available only in compiled binary code format.

**closure** In functional programming languages, a construct to provide a simple syntax for parallel applications.

**cloud computing** A computing environment in which hardware, software, or other resources are made available to customers across a WAN, such as the Internet, usually with APIs for management. A type of computing that delivers computing, storage, and even applications “as a service” across a network.

**cloud storage** Storage accessed from a computer over a network to a distant, shared resource data center.

**cluster** In Windows storage, a power-of-2 number of disk sectors collected for I/O optimization.

**cluster-based model** In distributed computing, a model of resource sharing where all systems are equal users and providers of resources.

**clustered file system (CFS)** A file system that is LAN-based and treats N systems storing data and Y clients accessing the data as a single client-server

instance; more complex than a client-server DFS but less complex than a cluster-based DFS. GPFS and Lustre are examples.

**clustered page table** A page table similar to a hashed page table, but a table entry refers to several (a cluster of) pages.

**clustered system** A system that gathers together multiple CPUs. Clustered systems differ from multiprocessor systems in that they are composed of two or more individual systems—or nodes—joined together.

**clustering** In general, gathering N items together. In virtual memory, paging in a group of contiguous pages when a single page is requested via a page fault.

**cluster** In file system block allocation, several contiguous blocks.

**coalescing** In general, combining. In the buddy memory allocation algorithm, freed memory in adjacent buddies can be coalesced into larger segments.

**code integrity** A Windows 7 module that checks the digital signatures of kernel modules to be sure they have not been tampered with by attackers.

**code review** A software development method in which the developer submits code to other developers for review and approval.

**code signing** The use of a digital signature to authenticate a program.

**code-injection attack** An attack that modifies otherwise well-behaved executable code.

**command interpreter** The operating system component that interprets user commands and causes actions based on them.

**command-line interface (CLI)** A method of giving commands to a computer based on a text input device (such as a keyboard).

**Common Criteria** The international 2005 successor to the Orange Book standard developed by the U.S. Department of Defense.

**common Internet file system (CIFS)** The Windows network file system, now used on many systems.

**communication port** In Windows OS, a port used to send messages between two processes.

**communications** A category of system calls.

**compaction** The shuffling of storage to consolidate used space and leave one or more large holes available for more efficient allocation.

**compartmentalization** The process of protecting each system component through the use of specific permissions and access restrictions.

**Completely Fair Queuing (CFQ)** In Linux, the default I/O scheduler in kernel 2.6 and later versions.

**Completely Fair Scheduler (CFS)** In Linux, the priority-based, preemptive scheduler included with the 2.6 kernel.

**component object model (COM)** The Windows mechanism for interprocess communication.

**compression** The use of algorithms to reduce the amount of data stored or sent.

**compression ratio** In memory compression, a measurement of the effectiveness of the compression (the ratio of the compressed space to the original amount of uncompressed space).

**compression unit** In NTFS, a unit of 16 contiguous clusters used in memory compression.

**computation migration** The use of a network to allow a task to remotely request resources to speed a computation.

**computation speedup** An increase in the amount of CPU compute power.

**computational kernel** A Windows mechanism for specifying tasks to run on GPUs.

**compute-server system** A server that provides an interface to which a client can send a request for an action (e.g., read data). In response, the server executes the action and sends the results to the client.

**Concurrency Runtime (ConcRT)** A Microsoft Windows concurrent programming framework for C++ that is designed for task-based parallelism on multicore processors.

**condition variable** A component of a monitor lock; a container of threads waiting for a condition to be true to enter the critical section.

**conditional-wait** A component of the monitor construct that allows for waiting on a variable with a priority number to indicate which process should get the lock next.

**confinement problem** The problem of guaranteeing that no information initially held in an object can migrate outside of its execution environment.

**conflict phase** During scheduling, the time the dispatcher spends moving a thread off a CPU and releasing resources held by lower-priority threads that are needed by the higher-priority thread that is about to be put onto the CPU.

**conflict-resolution mechanism** In Linux, the kernel module facility that allows different device drivers to reserve hardware resources and to protect those resources from accidental use by other drivers.

**congestion control** In networking, the attempt to approximate the state of networks between senders and receivers to avoid packet loss.

**connection port** In Windows OS, a communications port used to maintain connection between two processes, published by a server process.

**connectionless socket (UDP)** In Java, a mode of communication.

**connection-oriented socket (TCP)** In Java, a mode of communication.

**consistency checker** A system program, such as UNIX fsck, that reads file system metadata and tries to correct any inconsistencies (such as a file's length being incorrectly recorded).

**consolidation** In virtualization, the practice of running multiple guests per host, reducing the number of physical servers needed for a given workload.

**constant angular velocity (CAV)** A device-recording method in which the medium spins at a constant velocity and the bit density decreases from inner to outer tracks.

**constant linear velocity (CLV)** A device-recording method that keeps a constant density of bits per track by varying the rotational speed of the medium.

**consumer** A process role in which the process consumes information produced by a producer process.

**container** In application containment, a virtual layer between the operating system and a process in which the application runs, limiting its normal access to system resources.

**container object** In Windows 10, a category of objects that contain other objects (e.g., directories, which contain files).

**containers** In APFS, large free spaces in storage from which file systems can draw allocations.

**containment** A method for preventing deadlocks by creation and lock management of a higher-order resource that contains the locks of lower-order resources.

**contended** A term describing the condition of a lock when a thread blocks while trying to acquire it.

**content addressable storage** Another term for object storage; so called because objects can be retrieved based on their contents.

**context** When describing a process, the state of its execution, including the contents of registers, its program counter, and its memory context, including its stack and heap.

**context switch** The switching of the CPU from one process or thread to another; requires performing a state save of the current process or thread and a state restore of the other.

## G-8      Glossary

**contiguous allocation** A file-block allocation method in which all blocks of the file are allocated as a contiguous chunk on secondary storage.

**contiguous bit** In ARM v8 CPUs, a TLB bit indicating that the TLB holds a mapping to contiguous blocks of memory.

**contiguous memory allocation** A memory allocation method in which each process is contained in a single section of memory that is contiguous to the section containing the next process.

**control partition** In type 0 virtualization, a virtual hardware partition that provides services to the other partitions (and has higher privileges).

**control program** A program that manages the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.

**control register** A device I/O register where commands are placed by the computer.

**control-flow guard (CFG)** A Windows 7 exploit-mitigation feature.

**controller** A special processor that manages I/O devices.

**convoy effect** A scheduling phenomenon in which a number of threads wait for one thread to get off a core, causing overall device and CPU utilization to be suboptimal.

**cooperating process** A process that can affect or be affected by other processes executing in the system.

**cooperative** A form of scheduling in which threads voluntarily move from the running state.

**coordination** Ordering of the access to data by multiple threads or processes.

**copy protection** The implementation of a mechanism to prevent the unauthorized copying of a licensed work (e.g., media, programs, operating systems).

**copy semantics** The meaning assigned to data copying—e.g., whether a block write from a process allows the data to be modified after the write has been requested.

**copy-on-write** Generally, the practice by which any write causes the data to first be copied and then modified, rather than overwritten. In virtual memory, on a write attempt to a shared page, the page is first copied, and the write is made to that copy.

**core** Within a CPU, the component that executes instructions.

**core dump** A copy of the state of a process written to disk when a process crashes; used for debugging.

**core frameworks** In the layered macOS and iOS operating system design, the layer that defines frameworks that support graphics and media, including QuickTime and OpenGL.

**counting semaphore** A semaphore that has a value between 0 and N, to control access to a resource with N instances.

**CPU burst** Scheduling process state in which the process executes on CPU.

**CPU scheduler** Kernel routine that selects a thread from the threads that are ready to execute and allocates a core to that thread.

**CPU scheduling** The process by which the system chooses which job will run next if several jobs are ready to run at the same time.

**CPU-bound process** A process that spends more time executing on CPU than it does performing I/O.

**crash** Termination of execution due to a problem. A failure in the kernel results in a system crash and reboot, while a process failure results in a process crash.

**crash dump** A copy of the state of the kernel written to disk during a crash; used for debugging.

**critical section** A section of code responsible for changing data that must only be executed by one thread or process at a time to avoid a race condition.

**critical-section object** A user-mode mutex object that can often be acquired and released without kernel intervention; a Windows OS scheduling feature.

**cryptography** A tool used to constrain the potential senders and/or receivers of a message (or stored data).

**current-file-position pointer** A per-process pointer to the location in a file to which the next read or from which the next write will occur.

**cycle** A repeating loop.

**cycle stealing** The act of a device, such as a DMA controller, using the bus and preventing the CPU from using it temporarily.

**cylinder** On an HDD, the set of tracks under the read-write heads on all platters in the device.

**cylinder groups** A sequential collection of storage media (typically a group of cylinders) that are managed as a single unit and subdivided into blocks.

**daemon** A service that is provided outside of the kernel by system programs that are loaded into memory at boot time and run continuously.

**daisy chain** In computer I/O, a connection method involving connecting devices to each other in a string (device A to B, B to C, C to D, etc.).

**dark web** The part of the World Wide Web that is not easy to reach (say, by search engines) and that is sometimes used for bad behavior (such as selling information stolen in successful attacks).

**Darwin** The Apple code name for its open-source kernel.

**data execution prevention (DEP)** A Windows 7 exploit-mitigation feature.

**data migration** The entire transfer of one or more files from one site to another.

**data parallelism** A computing method that distributes subsets of the same data across multiple cores and performs the same operation on each core.

**data section** The data part of a program or process; it contains global variables.

**data striping** The splitting of data across multiple devices.

**data-encryption standard (DES)** A cipher (algorithm for doing encryption and decryption) provided by the U.S. National Institute of Standards and Technology (NIST).

**datagram** A basic transfer unit on a packet-switched network like the Internet protocol (i.e., a network packet).

**data-in register** A device I/O register where data is placed to be sent to the device.

**data-out register** A device I/O register where data is placed by the device to be read by the computer.

**DCOM** The distributed computing extension to object linking and embedding (OLE).

**deadlock** The state in which two processes or threads are stuck waiting for an event that can only be caused by one of the processes or threads.

**deadlock avoidance** An operating system method in which processes inform the operating system of which resources they will use during their lifetimes so the system can approve or deny requests to avoid deadlock.

**deadlock prevention** A set of methods intended to ensure that at least one of the necessary conditions for deadlock cannot hold.

**deadlocked** The state in which two processes or threads are stuck waiting for an event that can only be caused by one of the processes or threads.

**Debian** A popular Linux distribution.

**debugger** A system program designed to aid programmers in finding and correcting errors.

**debugging** The activity of finding and removing errors.

**deduplication** The removal of duplicate information (bits, blocks, or files).

**default heap** The heap data structure created when a Win32 process is initialized.

**default signal handler** The signal handler that receives signals unless a user-defined signal handler is provided by a process.

**defense in depth** The theory that more layers of defense provide stronger defense than fewer layers.

**deferred procedure call (DPC)** In Windows scheduling, a call initiated by the interrupt that occurs when a time quantum expires, eventually causing the expired thread to be moved off a core and replaced with the next thread in the ready queue.

**degree of multiprogramming** The number of processes in memory.

**delayed-write policy** In caching, a policy whereby data are first written to a cache; later, the cache writes the change to the master copy of the data.

**demand paging** In memory management, bringing in pages from storage as needed rather than, e.g., in their entirety at process load time.

**demand-zero memory** In Linux, a virtual memory region backed by nothing; when a process tries to read a page in such a region, it is given a page of memory filled with zeros.

**demilitarized zone** In firewalling, a security domain less trusted than some other security domain (e.g., the domain containing a web server compared to the domain containing the crucial company database).

**denial-of-service** Preventing legitimate use of a system.

**dirent object** The VFS representation of a directory.

**desktop** In a GUI, the standard workspace represented by the GUI on the screen, in which a user executes tasks.

**desktop activity moderator (DAM)** A Windows 8 component that supports the system state “connected standby” on mobile devices, freezing computer activity but allowing rapid return to full functionality.

**desktop window manager** A Windows Vista user-mode component to manage GUI windows.

**deterministic modeling** A type of analytic evaluation that takes a particular predetermined workload and defines the performance of each algorithm for that workload.

**development kernel** A kernel released for developers to use, rather than for production use.

**device controller** The I/O managing processor within a device.

## G-10 Glossary

**device driver** An operating system component that provides uniform access to various devices and manages I/O to those devices.

**device manipulation** A category of system calls.

**device object** In Windows, the object representing a device.

**device queue** The list of processes waiting for a particular I/O device.

**device-status table** A kernel data structure for tracking the status and queues of operations for devices.

**digital certificate** A public key digitally signed by a trusted party.

**digital rights management** The implementation of a mechanism to prevent the unauthorized copying of a licensed work (e.g., media, programs, operating systems).

**digital signature** The authenticator produced by a digital-signature algorithm.

**digital-signature algorithm** A cryptographic checksum calculated in asymmetric encryption; used to authenticate a message.

**dining-philosophers problem** A classic synchronization problem in which multiple operators (philosophers) try to access multiple items (chopsticks) simultaneously.

**direct access** A file-access method in which contents are read in random order, or at least not sequentially.

**direct blocks** In UFS, blocks containing pointers to data blocks.

**direct communication** In interprocess communication, a communication mode in which each process that wants to communicate must explicitly name the recipient or sender of the communication.

**direct I/O** Block I/O that bypasses operating-system block features such as buffering and locking.

**direct memory access (DMA)** A resource-conserving and performance-improving operation for device controllers allowing devices to transfer large amounts of data directly to and from main memory.

**direct virtual memory access (DVMA)** DMA that uses virtual addresses rather than physical memory addresses as transfer sources and destinations.

**dirty bit** An MMU bit used to indicate that a frame has been modified (and therefore must have its contents saved before page replacement).

**discretionary access control (DAC)** Optional, as opposed to mandatory, access control.

**disinfecting** In the context of computer viruses, removing the components of a virus.

**disk arm** An HDD component that holds the read-write head and moves over cylinders of platters.

**disk image** Generally, the contents of a disk encapsulated in a file. In virtualization, a guest virtual machine's boot file system contents contained in a disk image.

**diskless** A term describing systems that have no local storage.

**dispatch latency** The amount of time the dispatcher takes to stop one thread and put another thread onto the CPU.

**dispatch queue** An Apple OS feature for parallelizing code; blocks are placed in the queue by Grand Central Dispatcher (GCD) and removed to be run by an available thread.

**dispatched** Selected by the process scheduler to be executed next.

**dispatcher** The kernel routine that gives control of a core to the thread selected by the scheduler.

**dispatcher objects** A Windows scheduler feature that controls dispatching and synchronization. Threads synchronize according to several different mechanisms, including mutex locks, semaphores, events, and timers.

**distributed denial-of-service attack (DDoS)** An attack from multiple sources (frequently a botnet of zombies) with the purpose of denying legitimate use of the attacked resource.

**distributed file system (DFS)** A file system that works across a network in which remote directories are visible from a local machine.

**distributed information system** A set of protocols providing unified information needed for remote computing.

**distributed lock manager (DLM)** A function used by a clustered system to supply access control and locking to ensure that no conflicting operations occur.

**distributed naming service** A set of protocols providing unified information needed for remote computing.

**distributed system** A collection of loosely coupled nodes interconnected by a communication network.

**distribution** A release of a version of an operating system.

**docker** An orchestration tool for containers.

**domain switching** The mechanism for switching dynamic domains.

**domain-name system (DNS)** The Internet system for resolving host names to host-ids, in which

a distributed information system provides host-name-to-network-address translations for the Internet.

**double buffering** The copying of data twice (e.g., from a device to the kernel and then from the kernel to a process's address space), or the use of two buffers to decouple producers and consumers.

**double caching** The problem in which the same data might be in two different caches; solved by a unified buffer cache.

**double indirect block** In UFS, a block containing pointers to a single indirect block, which points to data blocks.

**down time** Generally, time during which a facility, system, or computing component are off-line and unavailable.

**driver end** The interface between STREAMS and the device being controlled.

**driver object** In Windows, the object representing a device driver.

**driver-registration system** In Linux, the kernel module facility that tells the rest of the kernel that a new driver is available.

**DTrace** A facility originally developed at Sun Microsystems that dynamically adds probes to a running system, both in user processes and in the kernel, for state analysis, performance tuning, and debugging.

**dual-booted** A term describing a computer that can boot one of two or more installed operating systems.

**dynamic loading** The loading of a process routine when it is called rather than when the process is started.

**dynamic random-access memory (DRAM)** The common version of RAM, which features high read and write speeds.

**dynamic storage-allocation** The class of file-block allocation methods that satisfy a request for n blocks from a list of free holes available.

**dynamic storage-allocation problem** The problem of how to satisfy a request for size n of memory from a list of free holes.

**dynamically linked libraries (DLLs)** System libraries that are linked to user programs when the processes are run, with linking postponed until execution time.

**earliest-deadline-first (EDF)** A real-time scheduling algorithm in which the scheduler dynamically assigns priorities according to completion deadlines.

**ease of use** The amount of difficulty and complexity involved in using some aspect of computing.

**EEPROM** Electrically erasable programmable read-only memory; a type of firmware.

**effective access time** The measured or statistically calculated time it takes to access something; e.g., see effective memory-access time.

**effective memory-access time** The statistical or real measure of how long it takes the CPU to read or write to memory.

**effective transfer rate** The actual, measured transfer rate of data between two devices (such as a computer and a disk drive).

**effective UID** The UID the process is currently using, which can be different from the login UID due to, e.g., escalating privileges.

**electrical storage** Storage devices made from electrical components, such as flash memory; one form of nonvolatile storage.

**ELF** The UNIX standard format for relocatable and executable files.

**embedded computer** A computer system within some other, larger system (such as a car) that performs specific, limited functions and has little or no user interface.

**emulation** A methodology used to enable a process to run when the compiled program's original (source) CPU type is different from the target CPU type on which the program is to run.

**emulator** A program that allows applications written for one hardware environment to run in a different hardware environment, such as a different type of CPU.

**encapsulate** In general, to enclose. In Java, encapsulation gives a class the ability to protect its data and methods from other classes loaded in the same JVM.

**encryption** The use of cryptography to limit the receivers of a message or access to data.

**entry section** The section of code within a process that requests permission to enter its critical section.

**entry set** In Java, the set of threads waiting to enter a monitor.

**environment vector** In Linux and UNIX, a list containing all of the environment variables of the shell that invoked a process (and are available to the process).

**environmental subsystem** In Windows, an operating environment that emulates a specific operating system by translating process API calls from that operating system to Windows calls.

## G-12 Glossary

**equal allocation** An allocation algorithm that assigns equal amounts of a resource to all requestors. In virtual memory, assigning an equal number of frames to each process.

**error-correcting code (ECC)** A value calculated from bytes of data and recalculated later to check for changes in the data.

**eSATA** A type of I/O bus.

**escalate privileges** To gain extra permissions for an activity, as when a user needs access to a device that is restricted.

**escape** Generally, a method of passing arbitrary commands or information when an interface does not provide a standard method.

**event latency** The amount of time between when an event occurs and when it is serviced.

**event** A Windows OS scheduling feature that is similar to a condition variable.

**eVM** An example of a partitioning hypervisor.

**exception** A software-generated interrupt caused either by an error (such as division by zero or invalid memory access) or by a specific request from a user program than an operating-system service be performed.

**exception dispatcher** The Windows component that processes exceptions.

**exclusive lock** A file lock similar to a writer lock in that only one process at a time can obtain the lock.

**executable and linkable format (ELF)** The UNIX standard format for relocatable and executable files.

**executable file** A file containing a program that is ready to be loaded into memory and executed.

**execution tracing** A Windows method of monitoring process execution and recording details for analysis and debugging.

**exit section** The section of code within a process that cleanly exits the critical section.

**expansion bus** A computer bus for connecting slow devices like keyboards.

**exponential average** A calculation used in scheduling to estimate the next CPU burst time based on the previous burst times (with exponential decay on older values).

**export list** The list of file systems available for remote mounting from a server.

**ext3** One of the most common types of Linux file systems.

**extended file attributes** Extended metadata about a file, including items such as character encoding details, file checksums, etc.

**extended file system** The most common class of Linux file systems, with ext3 and ext4 being the most commonly used file system types.

**extended file system (extfs)** The most common class of Linux file systems, with ext3 and ext4 being the most commonly used file system types.

**extensibility** The ability of an operating system to accommodate advances in computing technology via extensions (such as new kernel modules and new device drivers).

**extent** A chunk of contiguous storage space added to previously allocated space; a pointer is used to link the spaces.

**external data representation** A system used to resolve differences when data are exchanged between big- and little-endian systems.

**external fragmentation** Fragmentation in which available memory contains holes that together have enough free space to satisfy a request but no single hole is large enough to satisfy the request. More generally, the fragmentation of an address space into small, less usable chunks.

**fair scheduling** In the Completely Fair Scheduler, the scheduler algorithm that allots a proportion of the processor's time rather than time slices.

**false negatives** Results indicating that something is not a match to what is being detected, even though it is.

**false positives** Results indicating that something is a match to what is being detected, even though it isn't.

**fast directory sizing** In APFS, a feature that tracks and rapidly reports on directory sizes.

**fast I/O mechanism** In Windows, a high-speed bypass of the standard I/O handling mechanism in which the driver stack is called directly rather than having an IRP sent and processed.

**fault-tolerant system** A system that can suffer a failure of any single component and still continue operation.

**fiber** User-mode code that can be scheduled according to a user-defined scheduling algorithm.

**fibre channel (FC)** A type of storage I/O bus used in data centers to connect computers to storage arrays. A storage-attachment network.

**file** The smallest logical storage unit; a collection of related information defined by its creator.

**file attributes** Metadata about a file, such as who created it and when, file size, etc.

**file descriptor (fd)** UNIX open-file pointer, created and returned to a process when it opens a file.

**file handle** Windows name for the open-file file descriptor.

**file info window** A GUI view of the file metadata.

**file manipulation** A category of system calls.

**file mapping** In Windows, the first step in memory-mapping a file.

**file migration** A file system feature in which a file's location is changed automatically by the system.

**file object** The VFS representation of an open file.

**file reference** In Windows NTFS, a unique file ID that is the index of the file in the master file table (much like a UNIX inode number).

**file session** The series of accesses between open() and close() system calls that make up all the operations on a file by a process.

**file system** The system used to control data storage and retrieval; provides efficient and convenient access to storage devices by allowing data to be stored, located, and retrieved easily.

**File System Hierarchy Standard** A standard document specifying the overall layout of the standard Linux file systems.

**file-allocation table (FAT)** A simple but effective method of disk-space allocation used by MS-DOS. A section of storage at the beginning of each volume is set aside to contain the table, which has one entry per block, is indexed by block number, and contains the number of the next block in the file.

**file-control block (FCB)** A per-file block that contains all the metadata about a file, such as its access permissions, access dates, and block locations.

**file-organization module** A logical layer of the operating system responsible for files and for translation of logical blocks to physical blocks.

**file-server system** A server that provides a file-system interface where clients can create, update, read, and delete files (e.g., a web server that delivers files to clients running web browsers).

**filter drivers** In Windows, drivers allowed to insert themselves into the I/O processing chain.

**firewall** A computer, appliance, process, or network router that sits between trusted and untrusted systems or devices. It protects a network from security breaches by managing and blocking certain types of communications.

**firewall chains** In Linux, an ordered list of rules that specifies one of a number of possible firewall-decision functions plus matching components. A configuration of the firewall rules.

**firmware** Software stored in ROM or EEPROM for booting the system and managing low level hardware.

**first-come first-served (FCFS)** The simplest scheduling algorithm. The thread that requests a core first is allocated the core first.

**first-fit** In memory allocation, selecting the first hole large enough to satisfy a memory request.

**first-level interrupt handler** In some operating systems, an interrupt handler responsible for reception and queuing of interrupts; the interrupts are actually handled at another level (by the second-level handler).

**flash translation layer (FTL)** For nonvolatile memory, a table that tracks currently valid blocks.

**flow control** Generally, a method to pause a sender of I/O. In networking, a technique to limit the rate of data flow (e.g., to avoid buffer overflow and packet loss on a router).

**flush** Erasure of entries in, e.g., a TLB or other cache to remove invalid data.

**folder** A file system component that allows users to group files together.

**folder redirection** In Windows, for roaming users, a method for automatically storing a user's documents and other files on a remote server.

**foreground** Describes a process or thread that is interactive (has input directed to it), such as a window currently selected as active or a terminal window currently selected to receive input.

**foreground process** A process currently open and appearing on the display, with keyboard or other I/O device input directed to it.

**fork-join** A strategy for thread creation in which the main parent thread creates (forks) one or more child threads and then waits for the children to terminate and join with it.

**forward-mapped** Describes a scheme for hierarchical page tables in which address translation starts at the outer page table and moves inward.

**fourth extended file system (ext4)** In Linux, a current version of the extended file system, the successor to ext3.

**fragments** In networking, parts of messages. A message is split into fragments if it is too large to fit in one packet.

**frame table** In paged memory, the table containing frame details, including which frames are allocated, which are free, total frames in the system, etc.

**frame-allocation algorithm** The operating-system algorithm for allocating frames among all demands for frames.

**frames** Fixed-sized blocks of physical memory.

## G-14 Glossary

**free operating system** An operating system released under a license that makes its source code available and allows no-cost use, redistribution, and modification.

**free-behind** Sequential I/O performance optimization that removes a page or block from a buffer as soon as I/O to the next page is requested.

**free-frame list** A kernel-maintained data structure containing the list of currently available free frames of physical memory.

**free-space list** In file-system block allocation, the data structure tracking all free blocks in the file system.

**front-end processors** Small computers that perform certain tasks in an overall system; used by some systems to manage I/O and offload the general-purpose CPU.

**fsgid** In Linux, an added process property that allows file system access on behalf of another group.

**fsuid** In Linux, an added process property that allows file system access on behalf of another user.

**full backup** In file systems, a backup that includes all of the contents of a file system.

**functional language** A programming language that does not require states to be managed by programs written in that language (e.g., Erlang and Scala).

**fuzzing** A technique that tests for proper input validation (e.g., to avoid undetected buffer overruns).

**Galois field math** An advanced error-correcting calculation done in some forms of RAID.

**Gantt chart** A bar chart that can be used to illustrate a schedule.

**garbage collection** In general, recovery of space containing no-longer-valid data.

**general tree** A tree data structure in which a parent may have unlimited children.

**gestures** A user interface component in which motions cause computer actions (e.g., “pinching” the screen).

**gigabyte (GB)**  $1,024^3$  bytes.

**git** A version control system used for GNU/Linux and other programs.

**global replacement** In virtual memory frame allocation, allowing a process to select a replacement frame from the set of all frames in the system, not just those allocated to the process.

**GNU C compiler (gcc)** The standard C compiler of the GNU project, used throughout the industry on Linux and many other systems.

**GNU General Public License (GPL)** A license agreement that codifies copylefting (allowing and requiring open sourcing of the associated programs); a common license under which free software is released.

**GNU/Linux (aka Linux)** An open-source operating system composed of components contributed by the GNU foundation and Linus Torvalds, as well as many others.

**Google Android** The mobile operating system created by Google Inc.

**Google file system (GFS)** A cluster-based distributed file system designed and used by Google.

**GPFS** A common commercial clustered file system by IBM.

**graceful degradation** The ability of a system to continue providing service proportional to the level of surviving hardware.

**graphical user interface (GUI)** A computer interface comprising a window system with a pointing device to direct I/O, choose from menus, and make selections and, usually, a keyboard to enter text.

**graphics processing unit (GPU)** A hardware component, sometimes part of the CPU and sometimes a separate device, that provides graphics computation and sometimes graphics output to a display.

**graphics shaders** Processes that provide the shading within graphics images.

**group** In file permissions, a collection of users that have access to a file based on assigned access rights.

**group identifier** Similar to a user identifier, but used to identify a group of users to determine access rights.

**group rights** In file permissions, the access rights belonging to a user group.

**GRUB** A common open-source bootstrap loader that allows selection of boot partitions and options to be passed to the selected kernel.

**guard pages** In Windows, no-access-allowed pages at the tops of the kernel-mode and user-mode stacks that detect stack overflows.

**guest** In virtualization, an operating system running in a virtual environment (rather than natively on the computer hardware).

**hacker** Someone attempting to breach computer security.

**Hadoop distributed file system (HDFS)** A cluster-based distributed file system used for big-data projects.

**Hadoop file system** An example of object storage management software.

**haiku** A three-line poem in which the first line contains five syllables, the second line contains seven syllables, and the third line contains five syllables.

**handle** Generally, an opaque value that provides access to an object or data structure; e.g., a file handle is returned when a file is opened and is a pointer to an entry in an open-file table.

**handle table** In Windows, a per-process handle table containing entries that track (by their handles) the objects used by the process.

**hard affinity** The situation in which an operating system allows a process's threads to run on the same processor at all times (as opposed to being moved to various processors).

**hard disk drive (HDD)** A secondary storage device based on mechanical components, including spinning magnetic media platters and moving read-write heads.

**hard error** An unrecoverable error (possibly resulting in data loss).

**hard links** File-system links in which a file has two or more names pointing to the same inode.

**hard real-time systems** Systems in which a thread must be serviced by its deadline; service after the deadline has expired is the same as no service at all.

**hard working-set limit** In Windows memory management, the maximum amount of physical memory that a process is allowed to use.

**hardware** The CPU, memory devices, input/output (I/O) devices, and any other physical components that are part of a computer.

**hardware objects** The CPU, memory devices, input/output (I/O) devices, and any other physical components that are part of a computer

**hardware threads** Threads associated with the processing core. A given CPU core may run multiple hardware threads to optimize core use—e.g., to avoid memory stalls by switching hardware threads if the current thread causes a stall.

**hardware transactional memory (HTM)** A transactional memory implementation using hardware cache hierarchies and cache coherency protocols to manage and resolve conflicts involving shared data residing in separate processors' caches.

**hardware-abstraction layer (HAL)** A kernel layer that isolates chipset-specific hardware aspects from general-purpose code.

**hash function** A function that takes data as its input, performs a numeric operation on the data,

and returns a numeric value. Also, an algorithm for creating a hash (a small, fixed-size block of data calculated from a larger data set, used to determine if a message has been changed).

**hash map** A data structure that maps [key:value] pairs using a hash function; a hash function can then be applied to a key to obtain its matching value.

**hash value** The calculation resulting from a hash function.

**hashed page table** A page table that is hashed for faster access; the hash value is the virtual page number.

**head crash** On an HDD, a mechanical problem involving the read-write head touching a platter.

**heap section** The section of process memory that is dynamically allocated during process run time; it stores temporary variables.

**heartbeat** In computer clustering, a repeating signal to determine the state of the members of the cluster (e.g., to determine if a system is down).

**heterogeneous multiprocessing (HMP)** A feature of some mobile computing CPUs in which cores vary in their clock speeds and power management.

**high-availability** Describes a service that will continue even if one or more systems in the cluster fail.

**high-performance computing** A computing facility designed for use with a large number of resources to be used by software designed for parallel operation.

**high-performance event timer** A hardware timer provided by some CPUs.

**hit ratio** The percentage of times a cache provides a valid lookup (used, e.g., as a measure of a TLB's effectiveness).

**hives** In Windows, an internal repository of data.

**hole** In variable partition memory allocation, a contiguous section of unused memory. Also an alternative rock band formed by Courtney Love.

**honeypot** A false resource exposed to attackers; the resource appears real and enables the system to monitor and gain information about the attack.

**horizontal scalability** The ability to scale capacity not by expanding one item but by adding more items.

**host** In virtualization, the location of the virtual machine manager, which runs guest operating systems; generally, a computer.

**host bus adapter (HBA)** A device controller installed in a host bus port to allow connection of one or more devices to the host.

## G-16 Glossary

**host controller** The I/O-managing processors within a computer (e.g., inside a host bus adapter).

**host name** A human-readable name for a computer.

**host-attached storage** Storage accessed through local I/O ports (directly attached to a computer, rather than across a network or SAN).

**host-id** In networking, the unique number identifying a system on a network.

**hostname** The alphanumeric name for a system (such as becca.colby.edu).

**hot spare** An unused storage device ready to be used to recover data (e.g., in a RAID set).

**hot-standby mode** A condition in which a computer in a cluster does nothing but monitor the active server. If that server fails, the hot-standby host becomes the active server.

**huge pages** A feature that designates a region of physical memory where especially large pages can be used.

**hybrid cloud** A type of cloud computing that includes both public and private cloud components.

**hypcall** In paravirtualization, a call from a guest to the hypervisor to request a virtualization service, such as a page table change.

**hyper-threading** Intel's technology for assigning multiple hardware threads to a single processing core.

**hypervisor** The computer function that manages the virtual machine; also called a virtual machine manager (VMM).

**I/O burst** Scheduling process state in which the CPU performs I/O.

**I/O bus** A physical connection of an I/O device to a computer system.

**I/O channel** A dedicated, special-purpose CPU found in large systems like mainframes for performing I/O or offloading the general-purpose CPU.

**I/O control** A logical layer of the operating system responsible for controlling I/O, consisting of device drivers and interrupt handlers.

**I/O manager** In Windows, the system component responsible for I/O.

**I/O port** A hardware connector allowing connection of an I/O device.

**I/O request packet (IRP)** In Windows, a data structure to request file I/O that is sent from the I/O manager to the appropriate device driver.

**I/O subsystem** The I/O devices and the part of the kernel that manages I/O.

**I/O-bound process** A process that spends more of its time doing I/O than doing computations

**icons** Images representing objects (such as files or applications) that users can choose via the GUI.

**idempotent** Describes a function that, when applied more than once, has the same result every time.

**identifier** Generally, a numeric tag for a device or object. In networking, the unique host-id number identifying a system on a network.

**idle process** In Windows, a process that serves as the container of all idle threads.

**idle thread** In some operating systems, a special thread that runs on the CPU when no other thread is ready to run.

**immutable shared file** In a remote file system, a file that, once shared by its creator, cannot be modified.

**imperative language** Language for implementing algorithms that are state-based (e.g., C, C++, Java, and C#).

**impersonation** In Windows, the representation of a thread by a token for security purposes.

**implicit threading** A programming model that transfers the creation and management of threading from application developers to compilers and run-time libraries.

**incremental backup** In file systems, a backup that contains only some parts of a file system (the parts that have changed since the last full and incremental backups).

**indefinite blocking** A situation in which one or more processes or threads waits indefinitely within a semaphore.

**index** In file systems, an access method built on top of direct access in which a file contains an index with pointers to the contents of the file.

**index block** In indexed allocation, a block that contains pointers to the blocks containing the file's data.

**index root** In NTFS, the part of the directory containing the top level of the B+ tree.

**indexed allocation** In file-system block allocation, combining all block pointers in one or more index blocks to address limits in linked allocation and allow direct access to each block.

**indirect block** In UFS, a block containing pointers to direct blocks, which point to data blocks.

**InfiniBand (IB)** A high-speed network communications link.

**infinite blocking** A scheduling risk in which a thread that is ready to run is never put onto the

CPU due to the scheduling algorithm; it is starved for CPU time.

**information maintenance** A category of system calls.

**infrastructure as a service (IaaS)** A type of computing in which servers or storage are available over the Internet (e.g., storage available for making backup copies of production data).

**inode** In many file systems, a per-file data structure holding most of the metadata of the file. The FCB in most UNIX file systems.

**inode object** The VFS representation of an individual file.

**input/output operations per second** A measure of random access I/O performance; the number of inputs + outputs per second.

**integrity label** In Windows Vista and later versions, a mandatory access control component assigned to each securable object and subject.

**integrity levels** A mechanism, introduced in Windows Vista, that acts as a rudimentary capability system for controlling access.

**Intel 64** Intel 64 bit CPUs, part of a class of CPUs collectively known as x86-64

**interactive** Describes a type of computing that provides direct communication between the user and the system.

**intermachine interface** In distributed computing, a set of low-level functions for cross-machine interaction.

**internal fragmentation** Fragmentation that is internal to a partition.

**Internet** A worldwide system of interconnected computer networks.

**Internet key exchange (IKE)** A protocol that uses public key encryption to allow secure symmetric key exchange for IPSec.

**Internet protocol (IP)** The low-level networking protocol on which TCP and UDP are layered; responsible for routing IP datagrams through networks such as the Internet.

**Internet protocol security (IPSec)** A network protocol suite providing authentication and symmetric-key encryption of packets of network data.

**Internet Service Providers (ISPs)** Companies that provide Internet access.

**interpretation** A methodology that allows a program in computer language to be either executed in its high-level form or translated to an intermediate form rather than being compiled to native code.

**interprocess communication (IPC)** Communication between processes.

**interrupt** A hardware mechanism that enables a device to notify the CPU that it needs attention.

**interrupt address** A variable provided along with an interrupt signal to indicate the source of the interrupt.

**interrupt chaining** A mechanism by which each element in an interrupt vector points to the head of a list of interrupt handlers, which are called individually until one is found to service the interrupt request.

**Interrupt latency** The period of time from the arrival of an interrupt at the CPU to the start of the routine that services the interrupt.

**interrupt object** The Windows representation of an interrupt.

**interrupt priority level** Prioritization of interrupts to indicate handling order.

**interrupt request level (IRQL)** A prioritization method used in interrupt management.

**interrupt service routine (ISR)** An operating system routine that is called when an interrupt signal is received.

**interrupt vector** An operating-system data structure indexed by interrupt address and pointing to the interrupt handlers. A kernel memory data structure that holds the addresses of the interrupt service routines for the various devices.

**interrupt-controller hardware** Computer hardware components for interrupt management.

**interrupt-dispatch table** The Windows term for its interrupt vector.

**interrupt-handler routine** An operating system routine that is called when an interrupt signal is received.

**interrupt-request line** The hardware connection to the CPU on which interrupts are signaled.

**intruder** Someone attempting to breach security.

**intrusion prevention** The attempt to detect attempted and successful intrusions and properly respond to them.

**intrusion-prevention systems (IPS)** Systems to detect and prevent intrusions, usually in the form of self-modifying firewalls.

**inverted page table** A page-table scheme that has one entry for each real physical page frame in memory; the entry maps to a logical page (virtual address) value.

**iSCSI** The protocol used to communicate with SCSI devices; used across a network for more distant access.

## G-18 Glossary

**Itanium** Intel IA-64 CPU.

**iteration space** In Intel threading building blocks, the range of elements that will be iterated.

**Java virtual machine** In programming-environment virtualization, the process that implements the Java language and allows execution of Java code.

**job objects** In Windows, data structures for tracking collections of processes (e.g., to set CPU usage limits).

**job pool** The location where jobs are kept on disk while waiting for main memory to become available.

**job scheduling** The task of choosing which jobs to load into memory and execute.

**job** A set of commands or processes executed by a batch system.

**journaling** In a file system that features a write transaction log, logging of write activities for replay across actual file-system structures and consistency protection.

**journaling file system** A file system that features a write transaction log where all write activities are logged for replay across actual file-system structures and consistency protection.

**just-in-time (JIT)** In Java virtual machine implementations, describes a compiler that converts bytecode to native CPU instructions the first time the bytecode is interpreted to speed later execution.

**Kerberos** A network authentication protocol invented at M.I.T. that forms the basis for the Microsoft network authentication protocol.

**kernel** The operating system component running on the computer at all times after system boot.

**kernel abstractions** Components provided with the Mach microkernel to add functionality beyond the microkernel, such as tasks, threads, memory objects, and ports.

**kernel environment** In the layered macOS and iOS operating system design, the Darwin layer that includes the Mach microkernel and the BSD UNIX kernel.

**kernel extensions (kexts)** Third-party components added to the kernel (usually to support third-party devices or services).

**kernel mode** A CPU mode in which all instructions are enabled. The kernel runs in this mode. See also user mode.

**kernel threads** Threads running in kernel mode.

**kernel-mode driver framework (KMDF)** A framework in Windows to facilitate the writing of kernel-mode device drivers.

**kernel-mode thread (KT)** In Windows, the name for the state of a thread when it is running in kernel mode.

**Kernighan's Law** "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

**keys** In the context of protection, unique bit patterns held by domains corresponding with unique bit patterns (locks) held by objects. Generally, secrets used in cryptography.

**keystream** An infinite set of bits used to encrypt a plain-text stream through an XOR operation in a stream cipher.

**keystroke logger** A program that captures keystrokes entered by users.

**kilobyte (KB)** 1,024 bytes.

**Kubernetes** An orchestration tool for containers.

**labels** In mandatory access control, identifiers assigned to objects and/or subjects. The label is checked by the operating system when an operation is requested to determine if it is allowed.

**layered approach** A kernel architecture in which the operating system is separated into a number of layers (levels); typically, the bottom layer (layer 0) is the hardware, and the highest (layer N) is the user interface.

**lazy swapper** A swapping method in which only pages that are requested are brought from secondary storage into main memory.

**least frequently used (LFU)** In general, an algorithm that selects the item that has been used least frequently. In virtual memory, when access counts are available, selecting the page with the lowest count.

**least privilege** Design principle stating that every program and every privileged user of the system should operate using the least amount of privilege necessary to complete the job.

**least recently used (LRU)** In general, an algorithm that selects the item that has been used least recently. In memory management, selecting the page that has not been accessed in the longest time.

**lgroups** In Solaris, locality groups located in the kernel; each lgroup gathers together CPUs and memory, and each CPU in that group can access any memory in the group within a defined latency interval. A method for dealing with NUMA.

**library operating systems** The applications that run on unikernels, containing both the kernel and the application code.

**lightweight directory-access protocol (LDAP)** A secure distributed naming service used throughout the computer industry.

**lightweight process (LWP)** A virtual processor-like data structure allowing a user thread to map to a kernel thread.

**limit register** A CPU register that defines the size of the range. Together with the base register, it defines the logical address space.

**line discipline** In Linux, an interpreter for the information from a terminal device.

**link** In file naming, a file that has no contents but rather points to another file.

**linked allocation** A type of file-system block allocation in which each file is a linked list of allocated blocks containing the file's contents. Blocks may be scattered all over the storage device.

**linked list** A data structure in which items are linked to one another.

**linker** A system service that combines relocatable object files into a single binary executable file.

**Linux distribution** A Linux system plus administrative tools to simplify the installation, upgrading, and management of the system.

**Linux instance** A set of Pico processes running in Windows created by WSL containing an init process and a bash shell (allowing executing of Linux commands within Windows)

**Linux kernel** The operating-system kernel of a Linux system.

**Linux system** The kernel, programs, and files that comprise a complete, runnable Linux system.

**list** A data structure that presents a collection of data values as a sequence.

**little-endian** A system architecture that stores the least significant byte first in a sequence of bytes.

**Little's formula** A scheduling equation ( $n = ? \times W$ ) that is particularly useful because it is valid for any scheduling algorithm and arrival distribution.

**live migration** In virtualization, the movement of a running guest between two separate physical hosts.

**LiveCD** An operating system that can be booted and run from a CD-ROM (or more generally from any media) without being installed on a system's boot disk(s).

**LiveDVD** An operating system that can be booted and run from a DVD (or more generally from any media) without being installed on a system's boot disk(s).

**livelock** A condition in which a thread continuously attempts an action that fails.

**liveness** A set of properties that a system must satisfy to ensure that processes make progress during their execution life cycle.

**living document** A document that is modified over time to keep it up to date.

**load balancing** The movement of jobs or network packets between various components (say, computers in a network) to distribute the load or route around failures. Load balancing attempts to keep the workload evenly distributed across all processors in an SMP system.

**load sharing** The ability of a system with multiple CPU cores to schedule threads on those cores.

**loadable kernel module (LKM)** A kernel structure in which the kernel has a set of core components and can link in additional services via modules, either at boot time or during run time.

**loader** A system service that loads a binary executable file into memory, where it is eligible to run on a CPU core.

**local procedure call** In Windows, a method used for communication between two processes on the same machine.

**local replacement** In virtual-memory frame allocation, allowing a process to select a replacement frame only from the set of all frames allocated to the process.

**local replacement algorithm** A virtual-memory page replacement algorithm that avoids thrashing by not allowing a process to steal frames from other processes.

**local-area network (LAN)** A network that connects computers within a room, a building, or a campus.

**locality** The tendency of processes to reference memory in patterns rather than randomly.

**locality model** A model for page replacement based on the working-set strategy.

**locality of reference** The tendency of processes to reference memory in patterns rather than randomly.

**location independence** In distributed computing, a feature in which the name of an object does not need to be changed when the object's physical location changes.

**location transparency** In distributed computing, a feature in which the name of an object does not reveal its physical location.

**lock** A mechanism that restricts access by processes or subroutines to ensure integrity of shared data.

## G-20 Glossary

**locked** In general, fixed in place. In memory management, pages can be locked into memory to prevent them from being paged out.

**lock-free** An algorithm that provides protection from race conditions without requiring the overhead of locking.

**locking** Protecting critical sections, data structures, or objects through the use of code that coordinates access.

**lock-key scheme** In protection, a compromise between access lists and capability lists in which each object has a unique bit pattern (a lock) and each domain has a unique bit pattern (a key).

**log file** A file containing error or “logging” information; used for debugging or understanding the state or activities of the system.

**log-based transaction-oriented file system** A file system that features a write transaction log where all write activities are logged for replay across actual file-system structures and consistency protection.

**logic bomb** A remote-access tool designed to operate only when a specific set of logical conditions is met.

**logical address** Address generated by the CPU; must be translated to a physical address before it is used.

**logical address space** The set of all logical addresses generated by a program.

**logical blocks** Logical addresses used to access blocks on storage devices.

**logical cluster numbers** In Windows, the name given to secondary storage physical addresses.

**logical file system** A logical layer of the operating system responsible for file and file-system metadata management; maintains the FCBs.

**logical formatting** The creation of a file system in a volume to ready it for use.

**logical memory** Memory as viewed by the user; usually a large uniform array, not matching physical memory in virtual memory systems.

**logical records** File contents logically designated as fixed-length structured data.

**LOOK** An HDD I/O scheduling algorithm modification of SCAN that stops the head after the last request is complete (rather than at the innermost or outermost cylinder).

**loopback** Communication in which a connection is established back to the sender.

**loosely coupled** Describes a kernel design in which the kernel is composed of components that have specific and limited functions.

**lottery scheduling** A scheduling algorithm in which “lottery tickets” are given to threads and a lottery number is chosen at random to determine the next thread to get CPU time.

**low-fragmentation heap (LFH)** An optimization of the Windows default heap designed to decrease fragmentation.

**low-level formatting** The initialization of a storage medium in preparation for its use as a computer storage device.

**Lustre** A common open-source clustered file system.

**LXC** A Linux container technology.

**Mach** An operating system with microkernel structure and threading; developed at Carnegie Mellon University.

**Mach-O** The macOS format of executable files.

**magic cookie** A crude method of storing a text string at the start of a text file to indicate the type of text in the file.

**magic number** A crude method of storing a number at the start of a file to indicate the type of the data in the file.

**magnetic tape** A magnetic media storage device consisting of magnetic tape spooled on reels and passing over a read-write head. Used mostly for backups.

**main queue** Apple OS per-process block queue.

**main TLB** ARM CPU outer-level TLB; checked after the micro TLB lookup and before a miss causes a page table walk.

**mainframe** The largest class of computers (along with supercomputers), hosting hundreds of users and many and/or large jobs.

**major fault** In virtual memory, a page fault that can be resolved without having to page in data from secondary storage.

**malware** Software designed to exploit, disable, or damage computer systems.

**mandatory access control (MAC)** Access control settings enforced in the form of system policy.

**mandatory file-lock mechanism** A file-locking system in which the operating system enforces locking and file access.

**man-in-the-middle attack** An attack in which the attacker sits in the middle of the data flow of a communication, masquerading as the sender to the receiver and vice versa.

**MapReduce** A Google-created big data programming model and implementation for parallel processing across nodes in a distributed cluster. A layer on top of the Google file system (GFS), it

allows developers to carry out large-scale parallel computations easily.

**marshaling** Packaging a communication into an expected format for transmittal and reception.

**maskable** Describes an interrupt that can be delayed or blocked (such as when the kernel is in a critical section).

**masquerading** A practice in which a participant in a communication pretends to be someone else (another host or another person).

**master boot record (MBR)** Windows boot code, stored in the first sector of a boot partition.

**master file directory (MFD)** In two-level directory implementation, the index pointing to each UFD.

**master file table** The NTFS volume control block.

**master key** In the lock-key protection scheme, a key that is associated with each object and can be defined or replaced with the set-key operation to revoke or change access rights.

**matchmaker** A function that matches a caller to a service being called (e.g., a remote procedure call attempting to find a server daemon).

**mean time between failure (MTBF)** The statistical mean time that a device is expected to work correctly before failing.

**mean time to data loss** The statistical mean of the time until data is lost.

**mean time to repair** The statistical mean of the time to repair a device (e.g., to get a replacement and install it).

**mechanical storage device** A storage device based on moving mechanical parts (such as HDDs, optical disks, and magnetic tape); one form of nonvolatile storage.

**mechanism** An operation that defines how something will be done.

**medium access control (MAC) address** A unique byte number assigned to every Ethernet device allowing it to be located by packets sent across a LAN.

**megabyte (MB)**  $1,024^2$  bytes.

**memory** Volatile storage within a computer system.

**memory barriers** Computer instructions that force any changes in memory to be propagated to all other processors in the system.

**memory compression** In memory management, an alternative to paging involving compressing the contents of frames to decrease the memory used.

**memory compression process** In Windows 10, a process that maintains a working set of compressed standby pages.

**memory fences** Computer instructions that force any changes in memory to be propagated to all other processors in the system.

**memory manager (MM)** The Windows name for the component that manages memory.

**memory mapping** A file-access method in which a file is mapped into the process memory space so that standard memory access instructions read and write the contents of the file; an alternative to the use of read() and write() calls.

**memory model** Computer architecture memory guarantee, usually either strongly ordered or weakly ordered.

**memory resident** Objects, such as pages, that are in main memory and ready for threads to use or execute.

**memory stall** An event that occurs when a thread is on CPU and accesses memory content that is not in the CPU's cache. The thread's execution stalls while the memory content is fetched.

**memory transaction** A sequence of memory read-write operations that are atomic.

**memory-management unit (MMU)** The hardware component of a computer CPU or motherboard that allows it to access memory.

**memory-mapped file** A file that is loaded into physical memory via virtual memory methods, allowing access by reading and writing to the memory address occupied by the file.

**memory-mapped I/O** A device I/O method in which device-control registers are mapped into the address space of the processor.

**message** In networking, a communication, contained in one or more packets, that includes source and destination information to allow correct delivery. In message-passing communications, a packet of information with metadata about its sender and receiver.

**message digest** The calculation resulting from a hash function.

**message passing** In interprocess communication, a method of sharing data in which messages are sent and received by processes. Packets of information in predefined formats are moved between processes or between computers.

**message-authentication code (MAC)** A cryptographic checksum calculated in symmetric encryption; used to authenticate short values.

**message-passing model** A method of interprocess communication in which messages are exchanged.

**metadata** A set of attributes of an object. In file systems, e.g., all details of a file except the file's contents.

## G-22 Glossary

**metaslabs** Chunks of blocks. In ZFS, a pool of storage is split into metaslabs for easier management.

**methods** In Java, functions that act on objects and data fields.

**metropolitan-area network (MAN)** A network linking buildings within a city.

**micro TLB** ARM CPU inner-level TLBs, one for instructions and one for data.

**microkernel** An operating-system structure that removes all nonessential components from the kernel and implements them as system and user-level programs.

**Microsoft interface definition language** The Microsoft text-based interface definition language; used, e.g., to write client stub code and descriptors for RPC.

**middleware** A set of software frameworks that provide additional services to application developers.

**minicomputer** A mid-sized computer, smaller than a mainframe but larger (in resources and users) than a workstation.

**minidisks** Virtual disks used in early IBM virtual systems.

**minimum granularity** In the Completely Fair Scheduler, a configurable variable representing the minimum length of time any process is allocated to the processor.

**miniport driver** In Windows I/O, the device-specific driver.

**minor fault** In virtual memory, a page fault resolved by executing an I/O to bring in the page from secondary storage.

**mirrored volume** A volume in which two devices are mirrored.

**mirroring** In storage, a type of RAID protection in which two physical devices contain the same content. If one device fails, the content can be read from the other.

**mobile computing** A mode of computing involving small portable devices like smartphones and tablet computers.

**mode bit** A CPU status bit used to indicate the current mode: kernel (0) or user (1).

**modify bit** An MMU bit used to indicate that a frame has been modified (and therefore must have its contents saved before page replacement).

**module loader and unloader** In Linux, user-mode utilities that work with the module-management system to load modules into the kernel.

**module-management system** In Linux, the facility that allows kernel modules to be loaded into

memory and to communicate with the rest of the kernel.

**monitor** A high-level language synchronization construct that protects variables from race conditions.

**monitor call** A software-triggered interrupt allowing a process to request a kernel service.

**monoculture** A community of computer systems that are very similar to one another. This similarity makes them easier to attack and thus represents a threat to security.

**monolithic** In kernel architecture, describes a kernel without structure (such as layers or modules).

**Moore's Law** A law predicting that the number of transistors on an integrated circuit would double every eighteen months.

**most frequently used (MFU)** In general, an algorithm that selects the item that has been used most frequently. In virtual memory, when access counts are available, selecting the page with the highest count.

**mount point** The location within the file structure where a file system is attached.

**mount protocol** The protocol for mounting a file system in a remote file system.

**mount table** An in-memory data structure containing information about each mounted volume. It tracks file systems and how they are accessed.

**mounting** Making a file system available for use by logically attaching it to the root file system.

**multicore** Multiple processing cores within the same CPU chip or within a single system.

**multicore processor** Multiple processing cores within the same CPU chip.

**multicore systems** Systems that have two or more hardware processors (CPU cores) in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices.

**multifactor authentication** Authentication based on two or more sources of data, with more sources generally providing stronger authentication.

**multilevel feedback queue** A scheduling algorithm that allows a process to move between queues.

**multilevel queue** A scheduling algorithm that partitions the ready queue into several separate queues.

**multiple universal-naming-convention provider (MUP)** The component within Windows that executes remote file accesses.

**multiple user interface (MUI)** A Windows Vista feature that allows multiple user interfaces,

possibly configured for different locales, to be used concurrently.

**multiprocessor** Multiple processors within the same CPU chip or within a single system.

**multiprocessor systems** Systems that have two or more hardware processors (CPU cores) in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices.

**multiprogramming** A technique that increases CPU utilization by organizing jobs (code and data) so that the CPU always has a job to execute.

**multitasking** The concurrent performance of multiple jobs. A CPU executes multiple jobs by switching among them, but the switches occur so frequently that users can interact with the processes.

**multithreaded** A term describing a process or program with multiple threads of control, allowing multiple simultaneous execution points.

**mutex lock** A mutual exclusion lock; the simplest software tool for assuring mutual exclusion.

**mutual exclusion** A property according to which only one thread or process can be executing code at once.

**name server** In the domain-name system, a host or software that provides resolving services.

**named pipes** A connection-oriented messaging mechanism—e.g., allowing processes to communicate within a single computer system.

**named semaphore** A POSIX scheduling construct that exists in the file system and can be shared by unrelated processes.

**named shared-memory object** In Windows API, a section of a memory-mapped file accessible by name from multiple processes.

**namespace** In Linux, a process's specific view of the file system hierarchy.

**naming** In distributed computing, the mapping between logical and physical objects.

**national-language-support (NLS)** A Windows API providing support for localization (including date, time, and money formats).

**need-to-know principle** The principle that only those resources currently needed should be available to use at a given time.

**nested page tables (NPTs)** In virtualization, a method used by the virtual machine manager to maintain page-table state both for guests and for the system.

**network** In the simplest terms, a communication path between two or more systems.

**network address translation** In networking, the mapping of one header address to another by modifying the network packets (e.g., allowing a host to provide IP addresses to multiple guests while presenting only one IP address to the connected network).

**network computer** A limited computer that understands only web-based computing.

**network device interface specification (NDIS)** An internal Windows networking interface separating network adapters from transport protocols.

**network devices** I/O devices that connect to a network.

**network file system (NFS)** A common network file system used by UNIX, Linux, and other operating systems to share files across a network.

**network operating system** A type of operating system that provides features such as file sharing across a network, along with a communication scheme that allows different processes on different computers to exchange messages. It provides an environment in which users can access remote resources by remote login or data transfer between remote and local systems.

**network time protocol** A network protocol for synchronizing system clocks.

**network virtual memory** A distributed computing feature similar to virtual memory but with the backing store on a remote system.

**network-attached storage (NAS)** Storage accessed from a computer over a network.

**NFS protocol** The protocol used for remote file access, remote file system mounting, etc., by the NFS file system.

**nice value** One of a range of values from ?20 to +19, where a numerically lower value indicates a higher relative scheduling priority.

**NIS** A distributed naming service that provides username, password, hostname, and printer information to a set of computers.

**nonblocking** A type of I/O request that allows the initiating thread to continue while the I/O operation executes. In interprocess communication, a communication mode in which the sending process sends the message and resumes operation and the receiver process retrieves either a valid message or a null if no message is available. In I/O, a request that returns whatever data is currently available, even if it is less than requested.

**noncontainer objects** In Windows 10, a category of objects that cannot contain other objects.

## G-24 Glossary

**nonmaskable interrupt** An interrupt that cannot be delayed or blocked (such as an unrecoverable memory error)

**nonpreemptive** Scheduling in which, once a core has been allocated to a thread, the thread keeps the core until it releases the core either by terminating or by switching to the waiting state.

**nonpreemptive kernels** A type of kernel that does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

**nonrepudiation** Proof that an entity performed an action (frequently performed by digital signatures).

**non-uniform memory access (NUMA)** An architecture aspect of many computer systems in which the time to access memory varies based on which core the thread is running on (e.g., a core interlink is slower than accessing DIMMs directly attached to core).

**nonvolatile memory (NVM)** Persistent storage based on circuits and electric charges.

**nonvolatile storage (NVS)** Storage in which data will not be lost in a power outage or similar event.

**NOOP** The Linux NVM scheduling algorithm, first come first served but with adjacent requests merged into fewer, larger I/O requests.

**NUMA node** One or more cores (e.g., cores that share a cache) that are grouped together as a scheduling entity for affinity or other uses.

**NVMe express (NVMe)** A high-speed I/O bus for NVM storage.

**NVRAM** DRAM with battery or other backup power, rendering it nonvolatile.

**object** An instance of a class or an instance of a data structure. In Windows and generally, an instance of an object type.

**object linking and embedding (OLE)** A Microsoft technology allowing services to provide functions to components (e.g., for inserting spreadsheets into Word documents).

**object manager** In Windows, the kernel (executive) component that manipulates objects.

**object type** In Windows, a system-defined data type that has a set of attributes and methods that help define its behavior.

**off-line** Generally, a facility, system, or computing component that is unavailable. In file system implementation, operations executing while the file system is unavailable for use.

**one-time password** A password that is only valid once.

**on-line** Generally, a facility, system, or computing component that is available. In file system implementation, operations executing while the file system is available for use.

**open count** The number of processes having an open file.

**open-file table** An operating system data structure containing details of every file open within the system.

**open-source operating system** An operating system or other program available in source-code format rather than as compiled binary code.

**operating system** A program that manages a computer's hardware, provides a basis for application programs, and acts as an intermediary between the computer user and the computer hardware.

**optimal page-replacement algorithm** A theoretically optimal page replacement algorithm that has the lowest page-fault rate of all algorithms and never suffers from Belady's anomaly.

**Orange Book** U.S. Department of Defense Trusted Computer System Evaluation Criteria; a method of classifying the security of a system design.

**orphan** The child of a parent process that terminates in a system that does not require a terminating parent to cause its children to be terminated.

**OS/2** A PC operating system from the mid 1980s co-developed by IBM and Microsoft to replace MS-DOS; generally considered to be a failure.

**OSI protocol stack** A set of cooperating network protocols that form a fully functional network. The OSI model formalized some of the earlier work done in network protocols but is currently not in widespread use.

**out-of-band** In networking, a term describing data delivered in a manner independent of the main data stream (e.g., delivery of a symmetric key in a paper document).

**out-of-memory (OOM) killer** In Linux, a routine that executes when free memory is very low, terminating processes to free memory.

**over-allocating** Generally, providing access to more resources than are physically available. In virtual memory, allocating more virtual memory than there is physical memory to back it.

**overcommitment** In virtualization, providing resources to guests that exceed available physical resources.

**over-provisioning** In non-volatile memory, space set aside for data writes that is not counted in the device free space.

**owner** In file permissions, the userid that owns and controls access to a file.

**owner rights** In file permissions, the userid that owns and controls access to a file.

**page address extension (PAE)** Intel IA-32 CPU architecture hardware that allows 32-bit processors to access physical address space larger than 4GB.

**page allocator** The kernel routine responsible for allocating frames of physical memory.

**page cache** In file I/O, a cache that uses virtual memory techniques to cache file data as pages rather than file-system-oriented blocks for efficiency.

**page directory** In Intel IA-32 CPU architecture, the outermost page table.

**page directory pointer table** PAE pointer to page tables.

**page fault** A fault resulting from a reference to a non-memory-resident page of memory.

**page frame** A Windows virtual memory data structure.

**page frame number (PFN)** In Windows, the name of the indicator of the page frame address.

**page number** Part of a memory address generated by the CPU in a system using paged memory; an index into the page table.

**page offset** Part of a memory address generated by the CPU in a system using paged memory; the offset of the location within the page of the word being addressed.

**page replacement** In virtual memory, the selection of a frame of physical memory to be replaced when a new page is allocated.

**page slot** In Linux swap-space management, a part of the data structure tracking swap-space use.

**page table** In paged memory, a table containing the base address of each frame of physical memory, indexed by the logical page number.

**page-directory entry (PDE)** A Windows virtual-memory data structure.

**page-fault frequency** The frequency of page faults.

**page-fault rate** A measure of how often a page fault occurs per memory access attempt.

**pageout policy** Generally, an algorithm for deciding which memory pages to page out. In Linux, the virtual memory pageout policy uses a modified version of the second-chance algorithm.

**pager** The operating-system component that handles paging.

**page-replacement algorithm** In memory management, the algorithm that chooses which victim frame of physical memory will be replaced by a needed new frame of data.

**page** A fixed-sized block of logical memory.

**page-table base register** In paged memory, the CPU register pointing to the in-memory page table.

**page-table entry (PTE)** A Windows virtual memory data structure.

**page-table length register** A CPU register indicating the size of the page table.

**paging** A common memory management scheme that avoids external fragmentation by splitting physical memory into fixed-sized frames and logical memory into blocks of the same size called pages.

**paging file** The Windows term for backing store.

**paging mechanism** In Linux, the kernel component that carries out the transfer of pages back and forth to backing store.

**paired password** In authentication, a challenge-response set of secret keys, where only the correct response to the challenge provides authentication.

**parallel file system (PFS)** A file system that is LAN-based and treats N systems storing data and Y clients accessing the data as a single client-server instance; more complex than a client-server DFS but less complex than a cluster-based DFS. GPFS and Lustre are examples.

**parallel regions** Blocks of code that may run in parallel.

**parallel systems** Systems that have two or more hardware processors (CPU cores) in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices.

**parallelization** The process of dividing a program into separate components that run in parallel on individual cores in a computer or computers in a cluster.

**paravirtualization** A technique in which a guest operating system is modified to work in cooperation with a virtual machine manager.

**parent** In a tree data structure, a node that has one or more nodes connected below it.

**partition** Logical segregation of storage space into multiple areas; e.g., on HDDs, creating several groups of contiguous cylinders from the devices' full set of cylinders.

**partition boot sector** The NTFS boot control block.

**passphrase** A longer, generally more secure password composed of multiple words.

## G-26 Glossary

**password** A secret key, usually used to authenticate a user to a computer.

**path name** The file-system name for a file, which contains all the mount-point and directory-entry information needed to locate the file (e.g., “C:/foo/bar.txt” and “/foo/bar.txt”)

**path-name translation** The parsing of a file name into separate directory entries, or components.

**PCIe bus** A common computer I/O bus connecting the CPU to I/O devices.

**peer-to-peer (p2p)** A mode of distributed computing in which all nodes act as both clients of other nodes and servers to other nodes.

**penetration test** The scanning of a target entity to look for known vulnerabilities.

**performance tuning** The activity of improving performance by removing bottlenecks.

**periodic** A type of real-time process that repeatedly moves at fixed intervals between two modes: needing CPU time and not needing CPU time.

**permissions** An entity’s access rights to an object (e.g., a user’s access rights to a file).

**per-process open-file table** A kernel in-memory per-process data structure containing pointers to the system-wide open-file table, as well as other information, for all files the process has open.

**personal firewall** A software layer, either part of the operating system or added to a computer, limiting communication to and from a given host.

**personal identification number** A usually short and not very secure password composed of some combination of digits 0-9.

**personal-area network (PAN)** A network linking devices within several feet of each other (e.g., on a person).

**petabyte (PB)**  $1,024^5$  bytes.

**Peterson’s solution** A historically interesting algorithm for implementing critical sections.

**phishing** A class of social engineering attacks in which a legitimate-looking e-mail or website tricks a user into breaching confidentiality or enabling privilege escalation.

**PHY** The physical hardware component that connects to a network (implements layer 1 in the OSI model).

**physical address** Actual location in physical memory of code or data.

**physical address space** The set of all physical addresses generated by a program.

**physical formatting** The initialization of a storage medium in preparation for its use as a computer storage device.

**physical-to-virtual (P-to-V)** In virtualization, the conversion of a physical system’s operating system and applications to a virtual machine.

**Pico** In WSL, a special Linux-enabling process that translates Linux system calls to the LXCore and LXSS services.

**pinning** In memory management, locking pages into memory to prevent them from being paged out.

**pipe** A logical conduit allowing two processes to communicate.

**platform as a service (PaaS)** A software stack ready for application use via the Internet (e.g., a database server).

**platter** An HDD component that has a magnetic media layer for holding charges.

**plug-and-play (PnP) manager** In Windows, the manager responsible for detecting and enumerating devices when the system is booting and adding and removing devices when the system is running.

**pluggable authentication module (PAM)** A shared library that can be used by any system component to authenticate users.

**plug-in** An add-on functionality that expands the primary functionality of a process (e.g., a web browser plug-in that displays a type of content different from what the browser can natively handle).

**point-to-point tunneling protocol (PPTP)** A protocol in Windows and other systems allowing communication between remote-access server modules and client systems connected across a WAN.

**policy** A rule that defines what will be done.

**policy algorithm** In Linux, a part of the paging system that decides which pages to write out to backing store and when to write them.

**polling** An I/O loop in which an I/O thread continuously reads status information waiting for I/O to complete.

**pool** In virtual memory, a group of free pages kept available for rapid allocation (e.g., for copy-on-write). In ZFS, drives, partitions, or RAID sets that can contain one or more file systems.

**pop** The action of removing an item from a stack data structure.

**port** A communication address; a system may have one IP address for network connections but many ports, each for a separate communication. In computer I/O, a connection point for devices to attach to computers. In software development, to move code from its current platform to another platform (e.g., between operating systems or hardware systems). In the Mach OS, a mailbox for communication.

**port driver** In Windows I/O, the common driver for a class of devices.

**port number** In TCP/IP and UDP/IP networking, an address of a service on a system.

**port set** A collection of ports, as declared by a task, that can be grouped together and treated as one port for the purposes of the task.

**portable** An aspect of software that describes its ease of transfer between CPU architectures and computer systems.

**portable executable (PE)** The Windows format for executable files.

**portals** Gateways between requestors and services running on provider computers.

**position-independent code (PIC)** In Linux, binary code compiled from shared libraries that can be loaded anywhere in memory.

**positioning time** On an HDD, the time it takes the read-write head to position over the desired track.

**power manager** In Windows, the component that implements power management policies.

**power users** Users with unusually deep knowledge of a system.

**power-of-2 allocator** In the buddy system, an allocator that satisfies memory requests, in units sized as a power of 2, from a fixed-sized segment consisting of contiguous pages.

**power-on self-test (POST)** A firmware routine run at system power-on that tests the system for hardware issues, identifies and initializes many of the attached devices, and builds the description of the devices used by the advanced configuration and power interface (ACPI).

**preemptive** A form of scheduling in which processes or threads are involuntarily moved from the running state (e.g., by a timer signaling the kernel to allow the next thread to run).

**preemptive kernel** A type of kernel that allows a process to be preempted while it is running in kernel mode.

**preemptive multitasking** A model of multitasking in which threads on cores may be preempted by higher-priority threads before finishing their scheduled time quanta.

**prepaging** In virtual memory, bringing pages into memory before they are requested.

**principle of least privilege** A design principle stating that every program and every privileged user of the system should operate using the least amount of privilege necessary to complete the job.

**priority inversion** A scheduling challenge arising when a higher-priority process needs to read

or modify kernel data that are currently being accessed by a lower-priority process.

**priority number** A number indicating the position of a process in a conditional-wait queue in a monitor construct.

**priority paging** Prioritizing selection of victim frames based on some criteria, such as avoiding selection of shared library pages.

**priority replacement algorithm** A virtual memory page replacement algorithm that avoids thrashing by not allowing a process to steal frames from other processes.

**priority-inheritance protocol** A protocol for solving priority inversion in which all processes that are accessing resources needed by a higher-priority process inherit that higher priority until they are finished with the resources in question.

**priority scheduling** A scheduling algorithm in which a priority is associated with each thread and the free CPU core is allocated to the thread with the highest priority.

**private cloud** Cloud computing run by a company for that company's own use.

**private key** In an asymmetric encryption algorithm, a key that must be kept private for use in authenticating, encrypting, and decrypting.

**privilege escalation** The enabling of more privileges than an entity (process, system, person) should have.

**privileged instructions** Instructions that can execute only if the CPU is in kernel mode.

**privileged mode** A CPU mode in which all instructions are enabled. The kernel runs in this mode. See also user mode.

**proc file system (/proc)** A pseudo file system using file-system interfaces to provide access to a system's process name space.

**procedural language** A language that implements state-based algorithms (e.g., C, C++, Java, and C#).

**process** A program loaded into memory and executing.

**process control** A category of system calls.

**process control block** A per-process kernel data structure containing many pieces of information associated with the process.

**process identifier (pid)** A unique value for each process in the system that can be used as an index to access various attributes of a process within the kernel.

**process lifetime management (PLM)** A Windows power-saving feature that suspends all

## G-28 Glossary

threads within a process that has not been used for a few seconds.

**process migration** The movement of a process between computers.

**process name** A human-readable name for a process.

**process scheduler** A scheduler that selects an available process (possibly from a set of several processes) for execution on a CPU.

**process synchronization** Coordination of access to data by two or more threads or processes.

**process-contention scope (PCS)** A scheduling scheme, used in systems implementing the many-to-one and many-to-many threading models, in which competition for the CPU takes place among threads belonging to the same process.

**processor affinity** A kernel scheduling method in which a process has an affinity for (prefers) the processor on which it is currently running.

**processor groups** In Windows 7, processors grouped together for management and scheduling.

**producer** A process role in which the process produces information that is consumed by a consumer process.

**production kernels** Kernels released for production use (as opposed to development use).

**profiling** Periodically sampling the instruction pointer to determine which code is being executed; used in debugging and performance tuning.

**program counter** A CPU register indicating the main memory location of the next instruction to load and execute.

**programmable interval timer** A hardware timer provided by many CPUs.

**programmed I/O (PIO)** A method of transferring data between a CPU and a peripheral device in which data are transferred one byte at a time.

**programming-environment virtualization** Virtualization in which a virtual machine manager does not virtualize real hardware but instead creates an optimized virtual system (examples include Oracle Java and Microsoft .Net).

**project** In Solaris scheduling, a group of processes scheduled as a unit.

**proportional allocation** An allocation algorithm that assigns a resource in proportion to some aspect of the requestor. In virtual memory, the assignment of page frames in proportion to the size each process.

**proportional share** A scheduler that operates by allocating T shares among all applications, ensuring that each gets a specific portion of CPU time.

**protection** A category of system calls. Any mechanism for controlling the access of processes or users to the resources defined by a computer system.

**protection domain** In protection, a set of resources that a process may access. In virtualization, a virtual machine manager creates a protection domain for each guest to inform the CPU of which physical memory pages belong to that guest.

**protection mask** In Linux and UNIX, a set of bits assigned to an object specifying which access modes (read, write, execute) are to be granted to processes with owner, group, or world access writes to the object.

**protection rings** A model of privilege separation consisting of a series of rings, with each successive ring representing greater execution privileges.

**pseudo device driver** In virtualization, a guest device driver that does not directly control system hardware but rather works with the virtual machine manager to access the device.

**PTE table** A Windows virtual-memory data structure.

**Pthreads** The POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization (a specification for thread behavior, not an implementation).

**public cloud** Cloud computing available via the Internet to anyone willing to pay for the services offered.

**public domain** The total absence of copyright protection. Software in the public domain can be used as desired by anyone, with no limits.

**public key** In asymmetric encryption algorithm, a key that can be distributed for encrypting and decrypting.

**public key encryption** A cipher algorithm in which different keys are used for encryption and decryption.

**pull migration** Migration that occurs when an idle processor pulls a waiting thread from a busy processor.

**pure demand paging** A demand paging scheme wherein no page is brought into memory until it is referenced.

**push** The action of placing a value on a stack data structure.

**push migration** Migration in which a task periodically checks the load on each processor and, if it finds an imbalance, evenly distributes the load by moving (or pushing) threads from overloaded to idle or less busy processors.

**Quest-V** An example of a partitioning hypervisor.

**queue** A sequentially ordered data structure that uses the first-in, first-out (FIFO) principle; items are removed from a queue in the order in which they were inserted.

**queueing-network analysis** An area of computing study in which algorithms are analyzed for various characteristics and effectiveness.

**race condition** A situation in which two threads are concurrently trying to change the value of a variable.

**RAID levels** The various types of RAID protection.

**RAM drives** Sections of a system's DRAM presented to the rest of the system as if they were secondary storage devices.

**random-access memory (RAM)** Rewritable memory, also called main memory. Most programs run from RAM, which is managed by the kernel.

**ransomware** A class of malware that disables computer access (frequently by encrypting files or the entire system) until a ransom is paid.

**rate** Generally, a measure of speed or frequency. A periodic real-time process has a scheduling rate of  $1/p$ , where  $p$  is the length of its running period.

**rate-monotonic** A scheduling algorithm that schedules periodic tasks using a static priority policy with preemption.

**raw partition** A partition within a storage device not containing a file system.

**raw disk** Direct access to a secondary storage device as an array of blocks with no file system.

**raw I/O** Direct access to a secondary storage device as an array of blocks with no file system.

**read pointer** The location in a file from which the next read will occur.

**read-ahead** Sequential I/O performance optimization that reads and caches several subsequent pages when a read of one page is requested.

**readers-writers problem** A synchronization problem in which one or more processes or threads write data while others only read data.

**reader-writer lock** A lock appropriate for access to an item by two types of accessors, read-only and read-write.

**read-modify-write cycle** The situation in which a write of data smaller than a block requires the entire block to be read, modified, and written back.

**read-only memory (ROM)** A storage device whose contents are not modifiable.

**read-write (RW)** Access that allows reading and writing.

**ready queue** The set of processes ready and waiting to execute.

**real-time** A term describing an execution environment in which tasks are guaranteed to complete within an agreed-to time.

**real-time class** A scheduling class that segregates real-time threads from other threads to schedule them separately and provide them with their needed priority.

**real-time operating systems (RTOS)** Systems used when rigid time requirements have been placed on the operation of a processor or the flow of data; often used as control devices in dedicated applications.

**reapers** In memory management, routines that scan memory, freeing frames to maintain a minimum level of available free memory.

**recovery mode** A system boot state providing limited services and designed to enable the system admin to repair system problems and debug system startup.

**Red Hat** A popular Linux distribution.

**red-black tree** A tree containing  $n$  items and having at most  $\lg n$  levels, thus ensuring worst-case performance of  $O(\lg n)$ .

**redirector** In Windows, a client-side object that forwards I/O requests to a remote system.

**redundant arrays of independent disks (RAID)** A disk organization technique in which two or more storage devices work together, usually with protection from device failure.

**reentrant code** Code that supports multiple concurrent threads (and can thus be shared).

**reference bit** An MMU bit indicating that a page has been referenced.

**reference string** A trace of accesses to a resource. In virtual memory, a list of pages accessed over a period of time.

**referenced pointer** In Windows, a means by which kernel-mode code can access objects; must be obtained by calling a special API.

**regions** In ARM v8 CPUs, contiguous areas of memory with separate privilege and access rules.

**registry** A file, set of files, or service used to store and retrieve configuration information. In Windows, the manager of hives of data.

**regressive round-robin** A variation on round-robin scheduling in which a thread that uses its entire CPU scheduling quantum is given a longer quantum and higher priority.

**relative access** A file-access method in which contents are read in random order, or at least not sequentially.

## G-30 Glossary

**relative block number** An index relative to the beginning of a file. The first relative block of the file is block 0, the next is block 1, and so on through the end of the file.

**relative path name** A path name starting at a relative location (such as the current directory).

**relocatable code** Code with bindings to memory addresses that are changed at loading time to reflect where the code is located in main memory.

**relocatable object file** The output of a compiler in which the contents can be loaded into any location in physical memory.

**relocation** An activity associated with linking and loading that assigns final addresses to program parts and adjusts code and data in the program to match those addresses.

**relocation register** A CPU register whose value is added to every logical address to create a physical address (for primitive memory management).

**remainder section** Whatever code remains to be processed after the critical and exit sections.

**remote access tool (RAT)** A back-door daemon left behind after a successful attack to allow continued access by the attacker.

**remote desktop** The representation of a desktop session to another system across a network, for remote access to the computer's GUI.

**remote desktop protocol (RDP)** A network protocol to allow remote access to a computer's display contents and keyboard and mouse input devices.

**remote file transfer** A function of a network operating system providing a means to transfer files between network-attached computers.

**remote procedure calls (RPCs)** Procedure calls sent across a network to execute on another computer; commonly used in client-server computing.

**remote-service mechanism** A facility, implemented by a feature such as RPC, in which clients ask a remote system to perform a function for them.

**renderer** A process that contains logic for rendering contents (such as web pages) onto a display.

**rendezvous** In interprocess communication, when blocking mode is used, the meeting point at which a send is picked up by a receive.

**replay attack** The malicious or fraudulent repetition of a valid transmission.

**replication** In file systems, the duplication and synchronization of a set of data over a network to another system. In storage, the automatic duplication of writes between separate sites.

**request edge** In a system resource-allocation graph, an edge (arrow) indicating a resource request.

**request manager** In Linux, the kernel component that manages the reading and writing of buffer contents to and from a block-device driver.

**resolve** Generally, to translate from a symbolic representation to a numeric one. In networking, to translate from a host name to a host-id. With files, to follow a link and find the target file.

**resource allocator** An operating system or application that determines how resources are to be used.

**resource manager** The role of an operating system in managing the computer's resources.

**resource sharing** The ability for multiple users, computers, etc., to access computing resources.

**resource utilization** The amount of a given resource (hardware or software) that is being used.

**response time** The amount of time it takes the system to respond to user action.

**restore** In file systems, the act of repairing or recovering files or a file system from a backup.

**resume** In virtualization, the continuation of execution after a guest's suspension.

**reverse engineering** The procedure of converting a compiled binary file into a human-readable format.

**rich text format** A file format developed by Microsoft that includes formatting details but can be used by various applications and operating systems, enabling files to be transferred between programs and systems.

**risk assessment** A systemic security analysis that attempts to value the assets of the entity in question and determine the odds that a security incident will affect the entity.

**roaming profile** In Windows, a collection of user preferences and settings that are kept on a server and allow a user's environment to follow that user from computer to computer.

**role-based access control (RBAC)** A method of access control in which roles rather than users have access rights; applies the principle of least privilege to the protection of operating systems.

**role** In RBAC, a named set of privileges that can be available to a user.

**root partition** The storage partition that contains the kernel and the root file system; the one mounted at boot.

**rotational latency** On an HDD, the time it takes the read-write head, once over the desired cylinder, to access the desired track.

**round-robin (RR)** A scheduling algorithm similar to FCFS scheduling, but with preemption added to enable the system to switch between threads; designed especially for time-sharing systems.

**router** A device or software that connects networks to each other (e.g., a home network to the Internet).

**RSA** The most widely used public key cipher.

**run queue** The queue holding the threads that are ready to run on a CPU.

**running** The state of the operating system after boot when all kernel initialization has completed and system services have started. In general, the system state after booting and before crashing or being shut down.

**run-time environment (RTE)** The full suite of software needed to execute applications written in a given programming language, including its compilers, libraries, and loaders.

**safe computing** Human behavior aimed at avoiding viruses and other security problems (e.g., by avoiding downloading pirated software).

**safe sequence** “In deadlock avoidance, a sequence of processes  $\langle P_1, P_2, \dots, P_n \rangle$  in which, for each  $P_i$ , the resource requests that  $P_i$  can make can be satisfied by the currently available resources plus the resources held by all  $P_j$ , with  $j < i$ .”

**safe state** In deadlock avoidance, a state in which a system can allocate resources to each process in some order and still avoid deadlock.

**sandbox** A contained environment (e.g., a virtual machine).

**sandboxing** Restricting what an object can do by placing it in a contained environment (e.g., running a process on a virtual machine).

**SAS** A common type of I/O bus.

**scalability** Generally, the ability of a facility or service to increase capacity as needed by the users (e.g., to add more cores when the load increases).

**SCAN algorithm** An HDD I/O scheduling algorithm in which the disk head moves from one end of the disk to the other performing I/O as the head passes the desired cylinders; the head then reverses direction and repeats.

**scatter-gather** An I/O method in which multiple sources or destinations of I/O are specified in one command structure.

**scheduler** The part of the operating system that determines the next job to be done (e.g., the next process to be executed).

**scheduler activation** A threading method in which the kernel provides an application with a set of LWPs, and the application can schedule user threads onto an available virtual processor and receive upcalls from the kernel to be informed of certain events.

**scheduling classes** In Linux, classes on which scheduling is based; each class is assigned a specific priority.

**scheduling domain** A set of CPU cores that can be balanced against one another.

**scope** The time between when a lock is acquired and when it is released.

**script kiddie** An attacker who did not design the attack but instead is using an attack designed by a more sophisticated attacker.

**search path** In some operating systems, the sequence of directories searched for an executable file when a command is executed.

**second extended file system (ext2)** In Linux, an outdated version of the extended file system

**secondary storage** A storage system capable of holding large amounts of data permanently; most commonly, HDDs and NVM devices.

**second-chance page-replacement algorithm** A FIFO page replacement algorithm in which, if the reference bit is set, the bit is cleared and the page is not replaced.

**second-level interrupt handler** In some operating systems, the interrupt handler that actually handles interrupts; reception and queueing of interrupts are handled at another level (by the first-level handler).

**section object** The Windows data structure that is used to implement shared memory.

**sector forwarding** The replacement of an unusable HDD sector with another sector at some other location on the device.

**sector slipping** The renaming of sectors to avoid using a bad sector.

**sector sparing** The replacement of an unusable HDD sector with another sector at some other location on the device.

**sector** On an HDD platter, a fixed-size section of a track.

**secure** The state of a system whose resources are used and accessed as intended under all circumstances.

**secure by default** Describes a system or computer whose initial configuration decreases its attack surface.

## G-32 Glossary

**secure monitor call (SMC)** An ARM processor special instruction that can be used by the kernel to request services from the TrustZone.

**secure system process** In Windows, the process representing the fact that the secure kernel is loaded.

**security** The defense of a system from external and internal attacks. Such attacks include viruses and worms, denial-of-service attacks, identity theft, and theft of service.

**security access token** In Windows 10, a token created when a user logs in that contains the user's security ID, the security IDs of the groups the user belongs to, and a list of special privileges the user has.

**security context** In Windows 10, a characteristic, based on a user's access token, that enables a program run by the user to access what the user is allowed to access.

**security descriptor** In Windows 10, a feature that describes the security attributes of an object.

**security domain** The separation of systems and devices into classes, with each class having similar security needs.

**security ID (SID)** In Windows, a value used to uniquely identify a user or group for security purposes.

**security policy** A document describing the set of things being secured, how they are to be secured, and how users are to behave in matters relating to security.

**security reference monitor (SRM)** A Windows component that checks the effective security token whenever a thread opens a handle to a protected data structure.

**security through obscurity** A security layer in which information is kept private or obscured in the hope that it won't be discovered and used by attackers; an ineffective security method.

**security token** In Windows, a token associated with each process containing the SIDs of the user and the user's groups, the user's privileges, the integrity level of the process, the attributes and claims associated with the user, and any relevant capabilities.

**seek** The operation of changing the current file-position pointer.

**seek time** On an HDD, the time it takes the read-write head to position over the desired cylinder.

**semaphore** An integer variable that, apart from initialization, is accessed only through two standard atomic operations: `wait()` and `signal()`.

**semiconductor memory** The various types of memory constructed from semiconductors.

**sense key** In the SCSI protocol, information in the status register indicating an error.

**separation hypervisor** An experimental system that uses virtualization to partition separate system components into a chip-level distributed computing system.

**sequence number** In networking, a counter assigned to packets to order their assembly after delivery.

**sequential access** A file-access method in which contents are read in order, from beginning to end.

**serial-attached SCSI (SAS)** A common type of I/O bus.

**server** In general, any computer, no matter the size, that provides resources to other computers.

**server subject** In Windows 10 security, a process implemented as a protected server that uses the security context of the client when acting on the client's behalf.

**server system** A system providing services to other computers (e.g., a web server).

**server-message-block (SMB)** The Windows protocol for sending I/O requests over a network; a version was published as the common internet file system (CIFS).

**service** A software entity running on one or more machines and providing a particular type of function to calling clients. In Android, an application component with no user interface; it runs in the background while executing long-running operations or performing work for remote processes.

**service control manager (SCM)** In Windows 7, the component that manages services associated with plug-and-play devices.

**service-trigger** A mechanism in Windows 7 that allows plug-and-play device insertion to launch a service.

**session** In networking, a complete round of communication, frequently beginning with a login and ending with a logoff to terminate communications.

**session hijacking** The interception of a communication.

**session key** The TLS symmetric key, used for a web communication session, exchanged via asymmetric cryptography.

**SHA-1** An algorithm for creating a hash (a small, fixed-size block of data calculated from a larger data set, used to determine if a message has been changed).

**shared libraries** Libraries that can be loaded into memory once and used by many processes; used in systems that support dynamic linking.

**shared lock** A file lock similar to a reader lock in that several processes can obtain the lock concurrently.

**shared memory** In interprocess communication, a section of memory shared by multiple processes and used for message passing.

**shared system interconnect** A bus connecting CPUs to memory in such a way that all CPUs can access all system memory; the basis for NUMA systems.

**shared-memory model** An interprocess communication method in which multiple processes share memory and use that memory for message passing.

**shares** A basis for making scheduling decisions. The fair-share scheduling class uses CPU shares instead of priorities to allocate CPU time.

**shell** One of the command interpreters on a system with multiple command interpreters to choose from.

**shell script** A file containing a set series of commands (similar to a batch file) that are specific to the shell being used.

**shortest-job-first (SJF)** A scheduling algorithm that associates with each thread the length of the thread's next CPU burst and schedules the shortest first.

**shortest-remaining-time-first (SJRF)** A scheduling algorithm that gives priority to the thread with the shortest remaining time until completion.

**shortest-seek-time-first (SSTF) algorithm** An HDD I/O scheduling algorithm that sorts requests by the amount of seek time required to accomplish the request; the shortest time has the highest priority.

**shoulder surfing** Attempting to learn a password or other secret information by watching the target user at the keyboard.

**siblings** In a tree data structure, child nodes of the same parent.

**Siemens Jailhouse** An example of a partitioning hypervisor.

**signal** In UNIX and other operating systems, a means used to notify a process that an event has occurred.

**signature** In intrusion detection, a pattern of behavior associated with an attack.

**simple subject** In Windows 10 security, a subject that manages a user-initiated program's permissions.

**simultaneous multithreading (SMT)** The situation in which, in a CPU with multiple cores, each core supports multiple hardware threads.

**single indirect block** In UFS, a block containing pointers to direct blocks, which point to data blocks.

**single instruction multiple data (SIMD)** A form of parallelism in which multiple compute elements perform the same single instruction operating on multiple data points.

**single step** A CPU mode in which a trap is executed by the CPU after every instruction (to allow examination of the system state after every instruction); useful in debugging.

**single-threaded** A process or program that has only one thread of control (and so executes on only one core at a time).

**single-user mode** A system boot state providing limited services and designed to enable the system admin to repair system problems and debug system startup.

**Siri** The Apple voice-recognition system.

**sketch** An Arduino program.

**slab** A section of memory made up of one or more contiguous pages; used in slab allocation.

**slab allocation** A memory allocation method in which a slab of memory is allocated and split into chunks that hold objects of a given size. As the objects are freed, the chunks can coalesce into larger chunks, eliminating fragmentation.

**Slackware** An early but still widely used Linux distribution.

**slim reader-write lock (SRW)** A type of lock in modern Windows OS that favors neither readers nor writers.

**small computer-systems interface (SCSI)** One type of interface between a system and its storage (SCSI). See also ATA and SATA.

**snapshot** In file systems, a read-only view of a file system at a particular point in time; later changes do not affect the snapshot view.

**sniff** In network communication, to capture information by recording data as it is transmitted.

**sniffing** An attack in which the attacker monitors network traffic to obtain useful information.

**social engineering** A practice in which an attacker tricks someone into performing some task for the attacker (such as sending the attacker confidential information).

**socket** An endpoint for communication. An interface for network I/O.

## G-34    Glossary

**soft affinity** An operating system's policy of attempting to keep a process running on the same processor but not guaranteeing that it will do so.

**soft error** An error that is recoverable by retrying the operation.

**soft real-time systems** Systems that provide no guarantee as to when a critical real-time thread will be scheduled; they guarantee only that the thread will be given preference over noncritical threads.

**Software as a Service (SaaS)** A type of computing in which one or more applications (such as word processors or spreadsheets) are available as a service via the Internet.

**software engineering** A field of study and a career involving writing software (i.e., programming.)

**software interrupt** A software-generated interrupt; also called a trap. The interrupt can be caused either by an error (e.g., division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed.

**software objects** The software components that make up a computer or device (files, programs, semaphores, etc.).

**software transactional memory (STM)** Transactional memory implemented exclusively in software; no special hardware is needed.

**Solaris** A UNIX derivative that is the main operating system of Sun Microsystems (now owned by Oracle Corporation). There is an active open source version called Illumos.

**Solaris ZFS** An advanced file system, first included as part of Solaris.

**solid-state disk** A disk-drive-like storage device that uses flash-memory-based nonvolatile memory.

**source file** A file containing the source code of a program.

**space sharing** A feature of APFS in which storage is treated as a pool and space is shared among the file systems created in that pool (much like ZFS).

**SPARC** A proprietary RISC CPU created by Sun Microsystems and now owned by Oracle Corporation. There is an active open source version called OpenSPARC.

**sparse** In memory management, a term describing a page table that has noncontiguous, scattered entries. A sparse address space has many holes.

**spinlock** A locking mechanism that continuously uses the CPU while waiting for access to the lock.

**split-screen** Running multiple foreground processes (e.g., on an iPad) but splitting the screen among the processes.

**spoof** The imitation of a legitimate identifier (such as an IP address) by an illegitimate user or system.

**spool** A buffer that holds output for a device (such as a printer) that cannot accept interleaved data streams.

**springboard** The iOS touch-screen interface.

**spyware** A Trojan horse variation in which the installed malware gathers information about a person or organization.

**stack** A sequentially ordered data structure that uses the last-in, first-out (LIFO) principle for adding and removing items; the last item placed onto a stack is the first item removed.

**stack algorithm** A class of page-replacement algorithms that do not suffer from Belady's anomaly.

**stack inspection** In Java, a protection procedure in which a calling sequence is checked to ensure that some caller in the sequence has been granted access to the resource called.

**stack section** The section of process memory that contains the stack; it contains activation records and other temporary data.

**stall** A CPU state occurring when the CPU is waiting for data from main memory and must delay execution.

**starvation** The situation in which a process or thread waits indefinitely within a semaphore. Also, a scheduling risk in which a thread that is ready to run is never put onto the CPU due to the scheduling algorithm; it is starved for CPU time.

**state** The condition of a process, including its current activity as well as its associated memory and disk contents.

**state information** In remote file systems, the set of information pertaining to connections and ongoing file operations (e.g., which files are open).

**state restore** Copying a process's context from its saved location to the CPU registers in preparation for continuing the process's execution.

**state save** Copying a process's context to save its state in order to pause its execution in preparation for putting another process on the CPU.

**stateless** In remote file systems, a protocol in which state need not be maintained for proper operation.

**static linking** Linking in which system libraries are treated like other object modules and combined by the loader into a binary program image.

**status register** A device I/O register in which status is indicated.

**storage-area network (SAN)** A local-area storage network allowing multiple computers to connect to one or more storage devices.

**stream cipher** A cipher that encrypts or decrypts a stream of bits or bytes (rather than a block).

**stream head** The interface between STREAMS and user processes.

**stream modules** In STREAMS, modules of functionality loadable into a STREAM.

**STREAMS** A UNIX I/O feature allowing the dynamic assembly of pipelines of driver code.

**stub** A small, temporary place-holder function replaced by the full function once its expected behavior is known.

**subject** In Windows 10 security, an entity used to track and manage user permissions.

**subsystem** A subset of an operating system responsible for a specific function (e.g., memory management).

**SunOS** The predecessor of Solaris by Sun Microsystems Inc.

**superblock** The UFS volume control block.

**superblock object** The VFS representation of the entire file system.

**supervisor mode** A CPU mode in which all instructions are enabled. The kernel runs in this mode. See also user mode.

**SuSE** A popular Linux distribution.

**suspend** In virtualization, to freeze a guest operating system and its applications to pause execution.

**swap map** In Linux swap-space management, a part of the data structure tracking swap-space use.

**swap space** Secondary storage backing-store space used to store pages that are paged out of memory.

**swapped** Moved between main memory and a backing store. A process may be swapped out to free main memory temporarily and then swapped back in to continue execution.

**swapping** Moving a process between main memory and a backing store. A process may be swapped out to free main memory temporarily and then swapped back in to continue execution.

**swap-space management** The low-level operating-system task of managing space on secondary storage for use in swapping and paging.

**symmetric clustering** A situation in which two or more hosts are running applications and are monitoring each other.

**symmetric encryption algorithm** A cryptography algorithm in which the same keys are used to encrypt and decrypt the message or data.

**symmetric multiprocessing (SMP)** Multiprocessing in which each processor performs all tasks, including operating-system tasks and user processes. Also, a multiprocessor scheduling method in which each processor is self-scheduling and may run kernel threads or user-level threads.

**synchronous** In interprocess communication, a mode of communication in which the sending process is blocked until the message is received by the receiving process or by a mailbox and the receiver blocks until a message is available. In I/O, a request that does not return until the I/O completes.

**synchronous threading** Threading in which a parent thread creating one or more child threads waits for them to terminate before it resumes.

**synchronous writes** Writes that are stored in the order in which they were issued, are not buffered, and have requesting threads wait for the writes to complete before continuing.

**system administrators** Computer users that configure, monitor, and manage systems.

**system build** Creation of an operating-system build and configuration for a specific computer site.

**system call** Software-triggered interrupt allowing a process to request a kernel service.

**system call** The primary interface between processes and the operating system, providing a means to invoke services made available by the operating system.

**system-call filtering** An operating-system facility to limit which system calls can be executed by a process.

**system daemon** A service that is provided outside the kernel by system programs that are loaded into memory at boot time and run continuously.

**system disk** A storage device that has a boot partition and can store an operating system and other information for booting the computer.

**system integrity protection (SIP)** A feature of macOS 10.11 and later versions that uses extended file attributes to mark system files as restricted so that even the root user cannot tamper with them.

**system mode** A CPU mode in which all instructions are enabled. The kernel runs in this mode. See also user mode.

**system process** A service that is provided outside the kernel by system programs that are loaded into memory at boot time and run continuously. In

## G-36 Glossary

**Windows**, a process that serves as the container of all internal kernel worker threads and other system threads created by drivers for polling, house-keeping, and other background work.

**system program** A program associated with the operating system but not necessarily part of the kernel.

**system resource-allocation graph** A directed graph for precise description of deadlocks.

**system restore point** In Windows, a copy of the system hives taken before any significant change is made to system configuration.

**system service** A collection of applications included with or added to an operating system to provide services beyond those provided by the kernel.

**system utility** A collection of applications included with or added to an operating system to provide services beyond what are provided by the kernel.

**system-call firewall** A firewall within a computer that limits the system calls a process can trigger.

**system-call interface** An interface that serves as the link to system calls made available by the operating system and that is called by processes to invoke system calls.

**system-contention scope (SCS)** A thread-scheduling method in which kernel-level threads are scheduled onto a CPU regardless of which process they are associated with (and thus contend with all other threads on the system for CPU time).

**system-development time** The time during which an operating system is developed, before it is made available in final “release” form.

**system-wide open-file table** A kernel in-memory data structure containing a copy of the FCB of each open file, as well as other information.

**target latency** In the Completely Fair Scheduler, a configurable variable which is the interval of time during which every runnable task should run at least once.

**targeted latency** An interval of time during which every runnable thread should run at least once.

**task control block** A per-process kernel data structure containing many pieces of information associated with the process.

**task parallelism** A computing method that distributes tasks (threads) across multiple computing cores, with each task performing a unique operation.

**task** A process, a thread activity, or, generally, a unit of computation on a computer.

**templating** In virtualization, using one standard virtual-machine image as a source for multiple virtual machines.

**terabyte (TB)**  $1,024^4$  bytes.

**terminal concentrator** A type of front-end processor for terminals.

**tertiary storage** A type of storage that is slower and cheaper than main memory or secondary storage; frequently magnetic tape or optical disk.

**text file** A type of file containing text (alphanumeric characters).

**text section** The executable code of a program or process.

**thin client** A limited computer (terminal) used for web-based computing.

**third extended file system (ext3)** In Linux, a current version of the extended file system; the successor to ext2.

**thrashing** Paging memory at a high rate. A system thrashes when there is insufficient physical memory to meet virtual memory demand.

**thread** A process control structure that is an execution location. A process with a single thread executes only one task at a time, while a multi-threaded process can execute a task per thread.

**thread cancellation** Termination of a thread before it has completed.

**thread dump** In Java, a snapshot of the state of all threads in an application; a useful debugging tool for deadlocks.

**thread library** A programming library that provides programmers with an API for creating and managing threads.

**thread pool** A number of threads created at process startup and placed in a pool, where they sit and wait for work.

**thread-environment block (TEB)** In Win32, a user-mode threads data structure that contains numerous per-thread fields.

**thread-local storage (TLS)** Data available only to a given thread.

**threat** The potential for a security violation.

**throughput** Generally, the amount of work done over time. In scheduling, the number of threads completed per unit time.

**tightly coupled systems** Systems with two or more processors in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices.

**time quantum** A small unit of time used by scheduling algorithms as a basis for determining when to preempt a thread from the CPU to allow another to run.

**time sharing** A practice in which the CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with the processes.

**time slice** A small unit of time used by scheduling algorithms as a basis for determining when to preempt a thread from the CPU to allow another to run.

**timer** A hardware component that can be set to interrupt the computer after a specified period.

**timestamp counter (TSC)** In Windows Vista, a counter that tracks execution time.

**TLB miss** A translation look-aside buffer lookup that fails to provide the address translation because it is not in the TLB.

**TLB reach** The amount of memory addressable by the translation look-aside buffer.

**TLB walk** The steps involved in walking through page-table structures to locate the needed translation and then copying that result into the TLB.

**touch screen** A touch-sensitive screen used as a computer input device.

**touch-screen interface** A user interface in which touching a screen allows the user to interact with the computer.

**trace tapes** A tool used in the evaluation of scheduling algorithms. Thread details are captured on real systems, and various algorithms are analyzed to determine their effectiveness.

**track** On an HDD platter, the medium that is under the read-write head during a rotation of the platter.

**transaction** Generally, the execution of a set of steps that make up one activity. In log-based transaction-oriented file systems, a set of operations completed as part of a request (e.g., “write this block to that file”).

**transactional memory** A type of memory supporting memory transactions.

**transfer rate** The rate at which data flows.

**translation granules** Features of ARM v8 CPUs that define page sizes and regions.

**translation look-aside buffer (TLB)** A small, fast-lookup hardware cache used in paged memory address translation to provide fast access to a subset of memory addresses.

**translation table base register** ARM v8 CPU register pointing to the level 0 (outer) page table for the current thread.

**transmission control protocol/Internet protocol (TCP/IP)** The most common network protocol; it provides the fundamental architecture of the Internet.

**transparent** In distributed computing, a term describing the automatic sharing of resources so that users do not know if a resource is local or remote.

**transport driver interface (TDI)** In Windows networking, an interface that supports connect-based and connectionless transports on top of the transport layer.

**transport layer security (TLS)** A cryptographic protocol that enables two computers to communicate securely; the standard protocol by which web browsers communicate to web servers.

**trap** A software interrupt. The interrupt can be caused either by an error (e.g., division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed.

**trap door** A back-door daemon left behind after a successful attack to allow continued access by the attacker.

**trap-and-emulate** In virtualization, a method used to implement virtualization on systems lacking hardware support (such as CPU instructions) for virtualization; any action that would cause the guest to call the operating system is intercepted, and the result is emulated.

**tree** A data structure that can be used to represent data hierarchically; data values in a tree structure are linked through parent-child relationships.

**triple DES** A modification of DES that uses the same algorithm three times and uses two or three keys to make the encryption more difficult to break.

**triple indirect block** In UFS, a block containing pointers to double indirect blocks, which point to single indirect blocks, which point to data blocks.

**Trojan horse** A program that acts in a clandestine or malicious manner rather than simply performing its stated function.

**TrustZone (TZ)** ARM processor implementation of the most secure protection ring.

**tunnel** In computer communication, a container of communications within another type of communication (e.g., a VPN that allows web traffic).

**turnstile** A Solaris scheduling feature using a queue structure containing threads blocked on a lock.

**two-factor authentication** Authentication based on two separate sources of data (e.g., a brain providing a password and a finger providing a fingerprint).

## G-38 Glossary

**type 0 hypervisor** A hardware-based virtualization solution that provides support for virtual machine creation and management via firmware (e.g., IBM LPARs and Oracle LDOMs).

**type 1 hypervisor** Operating-system-like software built to provide virtualization (e.g., VMware ESX, Joyent SmartOS and Citrix Xenserver).

**type 2 hypervisor** An application that runs on standard operating systems but provides virtual machine management features to guest operating systems (e.g., VMware workstation and fusion, and Oracle Virtualbox)

**type safety** In Java, a feature that ensures that classes cannot treat integers as pointers, write past the end of an array, or otherwise access memory in arbitrary ways.

**unbounded buffer** A buffer with no practical limit on its memory size.

**uncontented** A term describing a lock that is available when a thread attempts to acquire it.

**unified buffer cache** In file I/O, a cache used for both memory-mapped I/O and direct file I/O.

**unified extensible firmware interface (UEFI)** The modern replacement for BIOS containing a complete boot manager.

**unified virtual memory** In file I/O, the use of page caching for all types of I/O (explicit file system I/O and page fault I/O).

**uniform memory access (UMA)** Access to all main memory by all processors, without performance differences based on CPU or memory location.

**uniform naming convention (UNC)** A name format that includes the system and its resources (e.g.m \\server\_name\share\_name\x\y\z).

**unikernels** Specialized machine images that contain both an operating system and applications for efficient execution and increased security.

**universal serial bus (USB)** A type of I/O bus.

**universal Windows platform (UWP)** Windows 10 architecture that provides a common app platform for all devices that run it, including mobile devices.

**UNIX file system (UFS)** An early UNIX file systems; uses inodes for FCB.

**UnixBSD** A UNIX derivative based on work done at the University of California at Berkeley (UCB).

**unnamed semaphore** A POSIX scheduling construct that can only be used by threads in the same process.

**unstructured data** Data that are not in a fixed format (like a database record) but rather are free-form (like a twitter.com tweet).

**upcall** A threading method in which the kernel sends a signal to a process thread to communicate an event.

**upcall handler** A function in a process that handles upcalls.

**USB drive** Nonvolatile memory in the form of a device that plugs into a USB port.

**user** The human using a computer, or the identification of the human to the computer.

**user account** In Windows 10, an account belonging to a user (rather than a system account used by the computer).

**user authentication** The identification of a user of a computer.

**user datagram protocol (UDP)** A communications protocol layered on IP that is connectionless, is low latency, and does not guarantee delivery.

**user experience layer** In the layered macOS and iOS operating system design, the layer that defines the software interface that allows users to interact with computing devices.

**user file directory (UFD)** In two-level directory implementation, a per-user directory of files.

**user identifier (user ID) (UID)** A unique numerical user identifier.

**user interface (UI)** A method by which a user interacts with a computer.

**user mode** A CPU mode for executing user processes in which some instructions are limited or not allowed. See also kernel mode.

**user programs** User-level programs, as opposed to system programs.

**user rights** Permissions granted to users.

**user thread** A thread running in user mode.

**user-defined signal handler** The signal handler created by a process to provide non-default signal handling.

**user-initiated** In the Grand Central Dispatch Apple OS scheduler, the scheduling class representing tasks that interact with the user but need longer processing times than user-interactive tasks.

**user-interactive** In the Grand Central Dispatch Apple OS scheduler, the scheduling class representing tasks that interact with the user.

**user-mode driver framework (UMDF)** A framework in Windows to facilitate the writing of user-mode device drivers.

**user-mode scheduling (UMS)** A Microsoft Windows 7 feature that allows applications to create and manage threads independently of the kernel. This feature supports task-based parallelism by

decomposing processes into tasks, which are then scheduled on available CPUs; it is used on AMD64 systems.

**user-mode thread (UT)** In Windows, the state of a thread when it is running in user mode.

**utility** In the Grand Central Dispatch Apple OS scheduler, the scheduling class representing tasks that require a longer time to complete but do not demand immediate results.

**utility storage** An inServ feature in which storage space can be increased as needed.

**valid-invalid** A page-table bit indicating whether a page-table entry points to a page within the logical address space of that process.

**variable-partition** A simple memory-allocation scheme in which each partition of memory contains exactly one process.

**vectored I/O** An I/O method in which multiple sources or destinations of I/O are specified in one command structure.

**version control system** Software that manages software distributions by allowing contributors to “push” changes into a repository and “pull” a version of the software source-code tree to a system (e.g., for compilation).

**victim frame** In virtual memory, the frame selected by the page-replacement algorithm to be replaced.

**view** In Windows, an address range mapped in shared memory. Also, the second step in memory-mapping a file, allowing a process to access the file contents.

**virtual address** An address generated by the CPU; must be translated to a physical address before it is used.

**virtual address control block (VACB)** The data structure in Windows that represents a cache block in the unified I/O cache.

**virtual address descriptor (VAD)** In Windows, a per-process descriptor of a virtual address range, kept in a tree data structure.

**virtual address space** The logical view of how a process is stored in memory.

**virtual CPU (VCPU)** In virtualization, a virtualized host CPU available to allocate to a guest operating system by the virtual machine manager.

**virtual file system (VFS)** The file-system implementation layer responsible for separating file-system-generic operations and their implementation and representing a file throughout a network.

**virtual machine (VM)** The abstraction of hardware allowing a virtual computer to execute on a

physical computer. Multiple virtual machines can run on a single physical machine (and each can have a different operating system).

**virtual machine control structures (VMCSs)** Hardware features provided by CPUs that support virtualization to track guest state.

**virtual machine manager (VMM)** The computer function that manages the virtual machine; also called a hypervisor.

**virtual machine sprawl** The situation in which there are so many virtual machines on a system that their use, history, and state become confusing and difficult to track; caused by the ease of creating virtual machines.

**virtual memory** A technique that allows the execution of a process that is not completely in memory. Also, separation of computer memory address space from physical into logical, allowing easier programming and larger name space.

**virtual memory fork** The vfork() system call, which forks a child process, suspends the parent, and lets the child share the parent’s address space for both read and write operations (changes are visible to the parent).

**virtual private network (VPN)** An encrypted tunnel between two systems, commonly using IPSec, allowing secure remote access.

**virtual run time** A Linux scheduling aspect that records how long each task has run by maintaining the virtual run time of each task.

**virtual trust level (VTL)** A Windows 10 virtualization feature using Hyper-V to add more secure system modes.

**virtualization** A technology for abstracting the hardware of a single computer into several different execution environments, thereby creating the illusion that each environment is running on its own private computer.

**virtual-to-physical (V-to-P)** In virtualization, the conversion of a virtual machine guest to a physical system’s operating system and applications.

**virus** A fragment of code embedded in a legitimate program that, when executed, can replicate itself; may modify or destroy files and cause system crashes and program malfunctions.

**virus dropper** The part of a virus that inserts the virus into the system.

**virus signature** A pattern that can be used to identify a virus within a system.

**VMware** Virtualization software company.

**VMware Workstation** A popular commercial type 2 hypervisor for x86 Windows systems.

## G-40    Glossary

**vnode** The virtual file system file representation structure, similar to the FCB for local files but applied to remote files.

**voice recognition** A computer interface based on spoken commands, which the computer parses and turns into actions.

**volatile** Describes storage whose content can be lost in a power outage or similar event.

**volatile storage** Storage whose content can be lost in a power outage or similar event.

**volume** A container of storage; frequently, a device containing a mountable file system (including a file containing an image of the contents of a device).

**volume control block** A per-volume storage block containing data describing the volume.

**von Neumann architecture** The structure of most computers, in which both process instructions and data are stored in the same main memory.

**VT-x** Intel x86 CPU virtualization-supporting instructions.

**wait queue** In process scheduling, a queue holding processes waiting for an event to occur before they need to be put on CPU.

**wait set** In Java, a set of threads, each waiting for a condition that will allow it to continue.

**wait-for graph** In deadlock detection, a variant of the resource-allocation graph with resource nodes removed; indicates a deadlock if the graph contains a cycle.

**wear leveling** In nonvolatile memory, the effort to select all NAND cells over time as write targets to avoid premature media failure due to wearing out a subset of cells.

**wide-area network (WAN)** A network that links buildings, cities, or countries.

**WiFi** Wireless networking, consisting of devices and protocols that allow devices to attach to a network via radio waves rather than cables.

**Win32 API** The fundamental interface to the capabilities of Windows.

**Windows 10** A release of Microsoft Windows from 2009.

**Windows group policy** In Windows, a policy providing centralized management and configuration of operating systems, applications, and user settings in an Active Directory environment.

**Windows Subsystem for Linux (WSL)** A Windows 10 component allowing native Linux applications (ELF binaries) to run on Windows.

**Windows XP** A widely popular version of Microsoft Windows released in 2001.

**Winsock** The Windows socket API (similar to BSD sockets) for network communications.

**wired down** A term describing a TLB entry that is locked into the TLB and not replaceable by the usual replacement algorithm.

**wireless network** A communication network composed of radio signals rather than physical wires.

**witness** A lock order verifier.

**word** A unit made up of one or more bytes. For example, a computer that has 64-bit registers and 64-bit memory addressing typically has 64-bit (8-byte) words.

**working set** The set of pages in the most recent page references.

**working-set maximum** The maximum number of frames allowed to a process in Windows.

**working-set minimum** The minimum number of frames guaranteed to a process in Windows.

**working-set model** A model of memory access based on tracking the set of most recently accessed pages.

**working-set window** A limited set of most recently accessed pages (a “window” view of the entire set of accessed pages).

**workstation** A powerful personal computer (PC) for engineering and other demanding workloads.

**world rights** A category of file access rights.

**World Wide Web (WWW)** The Internet; a worldwide system of interconnected computer networks.

**WORM** Write-once, read-many-times storage.

**worm** A program that spreads malware between computers without intervention from humans.

**worst-fit** In memory allocation, selecting the largest hole available.

**write amplification** The creation of I/O requests not by applications but by NVM devices doing garbage collection and space management, potentially impacting the devices’ write performance.

**write pointer** The location in a file to which the next write will occur.

**write-anywhere file layout (WAFL)** The file system that is the heart of the NetApp, Inc., storage appliances.

**write-back caching** In caching, a policy whereby data are first written to the cache; later, the cache writes the change to the master copy of the data.

**write-on-close policy** In caching, a policy whereby writes reside in the cache until the file is

closed and are only then written back to the master copy of the data.

**write-through policy** In caching, a policy whereby writes are not cached but are written through the cache to the master copy of the data.

**x86-64** A class of 64-bit CPUs running an identical instruction set; the most common CPUs in desktop and server systems.

**Xen** Virtualization software company.

**XML firewall** A firewall that examines and limits XML traffic.

**Xtratum** An example of a partitioning hypervisor.

**yellow pages** A distributed naming service that provides username, password, hostname, and printer information to a set of computers.

**zero-day attacks** Attacks that have not been seen before and therefore cannot be detected via their signatures.

**zero-fill-on-demand** The writing of zeros into a page before it is made available to a process (to keep any old data from being available to the process).

**ZFS** Oracle file system, created by Sun Microsystems, with modern algorithms and features and few limits on file and device sizes.

**zombie** A process that has terminated but whose parent has not yet called wait() to collect its state and accounting information.

**zombie systems** Compromised systems that are being used by attackers without the owners' knowledge.

**zones** In application containment, a virtual layer between the operating system and a process in which the application runs, limiting its normal access to system resources. In Linux, the four regions of kernel memory.

# **WILEY END USER LICENSE AGREEMENT**

Go to [www.wiley.com/go/eula](http://www.wiley.com/go/eula) to access Wiley's ebook EULA.