FUTURE VISION BIE

One Stop for All Study Materials & Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Operating System Concepts

Seventh Edition

ABRAHAM SILBERSCHATZ
Yale University

PETER BAER GALVIN
Corporate Technologies, Inc.

GREG GAGNE
Westminster College

WILEY

JOHN WILEY & SONS. INC

Chapter 9 Virtual Memory

PART FOUR • STORAGE MANAGEMENT

Chapter 10 File-System Interface

Chapter 11 File-System Implementation

Chapter 12 Mass-Storage Structure

Chapter 13 I/O Systems

Chapter 18 Distributed Coordination

PART SEVEN ■ SPECIAL-PURPOSE SYSTEMS

Chapter 19 Real-Time Systems

Chapter 20 Multimedia Systems

PART EIGHT ■ CASE STUDIES

Chapter 21 The Linux System

Chapter 22 Windows XP

Mass-Storage Structure

The file system can be viewed logically as consisting of three parts. In Chapter 10, we saw the user and programmer interface to the file system. In Chapter 11, we described the internal data structures and algorithms used by the operating system to implement this interface. In this chapter, we discuss the lowest level of the file system: the secondary and tertiary storage structures. We first describe the physical structure of magenetic disks and magnetic tapes. We then describe disk-scheduling algorithms that schedule the order of disk I/Os to improve performance. Next, we discuss disk formatting and management of boot blocks, damaged blocks, and swap space. We then examine secondary storage structure, covering disk reliability and stable-storage implementation. We conclude with a brief description of tertiary storage devices and the problems that arise when an operating system uses tertiary storage.

CHAPTER OBJECTIVES

- Describe the physical structure of secondary and tertiary storage devices and the resulting effects on the uses of the devices.
- Explain the performance characteristics of mass-storage devices.
- Discuss operating-system services provided for mass storage, including RAID and HSM.

12.1 Overview of Mass-Storage Structure

In this section we present a general overview of the physical structure of secondary and tertiary storage devices.

12.1.1 Magnetic Disks

Magnetic disks provide the bulk of secondary storage for modern computer systems. Conceptually, disks are relatively simple (Figure 12.1). Each disk platter has a flat circular shape, like a CD. Common platter diameters range from 1.8 to 5.25 inches. The two surfaces of a platter are covered with a magnetic material. We store information by recording it magnetically on the platters.

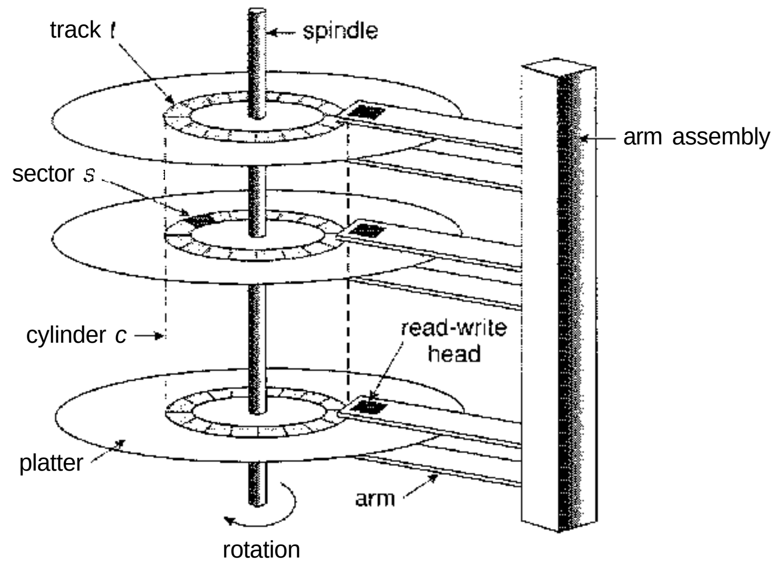


Figure 12.1 Moving-head disk mechanism.

A read–write head "flies" just above each surface of every platter. The heads are attached to a **disk arm** that moves all the heads as a unit. The surface of a platter is logically divided into circular **tracks,** which are subdivided into **sectors.** The set of tracks that are at one arm position makes up a **cylinder.** There may be thousands of concentric cylinders in a disk drive, and each track may contain hundreds of sectors. The storage capacity of common disk drives is measured in gigabytes.

When the disk is in use, a drive motor spins it at high speed. Most drives rotate 60 to 200 times per second. Disk speed has two parts. The **transfer rate** is the rate at which data flow between the drive and the computer. The **positioning time,** sometimes called the **random-access time,** consists of the time to move the disk arm to the desired cylinder, called the **seek time,** and the time for the desired sector to rotate to the disk head, called the **rotational latency.** Typical disks can transfer several megabytes of data per second, and they have seek times and rotational latencies of several milliseconds.

Because the disk head flies on an extremely thin cushion of air (measured in microns), there is a danger that the head will make contact with the disk surface. Although the disk platters are coated with a thin protective layer, sometimes the head will damage the magnetic surface. This accident is called a **head crash.** A head crash normally cannot be repaired; the entire disk must be replaced.

A disk can be **removable,** allowing different disks to be mounted as needed. Removable magnetic disks generally consist of one platter, held in a plastic case to prevent damage while not in the disk drive. **Floppy disks** are inexpensive removable magnetic disks that have a soft plastic case containing a flexible platter. The head of a floppy-disk drive generally sits directly on the disk surface, so the drive is designed to rotate more slowly than a hard-disk drive

DISK TRANSFER RATES

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

to reduce the wear on the disk surface. The storage capacity of a floppy disk is typically only 1.44 MB or so. Removable disks are available that work much like normal hard disks and have capacities measured in gigabytes.

A disk drive is attached to a computer by a set of wires called an **I/O bus.** Several kinds of buses are available, including **enhanced integrated drive electronics (EIDE), advanced technology attachment (ATA), serial ATA (SATA), universal serial bus (USB), fiber channel (FC),** and **SCSI** buses. The data transfers on a bus are carried out by special electronic processors called **controllers.** The **host controller** is the controller at the computer end of the bus. A **disk controller** is built into each disk drive. To perform a disk I/O operation, the computer places a command into the host controller, typically using memory-mapped I/O ports, as described in Section 9.7.3. The host controller then sends the command via messages to the disk controller, and the disk controller operates the disk-drive hardware to carry out the command. Disk controllers usually have a built-in cache. Data transfer at the disk drive happens between the cache and the disk surface, and data transfer to the host, at fast electronic speeds, occurs between the cache and the host controller.

12.1.2 Magnetic Tapes

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

A tape is kept in a spool and is wound or rewound past a read-write head. Moving to the correct spot on a tape can take minutes, but once positioned, tape drives can write data at speeds comparable to disk drives. Tape capacities vary greatly, depending on the particular kind of tape drive. Typically, they store from 20 GB to 200 GB. Some have built-in compression that can more than double the effective storage. Tapes and their drivers are usually categorized by width, including 4, 8, and 19 millimeters and 1/4 and 1/2 inch. Some are named according to technology, such as LTO-2 and SDLT. Tape storage is further described in Section 12.9.

FIREWIRE

FireWire refers fco^aiiijitiierftieE^siesagiiyd; for connecting peripheral devices such as hard drives, DVD drives, and digital video cameras to a computer system. FireWire was first developed by Apple Computer and became the IEEE 1394 standard in 1995. The original FireWire standard provided bandwidth up to 400 megabits per second. Recently, a new standard—FireWire 2—has emerged and is identified by the IEEE 1394b standard. FireWire 2 provides double the data rate of the original FireWire—800 megabits per second.

12.2 Disk Structure

Modern disk drives are addressed as large one-dimensional arrays of **logical blocks,** where the logical block is the smallest unit of transfer. The size of a logical block is usually 512 bytes, although some disks can be **low-level formatted** to have a different logical block size, such as 1,024 bytes. This option is described in Section 12.5.1. The one-dimensional array of logical blocks is mapped onto the sectors of the disk sequentially. Sector 0 is the first sector of the first track on the outermost cylinder. The mapping proceeds in order through that track, then through the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

By using this mapping, we can—at least in theory—convert a logical block number into an old-style disk address that consists of a cylinder number, a track number within that cylinder, and a sector number within that track. In practice, it is difficult to perform this translation, for two reasons. First, most disks have some defective sectors, but the mapping hides this by substituting spare sectors from elsewhere on the disk. Second, the number of sectors per track is not a constant on some drives.

Let's look more closely at the second reason. On media that use **constant linear velocity** (CLV), the density of bits per track is uniform. The farther a track is from the center of the disk, the greater its length, so the more sectors it can hold. As we move from outer zones to inner zones, the number of sectors per track decreases. Tracks in the outermost zone typically hold 40 percent more sectors than do tracks in the innermost zone. The drive increases its rotation speed as the head moves from the outer to the inner tracks to keep the same rate of data moving under the head. This method is used in CD-ROM and DVD-ROM drives. Alternatively, the disk rotation speed can stay constant, and the density of bits decreases from inner tracks to outer tracks to keep the data rate constant. This method is used in hard disks and is known as **constant** angular **velocity (CAV).**

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

12.3 Disk Attachment

Computers access disk storage in two ways. One way is via I/O ports (or host-attached storage); this is common on small systems. The other way is via a remote host in a distributed file system; this is referred to as network-attached storage.

12.3.1 Host-Attached Storage

Host-attached storage is storage accessed through local I/O ports. These ports use several technologies. The typical desktop PC uses an I/O bus architecture called IDE or ATA. This architecture supports a maximum of two drives per I/O bus. A newer, similar protocol that has simplified cabling is SATA. High-end workstations and servers generally use more sophisticated I/O architectures, such as SCSI and fiber channel (FC).

SCSI is a bus architecture. Its physical medium is usually a ribbon cable having a large number of conductors (typically 50 or 68). The SCSI protocol supports a maximum of 16 devices on the bus. Generally, the devices include one controller card in the host (the **SCSI initiator)** and up to 15 storage devices (the **SCSI targets).** A SCSI disk is a common SCSI target, but the protocol provides the ability to address up to 8 **logical units** in each SCSI target. A typical use of logical unit addressing is to direct commands to components of a RAID array or components of a removable media library (such as a CD jukebox sending commands to the media-changer mechanism or to one of the drives).

FC is a high-speed serial architecture that can operate over optical fiber or over a four-conductor copper cable. It has two variants. One is a large switched fabric having a 24-bit address space. This variant is expected to dominate in the future and is the basis of **storage-area networks** (SANs), discussed in Section 12.3.3. Because of the large address space and the switched nature of the communication, multiple hosts and storage devices can attach to the fabric, allowing great flexibility in I/O communication. The other PC variant is an **arbitrated loop (FC-AL)** that can address 126 devices (drives and controllers).

A wide variety of storage devices are suitable for use as host-attached storage. Among these are hard disk drives, RAID arrays, and CD, DVD, and tape drives. The I/O commands that initiate data transfers to a host-attached storage device are reads and writes of logical data blocks directed to specifically identified storage units (such as bus ID, SCSI ID, and target logical unit).

12.3.2 Network-Attached Storage

A network-attached storage (NAS) device is a special-purpose storage system that is accessed remotely over a data network (Figure 12.2). Clients access network-attached storage via a remote-procedure-call interface such as NFS for UNIX systems or CIFS for Windows machines. The remote procedure calls (RPCs) are carried via TCP or UDP over an IP network—-usually the same local-area network (LAN) that carries all data traffic to the clients. The network-attached storage unit is usua lly implemented as a RAID array with software that implements the RPC interface. It is easiest to think of NAS as simply another storage-access protocol. For example, rather than using a SCSI device driver and SCSI protocols to access storage, a system using NAS would use RPC over TCP/IP.

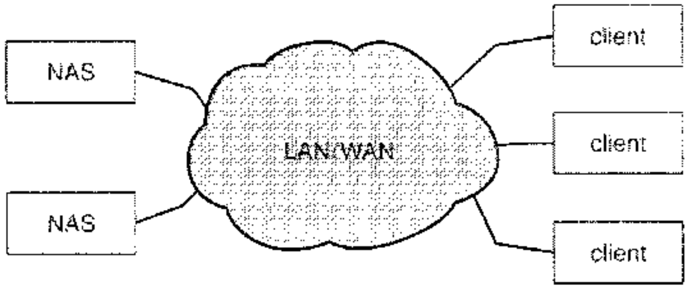


Figure 12.2 Network-attached storage.

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

ISCSI is the latest network-attached storage protocol. In essence, it uses the IP network protocol to carry the SCSI protocol. Thus, networks rather than SCSI cables can be used as the interconnects between hosts and their storage. As a result, hosts can treat their storage as if it were directly attached, but the storage can be distant from the host.

12.3.3 Storage-Area Network

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

A storage-area network (SAN) is a private network (using storage protocols rather than networking protocols) connecting servers and storage units, as shown in Figure 12.3. The power of a SAN lies in its flexibility. Multiple hosts and multiple storage arrays can attach to the same SAN, and storage can be dynamically allocated to hosts. A SAN switch allows or prohibits access between the hosts and the storage. As one example, if a host is running low-on disk space, the SAN can be configured to allocate more storage to that host. SANs make it possible for clusters of servers to share the same storage and for storage arrays to include multiple direct host connections. SANs typically have more ports, and less expensive ports, than storage arrays. FC is the most common. SAN interconnect.

An emerging alternative is a special-purpose bus architecture named InfiniBand, which provides hardware and software support for high-speed interconnection networks for servers and storage units.

12.4 Disk Scheduling

One of the responsibilities of the operating system is to use the hardware efficiently. For the disk drives, meeting this responsibility entails having

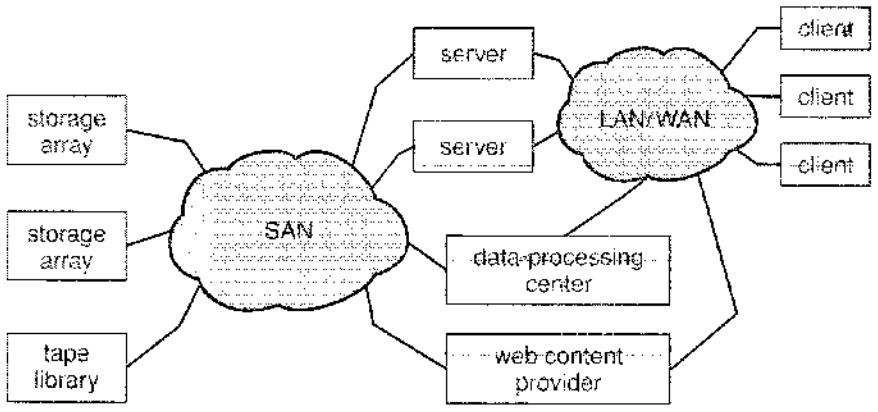


Figure 12.3 Storage-area network.

fast access time and large disk bandwidth. The access time has two major components (also see Section 12.1.1). The **seek time** is the time for the disk arm to move the heads to the cylinder containing the desired sector. The **rotational latency** is the additional time for the disk to rotate the desired sector to the disk head. The disk **bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer. We can improve both the access time and the bandwidth by scheduling the servicing of disk I/O requests in a good order.

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

- Whether this operation is input or output
- What the disk address for the transfer is
- What the memory address for the transfer is
- What the number of sectors to be transferred is

If the desired disk drive and controller are available, the request can be serviced immediately. If the drive or controller is busy, any new requests for service will be placed in the queue of pending requests for that drive. For a multiprogramming system with many processes, the disk queue may often have several pending requests. Thus, when one request is completed,, the operating system chooses which pending request to service next. How does the operating system make this choice? Any one of several disk-scheduling algorithms can be used, and we discuss them next.

12.4.1 FCFS Scheduling

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

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

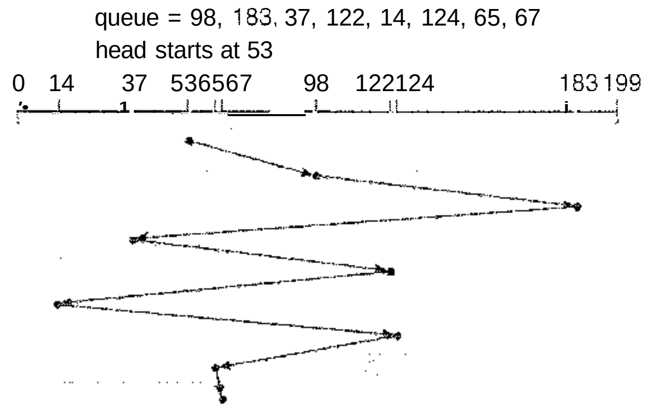


Figure 12.4 FCFS disk scheduling.

in that order. If the disk head is initially at cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 124/65, and finally to 67, for a total head movement of 640 cylinders. This schedule is diagrammed in Figure 12.4.

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

12.4.2 SSTF Scheduling

It seems reasonable to service all the requests close to the current head position before moving the head far away to service other requests. This assumption is the basis for the **shortest-seek-time-first (SSTF) algorithm.** The SSTF algorithm selects the request with the minimum seek time from the current head position. Since seek time increases with the number of cylinders traversed by the head, SSTF chooses the pending request closest to the current head position.

For our example request queue, the closest request to the initial head position (53) is at cylinder 65. Once we are at cylinder 65, the next closest request is at cylinder 67. From there, the request at cylinder 37 is closer than the one at 98, so 37 is served next. Continuing, we service the request at cylinder 14, then 98, 122, 124, and finally 183 (Figure 12.5). This scheduling method results in a total head movement of only 236 cylinders—little more than one-third of the distance needed for FCFS scheduling of this request queue. This algorithm gives a substantial improvement in performance.

SSTF scheduling is essentially a form of shortest-job-first (SJF) scheduling; and like SJF scheduling, it may cause starvation of some requests. Remember that requests may arrive at any time. Suppose that we have two requests in the queue, for cylinders 14 and 186, and while servicing the request from 14, a new request near 14 arrives. This new request will be serviced next, making the request at 186 wait. While this request is being serviced, another request close to 14 could arrive. In theory, a continual stream of requests near one another could arrive, causing the request for cylinder 186 to wait indefinitely.

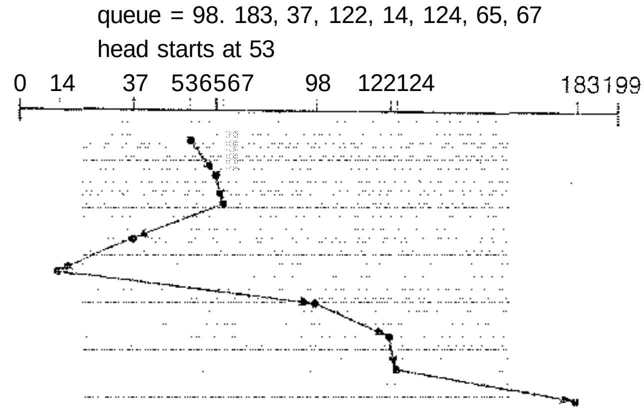


Figure 12.5 SSTF disk scheduling.

This scenario becomes increasingly likely if the pending-request queue grows long.

Although the SSTF algorithm is a substantial improvement over the FCFS algorithm, it is not optimal. In the example, we can do better by moving the head from 53 to 37, even though the latter is not closest, and then to 14, before turning around to service 65, 67, 98, 122, 124, and 183. This strategy reduces the total head movement to 208 cylinders.

12.4.3 SCAN Scheduling

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

Let's return to our example to illustrate. Before applying SCAN to schedule the requests on cylinders 98, 183, 37, 122, 14, 124, 65, and 67, we need to know the direction of head movement in addition to the head's current position (53). If the disk arm is moving toward 0, the head will service 37 and then 14. At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the requests at 65, 67, 98, 122, 124, and 183 (Figure 12.6). If a request arrives in the queue just in front of the head, it will be serviced almost immediately; a request arriving just behind the head will have to wait until the arm moves to the end of the disk, reverses direction, and comes back.

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

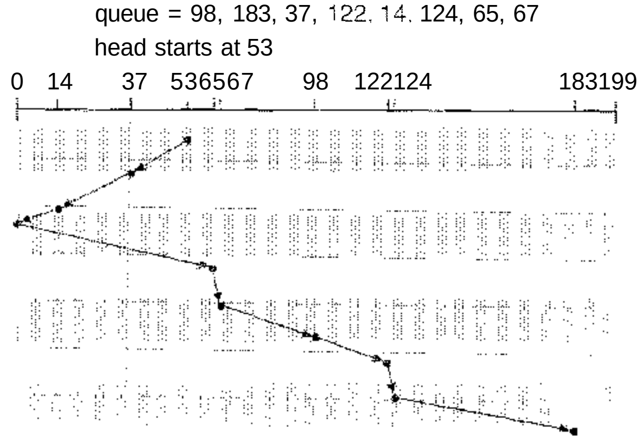


Figure 12.6 SCAN disk scheduling.

12.4.4 C-SCAN Scheduling

Circular SCAN (C-SCAN) **scheduling** is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip (Figure 12.7). The C-SCAN scheduling algorithm essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one.

12.4.5 LOOK Scheduling

As we described them, both SCAN and C-SCAN move the disk arm across the full width of the disk. In practice, neither algorithm is often implemented this way. More commonly, the arm goes only as far as the final request in each

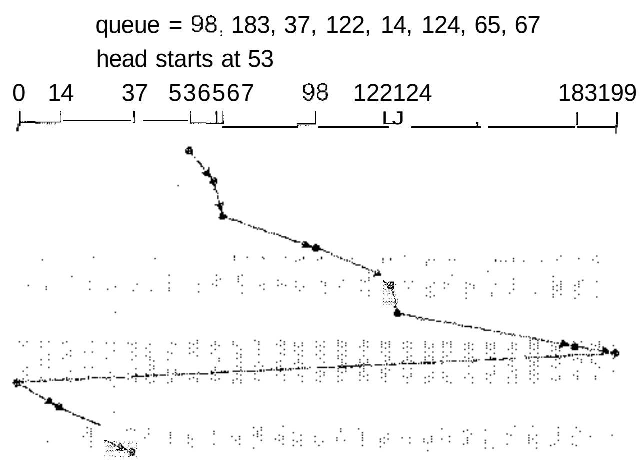


Figure 12.7 C-SCAN disk scheduling.

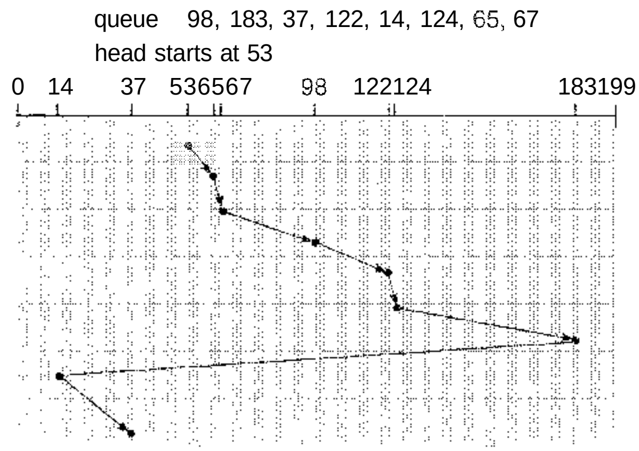


Figure 12.8 C-LOOK disk scheduling.

direction. Then, it reverses direction immediately, without going all the way to the end of the disk. Versions of SCAN and C-SCAN that follow this pattern are called **LOOK** and **C-LOOK scheduling,** because they *look* for a request before continuing to move in a given direction (Figure 12.8).

12.4.6 Selection of a Disk-Scheduling Algorithm

Given so many disk-scheduling algorithms, how do we choose the best one? SSTF is common and has a natural appeal because it increases performance over FCFS. SCAM and C-SCAN perform better for systems that place a heavy load on the disk, because they are less likely to cause a starvation problem. For any particular list of requests, we can define an optimal order of retrieval, but the computation needed to find an optimal schedule may not justify the savings over SSTF or SCAN. With any scheduling algorithm, however, performance depends heavily on the number and types of requests. For instance, suppose that the queue usually has just one outstanding request. Then, all scheduling algorithms behave the same, because they have only one choice for where to move the disk head: They all behave like FCFS scheduling.

Requests for disk service can be greatly influenced by the file-allocation method. A program reading a contiguously allocated file will generate several requests that are close together on the disk, resulting in limited head movement. A linked or indexed file, in contrast, may include blocks that are widely scattered on the disk, resulting in greater head movement.

The location of directories and index blocks is also important. Since every file must be opened to be used, and opening a file requires searching the directory structure, the directories will be accessed frequently. Suppose that a directory entry is on the first cylinder and a file's data are on the final cylinder. In this case, the disk head has to move the entire width of the disk. If the directory entry were on the middle cylinder, the head would have to move, at most, one-half the width. Caching the directories and index blocks in main memory can also help to reduce the disk-arm movement, particularly for read requests.

Because of these complexities, the disk-scheduling algorithm should be written as a separate module of the operating system, so that it can be replaced with a different algorithm if necessary. Either SSTF or LOOK is a reasonable choice for the default algorithm.

The scheduling algorithms described here consider only the seek distances. For modern disks, the rotational latency can be nearly as large as the average seek time. It is difficult for the operating system to schedule for improved rotational latency, though, because modern disks do not disclose the physical location of logical blocks. Disk manufacturers have been alleviating this problem by implementing disk-scheduling algorithms in the controller hardware built into the disk drive. If the operating system sends a batch of requests to the controller, the controller can queue them and then schedule them to improve both the seek time and the rotational latency.

If I/O performance were the only consideration, the operating system would gladly turn over the responsibility of disk scheduling to the disk hardware. In practice, however, the operating system may have other constraints on the service order for requests. For instance, demand paging may take priority over application I/O, and writes are more urgent than reads if the cache is running out of free pages. Also, it may be desirable to guarantee the order of a set of disk writes to make the file system robust in the face of system crashes. Consider what could happen if the operating system allocated a disk page to a file and the application wrote data into that page before the operating system had a chance to flush the modified inode and free-space list back to disk. To accommodate such requirements, an operating system may choose to do its own disk scheduling and to spoon-feed the requests to the disk controller, one by one, for some types of I/O.

12.5 Disk Management

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

12.5.1 Disk Formatting

A new magnetic disk is a blank slate: It is just a platter of a magnetic recording material. Before a disk can store data, it must be divided into sectors that the disk controller can read and write. This process is called low-level formatting, or physical formatting. Low-level formatting fills the disk with a special data structure for each sector. The data structure for a sector typically consists of a header, a data area (usually 512 bytes in size), and a trailer. The header and trailer contain information used by the disk controller, such as a sector number and an error-correcting code (ECC). When the controller writes a sector of data during normal I/O, the ECC is updated with a value calculated from all the bytes in the data area. When the sector is read, the ECC is recalculated and is compared with the stored value. If the stored and calculated numbers are different, this mismatch indicates that the data area of the sector has become corrupted and that the disk sector may be bad (Section 12.5.3). The ECC is an error-*correcting* code because it contains enough information that, if only a few bits of data have been corrupted, the controller can identify which bits, have changed and can calculate what their correct values should be. It then reports a recoverable soft error. The controller automatically does the ECC processing whenever a sector is read or written.

Most hard disks are low-level-formatted at the factory as a part of the manufacturing process. This formatting enables the manufacturer to test the disk and to initialize the mapping from logical block numbers to defect-free sectors on the disk. For many hard disks, when the disk controller is instructed to low-level-format the disk, it can also be told how many bytes of data space to leave between the header and trailer of all sectors. It is usually possible to choose among a few sizes, such as 256, 512, and 1,024 bytes. Formatting a disk with a larger sector size means that fewer sectors can fit on each track; but it also means that fewer headers and trailers are written on each track and more space is available for user data. Some operating systems can handle only a sector size of 512 bytes.

To use a disk to hold files, the operating system still needs to record its own data structures on the disk. It does so in two steps. The first step is to **partition** the disk into one or more groups of cylinders. The operating system can treat each partition as though it were a separate disk. For instance, one partition can hold a copy of the operating system's executable code, while another holds user files. After partitioning, the second step is **logical formatting** (or creation of a file system). In this step, the operating system stores the initial file-system data structures onto the disk. These data structures may include maps of free and allocated space (a FAT or modes) and an initial empty directory.

To increase efficiency, most file systems group blocks together into larger chunks, frequently called **clusters.** Disk I/O is done via blocks, but file system I/O is done via clusters, effectively assuring that I/O has more sequential-access and fewer random-access characteristics.

Some operating systems give special programs the ability to use a disk partition as a large sequential array of logical blocks, without any file-system data structures. This array is sometimes called the raw disk, and I/O to this array is termed raw I/O. For example, some database systems prefer raw I/O because it enables them to control the exact disk location where each database record is stored. Raw I/O bypasses all the file-system services, such as the buffer cache, file locking, prefetching, space allocation, file names, and directories. We can make certain applications more efficient by allowing them to implement their own special-purpose storage services on a raw partition, but most applications perform better when they use the regular file-system services.

12.5.2 Boot Block

For a computer to start running—for instance, when it is powered up or rebooted—it must have an initial program to run. This initial *bootstrap* program tends to be simple. It initializes all aspects of the system, from CPU registers to device controllers and the contents of main memory, and then starts the operating system. To do its job, the bootstrap program finds the operating-system kernel on disk, loads that kernel into memory, and jumps to an initial address to begin the operating-system execution.

For most computers, the bootstrap is stored in read-only memory (ROM). This location is convenient, because ROM needs no initialization and is at a fixed

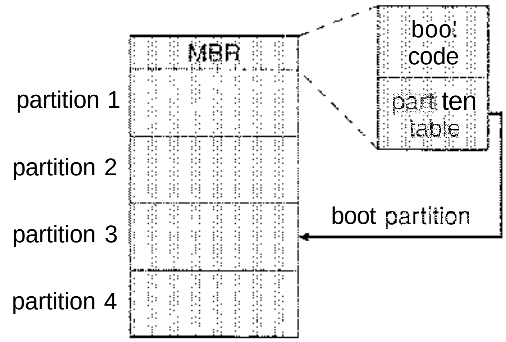


Figure 12.9 Booting from disk in Windows 2000.

location that the processor can start executing when powered up or reset. And, since ROM is read only, it cannot be infected by a computer virus. The problem is that changing this bootstrap code requires changing the ROM, hardware chips. For this reason, most systems store a tiny bootstrap loader program in the boot ROM whose only job is to bring in a full bootstrap program from disk. The full bootstrap program can be changed easily: A new version is simply written onto the disk. The full bootstrap program is stored in "the boot blocks" at a fixed location on the disk. A disk that has a boot partition is called a **boot disk** or **system disk.**

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

Let's consider as an example the boot process in Windows 2000. The Windows 2000 system places its boot code in the first sector on the hard disk (which it terms the **master boot record,** or MBR). Furthermore, Windows 2000 allows a hard disk to be divided into one or more partitions; one partition, identified as the **boot partition,** contains the operating system and device drivers. Booting begins in a Windows 2000 system by running code that is resident in the system's ROM memory. This code directs the system to read the boot code from, the MBR. In addition to containing boot code, the MBR contains a table listing the partitions for the hard disk and a flag indicating which partition the system is to be booted from. This is illustrated in Figure 12.9. Once the system identifies the boot partition, it reads the first sector from that partition (which is called the **boot sector)** and continues with the remainder of the boot process, which includes loading the various subsystems and system services.

12.5.3 Bad Blocks

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

On simple disks, such as some disks with IDE controllers, bad blocks are handled manually. For instance, the MS-DOS format command performs logical formatting and, as a part of the process, scans the disk to find bad blocks. If format finds a bad block, it writes a special value into the corresponding FAT entry to tell the allocation routines not to use that block. If blocks go bad during normal operation, a special program (such as chkdsk) must be run manually to search for the bad blocks and to lock them away as before. Data that resided on the bad blocks usually are lost.

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

A typical bad-sector transaction might be as follows:

- The operating system tries to read logical block 87.
- The controller calculates the ECC and finds that the sector is bad. It reports this finding to the operating system.
- The next time the system is rebooted, a special command is run to tell the SCSI controller to replace the bad sector with a spare.
- After that, whenever the system requests logical block 87, the request is translated into the replacement sector's address by the controller.

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

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

The replacement of a bad block generally is not totally automatic because the data in the bad block are usually lost. Several soft errors could trigger a process in which a copy of the block data is made and the block is spared or slipped. An unrecoverable hard error, however, results in lost data. Whatever file was using that block must be repaired (for instance, by restoration from a backup tape), and that requires manual intervention.

12.6 Swap-Space Management

Swapping was first presented in Section 8.2, where we discussed moving entire processes between disk and main memory. Swapping in that setting occurs when the amount of physical memory reaches a critically low point and processes (which are usually selected because they are the least active) are moved from memory to swap space to free available memory. In practice, very few modern operating systems implement swapping in this fashion. Rather, systems now combine swapping with virtual memory techniques (Chapter 9) and swap pages, not necessarily entire processes. In fact, some systems now use the terms *swapping* and *paging* interchangeably, reflecting the merging of these two concepts.

Swap-space management is another low-level task of the operating system. Virtual memory uses disk space as an extension of main memory. Since disk access is much slower than memory access, using swap space significantly decreases system performance. The main goal for the design and implementation of swap space is to provide the best throughput for the virtual memory system. In this section, we discuss how swap space is used, where swap space is located on disk, and how swap space is managed.

12.6.1 Swap-Space Use

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

Note that it may be safer to overestimate than to underestimate the amount of swap space required, because if a system runs out of swap space it may be forced to abort processes or may crash entirely. Overestimation wastes disk space that could otherwise be used for files, but it does no other harm. Some systems recommend the amount to be set aside for swap space. Solaris, for example, suggests setting swap space equal to the amount by which virtual memory exceeds pageable physical memory. Historically, Linux suggests setting swap space to double the amount of physical memory, although most Linux systems now use considerably less swap space. In fact, there is currently much debate in the Linux community about whether to set aside swap space at all!

Some operating systems—including Linux—allow the use of multiple swap spaces. These swap spaces are usually put on separate disks so the load placed on the I/O system by paging and swapping can be spread over the system's I/O devices.

12.6.2 Swap-Space Location

A swap space can reside in one of two places: It can be carved out of the normal file system, or it can be in a separate disk partition. If the swap space is simply a large file within the file system, normal file-system routines can be used to create it, name it, and allocate its space. This approach, though easy to implement, is inefficient. Navigating the directory structure and the disk-allocation data structures takes time and (potentially) extra disk accesses. External fragmentation can greatly increase swapping times by forcing multiple seeks during reading or writing of a process image. We can improve performance by caching the block location information in physical memory and by using special tools to allocate physically contiguous blocks for the swap file, but the cost of traversing the file-system data structures still remains.

Alternatively, swap space can be created in a separate raw partition, as no file system or directory structure is placed in this space. Rather, a separate swap-space storage manager is used to allocate and deallocate the blocks from the raw partition. This manager uses algorithms optimized for speed rather than for storage efficiency, because swap space is accessed much more frequently than file systems (when it is used). Internal fragmentation may increase, but this trade-off is acceptable because the life of data in the swap space generally is much shorter than that of files in the file system. Swap space is reinitialized at boot time so any fragmentation is short-lived. This approach creates a fixed amount of swap space during disk partitioning. Adding more swap space requires repartitioning the disk (which involves moving the other file-system, partitions or destroying them and restoring them from backup) or adding another swap space elsewhere.

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

12.6.3 Swap-Space Management: An Example

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

In Solaris 1 (SunOS), the designers changed standard UNIX methods to improve efficiency and reflect technological changes. When a process executes, text-segment pages containing code are brought in from the file system, accessed in main memory, and thrown away if selected for pageout. It is more efficient to reread a page from the file system than to write it to swap space and then reread it from there. Swap space is only used as a backing store for pages of anonymous memory, which includes memory allocated for the stack, heap, and uninitialized data of a process.

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

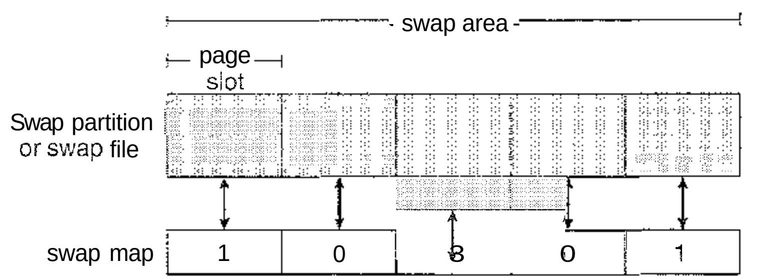


Figure 12.10 The data structures for swapping on Linux systems.

Linux is similar to Solaris in that swap space is only used for anonymous memory or for regions of memory shared by several processes. Linux allows one or more swap areas to be established. A swap area may be in either a swap file on a regular file system or a raw swap partition. Each swap area consists of a series of 4-KB page slots, which are used to hold swapped pages. Associated with each swap area is a swap **map**—an array of integer counters, each corresponding to a page slot in the swap area. Tf the value of a counter is 0, the corresponding page slot is available. Values greater than 0 indicate that the page slot is occupied by a swapped page. The value of the counter indicates the number of mappings to the swapped page; for example, a value of 3 indicates that the swapped page is mapped to three different processes (which can occur if the swapped page is storing a region of memory shared by three processes). The data structures for swapping on Linux systems are shown in Figure 12.10.

12.7 RAID Structure

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

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

12.7.1 Improvement of Reliability via Redundancy

Let us first consider the reliability of RAIDs. The chance that some disk out of a set of N disks will fail is much higher than the chance that a specific single disk will fail. Suppose that the mean time to failure of a single disk is 100,000 hours. Then the mean time to failure of some disk in an array of 100 disks

STRUCTURING RAID

RAID storage can be structured in a variety of ways. For example, a system can fes-e disks directly attached to its buses. In this case, the operating syste m software can implement RAID functionality. Alternatively, an in s and can implement RAID on those disks in hardware. Finally, a **storage array** or **RAID array** can be used. A RAID array is a standalone unit with its own controller, cache (usually), and disks. It is attached to the host via one or more standard AT ers. This common setup allows any operating system and software without RAID functionality to h is even used on systems that do have RAID software layers because of its simplicity and flexibility.

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

The solution to the problem of reliability is to introduce **redundancy;** we store extra information that is not normally needed but that can be used in the event of failure of a disk to rebuild the lost information. Thus, even if a disk fails, data are not lost.

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

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

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

Power failures are a particular source of concern, since they occur far more frequently than do natural disasters. Even with mirroring of disks, if writes are in progress to the same block in both disks, and power fails before both blocks are fully written, the two blocks can be in an inconsistent state. One solution to this problem is to write one copy first, then the next, so that one of the two copies is always consistent. Another is to add a nonvolatile' RAM (NVRAM) cache to the RAID array. This write-back cache is protected from data loss during power failures, so the write can be considered complete at that point, assuming the NVRAM has some kind of error protection and correction., such as ECC or mirroring.

12.7.2 Improvement in Performance via Parallelism

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

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

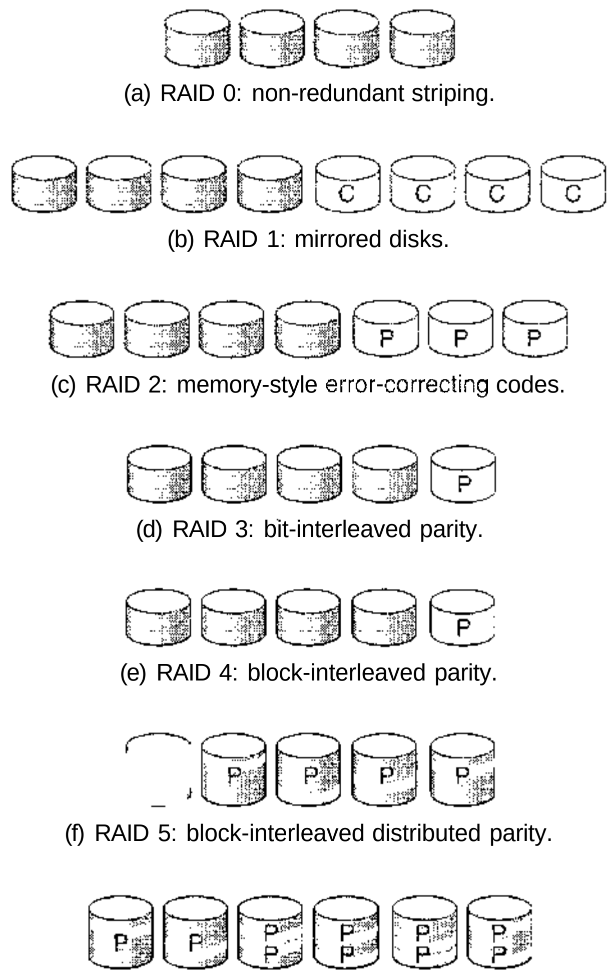
Bit-level striping can be generalized to include a number of disks that either is a multiple of 8 or divides 8. For example, if we use an array of four disks, bits i and $4 + i$ of each byte go to disk i. Further, striping need not be at the bit level. For example, in **block-level striping,** blocks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

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

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

12.7.3 RAID Levels

Mirroring provides high reliability, but it is expensive. Striping provides high data-transfer rates, but it does not improve reliability. Numerous schemes to provide redundancy at lower cost by using the idea of disk striping combined with "parity" bits (which we describe next) have been proposed. These schemes have different cost-performance trade-offs and are classified according to levels called **RAID levels.** We describe the various levels here; Figure 12.11 shows them pictorially (in the figure, P indicates error-correcting bits, and C indicates a second copy of the data). In all cases depicted in the figure, four disks' worth of data are stored, and the extra disks are used to store redundant information for failure recovery.



(a) RAID 0: non-redundant striping.

(b) RAID 1: mirrored disks.

(c) RAID 2: memory-style error-correcting codes.

(d) RAID 3: bit-interleaved parity.

(e) RAID 4: block-interleaved parity.

(f) RAID 5: block-interleaved distributed parity.

(g) RAID 6: P + Q redundancy.

Figure 12.11 RAID levels.

- RAID Level 0. RAID level 0 refers to disk arrays with striping at the level of blocks but without any redundancy (such as mirroring or parity bits), as shown in Figure 12.11(a).
- RAID **Level** 1. RAID level 1 refers to disk mirroring. Figure 12.11 (b) shows a mirrored organization.
- RAID Level 2. RAID level 2 is also known as **memory-style** error-correcting-**code (ECC) organization.** Memory systems have long detected certain errors by using parity bits. Each byte in a memory system may have a parity bit associated with it that records whether the number of bits in the byte set to 1 is even (parity = 0) or odd (parity = 1). If one of the bits in the byte is damaged (either a 1 becomes a 0, or a 0 becomes a 1), the parity of the byte changes and thus will not match the stored parity. Similarly, if the stored parity bit is damaged, it will not match the computed parity. Thus, all single-bit errors are detected by the memory system. Error-correcting

schemes store two or more extra bits and can reconstruct the data if a single bit is damaged. The idea of ECC can be used directly in disk arrays via striping of bytes across disks. For example, the first bit of each byte can be stored in disk 1, the second bit in disk 2, and so on until the eighth bit is stored in disk 8; the error-correction bits are stored in further disks. This scheme is shown pictorially in Figure 12.11(c), where the disks labeled *P* store the error-correction bits. If one of the disks fails, the remaining bits of the byte and the associated error-correction bits can be read from other disks and used to reconstruct the damaged data. Note that RAID level 2 requires only three disks' overhead for four disks of data, unlike RAID level 1, which requires four disks' overhead.

* RAID Level 3. RAID level 3, or bit-interleaved parity organization, improves on level 2 by taking into account the fact that, unlike memory systems, disk controllers can detect whether a sector has been read correctly, so a single parity bit can be used for error correction as well as for detection. The idea is as follows: If one of the sectors is damaged, we know exactly which sector it is, and we can figure out whether any bit in the sector is a 1 or a 0 by computing the parity of the corresponding bits from sectors in the other disks. If the parity of the remaining bits is equal to the stored parity, the missing bit is 0; otherwise, it is 1. RAID level 3 is as good as level 2 but is less expensive in the number of extra disks required (it has only a one-disk overhead), so level 2 is not used in practice. This scheme is shown pictorially in Figure 12.11(d).

  RAID level 3 has two advantages over level 1. First, the storage overhead is reduced because only one parity disk is needed for several regular disks, whereas one mirror disk is needed for every disk in level 1. Second, since reads and writes of a byte are spread out over multiple disks with *N*-way striping of data, the transfer rate for reading or writing a single block is *N* times as fast as with RAID level 1. On the negative side, RAID level 3 supports fewer I/Os per second, since every disk has to participate in every I/O request.

  A further performance problem with RAID 3—and with all parity-based RAID levels—is the expense of computing and writing the parity. This overhead results in significantly slower writes than with non-parity RAID arrays. To moderate this performance penalty, many RAID storage arrays include a hardware controller with dedicated parity hardware. This controller offloads the parity computation from the CPU to the array. The array has an NVRAM cache as well, to store the blocks while the parity is computed and to buffer the writes from the controller to the spindles. This combination can make parity RAID almost as fast as non-parity. In fact, a caching array doing parity RAID can outperform a non-caching non-parity RAID.

• RAID Level 4. RAID level 4, or block-interleaved parity organization, uses block-level striping, as in RAID 0, and in addition keeps a parity block on a separate disk for corresponding blocks from A! other disks. This scheme is diagramed in Figure 12.11(e). If one of the disks fails, the parity block can be used with the corresponding blocks from the other disks to restore the blocks of the failed disk.

A block read accesses only one disk, allowing other requests to be processed by the other disks. Thus, the data-transfer rate for each access is slower, but multiple read accesses can proceed in parallel, leading to a higher overall I/O rate. The transfer rates for large reads are high, since all the disks can be read in parallel; large writes also have high transfer rates, since the data and parity can be written in parallel.

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

WAFL (Chapter 11) uses RAID level 4 because this RAID level allows disks to be added to a RAID set seamlessly. If the added ciisks are initialized with blocks containing all zeros, then the parity value does not change, and the RAID set is still correct.

- RAID Level 5. RAID level 5, or block-interleaved distributed parity, differs from level 4 by spreading data and parity among all $N + 1$ disks, rather than storing data in N disks and parity in one disk. For each block, one of the disks stores the parity, and the others store data. For example, with an array of five disks, the parity for the nth block is stored in disk $(n \bmod 5)+1$; the nth blocks of the other four disks store actual data for that block. This setup is shown in Figure 12.11(f), where the Ps are distributed across all the disks. A parity block cannot store parity for blocks in the same disk, because a disk failure would result in loss of data as well as of parity, and hence the loss would not be recoverable. By spreading the parity across all the disks in the set, RAID 5 avoids the potential overuse of a single parity-disk that can occur with RAID 4. RAID 5 is the most common parity RAID system.

- RAID Level 6. RAID level 6, also called the P + Q redundancy scheme, is much like RAID level 5 but stores extra redundant information to guard against multiple disk failures. Instead of parity, error-correcting codes such as the Reed-Solomon codes are used. In the scheme shown in Figure 12.11(g), 2 bits of redundant data are stored for every 4 bits of data—compared with 1 parity bit in level 5—and the system can tolerate two disk failures.

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

  Another RAID option that is becoming available commercially is RAID level 1 + 0, in which disks are mirrored in pairs, and then the resulting mirror pairs are striped. This RAID has some theoretical advantages over RAID 0 + 1. For example, if a single disk fails in RAID 0 + 1, the entire

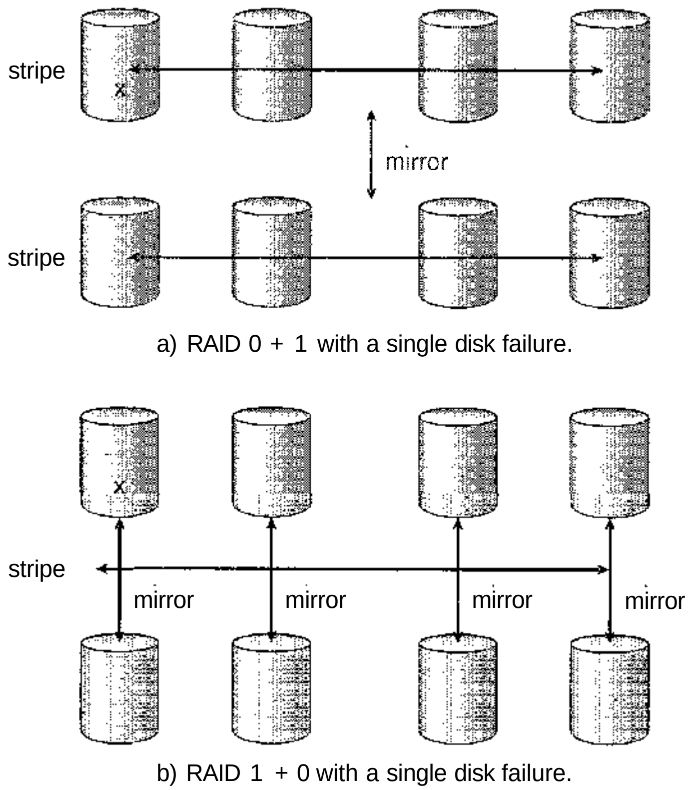


Figure 12.12 RAID 0 + 1 and 1 + 0.

stripe is inaccessible, leaving only the other stripe available. With a failure in RAID 1 + 0, the single disk is unavailable, but its mirrored pair is still available, as are all the rest of the disks (Figure 12.12).

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

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

- Volume-management software can implement RAID within the kernel or at the system software layer. In this case, the storage hardware can provide a minimum of features and still be part of a full RAID solution. Parity RAID is fairly slow when implemented in software, so typically RAID 0, 1, or 0 + 1 is used.

- RAID can be implemented in the host bus-adapter (HBA) hardware. Only the disks directly connected to the HBA can be part of a given RAID set. This solution is low in cost but not very flexible.

- RAID can be implemented in the hardware of the storage array. The storage array can create RAID sets of various levels and can even slice these sets into smaller volumes, which are then presented to the operating system.

The operating system need only implement the file system on each ©f the volumes. Arrays can have multiple connections available or can be part of a SAN, allowing multiple hosts to take advantage of the array's features.

- RAID can be implemented in the SAN interconnect layer by disk virtualization devices. In this case, a device sits between the hosts and the storage. It accepts commands from the servers and manages access to the storage. It could provide mirroring, for example, by writing each block to two separate storage devices.

Other features, such as snapshots and replication, can be implemented at each of these levels as well. Replication involves the automatic duplication of writes between separate sites for redimdancy and disaster recovery. Replication can be synchronous or asynchronous. In synchronous replication, each block must be written locally and remotely before the write is considered complete, whereas in asynchronous replication, the writes are grouped together and written periodically. Asynchronous replication can result in data loss if the primary site fails but is faster and has no distance limitations.

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

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

12.7.4 Selecting a RAID Level

Given the many choices they have, how do system designers choose a RAID level? One consideration is rebuild performance. If a disk fails, the time needed to rebuild its data can be significant and will vary with the RAID level used. Rebuilding is easiest for RAID level 1, since data can be copied from another disk; for the other levels, we need to access all the other disks in the array to rebuild data in a failed disk. The rebuild performance of a RAID system may be an important factor if a continuous supply of data is required, as it is in high-performance or interactive database systems. Furthermore, rebuild performance influences the mean time to failure. Rebuild times can be hours for RAID 5 rebuilds of large disk sets.

RAID level 0 is used in high-performance applications where data loss is not critical. RAID level 1 is popular for applications that require high reliability with fast recovery. RAID 0 + 1 and 1-0 are used where both performance and reliability are important—for example, for small databases. Due to RAID 1's high space overhead, RAID level 5 is often preferred for storing large volumes of data. Level 6 is not supported currently by many RAID implementations, but it should offer better reliability than level 5.

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

12.7.5 Extensions

The concepts of RAID have been generalized to other storage devices, including arrays of tapes, and even to the broadcast of data over wireless systems. When applied to arrays of tapes, RAID structures are able to recover data even if one of the tapes in an array is damaged. When applied to broadcast of data, a block of data is split into short units and is broadcast along with a parity unit; if one of the units is not received for any reason, it can be reconstructed from the other units. Commonly, tape-drive robots containing multiple tape drives will stripe data across all the drives to increase throughput and decrease backup time.

THE InServ STORAGE ARRAY

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

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

A further innovation is **utility storage**. Some file systems do not expand or shrink. On these file systems, the original size is the only size, and any changes require copying data. An administrator can configure InServ to provide a host with a lar amount of physical storage. As the host starts using the storage, unused disks are allocated to the host, up fe:|&::bggjHii logical level. In this manner, a host can believe that it has a large fixed storage space, create its file systems there, and so on. Disks can be added qf removed from the file system :by InServ without the file systems noticing the change, This feature can reduce the number of drives needed by hosts, or at least delay the purchase of disks until they are really needed.

12.7.6 Problems with RAID

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

The Solaris ZFS file system takes an innovative approach to solving these problems. It maintains internal checksums of all blocks, including data and metadata. Added functionality comes in the placement of the checksums. They are not kept with the block that is being checksummed. Rather, they are stored with the pointer to that block. Consider an inode with pointers to its data. Within the inode is the checksum of each block of data. If there is a problem with the data, the checksum will be incorrect, and the file system will know-about it. If the data are mirrored, and there is a block with a correct checksum and one with an incorrect checksum, ZFS will automatically update the bad block with the good one. Likewise, the directory entry that points to the inode has a checksum for the inode. Any problem in the mode is detected when the directory is accessed. This checksumming takes places throughout all ZFS structures, providing a much higher level of consistency, error detection, and error correction than is found in RAID disk sets or standard file systems. The extra overhead that is created by the checksum calculation and extra block read-modify-write cycles is not noticeable because the overall performance of ZFS is very fast.

12.8 Stable-Storage Implementation

In Chapter 6, we introduced the write-ahead log, which requires the availability of stable storage. By definition, information residing in stable storage is *never* lost. To implement such storage, we need to replicate the needed information on multiple storage devices (usually disks) with independent failure modes. We need to coordinate the writing of updates in a way that guarantees that a failure during an update will not leave all the copies in a damaged state and that, when we are recovering from a failure, we can force all copies to a consistent and correct value, even if another failure occurs during the recovery. In this section, we discuss how to meet these needs.

A disk write results in one of three outcomes:

1. **Successful completion.** The data were written correctly on disk.

2. Partial failure. A failure occurred in the midst of transfer, so only some of the sectors were written with the new data, and the sector being written during the failure may have been corrupted.

3. Total **failure.** The failure occurred before the disk write started, so the previous data values on the disk remain intact.

Whenever a failure occurs during writing of a block, the system needs to detect it and invoke a recovery procedure to restore the block to a consistent state. To do that, the system must maintain two physical blocks for each logical block. An output operation is executed as follows:

1. Write the information onto the first physical block.
2. When the first write completes successfully, write the same information onto the second physical block,
3. Declare the operation complete only after the second write completes successfully.

During recovery from a failure, each pair of physical blocks is examined. If both are the same and no detectable error exists, then no further action is necessary. If one block contains a detectable error, then we replace its contents with the value of the other block. If neither block contains a detectable error, but the blocks differ in content, then we replace the content of the first block with that of the second. This recovery procedure ensures that a write to stable storage either succeeds completely or results in no change.

We can extend this procedure easily to allow the use of an arbitrarily large number of copies of each block of stable storage. Although having a large number of copies further reduces the probability of a failure, it is usually reasonable to simulate stable storage with only two copies. The data in stable storage are guaranteed to be safe unless a failure destroys all the copies.

Because waiting for disk writes to complete (synchronous I/O) is time consuming, many storage arrays add NVRAM as a cache. Since the memory is nonvolatile (usually it has battery power as a backup to the unit's power), it can be trusted to store the data en route to the disks. It is thus considered part of the stable storage. Writes to it are much faster than to disk, so performance is greatly improved.

12.9 Tertiary-Storage Structure

Would you buy a VCR that had inside it only one tape that you could not take out or replace? Or a DVD or CD player that had one disk sealed inside? Of course not. You expect to use a VCR or CD player with many relatively inexpensive tapes or disks. On a computer as well, using many inexpensive cartridges with one drive lowers the overall cost. Low cost is the defining characteristic of tertiary storage, which we discuss in this section.

12.9.1 Tertiary-Storage Devices

Because costis so important, in practice, tertiary storage is built with removable media. The most common examples are floppy disks, tapes, and read-only, write-once, and rewritable CDs and DVDs. Many any other kinds of tertiary-storage devices are available as well, including removable devices that store data in flash memory and interact with the computer system via a USB interface.

12.9.1.1 Removable Disks

Removable disks are one kind of tertiary storage. Floppy disks are an example of removable magnetic disks. They are made from a thin, flexible disk coated with magnetic material and enclosed in a protective plastic case. Although common floppy disks can hold only about 1 MB, similar technology is used for removable magnetic disks that hold more than 1 GB. Removable magnetic disks can be nearly as fast as hard disks, although the recording surface is at greater risk of damage from scratches.

A **magneto-optic** disk is another kind of removable disk. It records data on a rigid platter coated with magnetic material, but the recording technology is quite different from that for a magnetic disk. The magneto-optic head flies much farther from the disk surface than a magnetic disk head does, and the magnetic material is covered with a thick protective layer of plastic or glass. This arrangement makes the disk much more resistant to head crashes.

The drive has a coil that produces a magnetic field; at room temperature, the field is too large and too weak to magnetize a bit on the disk. To write a bit, the disk head flashes a laser beam at the disk surface. The laser is aimed at a tiny spot where a bit is to be written. The laser heats this spot, which makes the spot susceptible to the magnetic field. Now the large, weak magnetic field can record a tiny bit.

The magneto-optic head is too far from the disk surface to read the data by detecting the tiny magnetic fields in the way that the head of a hard disk does. Instead, the drive reads a bit using a property of laser light called the **Kerr effect.** When a laser beam is bounced off of a magnetic spot, the polarization of the laser beam is rotated clockwise or counterclockwise, depending on the orientation of the magnetic field. This rotation is what the head detects to read a bit.

Another category of removable disk is the **optical disk.** Optical disks do not use magnetism at all. Instead, they use special materials that can be altered by laser light to have relatively dark or bright spots. One example of optical-disk technology is the **phase-change disk,** which is is coated with a material that can freeze into either a crystalline or an amorphous state. The crystalline state is more transparent, and hence a laser beam is brighter when it passes through the material and bounces off the reflective layer. The phase-change drive uses laser light at three different powers: low power to read data, medium power to erase the disk by melting and refreezing the recording medium into the crystalline state, and high power to melt the medium into the amorphous state to write to the disk. The most common examples of this technology are the re-recordable CD-RW and DVD-RW.

The kinds of disks just described can be used over and over. They are called **read-write disks.** In contrast, **write-once, read-many-times (WORM) disks** can be written only once. An old way to make a WORM disk is to manufacture a thin aluminum film sandwiched between two glass or plastic platters. To write a bit, the drive uses a laser light to burn a small hole through the aluminum,. This burning cannot be reversed. Although it is possible to destroy the information on a WORM disk by burning holes everywhere, it is virtually impossible to alter data on the disk, because holes can only be added, and the ECC code associated with each sector is likely to detect such additions. WORM disks are considered durable and reliable because the metal layer is safely encapsulated between the protective glass or plastic platters and magnetic fields cannot damage the recording. A newer write-once technology records on an organic polymer dye instead of an aluminum layer; the dye absorbs laser light to form marks. This technology is used in the recordable CD-R and DVD-R.

Read-only disks, such as CD-ROM and DVD-ROM, come from the factory with the data prerecorded. They use technology similar to that of WORM disks (although the bits are pressed, not burned), and they are very durable.

Most removable disks are slower than their nonremovable counterparts. The writing process is slower, as are rotation and sometimes seek time.

12.9.1.2 Tapes

Magnetic tape is another type of removable medium. As a general rule, a tape holds more data than an optical or magnetic disk cartridge. Tape drives and disk drives have similar transfer rates. But random access to tape is much slower than a disk seek, because it requires a fast-forward or rewind operation that takes tens of seconds or even minutes.

Although a typical tape drive is more expensive than a typical disk drive, the price of a tape cartridge is lower than the price of the equivalent capacity of magnetic disks. So tape is an economical medium for purposes that do not require fast random access. Tapes are commonly used to hold backup copies of disk data. They are also used in large supercomputer centers to hold the enormous volumes of data used in scientific research and by large commercial enterprises.

Large tape installations typically use robotic tape changers that move tapes between tape drives and storage slots in a tape library. These mechanisms give the computer automated access to many tape cartridges.

A robotic tape library can lower the overall cost of data storage. A disk-resident file that will not be neecied for a while can be **archived** to tape, where the cost per gigabyte is lower; if the file is needed in the future, the computer can **stage** it back into disk storage for active use. A robotic tape library is sometimes called **near-line** storage, since it is between the high performance of on-line magnetic disks and the low cost of off-line tapes sitting on shelves in a storage room.

12.9.1.3 Future Technology

In the future, other storage technologies may become important. One promising storage technology, **holographic storage,** uses laser light to record holographic photographs on special media. We can think of a hologram as a three-dimensional array of pixels. Each pixel represents one bit: 0 for black or 1 for white. And all the pixels in a hologram are transferred in one flash of laser light, so the data transfer rate is extremely high. With continued development, holographic storage may become commercially viable.

Another storage technology under active research is based on **micro-electronic mechanical systems** (MEMS). The idea is to apply the fabrication technologies that produce electronic chips to the manufacture of small data-storage machines. One proposal calls for the fabrication of an array of 10,000 tiny disk heads, with a square centimeter of magnetic storage material suspended above the array. When the storage material is moved lengthwise over the heads, each head accesses its own linear track of data on the material. The storage material can be shifted sideways slightly to enable all the heads to access their next track. Although it remains to be seen whether this technology can be successful, it may provide a nonvolatile data-storage technology that is faster than magnetic disk and cheaper than semiconductor DRAM.

Whether the storage medium is a removable magnetic disk, a DVD, or a magnetic tape, the operating system needs to provide several capabilities to use removable media for data storage. These capabilities are discussed in Section 12.9.2.

12.9.2 Operating-System Support

Two major jobs of an operating system are to manage physical devices and to present a virtual machine abstraction to applications. In this chapter, we have seen that, for hard disks, the operating system provides two abstractions. One is the raw device, which is just an array of data blocks. The other is a file system. For a file system on a magnetic disk, the operating system queues and schedules the interleaved requests from several applications. Now, we shall see how the operating system does its job when the storage media are removable.

12.9.2.1 Application Interface

Most operating systems can handle removable disks almost exactly as they do fixed disks. When a blank cartridge is inserted into the drive (or mounted), the cartridge must be formatted, and then an empty file system is generated on the disk. This file system is used just like a file system on a hard disk.

Tapes are often handled differently. The operating system usually presents a tape as a raw storage medium. An application does not open a file on the tape; it opens the whole tape drive as a raw device. Usually, the tape drive then is reserved for the exclusive use of that application until the application exits or closes the tape device. This exclusivity makes sense, because random access on a tape can take tens of seconds, or even a few minutes, so interleaving random accesses to tapes from more than one application would be likely to cause thrashing.

When the tape drive is presented as a raw device, the operating system does not provide file-system services. The application must decide how to use the array of blocks. For instance, a program that backs up a hard, disk to tape might store a list of file names and sizes at the beginning of the tape and then copy the data of the files to the tape in that order.

It is easy to see the problems that can arise from this way of using tape. Since every application makes up its own. rules for how to organize a tape, a tape full of data can generally be used by only the program that created it. For instance, even if we know that a backup tape contains a list of file names and file sizes followed by the file data in that order, we still would find it difficult to use the tape. How exactly are the file names stored? Are the file sizes in binary or in ASCII? Are the files written one per block, or are they all concatenated together in one tremendously long string of bytes? We do not even know the block size on the tape, because this variable is generally one that can be chosen separately for each block written.

For a disk drive, the basic operations are `readO`, `write()`, and `seek()`. Tape drives have a different set of basic operations. Instead of `seek()`, a tape drive uses the `locate()` operation. The tape `locate0` operation is more precise than the disk `seek()` operation, because it positions the tape to a specific logical block, rather than an entire track. Locating to block 0 is the same as rewinding the tape.

For most kinds of tape drives, it is possible to locate to any block that has been written on a tape. In a partly filled tape, however, it is not possible to locate into the empty space beyond the written area, because most tape drives do not manage their physical space in the same way disk drives do. For a disk drive, the sectors have a fixed size, and the formatting process must be used to place empty sectors in their final positions before any data can be written. Most tape drives have a variable block size, and the size of each block is determined on the fly when that block is written. If an area of defective tape is encountered during writing, the bad area is skipped and the block is written again. This operation explains why it is not possible to locate into the empty space beyond the written area—the positions and numbers of the logical blocks have not yet been determined.

Most tape drives have a `read_position()` operation that returns the logical block number where the tape head is. Many tape drives also support a `space ()` operation for relative motion. So, for example, the operation `space (-2)` would locate backward over two logical blocks.

For most kinds of tape drives, writing a block has the side effect of logically erasing everything beyond the position of the write. In practice, this side effect means that most tape drives are append-only devices, because updating a block in the middle of the tape also effectively erases everything beyond that block. The tape drive implements this appending by placing an end-of-tape (EOT) mark after a block that is written. The drive refuses to locate past the EOT mark, but it is possible to locate to the EOT and then start writing. Doing so overwrites the old EOT mark and places a new one at the end of the new blocks just written.

In principle, a file system can be implemented on a tape. But many of the file-system data structures and algorithms would be different from those used for disks, because of the append-only property of tape.

12.9.2.2 File Naming

Another question that the operating system needs to handle is how to name files on removable media. For a fixed disk, naming is not difficult. On a PC, the file name consists of a drive letter followed by a path name. In UNIX, the file name does not contain a drive letter, but the mount table enables the operating system to discover on what drive the file is located. If the disk is removable, however, knowing what drive contained the cartridge at some time in the past does not mean knowing how to find the file. If every removable cartridge in the world had a different serial number, the name of a file on a removable device could be prefixed with the serial number, but to ensure that no two serial numbers are the same would require each one to be about 12 digits in length. Who could remember the names of her files if she had to memorize a 12-digit serial number for each one?

The problem becomes even more difficult when we want to write data on a removable cartridge on one computer and then use the cartridge in another computer. If both machines are of the same type and have the same kind of removable drive, the only difficulty is knowing the contents and data layout on the cartridge. But if the machines or drives are different, many additional problems can arise. Even if the drives are compatible, different computers may store bytes in different orders and may use different encodings for binary numbers and even for letters (such as ASCII on PCs versus EBCDIC on mainframes).

Today's operating systems generally leave the name-space problem unsolved for removable media and depend on applications and users to figure out how to access and interpret the data. Fortunately, a few kinds of removable media are so well standardized that all computers use them the same way. One example is the CD. Music CDs use a universal format that is understood by any CD drive. Data CDs are available in only a few different formats, so it is usual for a CD drive and the operating-system device driver to be programmed to handle all the common formats. DVD formats are also well standardized.

12.9.2.3 Hierarchical Storage Management

A **robotic jukebox** enables the computer to change the removable cartridge in a tape or disk drive without human assistance. Two major uses of this technology are for backups and hierarchical storage systems. The use of a jukebox for backups is simple: When one cartridge becomes full, the computer instructs the jukebox to switch to the next cartridge. Some jukeboxes hold tens of drives and thousands of cartridges, with robotic arms managing the movement of tapes to the drives.

A hierarchical storage system extends the storage hierarchy beyond primary memory and secondary storage (that is, magnetic disk) to incorporate tertiary storage. Tertiary storage is usually implemented as a jukebox of tapes or removable disks. This level of the storage hierarchy is larger, cheaper, and slower.

Although the virtual memory system can be extended in a straightforward manner to tertiary storage, this extension is rarely carried out in practice. The reason is that a retrieval from a jukebox can take tens of seconds or even minutes, and such a long delay is intolerable for demand paging and for other forms of virtual memory use.

The usual way to incorporate tertiary storage is to extend the file system. Small and frequently used files remain on magnetic disk, while large and old files that are not actively used are archived to the jukebox. In some file-archiving systems, the directory entry for the file continues to exist, but the contents of the file no longer occupy space in secondary storage. If an application tries to open the file, the `open()` system call is suspended until the file contents can be staged in from tertiary storage. When the contents are again available from magnetic disk, the open() operation returns control to the application, which proceeds to use the disk-resident copy of the data.

Today, **hierarchical storage management** (HSM) is usually found in installations that have large volumes of data that are used seldom, sporadically, or periodically. Current work in HSM includes extending it to provide full **information life-cycle management** (ILM). Here, data move from disk to tape and back to disk, as needed, but are deleted on a schedule or according to policy. For example, some sites save e-mail for seven years but want to be sure that at the end oi seven years it is destroyed. At that point, the data could be on disk, HSM tape, and backup tape. ILM centralizes knowledge of where the data are so that policies can be applied across all these locations.

12.9.3 Performance Issues

As with any component of the operating system, the three most important aspects of tertiary-storage performance are speed, reliability, and cost.

12.9.3.1 Speed

The speed of tertiary storage has two aspects: bandwidth and latency. We measure the bandwidth in bytes per second. The sustained bandwidth is the average data rate during a large transfer—that is, the number of bytes divided by the transfer time. The effective bandwidth calculates the average over the entire I/O time, including the time for `seek()` or `locate()` and any cartridge-switching time in a jukebox. In essence, the sustained bandwidth is the data rate when the data stream is actually flowing, and the effective bandwidth is the overall data rate provided by the drive. The *bandwidth of a drive* is generally understood to mean the sustained bandwidth.

For removable disks, the bandwidth ranges from a few megabytes per second for the slowest to over 40 MB per second for the fastest. Tapes have a similar range of bandwidths, from a few megabytes per second to over 30 MB per second.

The second aspect of speed is the access latency. By this performance measure, disks are much faster than tapes: Disk storage is essentially two-dimensional—all the bits are out in the open. A disk access simply moves the arm to the selected cylinder and waits for the rotational latency, which may take less than 5 milliseconds. By contrast, tape storage is three-dimensional. At any time, a small portion of the tape is accessible to the head, whereas most of the bits are buried below hundreds or thousands of layers of tape wound on the reel. A random access on tape requires winding the tape reels until the selected block reaches the tape head, which can take tens or hundreds of seconds. So we can generally say that random access within a tape cartridge is more than a thousand times slower than random access on disk.

If a jukebox is involved, the access latency can be significantly higher. For a removable disk to be changed, the drive must stop spinning, then the robotic arm must switch the disk cartridges, and then the drive must spin up the new cartridge. This operation takes several seconds—about a hundred times longer than the random-access time within one disk. So switching disks in a jukebox incurs a relatively high performance penalty.

For tapes, the robotic-arm time is about the same as for disk. But for tapes to be switched, the old tape generally must rewind before it can be ejected, and that operation can take as long as 4 minutes. And, after a new tape is loaded into the drive, many seconds can be required for the drive to calibrate itself to the tape and to prepare for I/O. Although a slow tape jukebox can have a tape-switch time of 1 or 2 minutes, this time is not enormously greater than the random-access time within one tape.

So, to generalize, we say that random access in a disk jukebox has a latency of tens of seconds, whereas random access in a tape jukebox has a latency of hundreds of seconds; switching tapes is expensive, but switching disks is not. Be careful not to overgeneralize, though: Some expensive tape jukeboxes can rewind, eject, load a new tape, and fast-forward to a random item of data all in less than 30 seconds.

If we pay attention to only the performance of the drives in a jukebox, the bandwidth and latency seem reasonable. But if we focus our attention on the cartridges instead, we find a terrible bottleneck. Consider first the bandwidth. The bandwidth-to-storage-capacity ratio of a robotic library is much less favorable than that of a fixed disk. To read all the data stored on a large hard disk could take about an hour. To read all the data stored in a large tape library could take years. The situation with respect to access latency is nearly as bad. To illustrate this, if 100 requests are queued for a disk drive, the average waiting time will be about a second. If 100 requests are queued for a tape library, the average waiting time could be over an hour. The low-cost of tertiary storage results from having many cheap cartridges share a few expensive drives. But a removable library is best devoted to the storage of infrequently used data, because the library can satisfy only a relatively small number of I/O requests per hour.

12.9.3.2 Reliability

Although we often think *good performance* means *high speed,* another important aspect of performance is *reliability.* If we try to read some data and are unable to do so because of a drive or media failure, for all practical purposes the access time is infinitely long and the bandwidth is infinitely small. So it is important to understand the reliability of removable media.

Removable magnetic disks are somewhat less reliable than are fixed hard disks because the cartridge is more likely to be exposed to harmful environmental conditions such as dust, large changes in temperature and humidity, and mechanical forces such as shock and bending. Optical disks are considered very reliable, because the layer that stores the bits is protected by a transparent plastic or glass layer. The reliability of magnetic tape varies widely, depending on the kind of drive. Some inexpensive drives wear out tapes after a few dozen uses; other kinds are gentle enough to allow millions of reuses. By comparison with a magnetic-disk head, the head in a magnetic-tape drive is a weak spot. A disk head flies above the media, but a tape head is in close contact with the tape. The scrubbing action of the tape can wear out the head after a few thousands or tens of thousands of hours.

In summary, we say that a fixed disk drive is likely to be more reliable than a removable disk or tape drive, and an optical disk is likely to be more reliable than a magnetic disk or tape. But a fixed magnetic disk has one weakness. A head crash in a hard disk generally destroys the data, whereas the failure of a tape drive or optical disk drive often leaves the data cartridge unharmed.

12.9.3.3 Cost

Storage cost is another important factor. Here is a concrete example of how removable media may lower the overall storage cost. Suppose that a hard disk that holds *X* GB has a price of \$200; of this amount, \$190 is for the housing, motor, and controller, and \$10 is for the magnetic platters. The storage cost for this disk is \$200/ X per gigabyte. Now, suppose that we can manufacture the platters in a removable cartridge. For one drive and 10 cartridges, the total price is \$190 + \$100, and the capacity is 10X GB, so the storage cost is \$29/ *X* per gigabyte. Even if it is a little more expensive to make a removable cartridge,

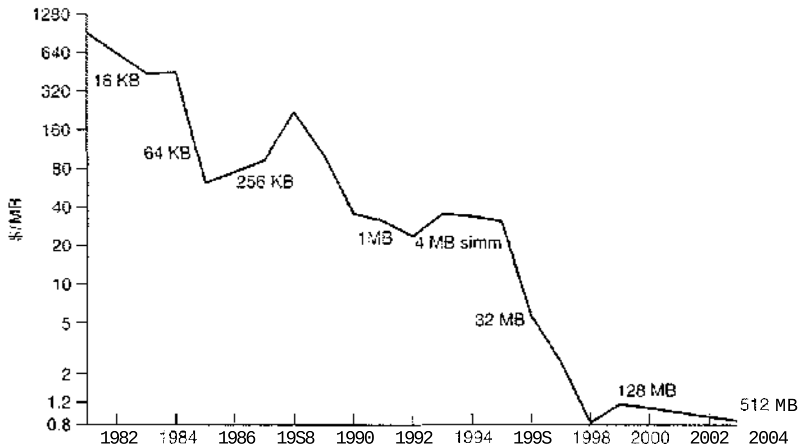


Figure 12.13 Price per megabyte of DRAM, from 1981 to 2004.

the cost per gigabyte of removable storage may well be lower than the cost per gigabyte of a hard disk, because the expense of one drive is averaged with the low price of many removable cartridges.

Figures 12.13, 12.14, and 12.15 show the cost trends per megabyte for DRAM memory, magnetic hard disks, and tape drives. The prices in the graphs are the lowest prices found in advertisements in various computer magazines and on the World Wide Web at the end of each year. These prices reflect the small-computer marketplace of the readership of these magazines, where prices are low by comparison with the mainframe and minicomputer markets. In the case of tape, the price is for a drive with one tape. The overall cost of tape storage becomes much lower as more tapes are purchased for use with the drive, because the price of a tape is a small fraction of the price of the drive. However, in a huge tape library containing thousands of cartridges, the storage

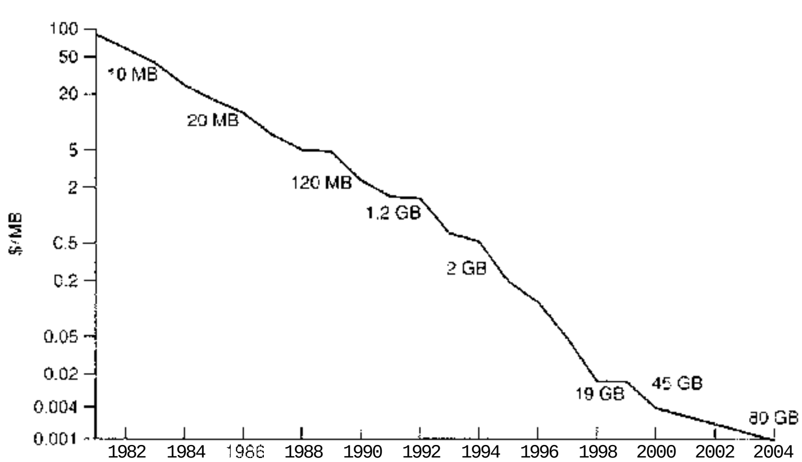


Figure 12.14 Price per megabyte of magnetic hard disk, from 1981 to 2004.

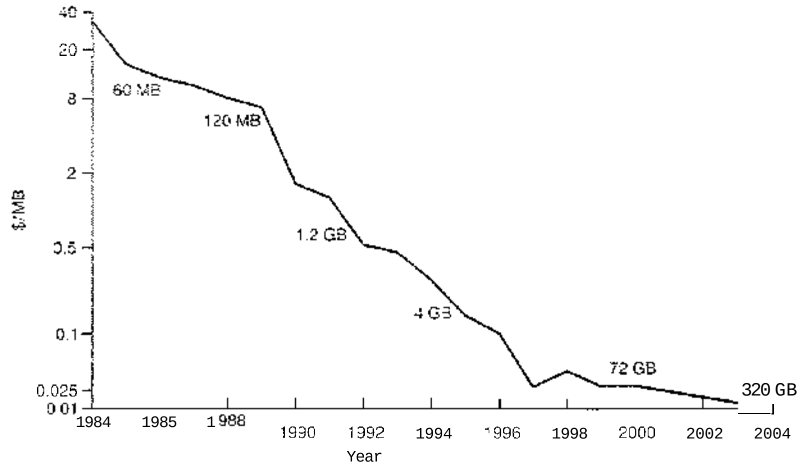


Figure 12.15 Price per megabyte of a tape drive, from 1984 to 2004.

cost is dominated by the cost of the tape cartridges. As of this writing in 2004, the cost per GB of tape cartridges can be approximated as somewhat less than $2.

The cost of DRAM fluctuates widely. In the period from 1981 to 2004, we can see three price crashes (around 1981, 1989, and 1996) as excess production caused a glut in the marketplace. We can also see two periods (around 1987 and 1993) where shortages in the marketplace caused significant price increases. In the case of hard disks, the price decline has been much steadier, although it appears to have accelerated since 1992. Tape-drive prices also fell steadily up to 1997. Since 1997, the price per gigabyte of inexpensive tape drives has ceased its dramatic fall, although the price of mid-range tape technology (such as DAT/DDS) has continued to fall and is now approaching that of the inexpensive drives. Tape-drive prices are not shown prior to 1984, because, as mentioned, the magazines used in tracking prices are targeted to the small-computer marketplace, and tape drives were not widely used with small computers prior to 1984.

We can see from these graphs that the cost of storage has fallen dramatically over the past twenty years or so. By comparing the graphs, we can also see that the price of disk storage has plummeted relative to the price of DRAM and tape.

The price per megabyte of magnetic disk has improved by more than four orders of magnitude during the past two decades, whereas the corresponding improvement for main memory has been only three orders of magnitude. Main memory today is more expensive than disk storage by a factor of 100.

The price per megabyte has dropped much more rapidly for disk drives than for tape drives as well. In fact, the price per megabyte of a magnetic disk drive is approaching that of a tape cartridge without the tape drive. Consequently, small- and medium-sized tape libraries have a higher storage cost than disk systems with equivalent capacity.

The dramatic fall in disk prices has largely rendered tertiary storage obsolete: We no longer have any tertiary storage technology that is orders of magnitude less expensive than magnetic disk. It appears that the revival of tertiary storage must await a revolutionary technology breakthrough. Meanwhile, tape storage will find its use mostly limited to purposes such as backups of disk drives and archival storage in enormous tape libraries that greatly exceed the practical storage capacity of large disk farms.

12.10 Summary

Disk drives are the major secondary-storage I/O devices on most computers. Most secondary storage devices are either magnetic disks or magnetic tapes. Modern disk drives are structured as a large one-dimensional array of logical disk blocks which is usually 512 bytes.

Disks may be attached to a computer system in one of two ways: (1) using the local I/O ports on the host computer or (2) using a network connection such as storage area networks.

Requests for disk I/O are generated by the file system and by the virtual memory system. Each request specifies the address on the disk to be referenced, in the form of a logical block number. Disk-scheduling algorithms can improve the effective bandwidth, the average response time, and the variance in response time. Algorithms such as SSTF, SCAN, C-SCAN, LOOK, and C-LOOK are designed to make such improvements through strategies for disk-queue ordering.

Performance can be harmed by external fragmentation. Some systems have utilities that scan the file system to identify fragmented files; they then move blocks around to decrease the fragmentation. Defragmenting a badly fragmented file system can significantly improve performance, but the system may have reduced performance while the defragmentation is in progress. Sophisticated file systems, such as the UNIX Fast File System, incorporate many strategies to control fragmentation during space allocation so that disk reorganization is not needed.

The operating system manages the disk blocks. First, a disk must be low-level-formatted to create the sectors on the raw hardware—new disks usually come preformatted. Then, the disk is partitioned, file systems are created, and boot blocks are allocated to store the system's bootstrap program. Finally, when a block is corrupted, the system must have a way to lock out that block or to replace it logically with a spare.

Because an efficient swap space is a key to good performance, systems usually bypass the file system and use raw disk access for paging I/O. Some systems dedicate a raw disk partition to swap space, and others use a file within the file system instead. Still other systems allow the user or system administrator to make the decision by providing both options.

Because of the amount of storage required on large systems, disks are frequently made redundant via RAID algorithms. These algorithms allow more than one disk to be used for a given operation and allow continued operation and even automatic recovery in the face of a disk failure. RAID algorithms are organized into different levels; each level provides some combination of reliability and high transfer rates.

The write-ahead log scheme requires the availability of stable storage. To implement such storage, we need to replicate the needed information on multiple nonvolatile storage devices (usually disks) with independent failure modes. We also need to update the information in a controlled manner to ensure that we can recover the stable data after any failure during data transfer or recovery.

Tertiary storage is built from disk and tape drives that use removable media. Many different technologies are available, including magnetic tape, removable magnetic and magneto-optic disks, and optical disks.

For removable disks, the operating system generally provides the full services of a file-system interface, including space management and request-queue scheduling. For many operating systems, the name of a file on a removable cartridge is a combination of a drive name and a file name within that drive. This convention is simpler but potentially more confusing than is using a name that identifies a specific cartridge.

For tapes, the operating system generally just provides a raw interface. Many operating systems have no built-in support for jukeboxes. Jukebox support can be provided by a device driver or by a privileged application designed for backups or for HSM.

Three important aspects of performance are bandwidth, latency, and reliability. Many bandwidths are available for both disks and tapes, but the random-access latency for a tape is generally much greater than that for a disk. Switching cartridges in a jukebox is also relatively slow. Because a jukebox has a low ratio of drives to cartridges, reading a large fraction of the data in a jukebox can take a long time. Optical media, which protect the sensitive layer with a transparent coating, are generally more robust than magnetic media, which are more likely to expose the magnetic material to physical damage.

Exercises

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

a. Explain why this assertion is true.

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

c. Explain why fairness is an important goal in a time-sharing system.

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

12.2 Suppose that a disk drive has 5,000 cylinders, numbered 0 to 4999. The drive is currently serving a request at cylinder 143, and the previous request was at cylinder 125. The queue of pending requests, in FIFO order, is:

86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130

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

a. FCFS

b. SSTF

c. SCAN

d. LOOK

e. C-SCAN

f. C-LOOK

12.3 Elementary physics states that when an object is subjected to a constant acceleration a, the relationship between distance d and time t is given by $d = \frac{1}{2}at^2$. Suppose that, during a seek, the disk in Exercise 12.2 accelerates the disk arm at a constant rate for the first half of the seek, then decelerates the disk arm at the same rate for the second half of the seek. Assume that the disk can perform a seek to an adjacent cylinder in 1 millisecond and a full-stroke seek over all 5,000 cylinders in 18 milliseconds.

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

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

c. Calculate the total seek time for each of the schedules in Exercise 12.2. Determine which schedule is the fastest (has the smallest total seek time).

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

12.4 Suppose that the disk in Exercise 12.3 rotates at 7,200 RPM.

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

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

12.5 Write a Java program for disk scheduling using the SCAN and C-SCAN disk-scheduling algorithms.

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

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

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

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

c. File systems typically find data blocks via an indirection table, such as a FAT in DOS or inodes in UNIX. Describe one or more ways to take advantage of this indirection to improve disk performance.

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

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

a. A write of one block of data

b. A write of seven continuous blocks of data

12.10 Compare the throughput achieved by a RAID Level 5 organization with that achieved by a RAID Level 1 organization for the following:

a. Read operations on single blocks

b. Read operations on multiple contiguous blocks

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

12.12 Assume that you have a mixed configuration comprising disks organized as RAID Level 1 and as RAID Level 5 disks. Assume that the system has flexibility in deciding which disk organization to use for storing a particular file. Which files should be stored in the RAID Level 1 disks and which in the RAID Level 5 disks in order to optimize performance?

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

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

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

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

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

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

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

12.17 The operating system generally treats removable disks as shared file systems but assigns a tape drive to only one application at a time. Give three reasons that could explain this difference in treatment of disks and tapes. Describe the additional features that an operating system would need to support shared file-system access to a tape jukebox. Would the applications sharing the tape jukebox need any special properties, or could they use the files as though the files were disk-resident? Explain your answer.

12.18 What would be the effects on cost and performance if tape storage had the same areal density as disk storage? (**Areal density** is the number of gigabits per square inch.)

12.19 You can use simple estimates to compare the cost and performance of a terabyte storage system made entirely from disks with one that incorporates tertiary storage. Suppose that magnetic disks each hold 10 GB, cost $1,000, transfer 5 MB per second, and have an average access latency of 15 milliseconds. Suppose that a tape library costs $10 per gigabyte, transfers 10 MB per second, and has an average access latency of 20 seconds. Compute the total cost, the maximum total data rate, and the average waiting time for a pure disk system. If you make any assumptions about the workload, describe and justify them. Now, suppose that 5 percent of the data are frequently used, so they must reside on disk, but the other 95 percent are archived in the tape library. Further suppose that the disk system handles 95 percent of the requests and the library handles the other 5 percent. What are the total cost, the maximum total data rate, and the average waiting time for this hierarchical storage system?

12.20 Imagine that a holographic storage drive has been invented. Suppose that the holographic drive costs $10,000 and has an average access time of 40 milliseconds. Suppose that it uses a $100 cartridge the size of a CD. This cartridge holds 40,000 images, and each image is a square black-and-white picture with a resolution of 6,000 x 6,000 pixels (each pixel stores 1 bit). Suppose that the drive can read or write one picture in 1 millisecond. Answer the following questions.

a. What would be some good uses for this device?

b. How would this device affect the I/O performance of a computing system?

c. Which other kinds of storage devices, if any, would become obsolete as a result of the invention of this device?

12.21 Suppose that a one-sided 5.25-inch optical-disk cartridge has an areal density of 1 gigabit per square inch. Suppose that a magnetic tape has an areal density of 20 megabits per square inch and is 1/2 inch wide and 1,800 feet long. Calculate an estimate of the storage capacities of these two kinds of storage cartridges. Suppose that an optical tape exists that has the same physical size as the tape but the same storage density as the optical disk. What volume of data could the optical tape hold? What would be a marketable price for the optical tape if the magnetic tape cost $25?

12.22 Discuss how an operating system could maintain a free-space list for a tape-resident file system. Assume that the tape technology is append-only and that it uses EOT marks and locate, space, and read position commands as described in Section 12.9.2.1.

Bibliographical Notes

Discussions of redundant arrays of independent disks (RAID) are presented by Patterson et al. [1988] and in the detailed survey of Chen et al. [1994]. Disk-system architectures for high-performance computing are discussed by Katz et al. [1989]. Enhancements to the RAID systems are discussed in Wilkes et al. [1996] and Yu et al. [2000]. Teorey and Pinkerton [1972] present an early comparative analysis of disk-scheduling algorithms. They use simulations that model a disk for which seek time is linear in the number of cylinders crossed. For this disk, LOOK is a good choice for queue lengths below 140, and C-LOOK is good for queue lengths above 100. King [1990] describes ways to improve the seek time by moving the disk arm when the disk is otherwise idle. Seltzer et al. [1990] and Jacobson and Wilkes [1991] describe disk-scheduling algorithms that consider rotational latency in addition to seek time. Scheduling optimizations that exploit disk idle times are discussed in Lumb et al. [2000]. Worthington et al. [1994] discuss disk performance and show the negligible performance impact of defect management. The placement of hot data to improve seek times has been considered by Ruemmler and Wilkes [1991] and Akyurek and Salem [1993]. Ruemmler and Wilkes [1994] describe an accurate performance model for a modern disk drive. Worthington et al. [1995] tell how to determine low-level disk properties such as the zone structure, and this work is further advanced by Schindler and Gregory [1999]. Disk power management issues are discussed in Douglis et al. [1994], Douglis et al. [1995], Greenawalt [1994], and Golding et al. [1995].

The I/O size and randomness of the workload has a considerable influence on disk performance. Ousterhout et al. [1985] and Ruemmler and Wilkes [1993] report numerous interesting workload characteristics, including that most files are small, most newly created files are deleted soon thereafter, most files that are opened for reading are read sequentially in their entirety, *and* most seeks are short. McKusick et al. [1984] describe the Berkeley Fast File System (FFS), which uses many sophisticated techniques to obtain good performance for a wide variety of workloads. McVoy and Kleiman [1991] discuss further improvements to the basic FFS. Quinlan [1991] describes how to implement a file system on WORM storage with a magnetic disk cache; Richards [1990] discusses a file-system approach to tertiary storage. Maher et al. [1994] give an overview of the integration of distributed file systems and tertiary storage.

The concept of a storage hierarchy has been studied for more than thirty years. For instance, a 1970 paper by Mattson et al. [1970] describes a mathematical approach to predicting the performance of a storage hierarchy. Alt [1993] describes the accommodation of removable storage in a commercial operating system, and Miller and Katz [1993] describe the characteristics of tertiary-storage access in a supercomputing environment. Benjamin [1990] gives an overview of the massive storage requirements for the EOSDIS project at NASA. Management and use of network-attached disks and programmable disks are discussed in Gibson et al. [1997b], Gibson et al. [1997a], Riedel et al. [1998], and Lee and Thekkath [1996].

Holographic storage technology is the subject of an article by Psaltis and Mok [1995]; a collection of papers on this topic dating from 1963 has been assembled by Sincerbox [1994]. Asthana and Finkelstein [1995] describe several emerging storage technologies, including holographic storage, optical tape, and electron trapping. Toigo [2000] gives an in-depth description of modern disk technology and several potential future storage technologies.

The Linux System

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 version of UNIX that has gained popularity in recent years. 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 internal methods by which Linux implements these interfaces. Linux is a rapidly evolving operating system. This chapter describes developments through the Linux 2.6 kernel, which was released in late 2003.

CHAPTER OBJECTIVES

- To explore the history of the UNIX operating system from which Linux is derived and the principles upon which Linux is designed.
- To examine the Linux process model and illustrate how Linux schedules processes and provides interprocess communication.
- To look at memory management in Linux.
- To explore how Linux implements file systems and manages I/O devices.

21.1 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 student, Linus Torvalds, wrote and christened **Linux,** 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 on the Internet. As a result, Linux's history has been one of collaboration by many users 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 much UNIX functionality.

In its early days, Linux development revolved largely around the central operating-system kernel—the core, privileged executive that manages all system resources and that interacts directly with the computer hardware. We need much more than this kernel to produce a full operating system, of course. It is useful to make the distinction between the Linux kernel and a Linux system. The **Linux kernel** is an entirely 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.

21.1.1 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. The only file system supported was the Minix file system —the first Linux kernels were cross-developed on a Minix platform. However, the kernel did implement proper UNIX processes with protected address spaces.

The next milestone version, 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 an Ethernet or (using 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 supported a range of SCSI controllers for high-performance disk access. The developers extended the virtual 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 also 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. Simple support for dynamically loadable and unloadable kernel modules was supplied as well.

At this point, development started on the 1.1 kernel stream, but numerous bug-fix patches were released subsequently against 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, 1.3, and 2.1, are **development kernels;** even-numbered minor-version numbers are stable **production kernels.** Updates against the stable kernels are intended only as remedial versions, whereas the development kernels may include newer and relatively untested functionality.

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 networking stack to provide support for the IPX protocol and made the IP implementation more complete by including accounting and firewalling functionality.

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 1.2 stable 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 had been deferred until after the stable 1.2 kernel had been released. As a result, the 1.3 development stream saw a great deal of new functionality added to the kernel.

This work was finally released as Linux 2.0 in June 1996. This release was given a major version-number increment on account of two major new capabilities: support for multiple architectures, including a fully 64-bit native Alpha port, and support for multiprocessor architectures. Linux distributions based on 2.0 are also available for the Motorola 68000-series processors and for Sun's SPARC systems. A derived version of Linux running on top of the Mach microkernel also runs on PC and PowerMac systems.

The changes in 2.0 did not stop there. 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 also were supported.

The 2.0 kernel also included much improved TCP/IP performance, and a number of new networking protocols were added, including AppleTalk, AX.25 amateur radio networking, and ISDN support. The ability to mount remote netware and SMB (Microsoft LanManager) network volumes was added.

Other major improvements in 2.0 were support for internal kernel threads, for handling dependencies between loadable modules, and for automatic loading of modules on demand. Dynamic configuration of the kernel at run time was much improved through a new, standardized configuration interface.

Additional new features included file-system quotas and POSIX-compatible real-time process-scheduling classes.

Improvements continued with the release of Linux 2.2 in January 1999. A port for UltraSPARC systems was added. Networking was enhanced with more flexible firewalling, better routing and traffic management, and support for TCP large window and selective acks. Acorn, Apple, and NT disks could now be read, and NFS was enhanced and a kernel-mode NFS daemon added. 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 include increased support for SMP systems, journaling file systems, and enhancements to the memory-management system. The process scheduler has been modified in version 2.6, providing an efficient 0(1) scheduling algorithm. In addition, the Linux 2.6 kernel is now preemptive, allowing a process to be preempted while running in kernel mode.

21.1.2 The Linux System

In many ways, the Linux kernel forms the core of the Linux project, but other components make up the 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 networking-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 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 keeping 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.

21.1.3 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 operation was often precisely what a Linux user had to carry out. As Linux has matured, however, various individuals and groups have attempted to make this job less painful by providing a standard, precompiled set 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; 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 **Caldera, Craftworks,** and **WorkGroup Solutions.** A large Linux following in Germany has resulted in several dedicated German-language distributions, including versions from **SuSE** and **Unifix**. There are too many Linux distributions in circulation for us to list all of them here. The variety of distributions does not prohibit compatibility across Linux distributions, 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.

21.1.4 Linux Licensing

The Linux kernel is distributed under 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, however, in the sense that people can copy it, modify it, use it in any manner they want, and give away their own copies, without any restrictions.

The main implications of Linux's licensing terms are that nobody using Linux, or creating her own derivative of Linux (a legitimate exercise), can make the derived product proprietary 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-only software distributions, as long as anybody who receives binaries is also given the opportunity to get source code, for a reasonable distribution charge.)

21.2 Design Principles

In its overall design, Linux resembles any other traditional, nonmicrokernel UNIX implementation. It is a multiuser, 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 implemented fully. 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 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 megabytes of main memory and many gigabytes of disk space, but it is still capable of operating usefully in under 4 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 of the 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 flavor 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 of 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 be compliant with the relevant POSIX documents; at least two Linux distributions have achieved official POSIX certification.

Because it presents 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 A.3) and user interface (Section A.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 are significantly different.

Many other standards exist in the UNIX world, but full certification of Linux against them is sometimes slowed because they are 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 if the implementation is not formally certified. 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.

21.2.1 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.
3. **System utilities.** The system utilities are programs that perform individual, specialized management tasks. Some system utilities may be invoked just once to initialize and configure some aspect of the system; others—known as *daemons* in UNIX terminology—may run permanently, handling such tasks as responding to incoming network connections, accepting logon requests from terminals, and updating log files.

Figure 21.1 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 with full access to all the physical resources of the computer. Linux refers to this privileged mode as **kernel mode.** Under Linux, no user-mode 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 instead.

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 to improve performance: Because all kernel code and data structures are kept in a single address space, no context switches are necessary when a process calls an operating-system function or when a hardware interrupt is delivered. Not

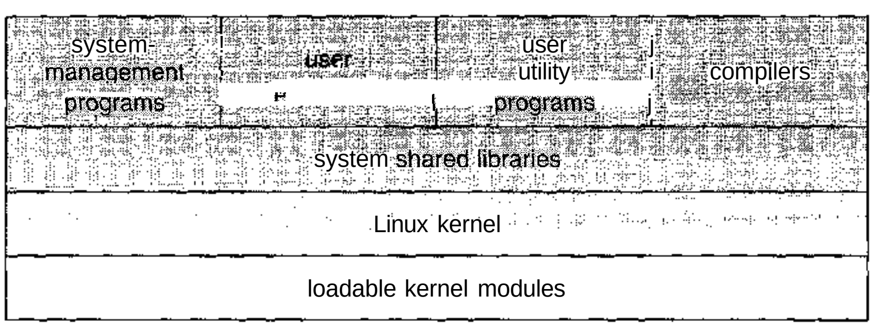


Figure 21.1 Components of the Linux system.

only the core scheduling and virtual memory code occupies this address space; *all* kernel code, including all device drivers, file systems, and networking code, is present in the same single address space.

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 necessarily 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 run processes, 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 looks nothing like a UNIX system. It is missing many of the extra features 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 kernel-system-service requests. 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 here 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 the system, such as those to configure network devices and to load kernel modules. Continually running server programs also count as system utilities; such programs handle user login requests, incoming network connections, and the printer queues.

Not all the standard utilities serve key system-administration functions. The UNIX user environment contains a large number of standard utilities to do simple everyday tasks, such as listing directories, moving and deleting files, and displaying the contents of a file. More complex utilities can perform text-processing functions, such as sorting textual data and performing pattern searches on input text. Together, these utilities form a standard tool set that users can expect on any UNIX system; although they do not perform any operating-system function, they are an important part of the basic Linux system.

21.3 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; typically, a module might 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 to load 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 CD-ROM driver might be loaded when a CD is mounted and unloaded from memory when the CD is dismounted from the file system.

The module support under Linux has three components:

1. The **module management** allows modules to be loaded into memory and to talk to the rest of the kernel.
2. The **driver registration** allows modules to tell the rest of the kernel that a new driver has become available.
3. A **conflict-resolution mechanism** allows different device drivers to reserve hardware resources and to protect those resources from accidental use by another driver.

21.3.1 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 exported explicitly by the kernel. 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 any 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 requestor. 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.

21.3.2 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 drivers and may register more than one driver if it wishes. For example, a device driver might want to register two separate mechanisms for accessing the device. Registration tables include 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 IPX, or simply a new set of packet-filtering rules for a network firewall.
- **Binary format.** This format specifies a way of recognizing, and loading, 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 21.7.4).

21.3.3 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. IBM PC hardware, however, comes in a vast number of configurations, with large numbers of possible drivers for devices such as network cards, SCSI controllers, 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—for example, as when both the parallel printer driver and the parallel-line IP (PLIP) network driver try to talk to the parallel printer 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 its initialization and request that it be unloaded if it cannot continue, or it may carry on, using alternative hardware resources.

21.4 Process Management

A process is the basic context within 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 from Section A.3.2 and introduce Linux's own threading model.

21.4.1 The fork() and exec() Process Model

The basic principle of UNIX process management is to separate two operations: the creation of a 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. A new process may be created with `fork()` without a new program being run—the new subprocess simply continues to execute exactly the same program that the first, parent process was running. Equally, running a new program does not require that a new process be created first: Any process may call exec 0 at any time. The currently running program is immediately terminated, and the new program starts executing in the context of the existing process.

This model has the advantage of great simplicity. Rather than having to specify every detail of the environment of a new program in the system call that runs that program, new programs simply run in their 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 program 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.

21.4.1.1 Process Identity

A process identity consists mainly of the following items:

- Process ID (PID). Each process has a unique identifier. PIDs are used to specify processes to the operating system when an application makes a system call to signal, modify, or wait for another 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 10.6.2) 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 modify slightly the semantics of certain system calls. Personalities are primarily used by emulation libraries to request that system calls be compatible with certain flavors of UNIX.

Most of these identifiers are under 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.

21.4.1.2 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 that its parent possesses. 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 in which language to display system messages for programs that include multilingual support.

The environment-variable mechanism custom tailors the operating system on a per-process basis, rather than for the system as a whole. Users can choose their own languages or select their own editors independently of one another.

21.4.1.3 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, so that 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 exclusively by kernel-mode code. Both system calls and interrupts that occur while the process is executing will use this stack.
- **Accounting.** The kernel maintains 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. When making file-I/O system calls, processes refer to files by their 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 current root and default directories to be used for new file searches are stored here.
- **Signal-handler table.** UNIX systems can deliver asynchronous signals to a process in response to various external events. The signal-handler table defines the routine in the process's address space to be called when specific signals arrive.
- **Virtual memory context.** The virtual memory context describes the full contents of a process's private address space; we discuss it in Section 21.6.

21.4.2 Processes and Threads

Linux provides the `fork()` system call with the traditional functionality of duplicating a process. Linux also provides the ability to create threads using the `clone()` system call. However, Linux does not distinguish between processes and threads. In fact, Linux generally uses the term *task*—rather than *process* or *thread*—when referring to a flow of control within a program. When `clone()` is invoked, it is passed a set of flags that determine how much sharing is to take place between the parent and child tasks. Some of these flags are listed below:

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	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. However, if none of these flags is set when `clone()` is invoked, no sharing takes place, resulting in functionality similar to 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 as appropriate.

The arguments to the `clone()` system call tell it which subcontexts to copy, and which to share, when it creates a new process. The new process always is given a new identity and a new scheduling context; according to the arguments passed, however, it may either create new subcontext data structures initialized 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.

21.5 Scheduling

Scheduling is the job of allocating CPU time to different tasks within an operating system. Normally, we think of scheduling as being the running and interrupting of processes, but another aspect of scheduling is also important to Linux: the running of the various kernel tasks. Kernel tasks encompass both tasks that are requested by a running process and tasks that execute internally on behalf of a device driver.

21.5.1 Process Scheduling

Linux has two separate process-scheduling algorithms. One is a time-sharing algorithm for fair, preemptive scheduling among multiple processes; 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 2.5 of the kernel. Prior to version 2.5, the Linux kernel ran a variation of the traditional UNIX scheduling algorithm. Among other issues, problems with the traditional UNIX scheduler are that it does not provide adequate support for SMP systems and that it does not scale well as the number of tasks on the system grows. The overhaul of the scheduler with version 2.5 of the kernel now provides a scheduling algorithm that runs in constant time—known as $O(1)$—regardless of the number of tasks on the system. The new scheduler also provides increased support for SMP, including

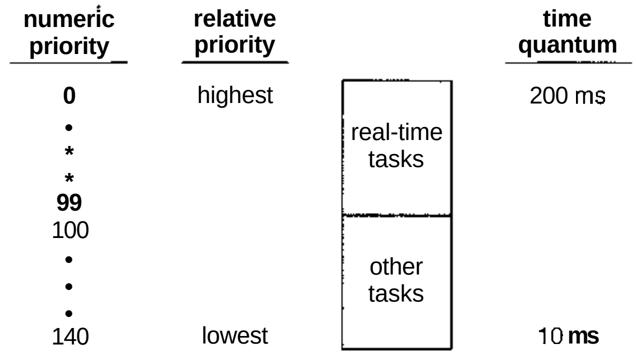


Figure 21.2 The relationship between priorities and time-slice length.

processor affinity and load balancing, as well as maintaining fairness and support for interactive tasks.

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 100 to 140. These two ranges map into a global priority scheme whereby numerically lower values indicate higher priorities.

Unlike schedulers for many other systems, Linux assigns higher-priority tasks longer time quanta and vice-versa. Because of the unique nature of the scheduler, this is appropriate for Linux, as we shall soon see. The relationship between priorities and time-slice length is shown in Figure 21.2.

A runnable task is considered eligible for execution on the CPU so long as it has time remaining in its time slice. When a task has exhausted its time slice, it is considered **expired** and is not eligible for execution again until all other tasks have also exhausted their time quanta. The kernel maintains a list of all runnable tasks in a **runqueue** data structure. Because of its support for SMP, each processor maintains its own runqueue and schedules itself independently. Each runqueue contains two priority arrays—**active** and **expired.** The active array contains all tasks with time remaining in their time slices, and the expired array contains all expired tasks. Each of these priority arrays includes a list of tasks indexed according to priority (Figure 21.3). The scheduler chooses the task with the highest priority from the active array for execution on the CPU. On multiprocessor machines, this means that each processor is scheduling the highest-priority task from its own runqueue structure. When all tasks have exhausted their time slices (that is, the active array is empty), the two priority arrays are exchanged as the expired array becomes the active array and vice-versa.

Tasks are assigned dynamic priorities that are based on the *nice* value plus or minus up to the value 5 based upon the interactivity of the task. Whether a value is added to or subtracted from a task's *nice* value depends on the interactivity of the task. A task's interactivity is determined by how long it has been sleeping while waiting for I/O. Tasks that are more interactive typically have longer sleep times and therefore are more likely to have an adjustment closer to -5, as the scheduler favors such interactive tasks. Conversely, tasks with shorter sleep times are often more CPU-bound and thus will have their priorities lowered.

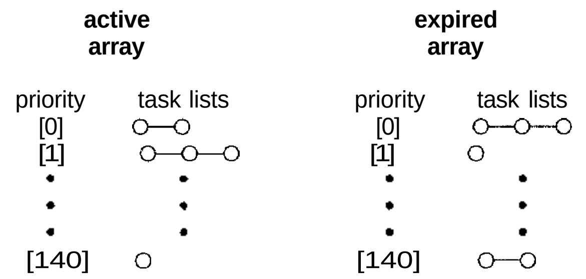


Figure 21.3 List of tasks indexed according to priority.

The recalculation of a task's dynamic priority occurs when the task has exhausted its time quantum and is to be moved to the expired array. Thus, when the two arrays are exchanged, all tasks in the new active array have been assigned new priorities and corresponding time slices.

Linux's real-time scheduling is simpler still. Linux implements the two real-time scheduling classes required by POSIX.lb: first-come, first-served (FCFS) and round-robin (Sections 5.3.1 and 5.3.4, respectively). In both cases, each process has a priority in addition to its scheduling class. Processes of different priorities can compete with one another to some extent in time-sharing scheduling; in real-time scheduling, however, the scheduler always runs the process with the highest priority. Among processes of equal priority, it runs the process that has been waiting longest. The only difference between FCFS and round-robin scheduling is that FCFS processes continue to run until they either exit or block, whereas a round-robin process will be preempted after a while and will be moved to the end of the scheduling queue, so round-robin processes of equal priority will automatically time-share among themselves. Unlike routine time-sharing tasks, real-time tasks are assigned static priorities.

Linux's real-time scheduling is soft—rather than hard—real time. The scheduler offers strict guarantees about the relative priorities of real-time processes, but the kernel does not offer any guarantees about how quickly a real-time process will be scheduled once that process becomes runnable.

21.5.2 Kernel Synchronization

The way the kernel schedules its own operations is fundamentally different from the way it schedules processes. 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 driver may deliver a hardware interrupt that causes the CPU to start executing a kernel-defined handler for that interrupt.

The problem posed to 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 that must not be allowed to execute concurrently. As a result, kernel synchronization involves much more than just process 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 process running in kernel mode could not be preempted—even if a higher-priority process became available to run. With version 2.6, the Linux kernel became fully preemptive; so a task can now 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; the kernel is designed so that the spinlock is held only for short durations. On single-processor machines, spinlocks are inappropriate for use and are replaced by enabling and disabling kernel preemption. That is, on single-processor machines, rather than holding a spinlock, the task disables 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 system calls—`preempt_disable()` and `preempt_enable()`—for disabling and enabling kernel preemption. However, in addition, the kernel is not preemptible if a kernel-mode task is holding a lock. 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. When a lock is acquired, `preempt_count` is incremented. Likewise, it is 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 enabling and disabling 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 that Linux uses 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 of shared data structures.

However, there is a penalty for disabling interrupts. On most hardware architectures, interrupt enable and disable instructions are expensive. Furthermore, 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; so performance degrades. 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

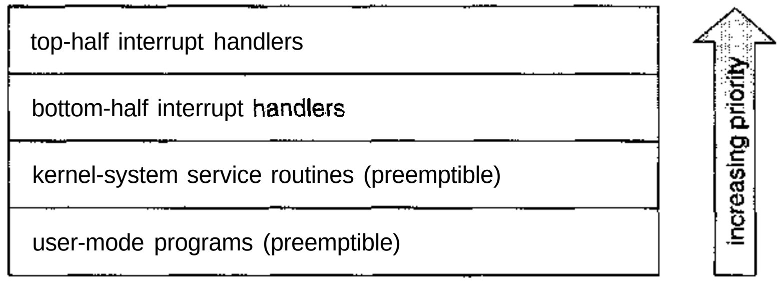


Figure 21.4 Interrupt protection levels.

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 a normal interrupt service routine and runs with recursive interrupts disabled; interrupts of a higher priority may interrupt the routine, but interrupts of the same or lower priority are disabled. 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 executing, 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 21.4 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 processes can always be preempted by another process when a time-sharing scheduling interrupt occurs.

21.5.3 Symmetric Multiprocessing

The Linux 2.0 kernel was the first stable Linux kernel to support **symmetric multiprocessor (SMP)** hardware, allowing separate processes to execute in parallel on separate processors. Originally, the implementation of SMP imposed the restriction that only one processor at a time could be executing kernel-anode code.

In version 2.2 of the kernel, a single kernel spinlock (sometimes termed BKL for "big kernel lock") was created to allow multiple processes (running on different processors) to be active in the kernel concurrently. However, the BKL provided a very coarse level of locking granularity. 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 are described in Section 21.5.2. The 2.6 kernel provided additional SMP enhancements, including processor affinity and load-balancing algorithms.

21.6 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 memory. 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.

21.6.1 Management of Physical Memory

Due to specific hardware characteristics, Linux separates physical memory into three different zones identifying different regions of memory. The zones are identified as:

- ZONE_DMA
- ZONE_NORMAL
- ZONE_HIGHMEM

These zones are architecture specific. For example, on the Intel 80x86 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. ZONE_NORMAL identifies physical memory that is mapped to the CPU's address space. This zone is used for most routine memory requests. For architectures that do not limit what DMA can access, ZONE_DMA is not present, and ZONE_NORMAL is used. Finally, 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. The relationship of zones and physical addresses on the Intel 80x86 architecture is shown in Figure 21.5. The kernel maintains a list of free pages for each zone. When a request for physical memory arrives, the kernel satisfies the request using the appropriate zone.

zone	physical memory
ZONE_DMA	< 16 MB
ZONE_NORMAL	16 .. 896 MB
ZONE_HIGHMEM	> 896 MB

Figure 21.5 Relationship of zones and physical addresses on the Intel 80x86.

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 it is capable of allocating ranges of physically contiguous pages on request. The allocator uses a **buddy system** (Section 9.8.1) 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 21.6 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 21.6.2; the `kmalloc()` variable-length allocator;

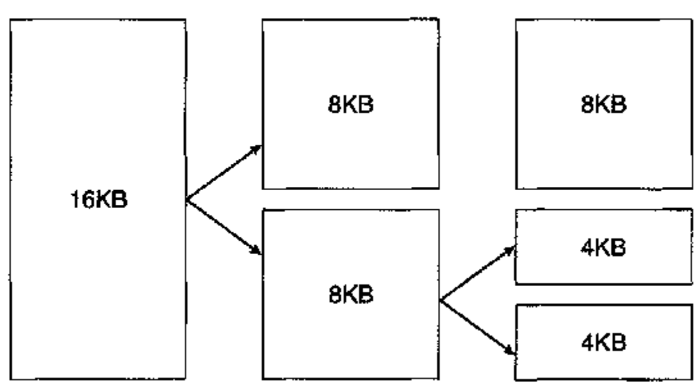


Figure 21.6 Splitting of memory in the buddy system.

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, rather than an entire page. Analogous to the C language's `malloc()` function, this `kmalloc()` service allocates entire pages on demand but then splits them into smaller pieces. The kernel maintains a set of lists of pages in use by the kmalloc () service. Allocating memory involves working out 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; the kmalloc () system cannot relocate 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 and 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 semaphores, 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 semaphores stores instances of semaphore objects, the cache representing process descriptors stores instances of process descriptor objects, etc. The relationship among slabs, caches, and objects is shown in Figure 21.7. 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—which are initially marked as **free**—are allocated to the cache. The number of objects in the cache depends on the size of

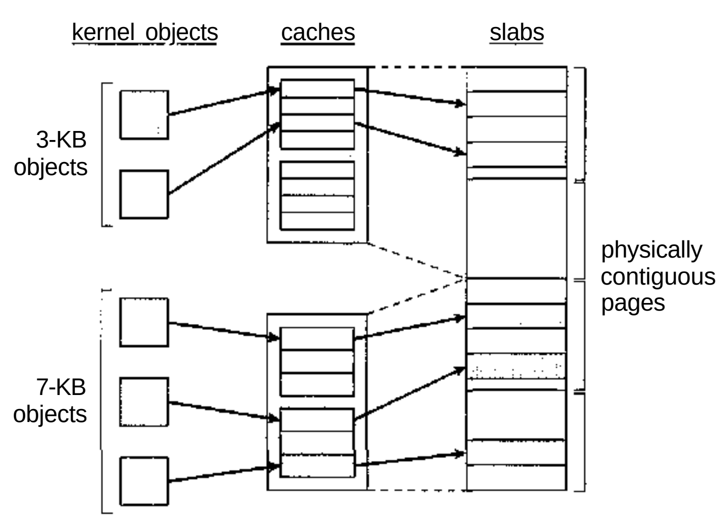


Figure 21.7 Slab allocator in Linux.

the associated slab. For example, a 12-KB slab (comprised of three contiguous 4-KB pages) could store six 2-KB objects. Initially, all 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 exist, 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.

The other two main subsystems in Linux that do their own management of physical pages are closely related to one another. These are the page cache and the virtual memory system. The page cache is the kernel's main cache for block-oriented devices and memory-mapped files and is the main mechanism through which I/O to these devices is performed. Both the native Linux disk-based file systems and the NFS networked file system use the page cache. The page cache caches 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 one another 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 sections, we look at the virtual memory system in greater detail.

21.6.2 Virtual Memory

The Linux virtual memory system is responsible for maintaining the address space visible to each process. It creates pages of virtual memory on demand and manages the loading of those pages from disk or their swapping 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, and 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 determine 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 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 that points to a table of functions that implement the key page-management functions 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.

21.6.2.1 Virtual Memory Regions

Linux implements several types of virtual memory regions. The first property that characterizes a type of virtual memory is the backing store for the region, which describes where the pages for a 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. 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 both by the page cache and by 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.

21.6.2.2 Lifetime of a Virtual Address Space

The kernel will create a new virtual address space in two situations: when a process runs a new program with the `exec()` system call and on creation of a new process 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; copies are made only when absolutely necessary.

21.6.2.3 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 disk 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 9.4.5.2. 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 disk blocks for improved performance. The allocator records the fact that a page has been paged out to disk 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.

21.6.2.4 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 processes 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.

21.6.3 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 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 extensibility: 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 Sections 21.6.3.1 and 21.6.3.2, we concentrate exclusively on the loading and running of ELF-format binaries. The procedure for loading a.out binaries is simpler but is similar in operation.

21.6.3.1 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 21.8 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

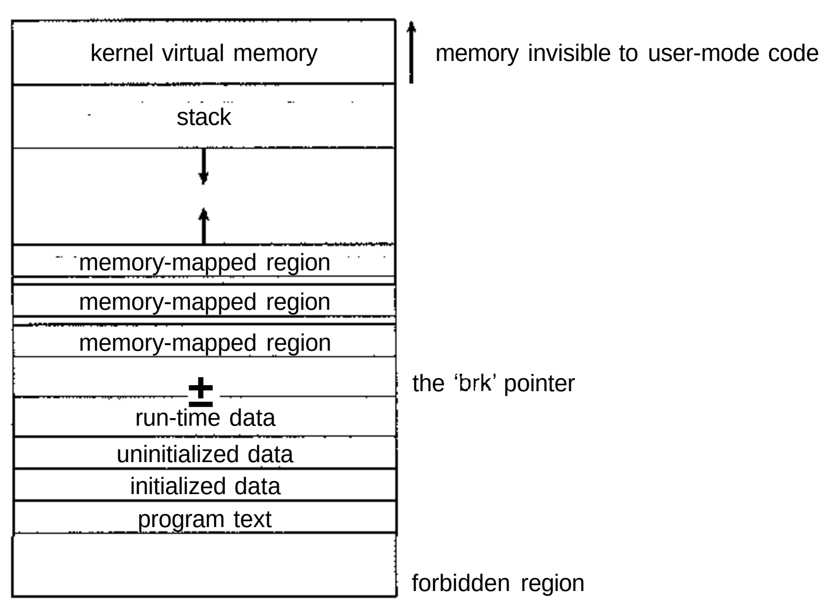


Figure **21.8** Memory layout for ELF programs.

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.

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.

21.6.3.2 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 also need to 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 this single loading 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.

21.7 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 file 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—ext2fs.

21.7.1 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 **file 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 process 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 whether the process asked for write permissions when the file was opened and tracks the process's activity if necessary to perform adaptive read-ahead, fetching file data into memory before the process requests the data, to improve performance.

File objects typically belong to a single process, but inode objects do not. Even when a file is no longer being used by any processes, 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 that 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 one 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 process 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 process 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.

21.7.2 The Linux **ext2fs** File System

The standard on-disk file system used by Linux is called **ext2fs,** 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 of this file system to improve performance and scalability and to add a few missing features led to the **second extended file system (ext2fs).**

Linuxs ext2fs has much in common with the BSD Fast File Systen] (FFS) (Section A.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; 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 ext2fs 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, ext2fs does not use fragments at all but performs all its allocations in smaller units. The default block size on ext2fs is 1 KB, although 2-KB and 4-KB blocks are also supported.

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 1-KB I/O request size is too small to maintain good performance, so ext2fs 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 ext2fs allocation policy comes in two parts. As in FFS, an ext2fs file system is partitioned into multiple **block groups.** FFS uses the similar concept of **cylinder groups,** where each group corresponds to a single cylinder of a physical disk. However, 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, ext2fs 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, ext2fs 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, ext2fs 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 ext2fs from leaving a hole between the most recently allocated block in the previous nonzero byte and the zero byte found.

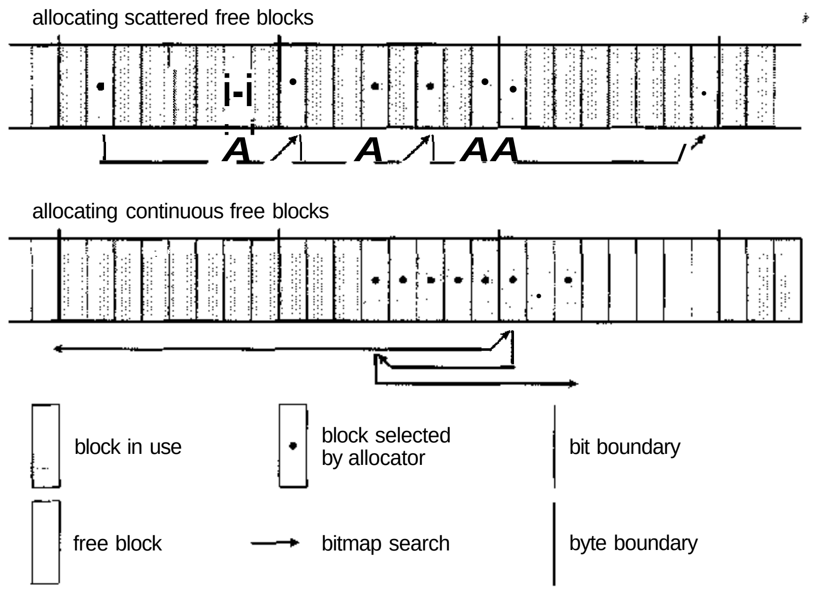


Figure 21.9 ext2fs block-allocation policies.

Once the next block to be allocated has been found by either bit or byte search, ext2fs 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 21.9 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, and 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 before it, so 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.

21.7.3 Journaling

Many different types of file systems are available for Linux systems. One popular feature in a file system is **journaling,** whereby modifications to the file system are sequentially written 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, and the system call modifying the file system (i.e. `write()`) can return to the user process, allowing it to continue execution. 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, there will be zero or more transactions 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. 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 are also typically faster than non-journaling 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. The 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.

Journaling is not provided in ext2fs. It is provided, however, in another common file system available for Linux systems, **ext3,** which is based on ext2fs.

21.7.4 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 simply provides an interface to some other functionality. The Linux **process file system,** known as the /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. SVR4 UNIX introduced a /proc file system as an efficient interface to the kernel's process debugging support: 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, which the standard

UNIX user environment provides powerful tools to process. 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. Given that 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, it 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 wide, 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 dealt with 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 the new desired values out 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 decide to which variables the application is referring.

21.8 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. A user can open an access channel to a device in the same way she opens any

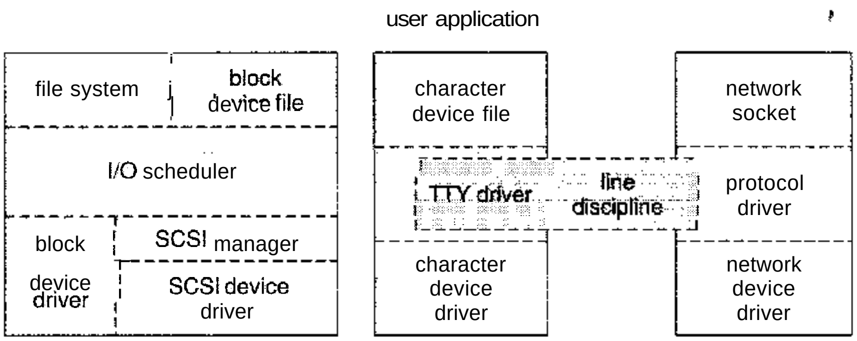


Figure 21.10 Device-driver block structure.

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 21.10 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 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 laying out of data onto the 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 may be accessed randomly, while character devices are only accessed serially. For example, seeking to a certain position in a file might be supported for a DVD but makes no sense to 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 21.10.

21.8.1 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 per-device 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 into the list before the active request. As new I/O requests are made, the request manager attempts to merge requests in the per-device lists.

The scheduling of I/O operations changed somewhat with version 2.6 of the kernel. The fundamental problem with the elevator algorithm is that I/O operations concentrated in a specific region of the disk can result in starvation of requests that need to occur in other regions of the disk. The **deadline I/O scheduler** used in version 2.6 works similarly to the elevator algorithm except that it also associates a deadline with each request, thus addressing the starvation issue. By default, the deadline for read requests is 0.5 second and that for write requests is 5 seconds. The deadline scheduler maintains a **sorted queue** of pending I/O operations sorted by sector number. However, it also maintains two other queues—a **read queue** for read operations and a **write queue** for write operations. These two queues are ordered according to deadline. Every I/O request is placed in both the sorted queue and either the read or the write queue, as appropriate. Ordinarily, I/O operations occur from the sorted queue. However, if a deadline expires for a request in either the read or the write queue, I/O operations are scheduled from the queue containing the expired request. This policy ensures that an I/O operation will wait no longer than its expiration time.

21.8.2 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.

21.9 Interprocess Communication

UNIX 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.

21.9.1 Synchronization and Signals

The standard UNIX mechanism for informing a process that an event has occurred is the signal. Signals canbe 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 are available, and they cannot carry information: Only the fact that a signal 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 normally use signals to receive notification of that event. Rather, communication about incoming asynchronous events within the kernel is performed 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, it will wake up every process on the wait queue. 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.

21.9.2 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 21.10.

Two other methods of sharing data among processes are available. First, 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 out to disk shared-memory pages, 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.

21.10 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 A.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 it has finished processing a set of packets, it passes them on, up to the socket interface if the data are destined for a local connection or downward to a device driver if the packet needs to be transmitted remotely. The protocol layer decides to which socket or device to send the packet.

All communication between the layers of the networking stack is performed by passing single skbuff structures. An skbuff 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 an `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 built 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 is used to carry various error and status messages between hosts.

Packets (skbuffs) arriving at the networking stack's protocol software are expected to be already tagged with an internal identifier indicating to which protocol the packet is relevant. Different networking-device drivers encode the protocol type in different ways over their communications media; 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 destined, it 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 an 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 to match against.

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 onto hash tables keyed on these four address-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.

21.11 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

21.11.1 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 problems. 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 times during which a user is permitted to connect to the system or 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).

21.11.2 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 of the process's gids match 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 A.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 onto 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 may wish to perform selected operations with all its privileges. Standard UNIX implementations achieve this capacity only by swapping the real and effective uids; 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 back 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 the process 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 could simply pass the server file descriptors for any files to be printed, denying the server access to any of the user's other files.

21.12 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. 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.

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 different 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.

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.

Exercises

21.1 What are the advantages and disadvantages of writing an operating system in a high-level language, such as C?

21.2 In what circumstances is the system-call sequence `fork()` `exec()` most appropriate? When is `vfork()` preferable?

21.3 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.

21.4 Linux runs on a variety of hardware platforms. What steps must the 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?

21.5 What are the advantages and disadvantages of making only some of the symbols defined inside a kernel accessible to a loadable kernel module?

21.6 What are the primary goals of the conflict-resolution mechanism used by the Linux kernel for loading kernel modules?

21.7 Discuss how the clone() operation supported by Linux is used to support both processes and threads.

21.8 Would one classify Linux threads as user-level threads or as kernel-level threads? Support your answer with the appropriate arguments.

21.9 What extra costs are incurred by the creation and scheduling of a process, compared with the cost of a cloned thread?

21.10 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?

21.11 Under what circumstances would an user process request an operation that results in the allocation of a demand-zero memory region?

21.12 What scenarios would cause a page of memory to be mapped into an user program's address space with the copy-on-write attribute enabled?

21.13 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.

21.14 The directory structure of a Linux operating system could comprise of files corresponding to different file systems, including the Linux /proc file system. What are the implications of having to support different file-system types on the structure of the Linux kernel?

21.15 In what ways does the Linux setuid feature differ from the setuid feature in standard Unix?

21.16 The Linux source code is freely and widely available over the Internet or from CD-ROM vendors. What are three implications of this availability for the security of the Linux system?

Bibliographical Notes

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 Pages at http://lxr.linux.no/ maintain current listings of the Linux kernel, browsable via the Web and fully cross-referenced.
- Linux-HQ at http://www.linuxhq.com/ hosts a large amount of information relating to the Linux 2.x kernels. This site also includes links to the home pages of most Linux distributions, as well as archives of the major mailing lists.
- The Linux Documentation Project at http://sunsite.unc.edu/linux/ lists many books on Linux that are available in source format as part of the Linux Documentation Project. The project also hosts the Linux *How-To* guides, which contain a series of hints and tips relating to aspects of Linux.
- The *Kernel Hackers' Guide* is an Internet-based guide to kernel internals in general. This constantly expanding site is located at http://www.redhat.com:8080/HyperNews/get/khg.html.
- The Kernel Newbies website (http://www.kernelnewbies.org/) provides a resource for introducing the Linux kernel to newcomers.

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. 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 can be obtained 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 are these:

- ftp://tsx-11.mit.edu/pub/linux/
- ftp://sunsite.unc.edu/pub/Linux/
- ftp://linux.kernel.org/pub/linux/

In addition to investigating Internet resources, you can read about the internals of the Linux kernel in Bovet and Cesati [2002] and Love [2004].