UNIT-IV

MASS-STORAGE STRUCTURE R13

FILE SYSTEM INTERFACE

FILE SYSTEM IMPLEMENTATION

OVERVIEW OF MASS-STORAGE STRUCTURE:

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

MAGNETIC DISKS:

Magnetic disks provide the bulk of secondary storage for modern computer systems. Conceptually, disks are relatively simple (Figure 4.1). Each disk **platter** has a flat circular shape, like a CD. Common platter diameters range from 1.8 to 3.5 inches. The two surfaces of a platter are covered with a magnetic material.We store information by recording it magnetically on the platters.

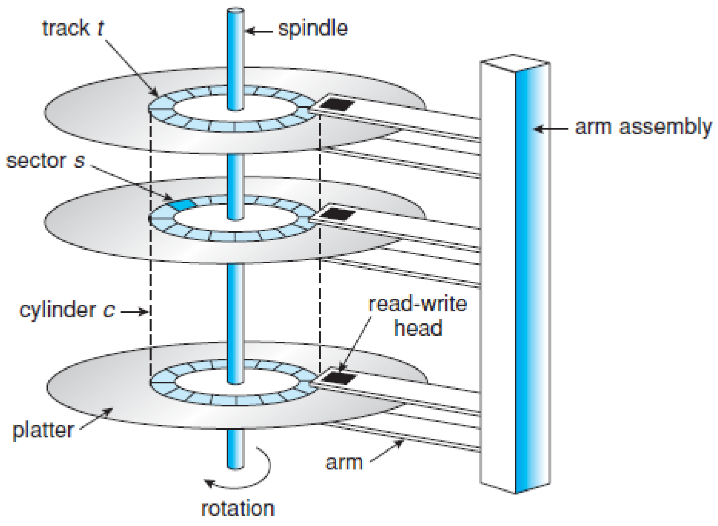


FIGURE 4.1: MOVING-HEAD DISK MECHANISM

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 250 times per second, specified in terms of rotations per minute **(RPM)**. Common drives spin at 5,400, 7,200, 10,000, and 15,000 RPM. Disk speed has two parts.

The **transfer rate** is the rate at which data flow between the drive and the computer. The **positioning time**, or **random-access time**, consists of two parts: the time necessary to move the disk arm to the desired cylinder, called the **seek time**, and the time necessary for the 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, the head will sometimes damage the magnetic surface. This accident is called a **head crash**. A head crash normally cannot be repaired; the entire disk must be replaced.

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. Other forms of removable disks include CDs, DVDs, and Blu-ray discs as well as removable flash-memory devices known as **flash drives** (which are a type of solid-state drive).

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 **advanced technology attachment (ATA)**, **serial ATA (SATA)**, **eSATA**, **universal serial bus (USB)**, and **fibre channel (FC)**. 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.

Solid-State Disks:

Sometimes old technologies are used in new ways as economics change or the technologies evolve. An example is the growing importance of **solid-state disks**, or **SSDs**. Simply described, an SSD is nonvolatile memory that is used like a hard drive. There are many variations of this technology, from DRAM with a battery to allow it to maintain its state in a power failure through flash-memory technologies like single-level cell (SLC) and multilevel cell (MLC) chips.

SSDs have the same characteristics as traditional hard disks but can be more reliable because they have no moving parts and faster because they have no seek time or latency. In addition, they consume less power.

However, they are more expensive per megabyte than traditional hard disks, have less capacity than the larger hard disks, and may have shorter life spans than hard disks, so their uses are somewhat limited. One use for SSDs is in storage arrays, where they hold file-system metadata that require high performance. SSDs are also used in some laptop computers to make them smaller, faster, and more energy-efficient.

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, with current capacities exceeding several terabytes. Some tapes have built-in compression that can more than double the effective storage. Tapes and their drivers are usually categorized by width, including 4, 8, and 19 millimeters and 1/4 and 1/2 inch.

DISK STRUCTURE:

Modern magnetic 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.

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.

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**.

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 fibre channel (FC), 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).** The other FC 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 andwrites of logical data blocks directed to specifically identified storage units (such as bus ID or target logical unit).

Network-Attached Storage:

A network-attached storage (NAS) device is a special-purpose storage system that is accessed remotely over a data network (Figure 4.2). Clients access network-attached storage via a remote-procedure-call interface such as NFS for UNIX systems or CIFS for Windows machines.

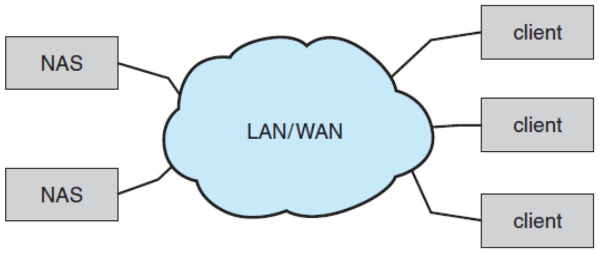


FIGURE 4.2: NETWORK-ATTACHED STORAGE

The remote procedure calls (RPCs) are carried via TCP or UDP over an IP network—usually the same localarea network (LAN) that carries all data traffic to the clients. Thus, it may be easiest to think of NAS as simply another storage-access protocol. The networkattached storage unit is usually implemented as a RAID array with software that implements the RPC interface.

Network-attached storage provides a convenientway 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.

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.

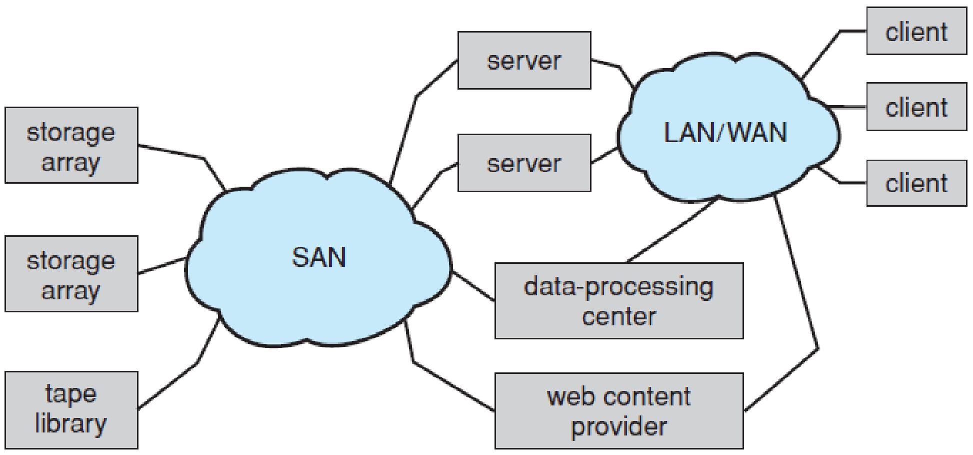


FIGURE 4.3: STORAGE-AREA NETWORK

Astorage-area network (SAN) is a private network (using storage protocols rather than networking protocols) connecting servers and storage units, as shown in Figure 4.3. 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.

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 fast access time and large disk bandwidth.

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 managing the order in which disk I/O requests are serviced.

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.

The Operating Ssystem make this choice with one of several disk-scheduling algorithms.

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***, in that order.

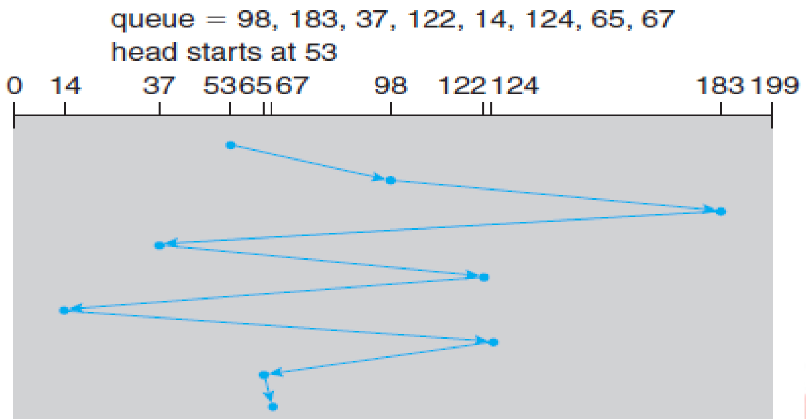


FIGURE 4.4: FCFS DISK SCHEDULING

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 4.4. ***The wild swing from 122 to 14 and then back to 124 illustrates the problem with this schedule.***

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 least seek time from the current head position. In other words, 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 4.5).

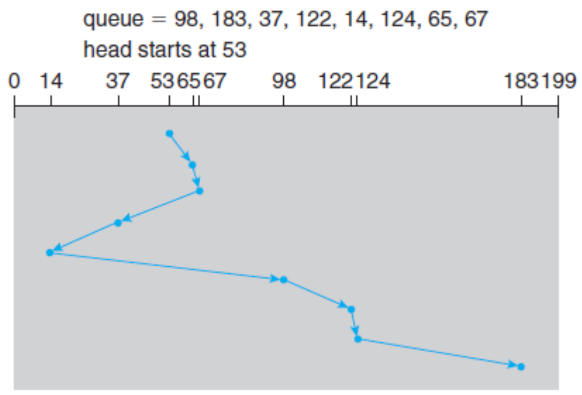


FIGURE 4.5: SSTF DISK SCHEDULING

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. Clearly, this algorithm gives a substantial improvement in performance. Although the SSTF algorithm is a substantial improvement over the FCFS algorithm, it is not optimal.

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.

Assuming that the disk arm is moving toward 0 and that the initial head position is again 53, the head will next service 37 and then 14. At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the requests at 65, 67, 98, 122, 124, and 183 (Figure 4.6). 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.

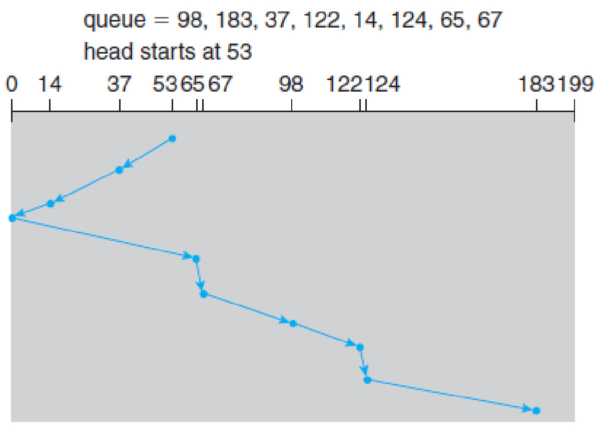


FIGURE 4.6: SCAN DISK SCHEDULING

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 4.7). The C-SCAN scheduling algorithm essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one.

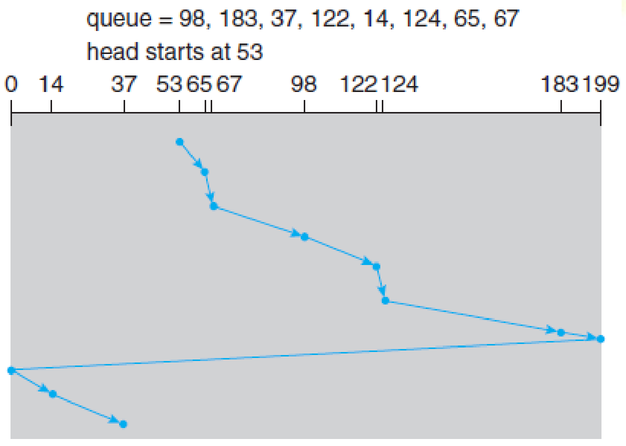


FIGURE 4.7: C-SCAN DISK SCHEDULING

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 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 4.8).

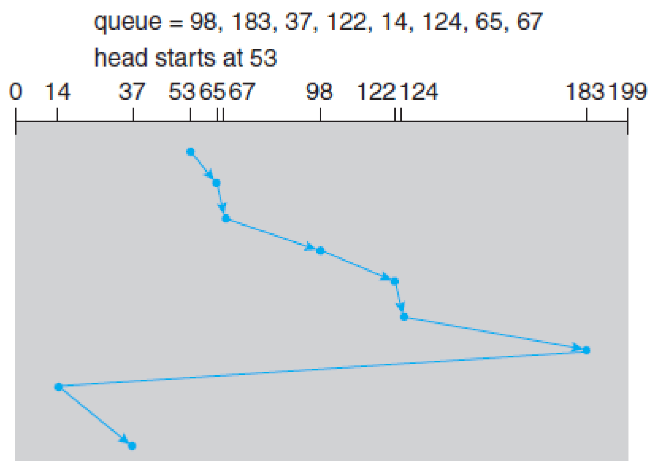


FIGURE 4.8: C-LOOK DISK SCHEDULING

SWAP-SPACE MANAGEMENT:

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.

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 froma few megabytes of disk space to gigabytes, depending on the amount of physical memory, the amount of virtual memory it is backing, and the way in which the virtual memory is used.

Note that it may be safer to overestimate than to underestimate the amount of swap space required, because if a system runs out of swap space it may be forced to abort processes or may crash entirely.

Overestimation wastes 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.

SWAP-SPACE LOCATION:

A swap space can reside in one of two places:

1) *It can be carved out of the normal file system*

2) *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 because navigating the directory structure and the disk allocation data structures takes time and (possibly) extra disk accesses. External fragmentation can greatly increase swapping times by forcing multiple seeks during reading or writing of a process image.

Alternatively, swap space can be created in a separate **raw partition**. No file system or directory structure is placed in this space. Rather, a separate swap-space storage manager is used to allocate and deallocate the blocks from the raw partition. This manager uses algorithms optimized for speed rather than for storage efficiency, because swap space is accessed much more frequently than file systems (when it is used). Internal fragmentation may increase which is acceptable.

Some operating systems are flexible and can swap both in raw partitions and in file-system space.

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 independent disks (RAID)**, are commonly used to address the performance and reliability issues.

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

IMPROVEMENT OF RELIABILITY VIA REDUNDANCY:

Let's 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.

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**. With mirroring, a logical disk consists of two physical disks, and every write is carried out on both disks. The result is called a **mirrored volume**. If one of the disks in the volume 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 of a mirrored volume—where failure is the loss of data—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.

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**.

In **block-level striping**, for instance, blocks of a file are striped across multiple disks; with *n* disks, block *i* of a file goes to disk (*i* mod *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.

RAID LEVELS:

Mirroring provides high reliability, but it is expensive. Striping provides high data-transfer rates, but it does not improve reliability. Numerous schemes to provide redundancy at lower cost by using disk striping combined with "parity" bits 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 4.9 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.

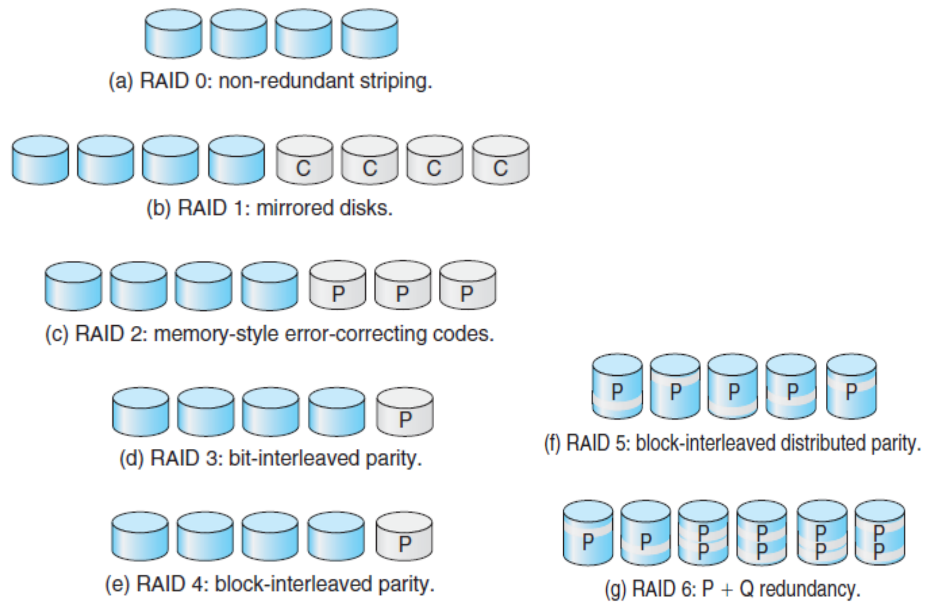

(a) RAID 0: non-redundant striping.
C C C C
(b) RAID 1: mirrored disks.
P P P
(c) RAID 2: memory-style error-correcting codes.
P
(d) RAID 3: bit-interleaved parity.
P
(e) RAID 4: block-interleaved parity.
P P P P P
(f) RAID 5: block-interleaved distributed parity.
P P P P P P P P P P
(g) RAID 6: P + Q redundancy.

Figure 4.9: 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 4.9(a).
- **RAID level 1**. RAID level 1 refers to disk mirroring. Figure 4.9(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.
- **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.
- **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 *N* other disks. This scheme is diagrammed in Figure 4.9(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.
- **RAID level 5**. RAID level 5, or block-interleaved distributed parity, differs from level 4 in that it spreads data and parity among all *N*+1 disks, rather than storing data in *N* disks and parity in one disk.
- **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.
- **RAID levels 0 + 1 and1 + 0**.
    - RAID level 0 + 1 refers to a combination of RAID levels 0 and 1. RAID 0 provides the performance, while RAID 1 provides the reliability. Generally, this level provides better performance than RAID 5.
    - Another RAID option that is becoming available commercially is RAID level 1 + 0, in which disks are mirrored in pairs and then the resulting mirrored pairs are striped. This scheme has some theoretical advantages over RAID 0 + 1.

For example, if a single disk fails in RAID 0 + 1, an entire stripe is inaccessible, leaving only the other stripe.With a failure in RAID 1 + 0, a single disk is unavailable, but the disk that mirrors it is still available, as are all the rest of the disks (Figure 4.10).

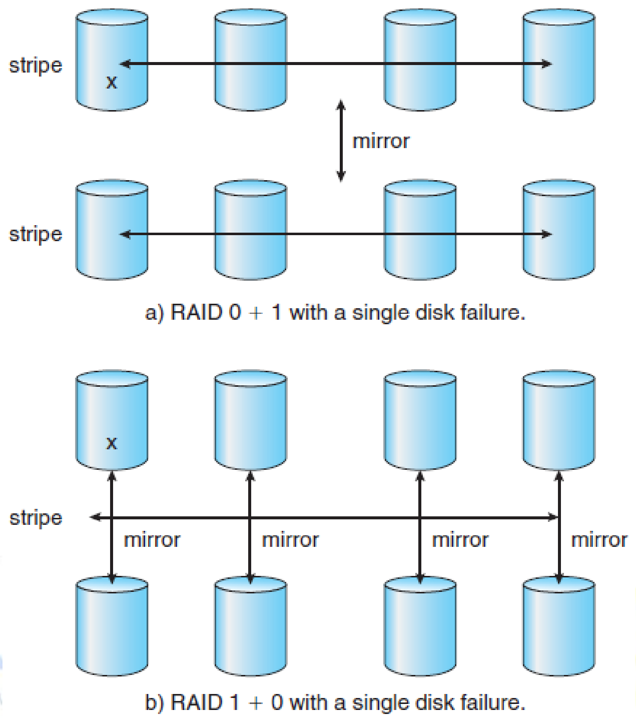

stripe
x
mirror
stripe
a) RAID 0 + 1 with a single disk failure.
x
stripe
mirror
mirror
mirror
mirror
b) RAID 1 + 0 with a single disk failure.

FIGURE 4.10: RAID 0 + 1 AND 1 + 0

STABLE-STORAGE IMPLEMENTATION:

By definition, information residing in stable storage is never lost. To implement such storage, we need to replicate the required information. on multiple storage devices (usually disks) with independent failure modes.

We also 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.

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.

FILE –SYSTEM INTERFACE:

FILE CONCEPT:

Computers can store information on various storage media, such as magnetic disks, magnetic tapes, and optical disks. So that the computer system will be convenient to use, the operating system provides a uniform logical view of stored information.

The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the **file**. Files are mapped by the operating system onto physical devices. These storage devices are usually nonvolatile, so the contents are persistent between system reboots.

A file is a named collection of related information that is recorded on secondary storage. From a user's perspective, a file is the smallest allotment of logical secondary storage; that is, data cannot be written to secondary storage unless they are within a file.

Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free form, such as text files, or may be formatted rigidly. In general, a file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user. The concept of a file is thus extremely general.

File Attributes:

A file is named, for the convenience of its human users, and is referred to by its name. A name is usually a string of characters, such as example.c. Some systems differentiate between uppercase and lowercase characters in names, whereas other systems do not. When a file is named, it becomes independent of the process, the user, and even the system that created it.

A file's attributes vary from one operating system to another but typically consist of these:

- **Name**. The symbolic file name is the only information kept in human readable form.
- **Identifier**. This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.
- **Type**. This information is needed for systems that support different types of files.
- **Location**. This information is a pointer to a device and to the location of the file on that device.
- **Size**. The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
- **Protection**. Access-control information determines who can do reading, writing, executing, and so on.
- **Time, date, and user identification**. This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

File Operations:

A file is an abstract data type. To define a file properly, we need to consider the operations that can be performed on files. The operating system can provide system calls to create, write, read, reposition, delete, and truncate files.

Let's examine what the operating system must do to perform each of these six basic file operations:

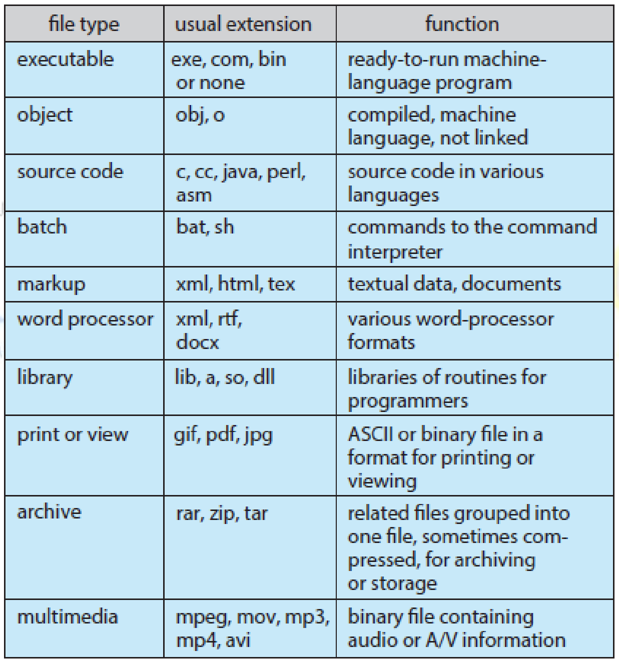
- **Creating a file**. Two steps are necessary to create a file. First, space in the file system must be found for the file.
- **Writing a file**. To write a file, we make a system call specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the file's location. The system must keep a **write pointer** to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.
- **Reading a file**. To read from a file, we use a system call that specifies the name of the file and where (in memory) the next block of the file should be put. Again, the directory is searched for the associated entry, and the system needs to keep a **read pointer** to the location in the file where the next read is to take place. Once the read has taken place, the read pointer is updated. Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process **currentfile-position pointer**. Both the read and write operations use this same pointer, saving space and reducing system complexity.
- **Repositioning within a file**. The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value. Repositioning within a file need not involve any actual I/O. This file operation is also known as a file **seek**.
- **Deleting a file**. To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space, so that it can be reused by other files, and erase the directory entry.
- **Truncating a file**. The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all

attributes to remain unchanged—except for file length—but lets the file be reset to length zero and its file space released.

File Types:

When we design a file system—indeed, an entire operating system—we always consider whether the operating system should recognize and support file types. If an operating system recognizes the type of a file, it can then operate on the file in reasonable ways.

A common technique for implementing file types is to include the type as part of the file name. The name is split into two parts—a name and an extension, usually separated by a period (figure 4.11).



file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
batch	bat, sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

FIGURE 4.11: COMMON FILE TYPES

ACCESS METHODS:

Files store information. When it is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways. Some systems provide only one access method for files.

While others support many access methods, and choosing the right one for a particular application is a major design problem.

SEQUENTIAL ACCESS:

The simplest access method is **sequential access**. Information in the file is processed in order, one record after the other. This mode of access is by far the most common; for example, editors and compilers usually access files in this fashion.

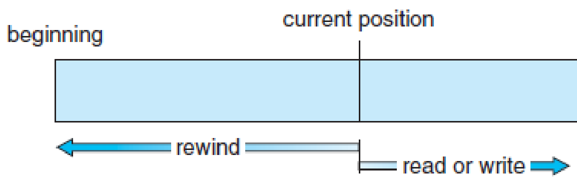


FIGURE 4.12: SEQUENTIAL-ACCESS FILE

Sequential access, which is depicted in Figure 4.12, is based on a tape model of a file and works as well on sequential-access devices as it does on random-access ones.

DIRECT ACCESS:

Another method is **direct access** (or **relative access**). Here, a file is made up of fixed-length **logical records** that allow programs to read and write records rapidly in no particular order. The direct-access method is based on a disk model of a file, since disks allow random access to any file block. For direct access, the file is viewed as a numbered sequence of blocks or records.

Thus, we may read block 14, then read block 53, and then write block 7. There are no restrictions on the order of reading or writing for a direct-access file. Direct-access files are of great use for immediate access to large amounts of information. Databases are often of this type.

As a simple example, on an airline-reservation system, we might store all the information about a particular flight (for example, flight 713) in the block identified by the flight number. Thus, the number of available seats for flight 713 is stored in block 713 of the reservation file.

We can easily simulate sequential access on a direct-access file by simply keeping a variable *cp* that defines our current position, as shown in Figure 4.13. Simulating a direct-access file on a sequential-access file, however, is extremely inefficient and clumsy.

Not all operating systems support both sequential and direct access for files. Some systems allow only sequential file access; others allow only direct access.

sequential access	implementation for direct access
reset	cp = 0;
read_next	read cp ; cp = cp + 1;

write_next	write cp;
	cp = cp + 1;

FIGURE 4.13: SIMULATION OF SEQUENTIAL ACCESS ON A DIRECT-ACCESS FILE

DIRECTORY AND DISK STRUCTURE:

A storage device can be used in its entirety for a file system. It can also be subdivided for finer-grained control. For example, a disk can be **partitioned** into quarters, and each quarter can hold a separate file system. Storage devices can also be collected together into RAID sets that provide protection from the failure of a single disk. Sometimes, disks are subdivided and also collected into RAID sets.

Partitioning is useful for limiting the sizes of individual file systems, putting multiple file-system types on the same device, or leaving part of the device available for other uses, such as swap space or unformatted (raw) disk space. Afile system can be created on each of these parts of the disk. Any entity containing a file system is generally known as a **volume**.

The volume may be a subset of a device, a whole device, or multiple devices linked together into a RAID set. Each volume can be thought of as a virtual disk. Volumes can also store multiple operating systems, allowing a system to boot and run more than one operating system.

Each volume that contains a file system must also contain information about the files in the system. This information is kept in entries in a **device directory** or **volume table of contents**. The device directory (more commonly known simply as the **directory**) records information—such as name, location, size, and type—for all files on that volume. Figure 4.14 shows a typical file-system organization.

STORAGE STRUCTURE:

As we have just seen, a general-purpose computer system has multiple storage devices, and those devices can be sliced up into volumes that hold file systems. Computer systems may have zero or more file systems, and the file systems maybe of varying types.

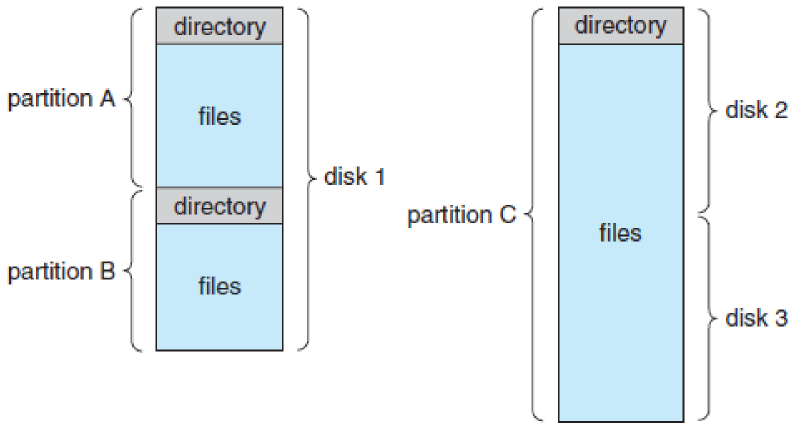


FIGURE 4.14: A TYPICAL FILE-SYSTEM ORGANIZATION

For example, a typical Solaris systemmayhave dozens of file systems of a dozen different types, as shown in the file system list in Figure 4.15.

/	ufs
/devices	devfs
/dev	dev
/system/contract	ctfs
/proc	proc
/etc/mnttab	mntfs
/etc/svc/volatile	tmpfs
/system/object	objfs
/lib/libc.so.1	lofs
/dev/fd	fd
/var	ufs
/tmp	tmpfs
/var/run	tmpfs
/opt	ufs
/zpbge	zfs
/zpbge/backup	zfs
/export/home	zfs
/var/mail	zfs
/var/spool/mqueue	zfs
/zpbg	zfs
/zpbg/zones	zfs

FIGURE 4.15: SOLARIS FILE SYSTEMS

DIRECTORY OVERVIEW:

The directory can be viewed as a symbol table that translates file names into their directory entries. If we take such a view,we see that the directory itself can be organized in many ways.

The organization must allow us to insert entries, to delete entries, to search for a named entry, and to list all the entries in the directory. In this section, we examine several schemes for defining the logical structure of the directory system.

When considering a particular directory structure, we need to keep in mind the operations that are to be performed on a directory:

• **Search for a file**.We need to be able to search a directory structure to find the entry for a particular file. Since files have symbolic names, and similar names may indicate a relationship among files, we may want to be able to find all files whose names match a particular pattern.

• **Create a file**. New files need to be created and added to the directory.

• **Delete a file**.When a file is no longer needed, we want to be able to remove it from the directory.

• **List a directory**. We need to be able to list the files in a directory and the contents of the directory entry for each file in the list.

• **Rename a file**. Because the name of a file represents its contents to its users, we must be able to change the name when the contents or use of the file changes. Renaming a file may also allow its position within the directory structure to be changed.

• **Traverse the file system**.Wemay wish to access every directory and every file within a directory structure. For reliability, it is a good idea to save the contents and structure of the entire file system at regular intervals. Often, we do this by copying all files to magnetic tape. This technique provides a backup copy in case of system failure. In addition, if a file is no longer in use, the file can be copied to tape and the disk space of that file released for reuse by another file.

SINGLE-LEVEL DIRECTORY:

The simplest directory structure is the single-level directory. All files are contained in the same directory, which is easy to support and understand (Figure 4.16). A single-level directory has significant limitations, however, when the number of files increases or when the system has more than one user. Since all files are in the same directory, they must have unique names. If two users call their data file test.txt, then the unique-name rule is violated.

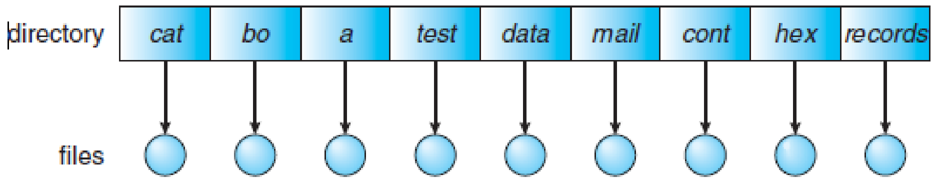


FIGURE 4.16: SINGLE-LEVEL DIRECTORY

TWO-LEVEL DIRECTORY:

As we have seen, a single-level directory often leads to confusion of file names among different users. The standard solution is to create a separate directory for each user.

In the two-level directory structure, each user has his own **user file directory (UFD)**. The UFDs have similar structures, but each lists only the files of a single user. When a user job starts or a user logs in, the system's **master file directory (MFD)** is searched. The MFD is indexed by user name or account number, and each entry points to the UFD for that user (Figure 4.17).

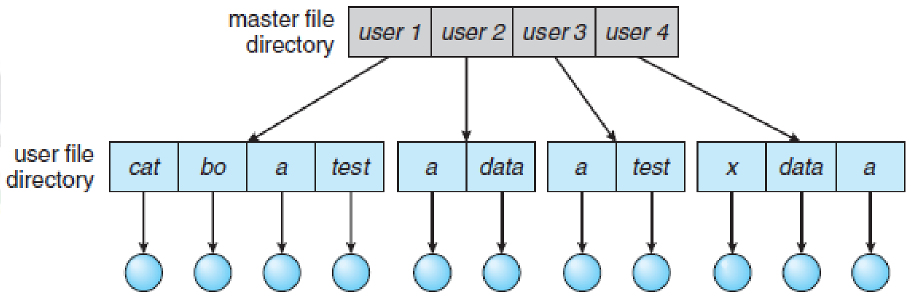


FIGURE 4.17: TWO-LEVEL DIRECTORY STRUCTURE

When a user refers to a particular file, only his own UFD is searched. Thus, different users may have files with the same name, as long as all the file names within each UFD are unique.

To create a file for a user, the operating system searches only that user's UFD to ascertain whether another file of that name exists. To delete a file, the operating system confines its search to the local UFD; thus, it cannot accidentally delete another user's file that has the same name.

The user directories themselves must be created and deleted as necessary. A special system program is run with the appropriate user name and account information. The program creates a new UFD and adds an entry for it to theMFD. The execution of this program might be restricted to system administrators.

The allocation of disk space for user directories can be handled with the techniques. Although the two-level directory structure solves the name-collision problem, it still has disadvantages. This structure effectively isolates one user from another.

Isolation is an advantage when the users are completely independent but is a ***disadvantage*** when the users want to cooperate on some task and to access one another's files. Some systems simply do not allow local user files to be accessed by other users.

TREE-STRUCTURED DIRECTORIES:

Once we have seen how to view a two-level directory as a two-level tree, the natural generalization is to extend the directory structure to a tree of arbitrary height (Figure 4.18). This generalization allows users to create their own subdirectories and to organize their files accordingly. A tree is the most common directory structure.

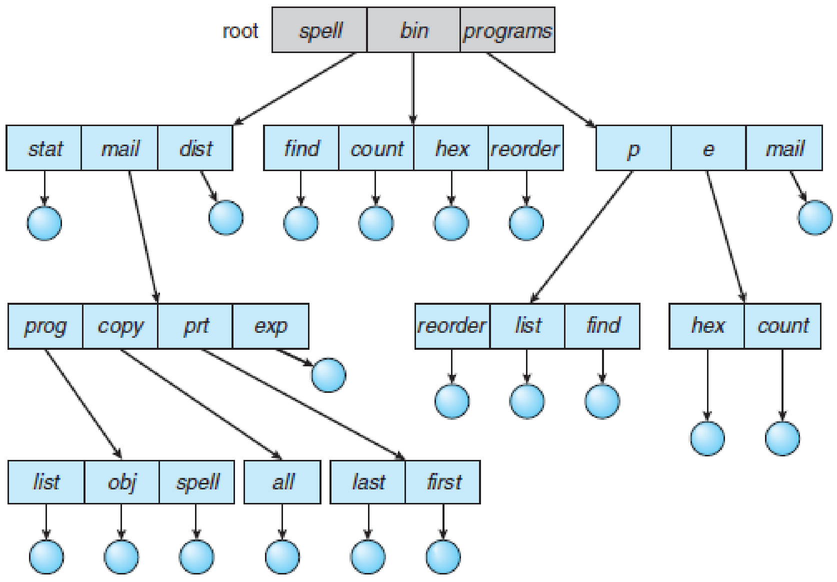


FIGURE 4.18: TREE-STRUCTURED DIRECTORY STRUCTURE

The tree has a root directory, and every file in the system has a unique path name. A directory (or subdirectory) contains a set of files or subdirectories. A directory is simply another file, but it is treated in a special way. All directories have the same internal format. One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1). Special system calls are used to create and delete directories. In normal use, each process has a current directory. The **current directory** should contain most of the files that are of current interest to the process.

When reference is made to a file, the current directory is searched. If a file is needed that is not in the current directory, then the user usually must either specify a path name or change the current directory to be the directory holding that file. To change directories, a system call is provided that takes a directory name as a parameter and uses it to redefine the current directory.

With a tree-structured directory system, users can be allowed to access, in addition to their files, the files of other users.

ACYCLIC-GRAPH DIRECTORIES:

Consider two programmers who are working on a joint project. The files associated with that project can be stored in a subdirectory, separating them from other projects and files of the two programmers. But since both programmers are equally responsible for the project, both want the subdirectory to be in their own directories. In this situation, the common subdirectory should be ***shared.***

A shared directory or file exists in the file system in two (or more) places at once. Atree structure prohibits the sharing of files or directories. An **acyclic graph** — that is, a graph with no cycles—allows directories to share subdirectories and files (Figure 4.19). The same file or subdirectory may be in two different directories. The acyclic graph is a natural generalization of the tree-structured directory scheme.

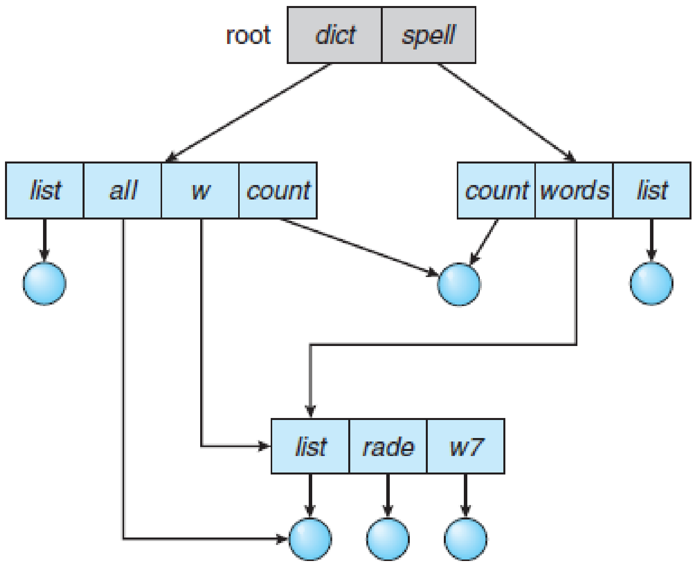


FIGURE 4.19: ACYCLIC-GRAPH DIRECTORY STRUCTURE

It is important to note that a shared file (or directory) is not the same as two copies of the file.With two copies, each programmer can view the copy rather than the original, but if one programmer changes the file, the changes will not appear in the other's copy.With a shared file, only one actual file exists, so any changes made by one person are immediately visible to the other. Sharing is particularly important for subdirectories; a new file created by one person will automatically appear in all the shared subdirectories.

Sharing is particularly important for subdirectories; a new file created by one person will automa Shared files and subdirectories can be implemented in several ways. A common way, exemplified by many of the UNIX systems, is to create a new directory entry called a link. A **link** is effectively a pointer to another file or subdirectory.

Another common approach to implementing shared files is simply to duplicate all information about them in both sharing directories. Thus, both entries are identical and equal. Consider the difference between this approach and the creation of a link. The link is clearly different fromthe original directory entry; thus, the two are not equal. Duplicate directory entries, however, make the original and the copy indistinguishable. A major problem with duplicate directory entries is maintaining consistency when a file is modified.

An acyclic-graph directory structure is more flexible than a simple tree structure, but it is also more complex.

General Graph Directory:

A serious problem with using an acyclic-graph structure is ensuring that there are no cycles. If we start with a two-level directory and allow users to create subdirectories, a tree-structured directory results. It should be fairly easy to see that simply adding new files and subdirectories to an existing tree-structured directory preserves the tree-structured nature. However, when we add links, the tree structure is destroyed, resulting in a simple graph structure (Figure 4.20).

The primary advantage of an acyclic graph is the relative simplicity of the algorithms to traverse the graph and to determine when there are no more references to a file. We want to avoid traversing shared sections of an acyclic graph twice, mainly for performance reasons. If cycles are allowed to exist in the directory, we likewise want to avoid searching any component twice, for reasons of correctness as well as performance. A poorly designed algorithm might result in an infinite loop continually searching through the cycle and never terminating.

One solution is to limit arbitrarily the number of directories that will be accessed during a search.

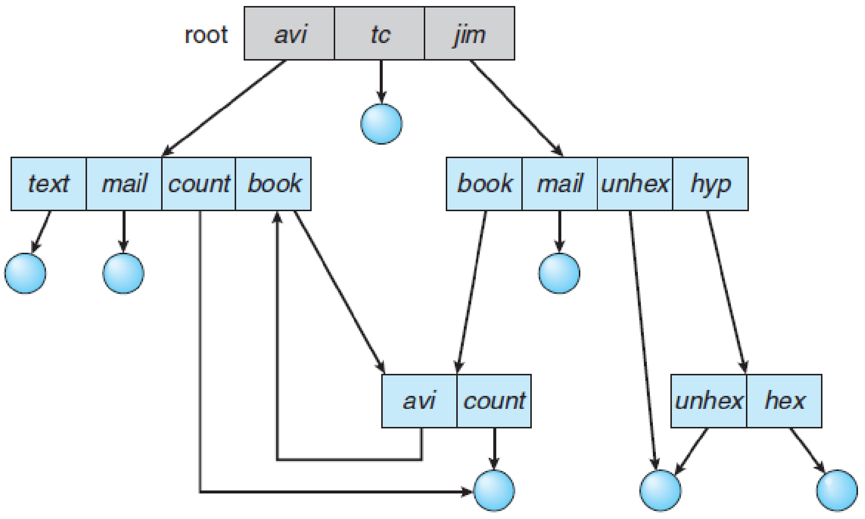


FIGURE 4.20: GENERAL GRAPH DIRECTORY

FILE-SYSTEM MOUNTING:

The mount procedure is straightforward. The operating system is given the name of the device and the **mount point**—the location within the file structure where the file system is to be attached. Some operating systems require that a file system type be provided, while others inspect the structures of the device and determine the type of file system. Typically, a mount point is an empty directory.

To illustrate file mounting, consider the file system depicted in Figure 4.21, where the triangles represent subtrees of directories that are of interest. Figure 4.21(a) shows an existing file system, while Figure 4.21(b) shows an unmounted volume residing on /device/dsk. At this point, only the files on the existing file system can be accessed. Figure 4.22 shows the effects of mounting the volume residing on /device/dsk over /users. If the volume is unmounted, the file system is restored to the situation depicted in Figure 4.21.

Systems impose semantics to clarify functionality. For example, a system may disallow a mount over a directory that contains files; or it may make the mounted file system available at that directory and obscure the directory's existing files until the file system is unmounted, terminating the use of the file system and allowing access to the original files in that directory.

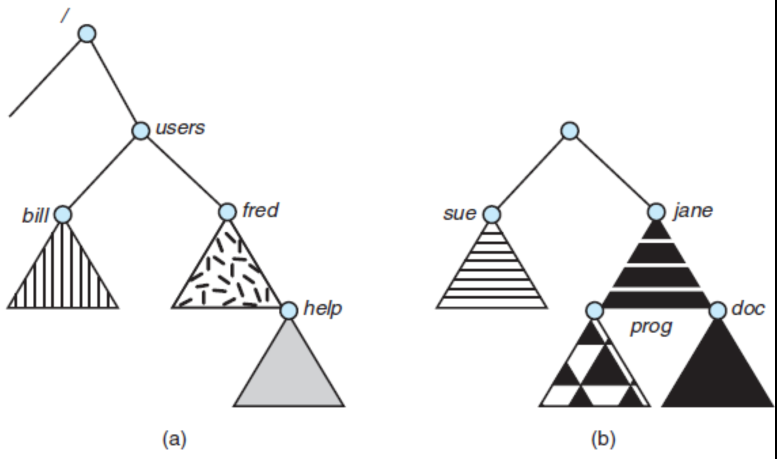


FIGURE 4.21: FILE SYSTEM. (A) EXISTING SYSTEM. (B) UNMOUNTED VOLUME

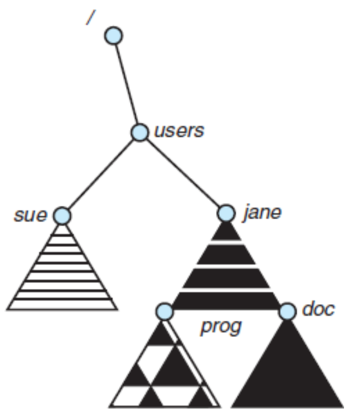


FIGURE 4.22: MOUNT POINT

FILE SHARING:

File sharing is very desirable for users who want to collaborate and to reduce the effort required to achieve a computing goal. Therefore, user-oriented operating systems must accommodate the need to share files in spite of the inherent difficulties.

MULTIPLE USERS:

When an operating system accommodates multiple users, the issues of file sharing, file naming, and file protection become preeminent. Given a directory structure that allows files to be shared by users, the system must mediate the file sharing. The system can either allow a user to access the files of other users by default or require that a user specifically grant access to the files.

To implement sharing and protection, the system must maintain more file and directory attributes than are needed on a single-user system. Although many approaches have been taken to meet this requirement, most systems have evolved to use the concepts of file (or directory) **owner** (or **user**) and **group**. The owner is the userwho can change attributes and grant access and who has the most control over the file. The group attribute defines a subset of users who can share access to the file.

The owner and group IDs of a given file (or directory) are stored with the other file attributes. When a user requests an operation on a file, the user ID can be compared with the owner attribute to determine if the requesting user is the owner of the file.

Likewise, the group IDs can be compared. The result indicates which permissions are applicable. The system then applies those permissions to the requested operation and allows or denies it.

REMOTE FILE SYSTEMS:

With the advent of networks communication among remote computers became possible. Networking allows the sharing of resources spread across a campus or even around the world. One obvious resource to share is data in the form of files.

Through the evolution of network and file technology, remote file-sharing methods have changed. The first implemented method involves manually transferring files between machines via programs like ftp. The second major method uses a **distributed file system (DFS)** in which remote directories are visible from a local machine. In some ways, the third method, the**WorldWide Web**, is a reversion to the first.

A browser is needed to gain access to the remote files, and separate operations (essentially a wrapper for ftp) are used to transfer files. ftp is used for both anonymous and authenticated access. **Anonymous access** allows a user to transfer files without having an account on the remote system. TheWorldWideWeb uses anonymous file exchange almost exclusively. DFS involves a much tighter integration between the machine that is accessing the remote files and the machine providing the files.

THE CLIENT–SERVER MODEL:

Remote file systems allow a computer to mount one or more file systems from one or more remote machines. In this case, the machine containing the files is the **server**, and the machine seeking access to the files is the **client**. The client–server relationship is common with networked machines.

Generally, the server declares that a resource is available to clients and specifies exactly which resource (in this case, which files) and exactly which clients. A server can serve multiple clients, and a client can use multiple servers, depending on the implementation details of a given client–server facility.

The server usually specifies the available files on a volume or directory level. Client identification is more difficult. A client can be specified by a network name or other identifier, such as an IP address, but these can be **spoofed**, or imitated.

As a result of spoofing, an unauthorized client could be allowed access to the server. More secure solutions include secure authentication of the client via encrypted keys.

DISTRIBUTED INFORMATION SYSTEMS:

To make client–server systems easier to manage, **distributed information systems**, also known as **distributed naming services**, provide unified access to the information needed for remote computing. The **domain name system (DNS)** provides host-name-to-network-address translations for the entire Internet.

Before DNS became widespread, files containing the same information were sent via e-mail or ftp between all networked hosts. Other distributed information systems provide ***user name/password/user ID/group ID*** space for a distributed facility.

UNIX systems have employed a wide variety of distributed information methods. Sun Microsystems (now part of Oracle Corporation) introduced **yellow pages** (since renamed **network information service**, or **NIS**), and most of the industry adopted its use.

FAILURE MODES:

Local file systems can fail for a variety of reasons, including failure of the disk containing the file system, corruption of the directory structure or other disk-management information (collectively called **metadata**), disk-controller failure, cable failure, and host-adapter failure.

User or system-administrator failure can also cause files to be lost or entire directories or volumes to be deleted. Many of these failures will cause a host to crash and an error condition to be displayed, and human intervention will be required to repair the damage.

Remote file systems have even more failure modes. Because of the complexity of network systems and the required interactions between remote machines, many more problems can interfere with the proper operation of remote file systems.

CONSISTENCY SEMANTICS:

Consistency semantics represent an important criterion for evaluating any file system that supports file sharing. These semantics specify how multiple users of a system are to access a shared file simultaneously. Consistency semantics are directly related to the process synchronization algorithms.

PROTECTION:

When information is stored in a computer system, we want to keep it safe from physical damage (the issue of reliability) and improper access (the issue of protection).

Reliability is generally provided by duplicate copies of files. Many computers have systems programs that automatically (or through computer-operator intervention) copy disk files to tape at regular intervals (once per day or week or month) to maintain a copy should a file system be accidentally destroyed.

File systems can be damaged by hardware problems (such as errors in reading or writing), power surges or failures, head crashes, dirt, temperature extremes, and vandalism. Files may be deleted accidentally. Bugs in the file-system software can also cause file contents to be lost.

TYPES OF ACCESS:

The need to protect files is a direct result of the ability to access files. Systems that do not permit access to the files of other users do not need protection. Thus, we could provide complete protection by prohibiting access. Alternatively, we could provide free access with no protection. Both approaches are too extreme for general use. What is needed is controlled access.

Protection mechanisms provide controlled access by limiting the types of file access that can be made. Access is permitted or denied depending on several factors, one of which is the type of access requested. Several different types of operations may be controlled:

- **Read**. Read from the file.
- **Write**.Write or rewrite the file.
- **Execute**. Load the file into memory and execute it.
- **Append**.Write new information at the end of the file.
- **Delete**. Delete the file and free its space for possible reuse.
- **List**. List the name and attributes of the file.

Other operations, such as renaming, copying, and editing the file, may also be controlled. For many systems, however, these higher-level functions may be implemented by a system program that makes lower-level system calls. Protection is provided at only the lower level. For instance, copying a file may be implemented simply by a sequence of read requests.

Many protection mechanisms have been proposed. Each has advantages and disadvantages and must be appropriate for its intended application. A small computer system that is used by only a few members of a research group, for example, may not need the same types of protection as a large corporate computer that is used for research, finance, and personnel operations.

ACCESS CONTROL:

The most common approach to the protection problem is to make access dependent on the identity of the user. Different users may need different types of access to a file or directory. The most general scheme to implement identitydependent access is to associate with each file and directory an **access-control list (ACL)** specifying user names and the types of access allowed for each user.

When a user requests access to a particular file, the operating system checks the access list associated with that file. If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs, and the user job is denied access to the file.

This approach has the advantage of enabling complex access methodologies. The main problem with access lists is their length. If we want to allow everyone to read a file, we must list all users with read access. This technique has two undesirable consequences:

- Constructing such a list may be a tedious and unrewarding task, especially if we do not know in advance the list of users in the system.
- The directory entry, previously of fixed size, now must be of variable size, resulting in more complicated space management.

These problems can be resolved by use of a condensed version of the access list. To condense the length of the access-control list, many systems recognize three classifications of users in connection with each file:

- **Owner**. The user who created the file is the owner.
- **Group**. A set of users who are sharing the file and need similar access is a group, or work group.
- **Universe**. All other users in the system constitute the universe.