Kernel I/O Subsystem

Kernels provide many services related to I/O. Several services-scheduling, buffering, caching, spooling, device reservation, and error handling-are provided by the kernel's I/O subsystem and build on the hardware and device driver infrastructure.

I/O Scheduling

To schedule a set of I/O requests means to determine a good order in which to execute them. The order in which applications issue system calls rarely is the best choice. Scheduling can improve overall system performance, can share device access fairly among processes, and can reduce the average waiting time for I/O to complete.

Operating-system developers implement scheduling by maintaining a queue of requests for each device. When an application issues a blocking I/O system call, the request is placed on the queue for that device. The I/O scheduler rearranges the order of the queue to improve the overall system efficiency and the average response time experienced by applications. The operating system may also try to be fair, so that no one application receives especially poor service, or it may give priority service for delay-sensitive requests. For instance, requests from the virtual-memory subsystem may take priority over application requests.

Buffering

A buffer is a memory area that stores data while they are transferred between two devices or between a device and an application. Buffering is done for three reasons.

One reason is to cope with a speed mismatch between the producer and consumer of a data stream.

Suppose, for example, that a file is being received via modem for storage on the hard disk. The modem is about a thousand times slower than the hard disk. So a buffer is created in main memory to accumulate the bytes received from the modem. When an entire buffer of data has arrived, the buffer can be written to disk in a single operation. Since the disk write is not instantaneous and the modem still needs a place to store additional incoming data, two buffers are used. After the modem fills the first buffer, the disk write is requested. The modem then starts to fill the second buffer while the first buffer is written to disk. By the time the modem has filled the second buffer, the disk write from the first one should have completed, so the modem can switch back to the first buffer while the disk writes the second one. This double buffering decouples the producer of data from the consumer, thus relaxing timing requirements between them.

A second use of buffering is to adapt between devices that have different data-transfer sizes.

Such disparities are especially common in computer networking, where buffers are used widely for fragmentation and reassembly of messages. At the sending side, a large message is fragmented into small network packets. The packets are sent over the network, and the receiving side places them in a reassembly buffer to form an image of the source data.

A third use of buffering is to support copy semantics for application I/O.

An example will clarify the meaning of "copy semantics." Suppose that an application has a buffer of data that it wishes to write to disk. It calls the write (1 system call, providing a pointer to the buffer and an integer specifying the number of bytes to write. After the system call returns, what happens if the application changes the contents of the buffer? With copy semantics, the version of the data written to disk is guaranteed to be the version at the time of the application

system call, independent of any subsequent changes in the application's buffer. A simple way that the operating system can guarantee copy semantics is for the write (1 system call to copy the application data into a kernel buffer before returning control to the application. The disk write is performed from the kernel buffer, so that subsequent changes to the application buffer have no effect.

Data space is common in operating systems, despite the overhead that this operation introduces, because of the clean semantics. The same effect can be obtained more efficiently by clever use of virtual-memory mapping and copy-on-write page protection.

Caching

A cache is a region of fast memory that holds copies of data. Access to the cached copy is more efficient than access to the original. For instance, the instructions of the currently running process are stored on disk, cached in physical memory, and copied again in the CPU's secondary and primary caches. The difference between a buffer and a cache is that a buffer may hold the only existing copy of a data item, whereas a cache, by definition, just holds a copy on faster storage of an item that resides elsewhere.

Spooling and Device Reservation

A spool is a buffer that holds output for a device, such as a printer, that cannot accept interleaved data streams. Although a printer can serve only one job at a time, several applications may wish to print their output concurrently, without having their output mixed together. The operating system solves this problem by intercepting all output to the printer. Each application's output is spooled to a separate disk file. When an application finishes printing, the spooling system queues the corresponding spool file for output to the printer.

The spooling system copies the queued spool files to the printer one at a time. In some operating systems, spooling is managed by a system daemon process. Some devices, such as tape drives and printers, cannot usefully multiplex the I/O requests of multiple concurrent applications. Spooling is one way that operating systems can coordinate concurrent output.

Error Handling

An operating system that uses protected memory can guard against many kinds of hardware and application errors, so that a complete system failure is not the usual result of each minor mechanical glitch. Devices and I/O transfers can fail in many ways, either for transient reasons, such as a network becoming overloaded, or for "permanent" reasons, such as a disk controller becoming defective. Operating systems can often compensate effectively for transient failures. For instance, a disk read0 failure results in a read0 retry, and a network send 0 error results in a resend 0, if the protocol so specifies. Unfortunately, if an important component experiences a permanent failure, the operating system is unlikely to recover.

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 a fast access time and disk bandwidth. The access time has two major components. 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 waiting 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. Both the access time and the bandwidth can improve by scheduling the servicing of disk I/O requests in a good order.

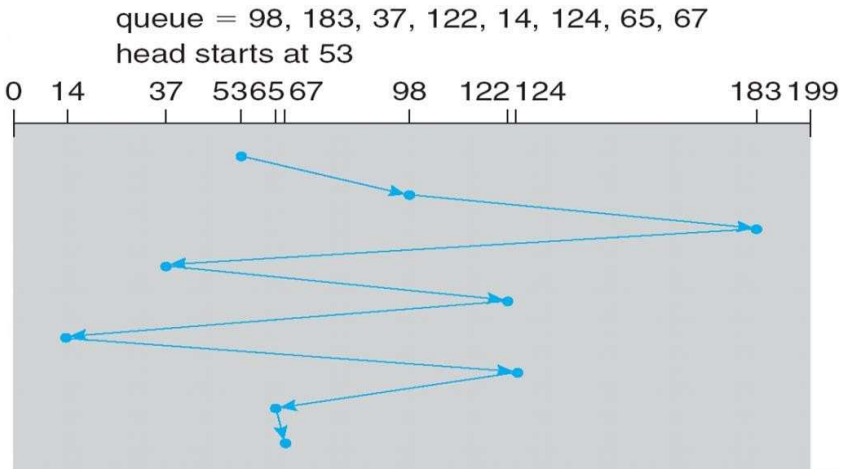
If the desired disk drive and controller are available, the request can be serviced immediately. If the drive or controller is busy, any new requests for service will be placed on the queue of pending requests for that drive. For a multiprogramming system with many processes, the disk queue may often have several pending requests. Thus, when one request is completed, the operating system chooses which pending request to service next.

1. FCFS Scheduling

The simplest form of disk scheduling is, 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. We are having a request queue of 0 to 199, and the disk head is initially at cylinder 53,

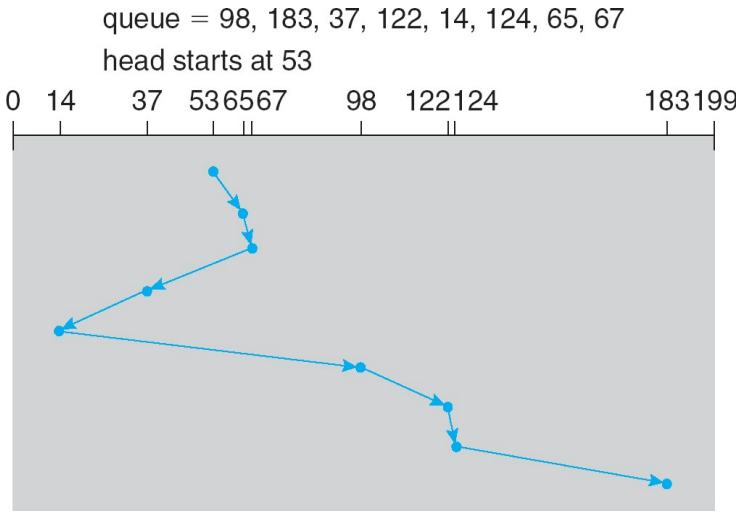


If the disk head is initially at cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 124, 65, and finally to 67, for a total head movement of 640 cylinders.
total head movement = (98-53)+ (183-98)+(183-37)+(122-37)+(122-14)+(124-14)+(124-65) +
(67-65) = 640 cylinders.

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.

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 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 98, so **37** is served next. Continuing, service the request at cylinder 14, then 98, 122, 124, and finally 183. 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.

total head movement = (53-65)+(65-67)+(67-37)+(37-14)+(98-14)+(122-98)+(124-122)
+(183-124) = 236 cylinders.

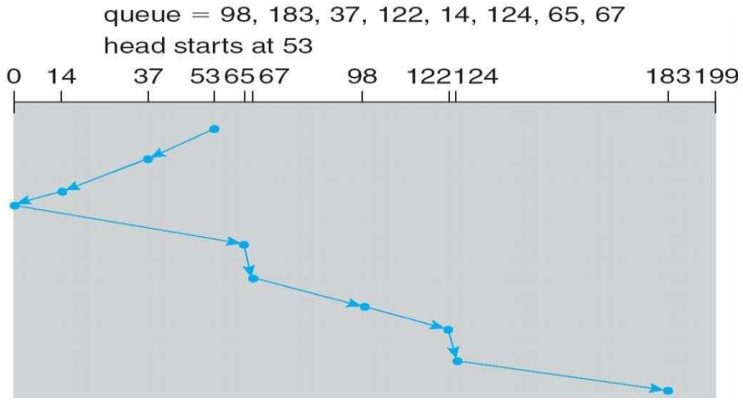
This algorithm gives a substantial improvement in performance. SSTF scheduling is essentially a form of shortest-job-first (SJF) scheduling, and, like SJF scheduling, it may cause starvation of some requests, that the requests may arrive at any time. Suppose that we have two requests in the queue, for 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. 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.

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.

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.



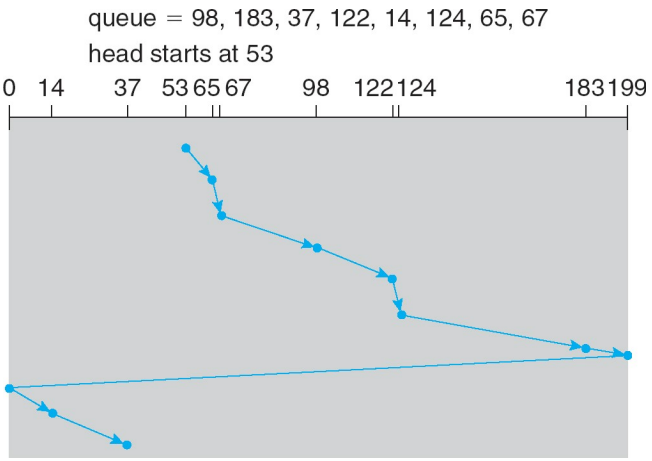
If a request arrives in the queue just in front of the head, it will be serviced almost immediately; a request arriving just behind the head will have to wait until the arm moves to the end of the disk, reverses direction, and comes back.

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.

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.

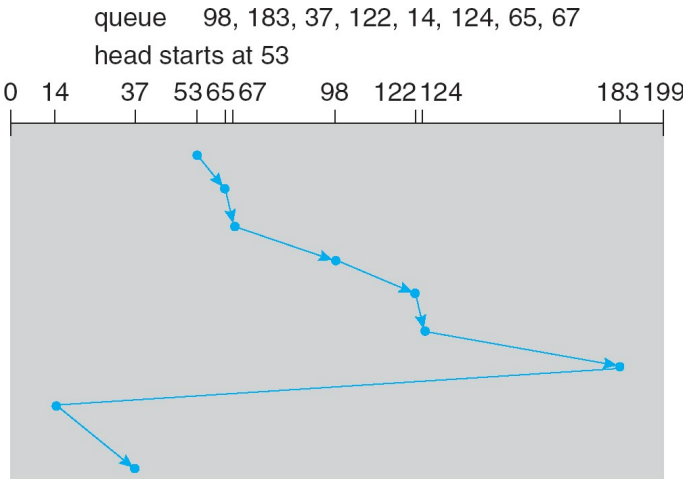
4. C-SCAN Scheduling

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



5. LOOK Scheduling

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. These versions of SCAN and C-SCAN are called **LOOK** and **C-LOOK** scheduling, because they look for a request before continuing to move in a given direction



Selection of a Disk-Scheduling Algorithm

- SSTF is common and has a natural appeal because it increases performance over FCFS. SCAN and C-SCAN perform better for systems that place a heavy load on the disk, because they are less likely to have 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 are forced to behave the same, because they have only one choice for where to move the disk head: They all behave like FCFS scheduling.
- The 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, on the other hand, may include blocks that are widely scattered on the disk, resulting in greater head movement.

RAID Structure

Disk drives have continued to get smaller and cheaper, so it is now economically feasible to attach a large number of 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 (RAID), are commonly used to address the performance and reliability issues.

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 tradeoffs and are classified into levels called RAID **levels.**

The RAID scheme consists of seven levels, zero through six. These levels do not imply a hierarchical relationship but designate different design architectures that share three common characteristics:

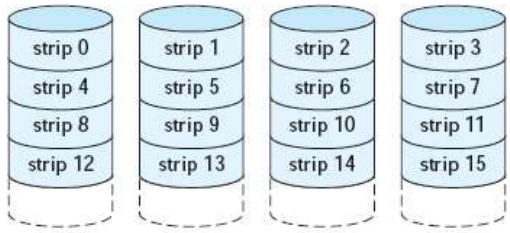
1. RAID is a set of physical disk drives viewed by the operating system as a single logical drive.
2. Data are distributed across the physical drives of an array in a scheme known as striping.
3. Redundant disk capacity is used to store parity information, which guarantees data recoverability in case of a disk failure.

The details of the second and third characteristics differ for the different RAID levels. RAID 0 and RAID 1 do not support the third characteristic.

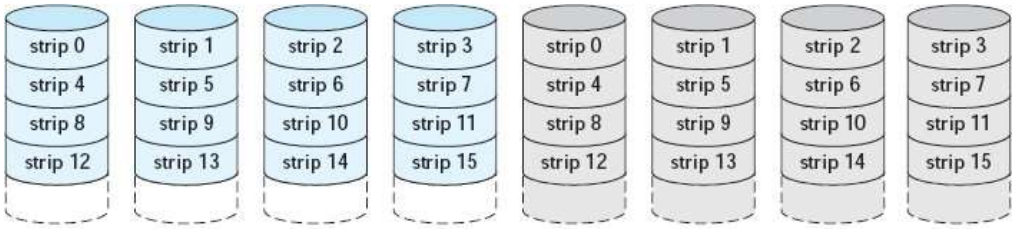
RAID Levels:

Category	Level	Description	Disks required	Data availability	Large I/O data transfer capacity	Small I/O request rate
Striping	0	Nonredundant	N	Lower than single disk	Very high	Very high for both read and write
Mirroring	1	Mirrored	$2N$	Higher than RAID 2, 3, 4, or 5; lower than RAID 6	Higher than single disk for read; similar to single disk for write	Up to twice that of a single disk for read; similar to single disk for write
Parallel access	2	Redundant via Hamming code	$N + m$	Much higher than single disk; comparable to RAID 3, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
	3	Bit-interleaved parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
Independent access	4	Block-interleaved parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 3, or 5	Similar to RAID 0 for read; significantly lower than single disk for write	Similar to RAID 0 for read; significantly lower than single disk for write
	5	Block-interleaved distributed parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 3, or 4	Similar to RAID 0 for read; lower than single disk for write	Similar to RAID 0 for read; generally lower than single disk for write
	6	Block-interleaved dual distributed parity	$N + 2$	Highest of all listed alternatives	Similar to RAID 0 for read; lower than RAID 5 for write	Similar to RAID 0 for read; significantly lower than RAID 5 for write

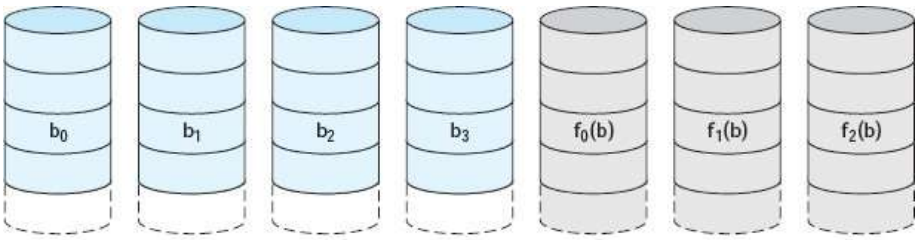
N = number of data disks; m proportional to $\log N$



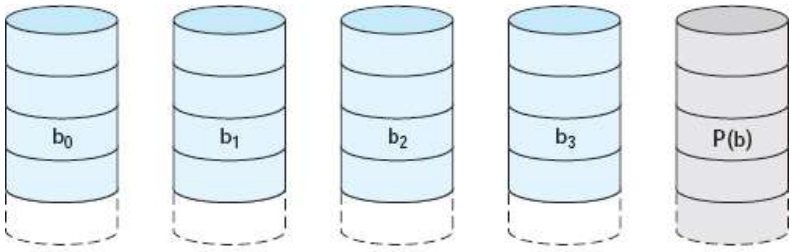
(a) RAID 0 (nonredundant)



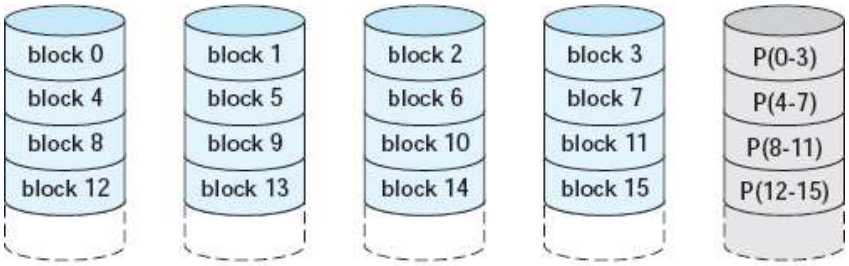
(b) RAID 1 (mirrored)



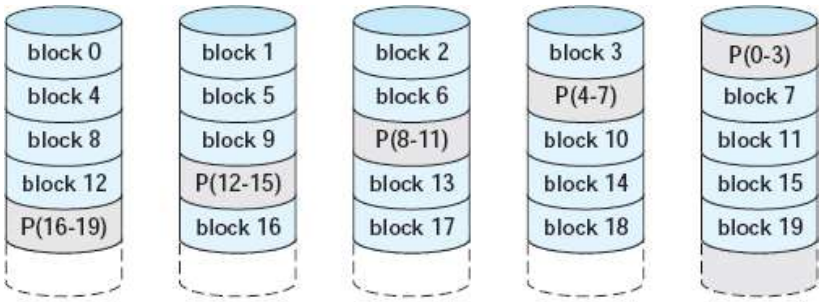
(c) RAID 2 (redundancy through Hamming code)



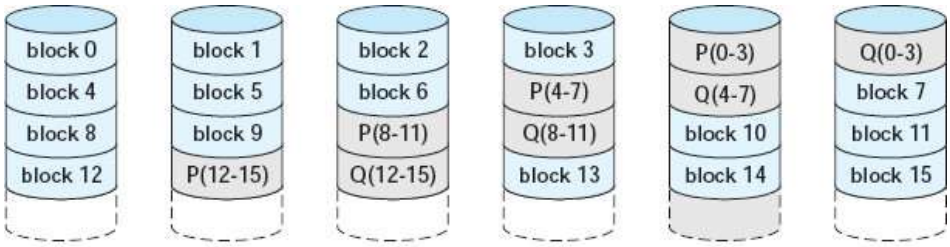
(d) RAID 3 (bit-interleaved parity)



(e) RAID 4 (block-level parity)



(f) RAID 5 (block-level distributed parity)

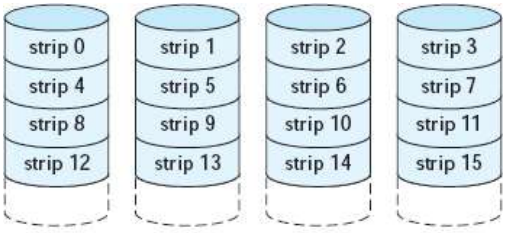


(g) RAID 6 (dual redundancy)

RAID Level 0:

For RAID 0, the user and system data are distributed across all of the disks in the array. This has a notable advantage over the use of a single large disk: If two different I/O requests are pending for two different blocks of data, then there is a good chance that the requested blocks are on different disks. Thus, the two requests can be issued in parallel, reducing the I/O queuing time. But RAID 0, as with all of the RAID levels, goes further than simply distributing the data across a disk array: the data are *striped* across the available disks. In figure all user and system data are

viewed as being stored on a logical disk. The logical disk is divided into strips; these strips may be physical blocks, sectors, or some other unit. The strips are mapped round robin to consecutive physical disks in the RAID array. A set of logically consecutive strips that maps exactly one strip to each array member is referred to as a **stripe**.



(a) RAID 0 (nonredundant)

In an *n*-disk array, the first *n* logical strips are physically stored as the first strip on each of the *n* disks, forming the first stripe; the second *n* strips are distributed as the second strips on each disk; and so on. The advantage of this layout is that if a single I/O request consists of multiple logically contiguous strips, then up to *n* strips for that request can be handled in parallel, greatly reducing the I/O transfer time.

- **RAID 0 for High Data Transfer Capacity**

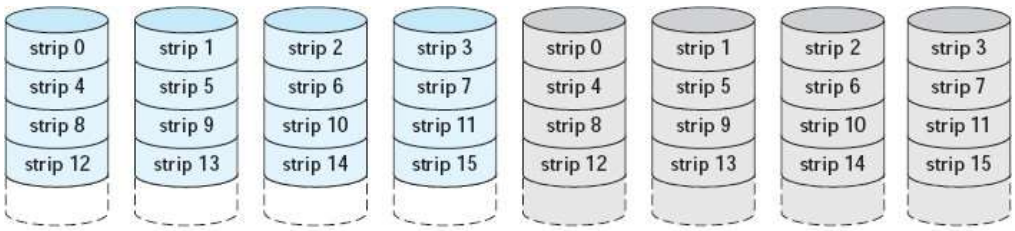
A high transfer capacity must exist along the entire path between host memory and the individual disk drives. This includes internal controller buses, host system I/O buses, I/O adapters, and host memory buses. The application must make I/O requests that drive the disk array efficiently. This requirement is met if the typical request is for large amounts of logically contiguous data, compared to the size of a strip. In this case, a single I/O request involves the parallel transfer of data from multiple disks, increasing the effective transfer rate compared to a single-disk transfer.

- **RAID 0 for High I/O Request Rate**

In a transaction-oriented environment, the user is typically more concerned with response time than with transfer rate. For an individual I/O request for a small amount of data, the I/O time is dominated by the motion of the disk heads (seek time) and the movement of the disk (rotational latency). In a transaction environment, there may be hundreds of I/O requests per second. A disk array can provide high I/O execution rates by balancing the I/O load across multiple disks. Effective load balancing is achieved only if there are typically multiple I/O requests outstanding. This, in turn, implies that there are multiple independent applications or a single transaction-oriented application that is capable of multiple asynchronous I/O requests. The performance will also be influenced by the strip size. If the strip size is relatively large, so that a single I/O request only involves a single disk access, then multiple waiting I/O requests can be handled in parallel, reducing the queuing time for each request.

RAID Level 1:

RAID 1 differs from RAID levels 2 through 6 in the way in which redundancy is achieved. In these other RAID schemes, some form of parity calculation is used to introduce redundancy, whereas in RAID 1, redundancy is achieved by the simple expedient of duplicating all the data. But in this case, each logical strip is mapped to two separate physical disks so that every disk in the array has a mirror disk that contains the same data. RAID 1 can also be implemented without data striping, though this is less common.



(b) RAID 1 (mirrored)

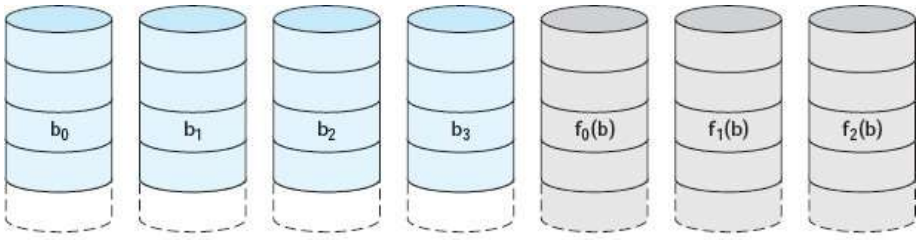
There are a number of positive aspects to the RAID 1 organization:

1. A read request can be serviced by either of the two disks that contains the requested data, whichever one involves the minimum seek time plus rotational latency.
2. A write request requires that both corresponding strips be updated, but this can be done in parallel. Thus, the write performance is dictated by the slower of the two writes (i.e., the one that involves the larger seek time plus rotational latency). However, there is no "write penalty" with RAID 1.RAID levels 2 through 6 involve the use of parity bits. Therefore, when a single strip is updated, the array management software must first compute and update the parity bits as well as updating the actual strip in question.
3. Recovery from a failure is simple. When a drive fails, the data may still be accessed from the second drive.

- The principal disadvantage of RAID 1 is the cost; it requires twice the disk space of the logical disk that it supports. Because of that, a RAID 1 configuration is likely to be limited to drives that store system software and data and other highly critical files. In these cases, RAID 1 provides real-time backup of all data so that in the event of a disk failure, all of the critical data is still immediately available.
- In a transaction-oriented environment, RAID 1 can achieve high I/O request rates if the bulk of the requests are reads. In this situation, the performance of RAID 1 can approach double of that of RAID 0.

RAID Level 2:

RAID levels 2 and 3 make use of a parallel access technique. In a parallel access array, all member disks participate in the execution of every I/O request. Typically, the spindles of the individual drives are synchronized so that each disk head is in the same position on each disk at any given time.



(c) RAID 2 (redundancy through Hamming code)

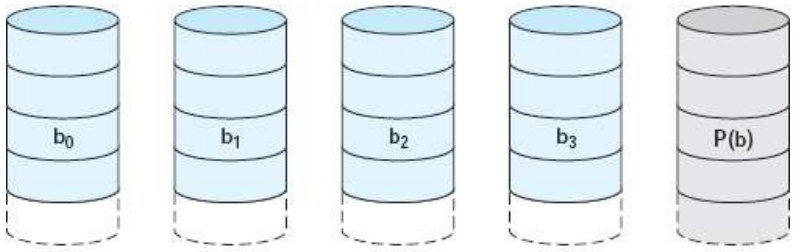
- As in the other RAID schemes, data striping is used. In the case of RAID 2 and 3, the strips are very small, often as small as a single byte or word. With RAID 2, an error-correcting code is calculated across corresponding bits on each data disk, and the bits of the code are stored in

the corresponding bit positions on multiple parity disks. Typically, a Hamming code is used, which is able to correct single-bit errors and detect double-bit errors.

- Although RAID 2 requires fewer disks than RAID 1, it is still rather costly. The number of redundant disks is proportional to the log of the number of data disks. On a single read, all disks are simultaneously accessed. The requested data and the associated error-correcting code are delivered to the array controller. If there is a single-bit error, the controller can recognize and correct the error instantly, so that the read access time is not slowed. On a single write, all data disks and parity disks must be accessed for the write operation.
- RAID 2 would only be an effective choice in an environment in which many disk errors occur. Given the high reliability of individual disks and disk drives, RAID 2 is overkill and is not implemented.

RAID Level 3:

- RAID 3 is organized in a similar fashion to RAID 2.The difference is that RAID 3 requires only a single redundant disk, no matter how large the disk array. RAID 3 employs parallel access, with data distributed in small strips. Instead of an error correcting code, a simple parity bit is computed for the set of individual bits in the same position on all of the data disks.



(d) RAID 3 (bit-interleaved parity)

- Redundancy In the event of a drive failure, the parity drive is accessed and data is reconstructed from the remaining devices. Once the failed drive is replaced, the missing data can be restored on the new drive and operation resumed. Data reconstruction is simple.

Consider an array of five drives in which X0through X3 contain data and X4 is the parity disk. The parity for the ith bit is calculated as follows:

$X4(i) = X3(i) \{ X2(i) \{ X1(i) \{ X0(i)$

where { is exclusive-OR function. Suppose that drive X1 has failed. If we add X4(i) { X1(i) to both sides of the preceding equation, we get

$X1(i) = X4(i) \{ X3(i) \{ X2(i) \{ X0(i)$

Thus, the contents of each strip of data on X1 can be regenerated from the contents of the corresponding strips on the remaining disks in the array. This principle is true for RAID levels 3 through 6.

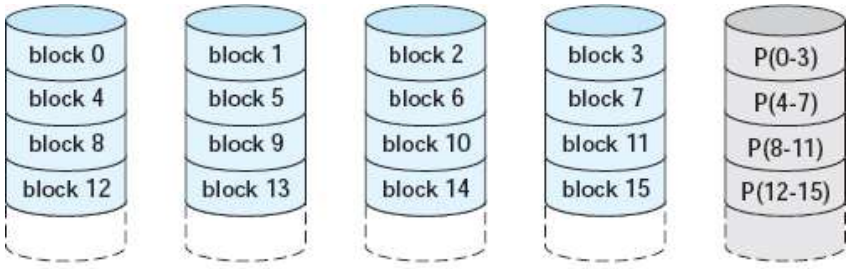
- In the event of a disk failure, all of the data are still available in what is referred to as reduced mode. In this mode, for reads, the missing data are regenerated on the fly using the exclusive-OR calculation. When data are written to a reduced RAID 3 array, consistency of the parity

must be maintained for later regeneration. Return to full operation requires that the failed disk be replaced and the entire contents of the failed disk be regenerated on the new disk.

- Because data are striped in very small strips, RAID 3 can achieve very high data transfer rates. Any I/O request will involve the parallel transfer of data from all of the data disks. For large transfers, the performance improvement is especially noticeable. On the other hand, only one I/O request can be executed at a time. Thus, in a transaction-oriented environment, performance suffers.

RAID Level 4:

RAID levels 4 through 6 make use of an independent access technique. In an independent access array, each member disk operates independently, so that separate I/O requests can be satisfied in parallel. Because of this, independent access arrays are more suitable for applications that require high I/O request rates and are relatively less suited for applications that require high data transfer rates.



(e) RAID 4 (block-level parity)

As in the other RAID schemes, data striping is used. In the case of RAID 4 through 6, the strips are relatively large. With RAID 4, a bit-by-bit parity strip is calculated across corresponding strips on each data disk, and the parity bits are stored in the corresponding strip on the parity disk. RAID 4 involves a write penalty when an I/O write request of small size is performed. Each time that a write occurs, the array management software must update not only the user data but also the corresponding parity bits. Consider an array of five drives in which X0 through X3 contain data and X4 is the parity disk. Suppose that a write is performed that only involves a strip on disk X1. Initially, for each bit *i*, we have the following relationship:

$$X4(i) = X3(i) \{ X2(i) \{ X1(i) \{ X0(i) \quad \text{--(1)}$$

After the update, with potentially altered bits indicated by a prime symbol:

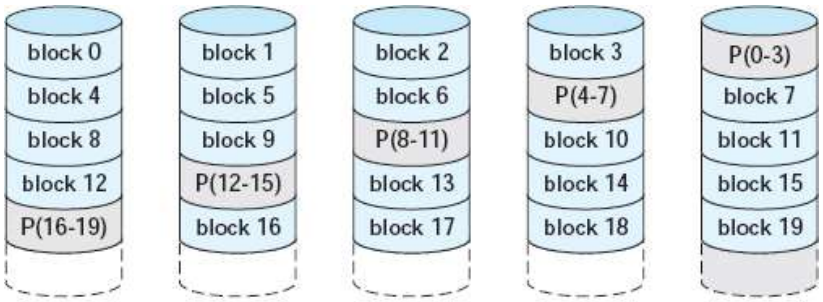
$$\begin{aligned} X4'(i) &= X3(i) \{ X2(i) \{ X1'(i) \{ X0(i) \\ &= X3(i) \{ X2(i) \{ X1'(i) \{ X0(i) \{ X1(i) \{ X1(i) \\ &= X3(i) \{ X2(i) \{ X1(i) \{ X0(i) \{ X1(i) \{ X1'(i) \\ &= X4(i) \{ X1(i) \{ X1'(i) \end{aligned}$$

The preceding set of equations is derived as follows. The first line shows that a change in X1 will also affect the parity disk X4. In the second line, we add the terms [{X1(*i*) {X1(*i*)]. Because the XOR of any quantity with itself is 0, this does not affect the equation. However, it is a convenience that is used to create the third line, by reordering. Finally, Equation (1) is used to replace the first four terms by X4(*i*). To calculate the new parity, the array management software must read the old user strip and the old parity strip. Then it can update these two strips with the new data and the newly calculated parity. Thus, each strip write involves two reads and two

writes. In the case of a larger size I/O write that involves strips on all disk drives, parity is easily computed by calculation using only the new data bits. Thus, the parity drive can be updated in parallel with the data drives and there are no extra reads or writes. In any case, every write operation must involve the parity disk, which therefore can become a bottleneck.

RAID Level 5:

RAID 5 is organized in a similar fashion to RAID 4.The difference is that RAID 5 distributes the parity strips across all disks. A typical allocation is a round-robin scheme. For an *n*-disk array, the parity strip is on a different disk for the first *n* stripes, and the pattern then repeats. The distribution of parity strips across all drives avoids the potential I/O bottleneck of the single parity disk found in RAID 4.



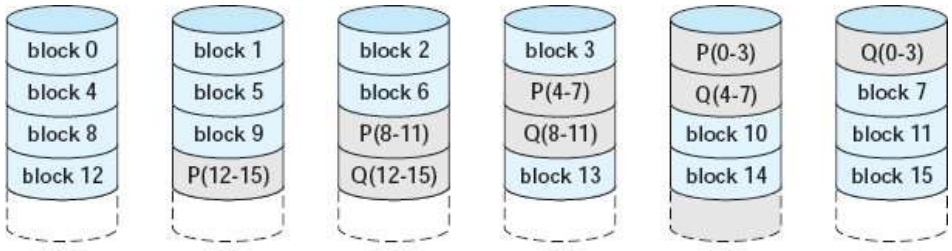
(f) RAID 5 (block-level distributed parity)

RAID Level 6:

RAID 6 was introduced in a subsequent paper by the Berkeley researchers [KATZ89]. In the RAID 6 scheme, two different parity calculations are carried out and stored in separate blocks on different disks. Thus, a RAID 6 array whose user data require *N* disks consists of $N \cdot 2$ disks.

- P and Q are two different data check algorithms. One of the two is the exclusive-OR calculation used in RAID 4 and 5. But the other is an independent data check algorithm. This makes it possible to regenerate data even if two disks containing user data fail.
- The advantage of RAID 6 is that it provides extremely high data availability.

Three disks would have to fail within the MTTR (mean time to repair) interval to cause data to be lost. On the other hand, RAID 6 incurs a substantial write penalty, because each write affects two parity blocks. Performance benchmarks [EISC07] show a RAID 6 controller can suffer more than a 30% drop in overall write performance compared with a RAID 5 implementation. RAID 5 and RAID 6 read performance is comparable.



(g) RAID 6 (dual redundancy)

File Concept

Computers can store information on several different 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 information storage.

- 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 through power failures and 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. The information in a file is defined by its creator. Many different types of information may be stored in a file-source programs, object programs, executable programs, numeric data, text, payroll records, graphic images, sound recordings, and so on.

A file has a certain defined **structure** according to its type.

- A *text* file is a sequence of characters organized into lines (and possibly pages).
- A *source* file is a sequence of subroutines and functions, each of which is further organized as declarations followed by executable statements.
- An *object* file is a sequence of bytes organized into blocks understandable by the system's linker.
- An *executable* file is a series of code sections that the loader can bring into memory and execute.

File Attributes

A file has certain other attributes, which 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 those systems that support different types.
- **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.

The information about all files is kept in the directory structure that also resides on secondary storage. Typically, the directory entry consists of the file's name and its unique identifier. The identifier in turn locates the other file attributes. It may take more than a kilobyte to record this

information for each file. In a system with many files, the size of the directory itself may be megabytes. Because directories, like files, must be nonvolatile, they must be stored on the device and brought into memory piecemeal, as needed.

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. The basic file operations are:

- **Creating a file:** Two steps are necessary to create a file. First, space in the file system must be found for the file. Second, an entry for the new file must be made in the directory. The directory entry records the name of the file and the location in the file system, and possibly other information.
- **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 location of the file. 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 directory 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. A given process is usually only reading or writing a given file, and the current operation location is kept as a per-process **current-file-position pointer.** Both the read and write operations use this same pointer, saving space and reducing the system complexity.
- **Repositioning within a file:** The directory is searched for the appropriate entry, and the current-file-position is set to a given value. Repositioning within a file does not need to 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.

Most of the file operations mentioned involve searching the directory for the entry associated with the named file. To avoid this constant searching, many systems require that an open system call be used before that file is first used actively. The operating system keeps a small table containing information about all open files (the **open-file table).** When a file operation is requested, the file is specified via an index into this table, so no searching is required. When the

file is no longer actively used, it is *closed* by the process and the operating system removes its entry in the open-file table.

The implementation of the open and close operations in a multiuser environment, such as UNIX, is more complicated. In such a system, several users may open the file at the same time. **Typically, the operating system uses two levels of internal tables: a per-process table and a system-wide table.**

The per-process table tracks all files that a process has open. Stored in this table is information regarding the use of the file by the process. For instance, the current file pointer for each file is found here, indicating the location in the file that the next read or write call will affect. Access rights to the file and accounting information can also be included. Each entry in the per-process table in turn points to a system-wide open-file table.

The system-wide table contains process-independent information, such as the location of the file on disk, access dates, and file size. Once a file is opened by one process, another process executing an open call simply results in a new entry being added to the process' open-file table, pointing to the appropriate entry in the system-wide table. Typically, the open-file table also has an *open count* associated with each file, indicating the number of processes that have the file open. Each close decreases this *count,* and when the *open count* reaches zero, the file is no longer in use, and the file's entry is removed from the open file table. In summary, several pieces of information are associated with an open file.

- **File pointer:** On systems that do not include a file offset as part of the read and write system calls, the system must track the last read-write location as a current-file-position pointer. This pointer is unique to each process operating on the file, and therefore must be kept separate from the on-disk file attributes.
- **File open count:** As files are closed, the operating system must reuse its open-file table entries, or it could run out of space in the table. Because multiple processes may open a file, the system must wait for the last file to close before removing the open-file table entry. This counter tracks the number of opens and closes and reaches zero on the last close. The system can then remove the entry.
- **Disk location of the file:** Most file operations require the system to modify data within the file. The information needed to locate the file on disk is kept in memory to avoid having to read it from disk for each operation.
- **Access rights:** Each process opens a file in an access mode. This information is stored on the per-process table so the operating system can allow or deny subsequent I/O requests.

File Types

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 character. In this way, the user and the operating system can tell from the name alone what the type of a file is. For example, in MS-DOS, a name can consist of up to eight characters followed by a period and terminated by an extension of up to three characters. The system uses the extension to indicate the type of the file and the type of operations that can be done on that file. For instance, only a file with a .corn, .exe, or .bat extension can be executed. The .corn and .exe files are two

forms of binary executable files, whereas a *.bat* file is a **batch file** containing, in ASCII format, commands to the operating system. MS-DOS recognizes only a few extensions, but application programs also use extensions to indicate file types in which they are interested. For example, assemblers expect source files to have an *.asm* extension, and the Wordperfect word processor expects its file to end with a *.wp* extension. These extensions are not required, so a user may specify a file without the extension (to save typing), and the application will look for a file with the given name and the extension it expects. Because these extensions are not supported by the operating system, they can be considered as "hints" to applications that operate on them.

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, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

File Structure

File types also may be used to indicate the internal structure of the file. The source and object files have structures that match the expectations of the programs that read them. Certain files must conform to a required structure that is understood by the operating system. For example, the operating system may require that an executable file have a specific structure so that it can determine where in memory to load the file and what the location of the first instruction is. Some

operating systems extend this idea into a set of system-supported file structures, with sets of special operations for manipulating files with those structures.

Internal File Structure

Internally, locating an offset within a file can be complicated for the operating system. All disk I/O is performed in units of one block (physical record), and all blocks are the same size. It is unlikely that the physical record size will exactly match the length of the desired logical record. Logical records may even vary in length. **Packing** a number of logical records into physical blocks is a common solution to this problem. For example, the UNIX operating system defines all files to be simply a stream of bytes. Each byte is individually addressable by its offset from the beginning (or end) of the file. In this case, the logical record is 1 byte. The file system automatically packs and unpacks bytes into physical disk blocks-say, 512 bytes per block-as necessary.

The logical record size, physical block size, and packing technique determine how many logical records are in each physical block. The packing can be done either by the user's application program or by the operating system. In either case, the file may be considered to be a sequence of blocks. All the basic I/O functions operate in terms of blocks. The conversion from logical records to physical blocks is a relatively simple software problem. Because disk space is always allocated in blocks, some portion of the last block of each file is generally wasted.

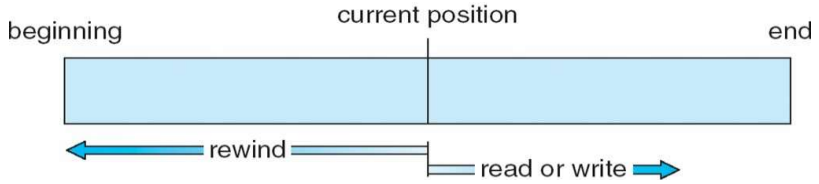
If each block were 512 bytes, then a file of 1,949 bytes would be allocated four blocks (2,048 bytes); the last 99 bytes would be wasted. The wasted bytes allocated to keep everything in units of blocks (instead of bytes) is *internal fragmentation.* All file systems suffer from internal fragmentation; the larger the block size, the greater the internal fragmentation.

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.

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.



The bulk of the operations on a file is reads and writes. A read operation reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location. Similarly, a write appends to the end of the file and advances to the end of the newly written material (the new end of file). Such a file can be reset to the beginning and, on some systems; a program may be able to skip forward or backward n records, for some integer n-perhaps only for $n = 1$. Sequential access 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).** 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. A direct-access file allows arbitrary blocks to be read or written. 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. When a query concerning a particular subject arrives, we compute which block contains the answer, and then read that block directly to provide the desired information.

For the direct-access method, the file operations must be modified to include the block number as a parameter. Thus, we have *read n,* where *n* is the block number, rather than *read next,* and *write n* rather than *write next.* An alternative approach is to retain *read next* and *write next,* as with sequential access, and to add an operation *position file to n,* where *n* is the block number. Then, to effect a *read n,* we would *position to n* and then *read next.*

The block number provided by the user to the operating system is normally a **relative block number.** A relative block number is an index relative to the beginning of the file. Thus, the first relative block of the file is 0, the next is 1, and so on, even though the actual absolute disk address of the block may be 14703 for the first block and 3192 for the second. The use of relative block numbers allows the operating system to decide where the file should be placed (called the *allocation problem,*), and helps to prevent the user from accessing portions of the file system that may not be part of his file.

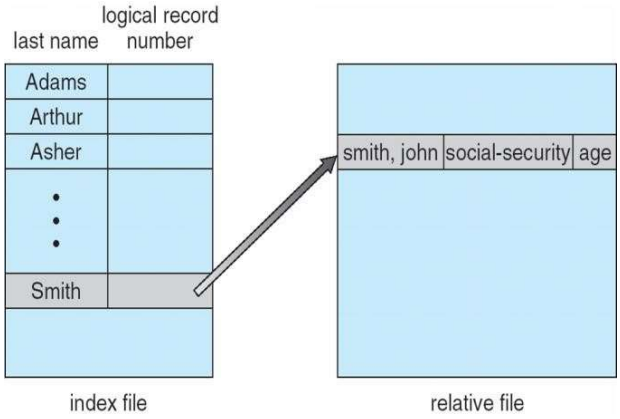
Not all operating systems support both sequential and direct access for files. Some systems allow only sequential file access; others allow only direct access. Some systems require that a file be defined as sequential or direct when it is created; such a file can be accessed only in a manner consistent with its declaration.

sequential access	implementation for direct access
reset	*cp* = 0;
read next	*read cp*; *cp* = *cp* + 1;
write next	*write cp*; *cp* = *cp* + 1;

However, it is easy to simulate sequential access on a direct-access file. If we simply keep a variable *cp* that defines our current position, then we can simulate sequential file operations, as shown in Figure. On the other hand, it is extremely inefficient and clumsy to simulate a direct-access file on a sequential-access file.

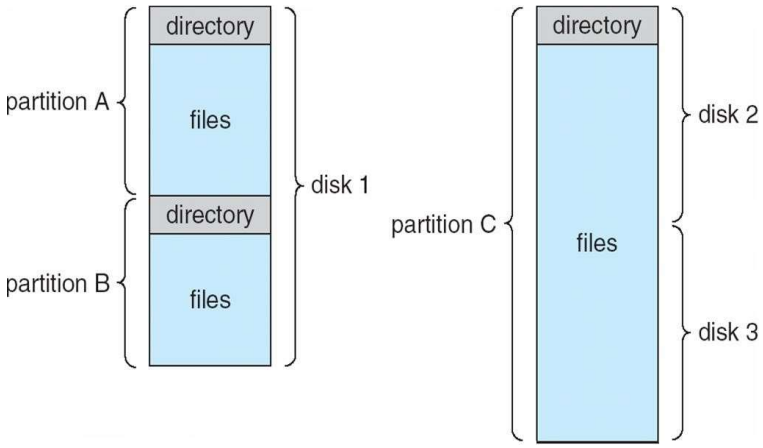
Other Access Methods

Other access methods can be built on top of a direct-access method. These methods generally involve the construction of an index for the file. The index, like an index in the back of a book, contains pointers to the various blocks. To find a record in the file, we first search the index, and then use the pointer to access the file directly and to find the desired record.



Directory Structure

The file systems of computers can be extensive. Some systems store millions of files on terabytes of disk. To manage all these data, we need to organize them. This organization is usually done in two parts. First, disks are split into one or more partitions, also known as minidisks in the IBM world or volumes in the PC and Macintosh arenas. Typically, each disk on a system contains at least one partition, which is a low-level structure in which files and directories reside. Sometimes, partitions are used to provide several separate areas within one disk, each treated as a separate storage device, whereas other systems allow partitions to be larger than a disk to group disks into one logical structure. In this way, the user needs to be concerned with only the logical directory and file structure, and can ignore completely the problems of physically allocating space for files. For this reason partitions can be thought of as virtual disks. Partitions can also store multiple operating systems, allowing a system to boot and run more than one.



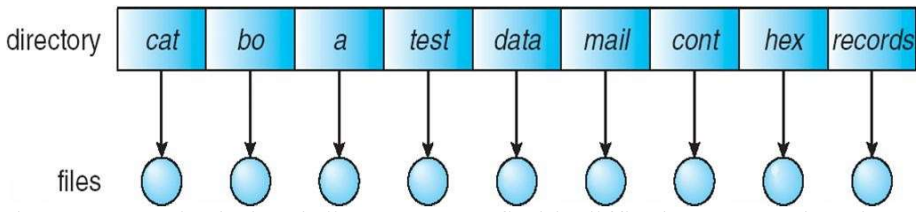
Second, each partition contains information about files within it. This information is kept in entries in a **device directory** or **volume table of contents.** The device directory (more commonly known simply as a *directory)* records information-such as name, location, size, and type-for all files on that partition.

The directory can be viewed as a symbol table that translates file names into their directory entries. The directory itself can be organized in many ways. We want to be able to insert entries, to delete entries, to search for a named entry, and to list all the entries in the directory.
Following common operations are to be performed on 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 between 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 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, the name must be changeable 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:** We may 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. This saving often consists of copying all files to magnetic tape. This technique provides a backup copy in case of system failure or if the file is simply no longer in use. In this case, 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 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,* then the unique-name rule is violated.

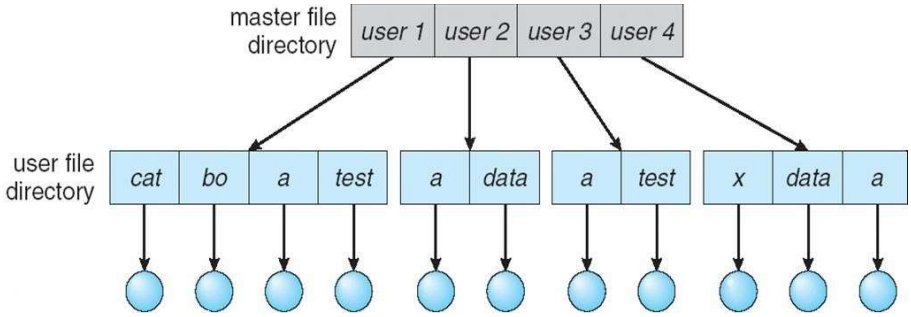


Even a single user on a single-level directory may find it difficult to remember the names of all the files, as the number of files increases. It is not uncommon for a user to have hundreds of files on one computer system and an equal number of additional files on another system. In such an environment, keeping track of so many files is a daunting task.

Two-Level Directory

A single-level directory often leads to confusion of file names between different users. The standard solution is to create a *separate* directory for each user. In the two-level directory structure, each user has her own **user file directory (UFD).** Each UFD has a similar structure, but 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.



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.

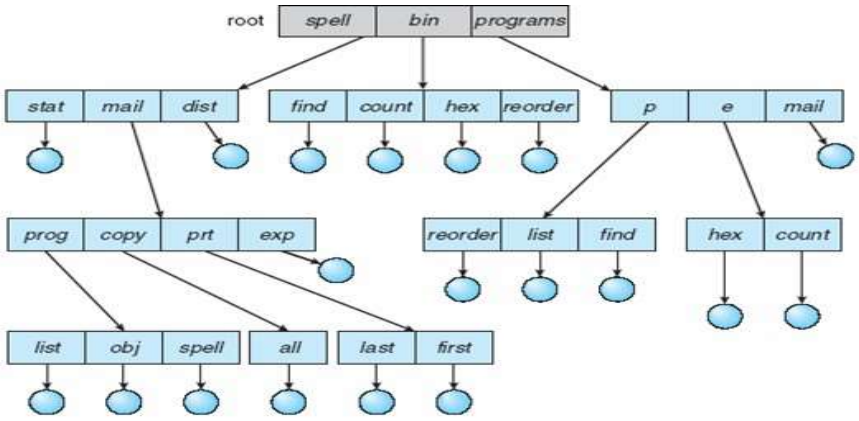
To create a file for a user, the operating system searches only that user's UFD to ascertain whether another file of that name exists. To delete a file, the operating system confines its search to the local UFD; thus, it cannot accidentally delete another user's file that has the same name.

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 the MFD. The execution of this program might be restricted to system administrators.

Although the two-level directory structure solves the name-collision problem, it still has disadvantages. This structure effectively isolates one user from another. This 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

The natural generalization of two-level directory is to extend the directory structure to a tree or arbitrary height. This generalization allows users to create their own sub directories and to organize their files accordingly. The tree has a root directory. Each file in the system has a unique path name. A path name is the path from the root, through all the subdirectories, to a specified file.



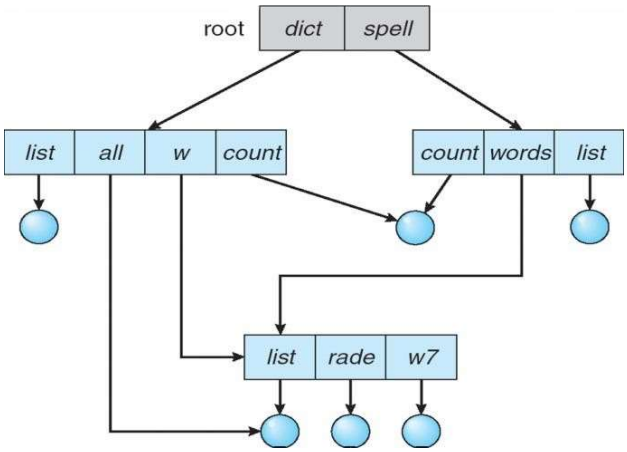
Path names can be of two types: *absolute* path names or *relative* path names. An **absolute path name** begins at the root and follows a path down to the specified file, giving the directory names on the path. A **relative path name** defines a path from the current directory. For example, in the tree-structured file system of Figure, if the current directory is *root/spell/mail,* then the relative path name *prt/first* refers to the same file as does the absolute path name *root/spell/mail/prt/fivst.*

With a tree-structured directory system, users can access, in addition to their files, the files of other users. For example, user B can access files of user A by specifying their path names. User B can specify either an absolute or a relative path name. Alternatively, user B could change her current directory to be user A's directory, and access the files by their file names. Some systems also allow users to define their own search paths. In this case, user B could define her search path to be (1) her local directory, (2) the system file directory, and **(3)** user A's directory, in that order. As long as the name of a file of user A did not conflict with the name of a local file or system file, it could be referred to simply by its name.

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. The common subdirectory should be shared. A shared directory or file will exist in the file system in two (or more) places at once. A tree structure prohibits the sharing of files or directories. An **acyclic graph** allows directories to have shared subdirectories and files.

The *same* file or subdirectory may be in two different directories. An acyclic graph, that is, a graph with no cycles, is a natural generalization of the tree structured directory scheme. 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.



When people are working as a team, all the files they want to share may be put into one directory. The UFDs of all the team members would each contain this directory of shared files as a subdirectory. Even when there is a single user, his file organization may require that some files

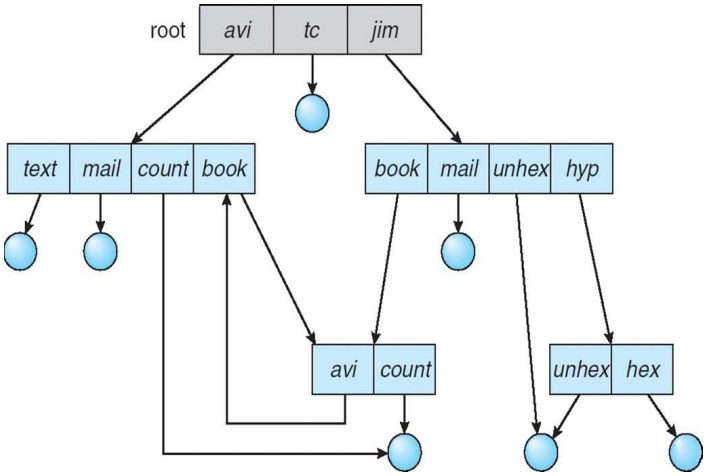
be put into different subdirectories. For example, a program written for a particular project should be both in the directory of all programs and in the directory for that project.

An acyclic-graph directory structure is more flexible than is a simple tree structure, but it is also more complex. Several problems must be considered carefully. A file may now have multiple absolute path names. Consequently, distinct file names may refer to the same file. This situation is similar to the aliasing problem for programming languages. If we are trying to traverse the entire file system-to find a file, to accumulate statistics on all files, or to copy all files to backup storage-this problem becomes significant, since we do not want to traverse shared structures more than once.

Another problem involves deletion. When can the space allocated to a shared file be deallocated and reused? One possibility is to remove the file whenever anyone deletes it, but this action may leave dangling pointers to the now-nonexistent file. Worse, if the remaining file pointers contain actual disk addresses, and the space is subsequently reused for other files, these **dangling pointers** may point into the middle of other files. In a system where sharing is implemented by symbolic links, this situation is somewhat easier to handle. The deletion of a link does not need to affect the original file; only the link is removed. If the file entry itself is deleted, the space for the file is deallocated, leaving the links dangling. We can search for these links and remove them also, but unless a list of the associated links is kept with each file, this search can be expensive. Alternatively, we can leave the links until an attempt is made to use them. At that time, we can determine that the file of the name given by the link does not exist, and can fail to resolve the link name; the access is treated just like any other illegal file name.

General Graph Directory

One 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 to an existing tree-structured directory, the tree structure is destroyed, resulting in a simple graph structure.



File Sharing

In a multiuser system, there is almost always a requirement for allowing files to be shared among a number of users.

– **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 either can allow a user to access the files of other users by default, or it may require that a user specifically grant access to the files.
- These are the issues of access control and protection,
- To implement sharing and protection, the system must maintain more file and directory attributes than on a single-user system.
- Most systems have evolved to the concepts of file/directory *owner* (or *user)* and *group.*
- The owner is the user who may change attributes, grant access, and has the most control over the file or directory. The group attribute of a file is used to define a subset of users who may share access to the file. Most systems implement owner attributes by managing a list of user names and associated **user identifiers (user IDS).**
- When a user logs in to the system, the authentication stage determines the appropriate user ID for the user. That user ID is associated with all of the user's processes and threads. When they need to be user readable, they are translated back to the user name via the user name list.

– **Remote File Systems**

- Networking allows the sharing of resources spread within 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, file-sharing methods have changed.
- In the first implemented method, users manually transfer files between machines via programs like ftp.
- The second major method is a **distributed file system (DFS)** in which remote directories are visible from the local machine.
- In some ways, the third method, the **World Wide 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. The World Wide Web uses anonymous file exchange almost exclusively.

➢ **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 wanting 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. Files are usually specified on a partition or subdirectory level. A server can serve multiple clients, and

a client can use multiple servers, depending on the implementation details of a given client-server facility.

→ **Distributed Information Systems**

To ease the management of client-server services, **distributed information systems,** also known as **distributed naming services,** have been devised to provide a unified access to the information needed for remote computing. **Domain name system (DNS)** provides host-name-to-network-address translations for the entire Internet (including the World Wide Web). Before DNS was invented and became widespread, files containing the same information were sent via email or ftp between all networked hosts.

→ **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, or host adapter failure. User or systems-administrator failure can also cause files to be lost, or entire directories or partitions to be deleted. Many of these failures would cause a host to crash and an error condition to be displayed, and require human intervention to repair. Some failures do not cause loss of data or loss of availability of data. **Redundant arrays of inexpensive disks (RAID)** can prevent the loss of a disk from resulting in the loss of data.

– **Consistency Semantics**

Consistency semantics is an important criterion for evaluating any file system that supports file sharing. It is a characterization of the system that specifies the semantics of multiple users accessing a shared file simultaneously. In particular, these semantics should specify when modifications of data by one user are observable by other users. The semantics are typically implemented as code with the file system.

– **UNIX Semantics**

The UNIX file system uses the following consistency semantics:

- Writes to an open file by a user are visible immediately to other users that have this file open at the same time.
- One mode of sharing allows users to share the pointer of current location into the file. Thus, the advancing of the pointer by one user affects all sharing users. Here, a file has a single image that interleaves all accesses, regardless of their origin.
- In the UNIX semantics a file is associated with a single physical image that is accessed as an exclusive resource. Contention for this single image results in user processes being delayed.

– **Session Semantics**

The Andrew file system (AFS) uses the following consistency semantics:

- Writes to an open file by a user are not visible immediately to other users that have the same file open simultaneously.

- Once a file is closed, the changes made to it are visible only in sessions starting later. Already open instances of the file do not reflect these changes.
- According to these semantics, a file may be associated temporarily with several (possibly different) images at the same time. Consequently, multiple users are allowed to perform both read and write accesses concurrently on their image of the file, without delay. Almost no constraints are enforced on scheduling accesses.

- **Immutable-Shared-Files Semantics**

A unique approach is that of **immutable shared files.** Once a file is declared as shared by its creator, it cannot be modified. An immutable file has two key properties:
Its name may not be reused and its contents may not be altered. Thus, the name of an immutable file signifies that the contents of the file are fixed, rather than the file being a container for variable information.
The implementation of these semantics in a distributed system is simple, because the sharing is disciplined (read-only).

Protection

When information is kept in a computer system, we want to keep it safe from physical damage (reliability) and improper access (protection).
Protection can be provided in many ways. For a small single-user system, we might provide protection by physically removing the floppy disks and locking them in a desk drawer or file cabinet. In a multiuser system, however, other mechanisms are needed.

- **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. 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, or 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. In this case, a user with read access can also cause the file to be copied, printed, and so on.

- **Access Control**

The most common approach to the protection problem is to make access dependent on the identity of the user. Various users may need different types of access to a file or directory. The

most general scheme to implement identity-dependent access is to associate with each file and directory an access-control list **(ACL)** specifying the user name 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 needs to 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.

– **Other Protection Approaches**

- Another approach to the protection problem is to associate a password with each file. Just as access to the computer system is often controlled by a password, access to each file can be controlled by a password. If the passwords are chosen randomly and changed often, this scheme may be effective in limiting access to a file to only those users who know the password. This scheme, however, has several disadvantages. First, the number of passwords that a user needs to remember may become large, making the scheme impractical. Secondly, if only one password is used for all the files, then, once it is discovered, all files are accessible.
- In a multilevel directory structure, we need to protect not only individual files, but also collections of files in a subdirectory; that is, we need to provide a mechanism for directory protection. The directory operations that must be protected are somewhat different from the file operations. We want to control the creation and deletion of files in a directory.
- Thus, listing the contents of a directory must be a protected operation. Therefore, if a path name refers to a file in a directory, the user must be allowed access to both the directory and the file. In systems where files may have numerous path names (such as acyclic or general graphs), a given user may have different access rights to a file, depending on the path name used.