UNIT-2

File Concept:- 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, which depends on 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'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:- The operating system can provide system calls to create, write, read, reposition, delete, and truncate files.
Creating a file. Two steps are necessary to create a file. First, space in the file system must be found for the file. We discuss how to allocate space for the file in Chapter 11. Second, an entry for the new file must be made in the directory.

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 **current file-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 bv other files, and erase the directory entry.

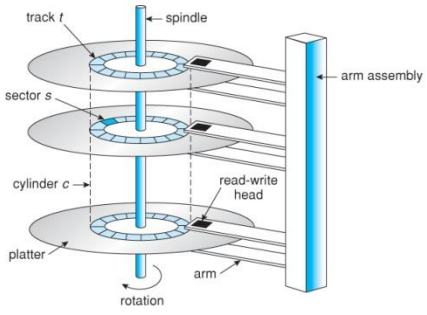
Truncating a file. The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged—except for file length—but lets the tile be reset to length zero and its file space released.

File Types:-

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

Figure 10.2 Common file types.

Disk Organization



Flowchart of Disk


Start
Introduce Disk Storage
Explain Disk Capacity
Input Disk Specifications
Calculate Disk Capacity
Display Disk Capacity
Explain Disk Performance
Input Rotational Latency and seek Time
Calculate Average Access Time
DIsplay Average Access Time
Explain Disk Utilization
Input Data Stored and Total Capacity
Calculate Disk Utilization
Display Disk Utilization
Stop

Numericals

Q1. Consider a disk with 4 platters, 2 surfaces per platter, 1000 tracks per surface, 50 sectors per track, and 512 bytes per sector. What is the disk capacity?

Solution – To calculate the disk capacity, we can follow these steps:

Step 1 – Calculate the total number of tracks on the disk–

$$Total tracks = No.of platters * No.of surfaces per platter * No.of tracks per surface$$

$$Total tracks = 4 * 2 * 1000 = 8000$$

Step 2 – Calculate the total number of sectors on the disk–

$$Total sectors = Total tracks * Number of sectors per track$$

Step 3 – Calculate the total storage capacity of the disk

$$Total capacity = Total sectors * Number of bytes per sector$$

$$Total capacity = 400,000 * 512 = 204,800,000 bytes$$

Therefore, the disk capacity is **204,800,000 bytes** or approximately **204.8 megabytes.**

Q2 – A storage system consists of 10 disk drives, each with 6 platters, 4 surfaces per platter, 2000 tracks per surface, 100 sectors per track, and 512 bytes per sector. What is the total capacity of the storage system? If 7 disk drives are currently filled with data, what is the overall disk utilization percentage?

Solution – To calculate the total capacity of the storage system, we can follow these steps–

Step 1 – Calculate the total number of tracks on each disk drive–

$$Total tracks = No.of platters * No.of surfaces per platter * No.of tracks per surface$$

$$Total tracks = 6 * 4 * 2000 = **48,000** tracks per disk drive$$

Step 2 – Calculate the total number of sectors on each disk drive –

$$Total sectors = Total tracks * Number of sectors per track$$

$$Total sectors = 48,000 * 100 = **4,800,000** sectors per disk drive$$

Step 3 – Calculate the total storage capacity of each disk drive–

$$Total capacity = Total sectors * Number of bytes per sector$$

$$Total capacity = 4,800,000 * 512 = **2,457,600,000** bytes per disk drive$$

Step 4 – Calculate the total capacity of the storage system–

$$Total capacity = Total capacity per disk drive * Number of disk drives$$

Total capacity = 2,457,600,000 * 10 = **24,576,000,000 bytes**or approximately **24.58 terabytes**

Therefore, the total capacity of the storage system is approximately **24.58 terabytes.**

Now, let's calculate the overall disk utilization percentage –

Step 1 – Calculate the total capacity of the filled disk drives –

$$Total filled capacity = Total capacity per disk drive * Number of filled disk drives$$

$$Total filled capacity = 2,457,600,000 * 7 = **17,202,200,000 bytes**$$

Step 2 – Calculate the overall disk utilization percentage:

Overall disk utilization % = (Total filled capacity / Total capacity of the storage system) * 100

$$Overall disk utilization percentage = (17,202,200,000 / 24,576,000,000) * 100$$

$$Overall disk utilization percentage ≈ **70.08%** $$

Therefore, the overall disk utilization percentage of the storage system, considering 7 filled disk drives, is **approximately 70.08%.**

Access Methods:-

There are several ways that the information in the file can be accessed. Some systems provide only one access method for files. On other systems, many different access methods are supported, and choosing the right one for a particular application is a major design problem.

Sequential Access Information in the file is processed in order, one record after the other. This is by far the most common mode of access of files. For example, computer editors usually access files in this fashion.

A read operation reads the next portion of the file and automatically advances the file pointer. Similarly, a write appends to the end of the file and the file pointer. Similarly, a write appends to the end of the file and the file pointer. Similarly, a write appends to the end of the end of the file and advances to the end of the newly written material (the new end of file).

Direct Access Direct access is based on a disk model of a file. For direct access, the file is viewed as a numbered sequence of block or records. A direct-access file allows arbitrary blocks to be read or written. Thus, after block 18 has been read, block 57 could be next, and then block 3. There are no restrictions on the order of reading and writing for a direct access file. Direct access files are of great use for intermediate access to large amounts of information.

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 that "write next". An alternative approach is to retain "read next" and "write next" and to add an operation; "position file to n" where n is the block number. Then, to effect a "read n", we would issue the commands "position to n" and then "read next".

Other Access Methods Other access methods can be built on top of a direct-access method. These additional methods generally involve the construction of an index for a file. The index contains pointers to the various blocks. To find an entry in the file, the index is searched first and the pointer is then used to access the file directly to find the desired entry.

With a large file, the index itself may become too large to be kept in memory. One solution is to create an index for the index file. The primary index file would contain pointers to secondary index files, which would point to the actual data items.

Free-Space Management:-

Since disk space is limited, we need to reuse the space from deleted files for new files, if possible. (Write-once optical disks only allow one write to any given sector, and thus such reuse is not physically possible.) To keep track of free disk space, the system maintains a **free-space list.** The free-space list records *all free* disk blocks—those not allocated to some file or directory. To create a

file, we search the free-space list for the required amount of space and allocate that space to the new file. This space is then removed from the free-space list. When a file is deleted, its disk space is added to the free-space list. The free-space list, despite its name, might not be implemented as a list, as we discuss next.

Bit-Vector

Frequently, the free-space list is implemented as a bit map or bit vector. Each block is represented by a 1 bit. If the block is free, the bit is 0; if the block is allocated, the bit is 1. For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free, and the rest of the blocks are allocated. The free-space bit map would be: 1100001100000011100111111100011111…

Linked List

Another approach is to link all the free disk blocks together, keeping a pointer to the first free block. This block contains a pointer to the next free disk block, and so on. In the previous example, a pointer could be kept to block 2, as the first free block. Block 2 would contain a pointer to block 3, and so on.

Grouping

A modification of the free-list approach is to store the addresses of n free blocks in the first free block. The first n-1 of these are actually free. The last one is the disk address of another block containing addresses of another n free blocks.

Counting

Another approach is to take advantage of the fact that, generally, several contiguous blocks may be allocated or freed simultaneously, particularly when contiguous allocation is used. Thus, rather than keeping a list of free disk addresses, the address of the first free block is kept and the number n of free contiguous blocks that follow the first block.

Allocation Methods:-

The direct-access nature of disks allows us flexibility in the implementation of files, in almost every case, many files are stored on the same disk. The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly. Three major methods of allocating disk space are in wide use: contiguous, linked, and indexed. Each method has advantages and disadvantages. Some systems (such as Data General's RDOS for its Nova line of computers) support all three. More commonly, a system vises one method for all files within a file system type.

Contiguous allocation:- Contiguous allocation requires that each file occupy a set of contiguous blocks on the disk. Disk addresses define a linear ordering on the disk. With this ordering, assuming that only one job is accessing the disk, accessing block *b* + 1 after block *b* normally requires no head movement. When head movement is needed (from the last sector of one cylinder to the first sector of the next cylinder), the head need only move from one track to the next. Thus, the number of disk seeks required for accessing contiguously allocated files is minimal, as is seek time when a seek is finally needed. The IBM VM/CMS operating system uses contiguous allocation because it provides such good performance. Contiguous allocation of a file is defined by the disk address and length (in block units) of the first block. If the file is *n* blocks long and starts at location *b*, then it occupies blocks *b*, *b* + 1, *b* + *2*, ..., *b* + *n* — 1. The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file (Figure 11.5).

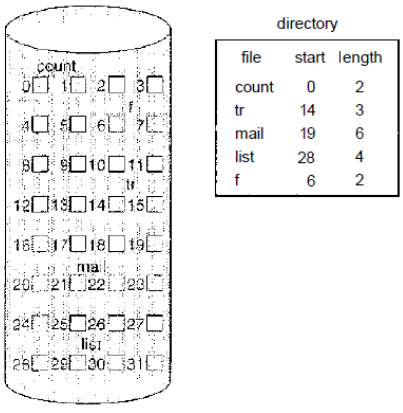


Figure 11.5 Contiguous allocation of disk space.

Accessing a file that has been allocated contiguously is easy. For sequential access, the file system remembers the disk address of the last block referenced and, when necessary, reads the next block. For direct access to block /' of a file that starts at block *b*, we can immediately access block *b* + *i*. Thus, both sequential and direct access can be supported by contiguous allocation.

Linked allocation:-

Linked allocation solves all problems of contiguous allocation. With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. For example, a file of five blocks might start at block 9 and continue at block 16, then block 1, then block 10, and finally block 25 (Figure 11.6). Each block contains a pointer to the next block. These pointers are not made available to the user. Thus, if each block is 512 bytes in size, and a disk address (the pointer) requires 4 bytes, then the user sees blocks of 508 bytes.

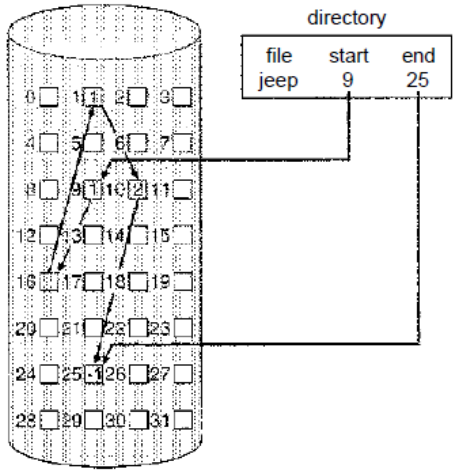


Figure 11.6 Linked allocation of disk space.

Linked allocation does have disadvantages, however. The major problem is that it can be used effectively only for sequential-access files. To find the ith block of a file, we must start at the beginning of that file and follow the pointers until we get to the ith block. Each access to a pointer requires a disk read, and some require a disk seek. Consequently, it is inefficient to support a direct-access capability for linked-allocation files.

Another disadvantage is the space required for the pointers. If a pointer requires 4 bytes out of a 512-byte block, then 0.78 percent of the disk is being used for pointers, rather than for information. Each file requires slightly more space than it would otherwise.

Indexed Allocation:-

Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation. However, in the absence of a FAT, linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and must be retrieved in order. **Indexed allocation** solves this problem by bringing all the pointers together into one location: the **index block.**

Each file has its own index block, which is an array of disk-block addresses. The /"' entry in the index block points to the /"' block of the file. The directory contains the address of the index block (Figure 11.8). To find and read the /th block, we use the pointer in the /"' index-block entry. This scheme is similar to the paging scheme described in Section 8.4.

When the file is created, all pointers in the index block are set to *nil.* When the ith block is first written, a block is obtained from the free-space manager, and its address is put in the zth index-block entry.

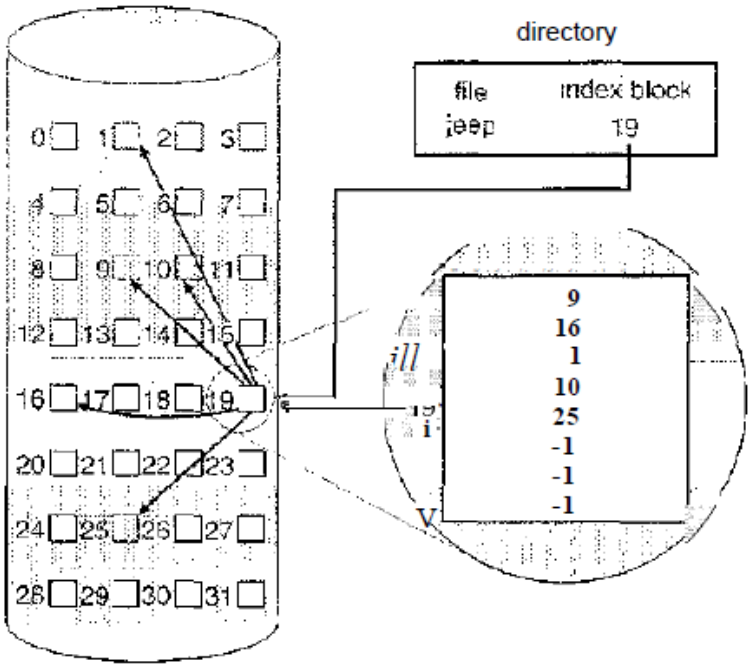


Figure 11.8 Indexed allocation of disk space.

Directory systems:-

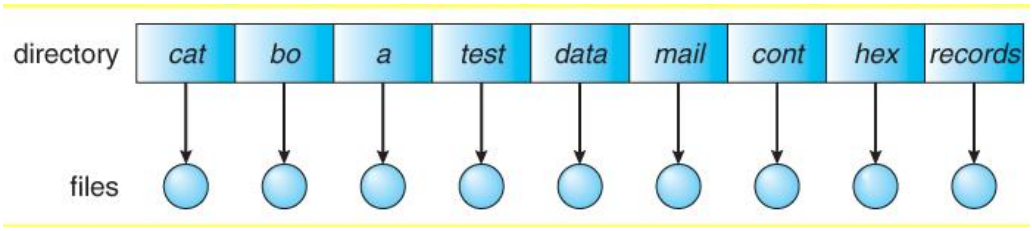
Directory Overview

Directory operations to be supported include:

- Search for a file
- Create a file - add to the directory
- Delete a file - erase from the directory
- List a directory - possibly ordered in different ways.
- Rename a file - may change sorting order
- Traverse the file system.

Single-Level Directory

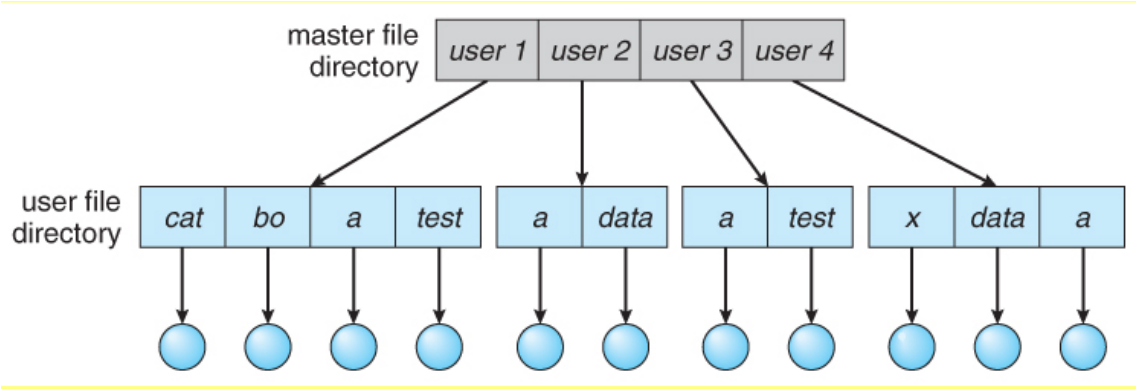
- Simple to implement, but each file must have a unique name.



Two-Level Directory

- Each user gets their own directory space.
- File names only need to be unique within a given user's directory.

- A master file directory is used to keep track of each users directory, and must be maintained when users are added to or removed from the system.
- A separate directory is generally needed for system (executable) files.

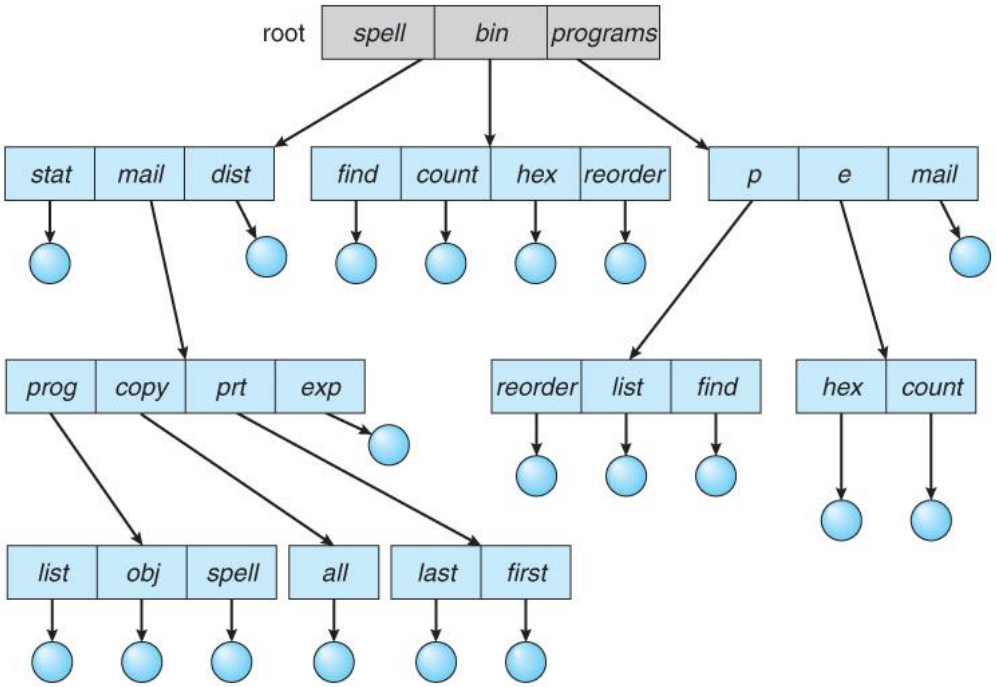


Tree-Structured Directories

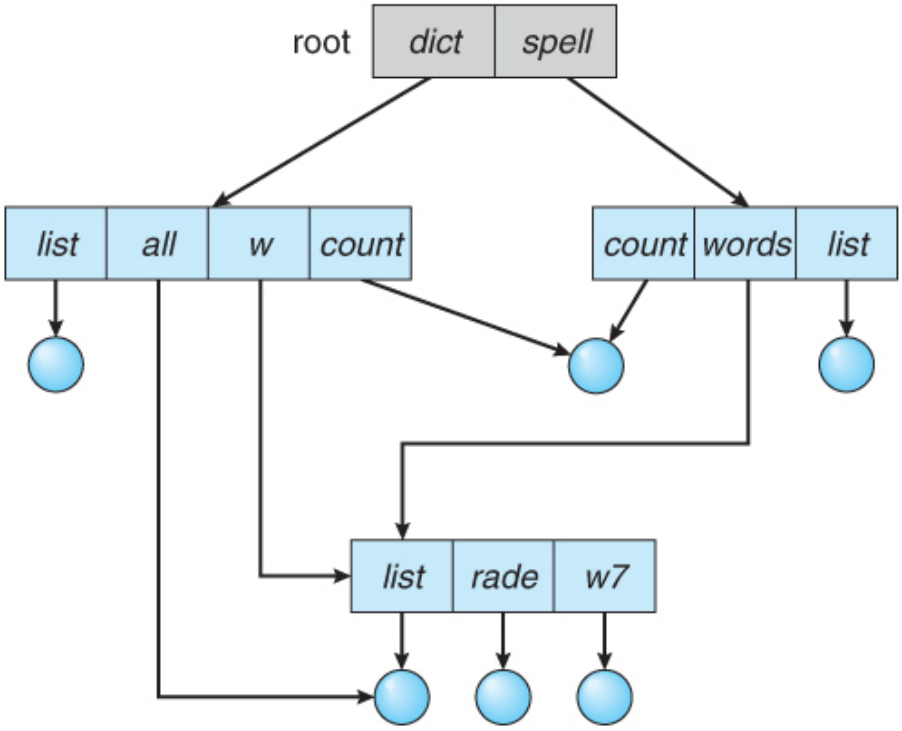
- An obvious extension to the two-tiered directory structure, and the one with which we are all most familiar.
- Each user / process has the concept of a ***current directory*** from which all (relative) searches take place.
- Files may be accessed using either absolute pathnames (relative to the root of the tree) or relative pathnames (relative to the current directory.)
- Directories are stored the same as any other file in the system, except there is a bit that identifies them as directories, and they have some special structure that the OS understands.

Acyclic-Graph Directories

- When the same files need to be accessed in more than one place in the directory structure (e.g. because they are being shared by more than one user / process), it can be useful to provide an acyclic-graph structure. (Note the ***directed*** arcs from parent to child.)



Tree-Structured Directories



Acyclic-Graph Directories

File-System Structure:-

Disks provide the bulk of secondary storage on which a file system is maintained. They have two characteristics that make them a convenient medium for storing multiple files:
1. A disk can be rewritten in place; it is possible to read a block from the disk, modify the block, and write it back into the same place.
2. A disk can access directly any given block of information it contains. Thus, it is simple to access any file either sequentially or randomly, and switching from one file to another requires only moving the read-write heads and waiting for the disk to rotate.

Rather than transferring a byte at a time, to improve I/O efficiency, I/O transfers between memory and disk are performed in units of *blocks*. Each block has one or more sectors. Depending on the disk drive, sectors vary from 32 bytes to 4,096 bytes; usually, they are 512 bytes.

To provide efficient and convenient access to the disk, the operating system imposes one or more file systems to allow the data to be stored, located, and retrieved easily. A file system poses two quite different design problems. The first problem is defining how the file system should look to the user. This task involves defining a file and its attributes, the operations allowed on a file, and the directory structure for organizing files. The second problem is creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices.

The file system itself is generally composed of many different levels. The structure shown in Figure 11.1 is an example of a layered design. Each level in the design uses the features of lower levels to create new features for use by higher levels.

The lowest level, the *I/O control*, consists of **device drivers** and interrupt handlers to transfer information between the main memory and the disk system. A device driver can be thought of as a translator. Its input consists of high-level commands such as "retrieve block 123." Its output consists of lowlevel, hardware-specific instructions that are used by the hardware controller, which interfaces the I/O device to the rest of the system. The device driver usually writes specific bit patterns to special locations in the I/O controller's memory to tell the controller which device location to act on and what actions to take. The details of device drivers and the I/O infrastructure are covered in Chapter 13.

The **basic file system** needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk. Each physical block is identified by its numeric disk address (for example, drive 1, cylinder 73, track 2, sector 10).

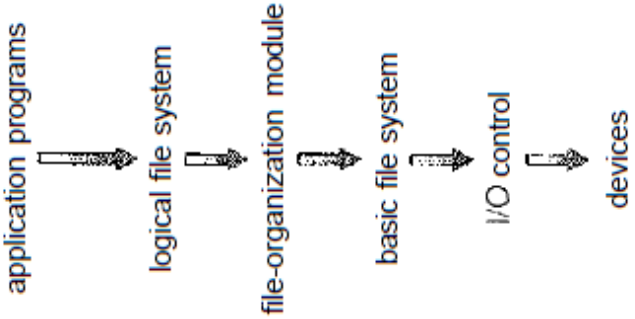


Figure 11.1 Layered file system.

The file-organization module knows about files and their logical blocks, as well as physical blocks. By knowing the type of file allocation used and the location of the file, the file-organization module can translate logical block addresses to physical block addresses for the basic file system to transfer. Each file's logical blocks are numbered from 0 (or 1) through *N*. Since the physical blocks containing the data usually do not match the logical numbers, a translation is needed to locate each block. The file-organization module also includes the free-space manager, which tracks unallocated blocks and provides these blocks to the file-organization module when requested.

Finally, the logical file system manages metadata information. Metadata includes all of the file-system structure except the actual *data* (or contents of the files). The logical file system manages the directory structure to provide the file organization module with the information the latter needs, given a symbolic file name. It maintains file structure via file-control blocks. A **file-control block** (FCB) contains information about the file, including ownership, permissions, and location of the file contents. The logical file system is also responsible for protection and security, as was discussed in Chapter 10 and will be further discussed in Chapter 14.

When a layered structure is used for file-system implementation, duplication of code is minimized. The I/O control and sometimes the basic file-system code can be used by multiple file systems. Each file system can then have its own logical file system and file-organization modules.

File-System implementation:-

Several on-disk and in-memory structures are used to implement a file system. These structures vary depending on the operating system and the file system, but some general principles apply.

On disk, the file system may contain information about how to boot an operating system stored there, the total number of blocks, the number and location of free blocks, the directory structure, and individual files. Many of these structures are detailed throughout the remainder of this chapter; here we describe them briefly:

A **boot control block** (per volume) can contain information needed by the system to boot an operating system from that volume. If the disk does not contain an operating system, this block can be empty. It is typically the first block of a volume. In UFS, it is called the boot block; in NTFS, it is the partition boot **sector.**

• A **volume control block** (per volume) contains volume (or partition) details, such as the number of blocks in the partition, size of the blocks, free block count and free-block pointers, and free FCB count and FCB pointers.
In UFS, this is called a **superblock;** in NTFS, it is stored in. the **master file table.**

• A directory structure per file system is used to organize the files. In UFS, this includes file names and associated **inode** numbers. In NTFS it is stored in the **master file table.**

• A per-file FCB contains many details about the file, including file permissions, ownership, size, and location of the data blocks. In UFS, this is called the i-node. In NTFS, this information is actually stored within the master file table, which uses a relational database structure, with a row per file.

The in-memory information is used for both file-system management and performance improvement via caching. The data are loaded at mount time and discarded at dismount. The structures may include the ones described below:

• An in-memory mount table contains information about each mounted volume.
• An in-memory directory-structure cache holds the directory information of recently accessed directories. (For directories at which volumes are mounted, it can contain a pointer to the volume table.)
• The **system-wide open-file table** contains a copy of the FCB of each open file, as well as other information.
• The **per-process open-file table** contains a pointer to the appropriate entry in the system-wide open-file table, as well as other information.

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

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 in that order. If the disk head is initially at cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 124/65, and finally to 67, for a total head movement of 640 cylinders. This schedule is diagrammed in Figure 12.4.
The wild swing from 122 to 14 and then back to 124 illustrates the problem with this schedule. If the requests for cylinders 37 and 14 could be serviced together, before or after the requests at 122 and 124, the total head movement
could be decreased substantially, and performance could be thereby improved.

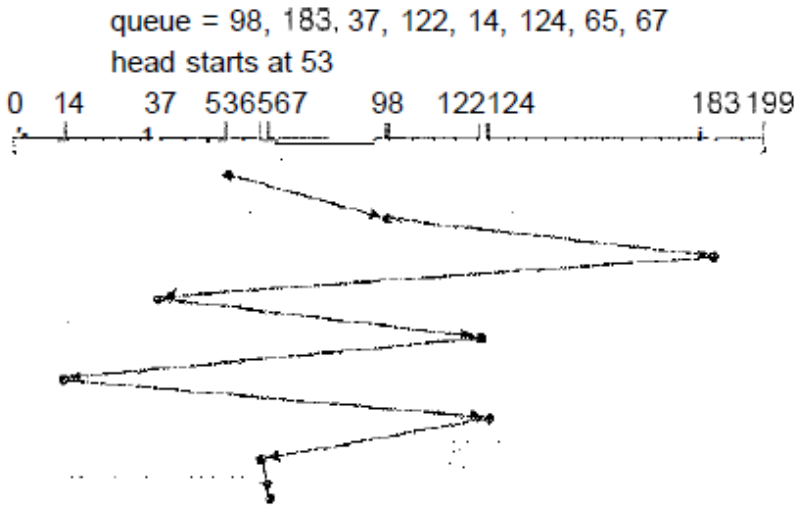


Figure 12.4 FCFS disk scheduling.

SSTF Scheduling:-

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

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

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

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.

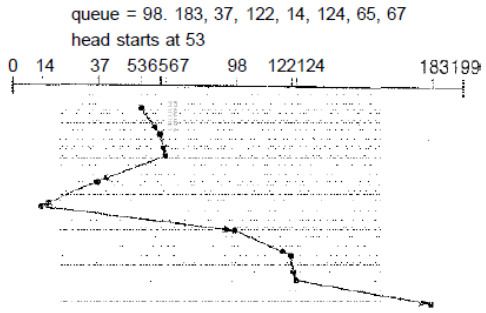


Figure 12.5 SSTF disk scheduling.

SCAN Scheduling:-

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

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

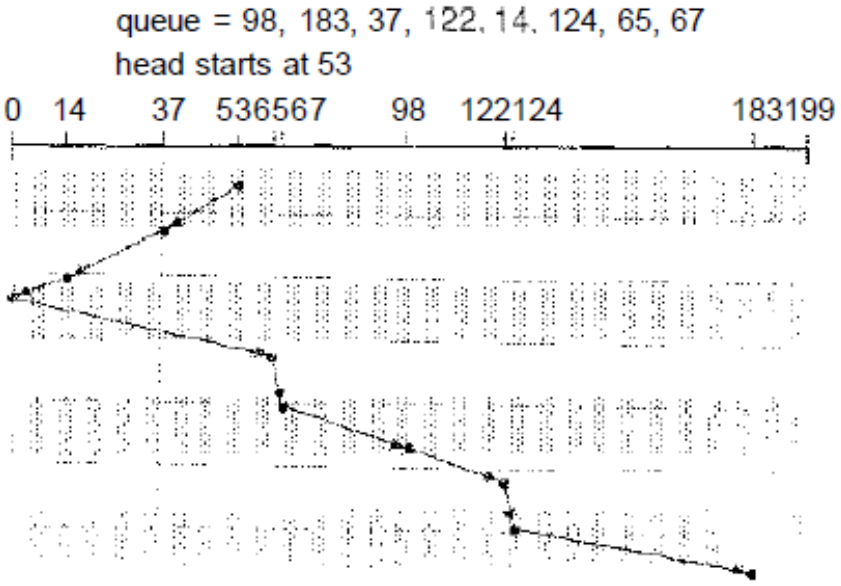


Figure 12.6 SCAN disk scheduling.

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

LOOK Scheduling:-

As we described them, both SCAN and C-SCAK 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 12.8).

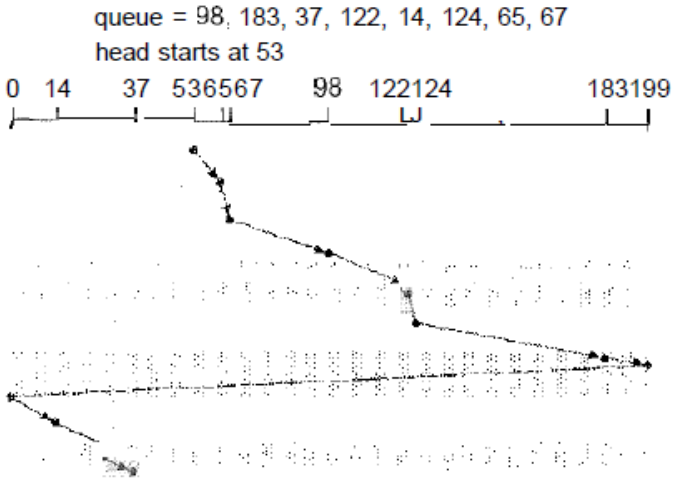


Figure 12.7 C-SCAN disk scheduling.

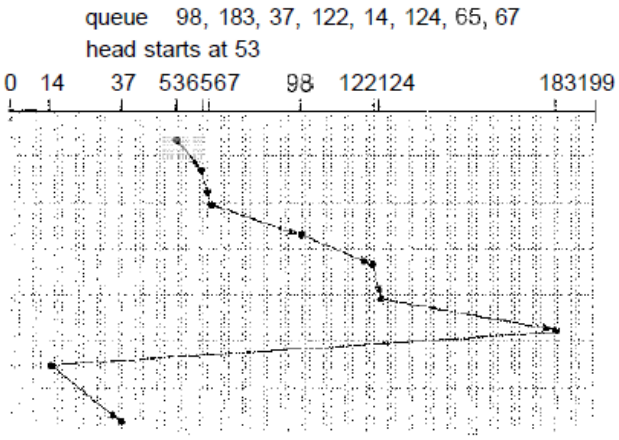


Figure 12.8 C-LOOK disk scheduling.

I/O Hardware:-

Computers operate a great many kinds of devices. Most fit into the general categories of storage devices (disks, tapes), transmission devices (network cards, modems), and human-interface devices (screen, keyboard, mouse).

A device communicates with, a computer system by sending signals over a cable or even through the air. The device communicates with the machine via a connection point (or port)—for example, a serial port. If devices use a common set of wires, the connection is called a *bus*. A **bus** is a set of wires and a rigidly defined protocol that specifies a set of messages that can be sent on the wires. In terms of the electronics, the messages are conveyed by patterns of electrical voltages applied to the wires with defined timings. When device *A* has a cable that plugs into device *B*, and device *B* has a cable that plugs into device C, and device C plugs into a port on the computer, this arrangement is called a daisy chain. A daisy chain usually operates as a bus.

Buses are used widely in computer architecture. A typical PC bus structure appears in Figure 13.1. This figure shows a PCI bus (the common PC system bus) that connects the processor-memory subsystem to the fast devices and an expansion bus that connects relatively slow devices such as the keyboard and serial and parallel ports. In the upper-right portion of the figure, four disks are connected together on a SCSI bus plugged into a SCSI controller.

A controller is a collection of electronics that can operate a port, a bus, or a device. A serial-port controller is a simple device controller. It is a single chip (or portion of a chip) in the computer that controls the signals on the wires of a serial port. By contrast, a SCSI bus controller is not simple. Because the SCSI protocol is complex, the SCSI bus controller is often implemented as a separate circuit board (or a **host** adapter) that plugs into the computer. It typically contains a processor, microcode, and some private memory to enable it to process the SCSI protocol messages. Some devices have their own built-in controllers. If you look at a disk drive, you will see a circuit board attached to one side. This board is the disk controller. It implements the disk side of the protocol for some kind of connection—SCSI or ATA, for instance. It has microcode and a processor to do many tasks, such as bad-sector mapping, pre-fetching, buffering, caching.

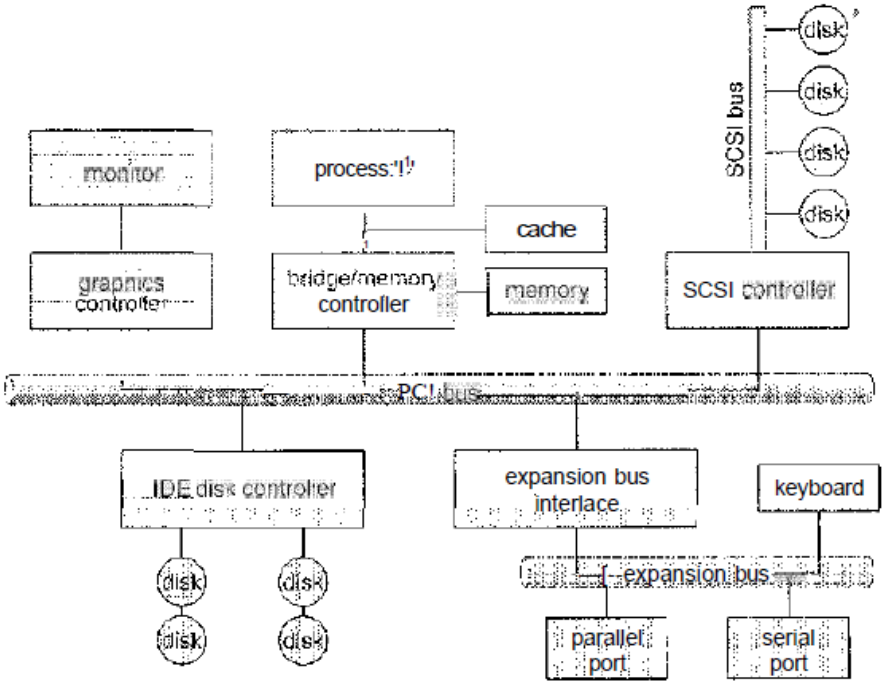


Figure 13.1 A typical PC bus structure.

How can the processor give commands and data to a controller to accomplish an I/O transfer? The short answer is that the controller has one or more registers for data and control signals. The processor communicates with the controller by reading and writing bit patterns in these registers. One way in which this communication can occur is through the use of special I/O instructions that specify the transfer of a byte or word to an I/O port address. The I/O instruction triggers bus lines to select the proper device and to move bits into or out of a device register. Alternatively, the device controller can support memory-mapped I/O. In this case, the device-control registers are mapped into the address space of the processor. The CPU executes I/O requests using the standard data-transfer instructions to read and write the device-control registers.

An I/O port typically consists of four registers, called the (1) status, (2) control, (3) data-in, and (4) data-out registers.
• The **data-in** register is read by the host to get input.
• The **data-out** register is written by the host to send output.
• The **status** register contains bits that can be read by the host. These bits indicate states, such as whether the current command has completed, whether a byte is available to be read from the data-in register, and whether a device error has occurred.
• The control register can be written by the host to start a command or to change the mode of a device. For instance, a certain bit in the control register of a serial port chooses between full-duplex and half-duplex communication, another bit enables parity checking, a third bit sets the word length to 7 or 8 bits, and other bits select one of the speeds supported by the serial port.

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. The I/O subsystem is also responsible for protecting itself from errant processes and malicious users.

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. Here is a simple example to illustrate the opportunity. Suppose that a disk arm is near the beginning of a disk and that three applications issue blocking read calls to that disk. Application 1 requests a block near the end of the disk, application 2 requests one

near the beginning, and application 3 requests one in the middle of the disk. The operating system can reduce the distance that the disk arm travels by serving the applications in the order 2, 3,1. Rearranging the order of service in this way is the essence of I/O scheduling.

Operating-system developers implement scheduling by maintaining a wait 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.

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.

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. In others, it is handled by an in-kernel thread. In either case, the operating system provides a control interface that enables users and system administrators to display the queue, to remove unwanted jobs before those jobs print, to suspend printing while the printer is serviced, and so on.

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, as when a network becomes overloaded, or for "permanent" reasons, as when a disk controller becomes defective. Operating systems can often compensate effectively for transient failures. For instance, a disk read() failure results in a read(C) retry, and a network send() error results in a resend, if the protocol so specifies. Unfortunately, if an important component experiences a permanent failure, the operating system is unlikely to recover.

I/O Protection:-

Errors are closely related to the issue of protection. A user process may accidentally or purposefully attempt to disrupt the normal operation of a system by attempting to issue illegal I/O instructions. We can use various mechanisms to ensure that such disruptions cannot take place in the system. To prevent users from performing illegal I/O, we define all I/O instructions to be privileged instructions. Thus, users cannot issue I/O instructions directly; they must do it through the operating system. To do I/O, a user program executes a system call to request that the operating system perform I/O on its behalf (Figure 13.11). The operating system, executing in monitor mode, checks that the request is valid and, if it is, does the I/O requested. The operating system then returns to the user.

Transforming I/O Requests to Hardware Operations:-

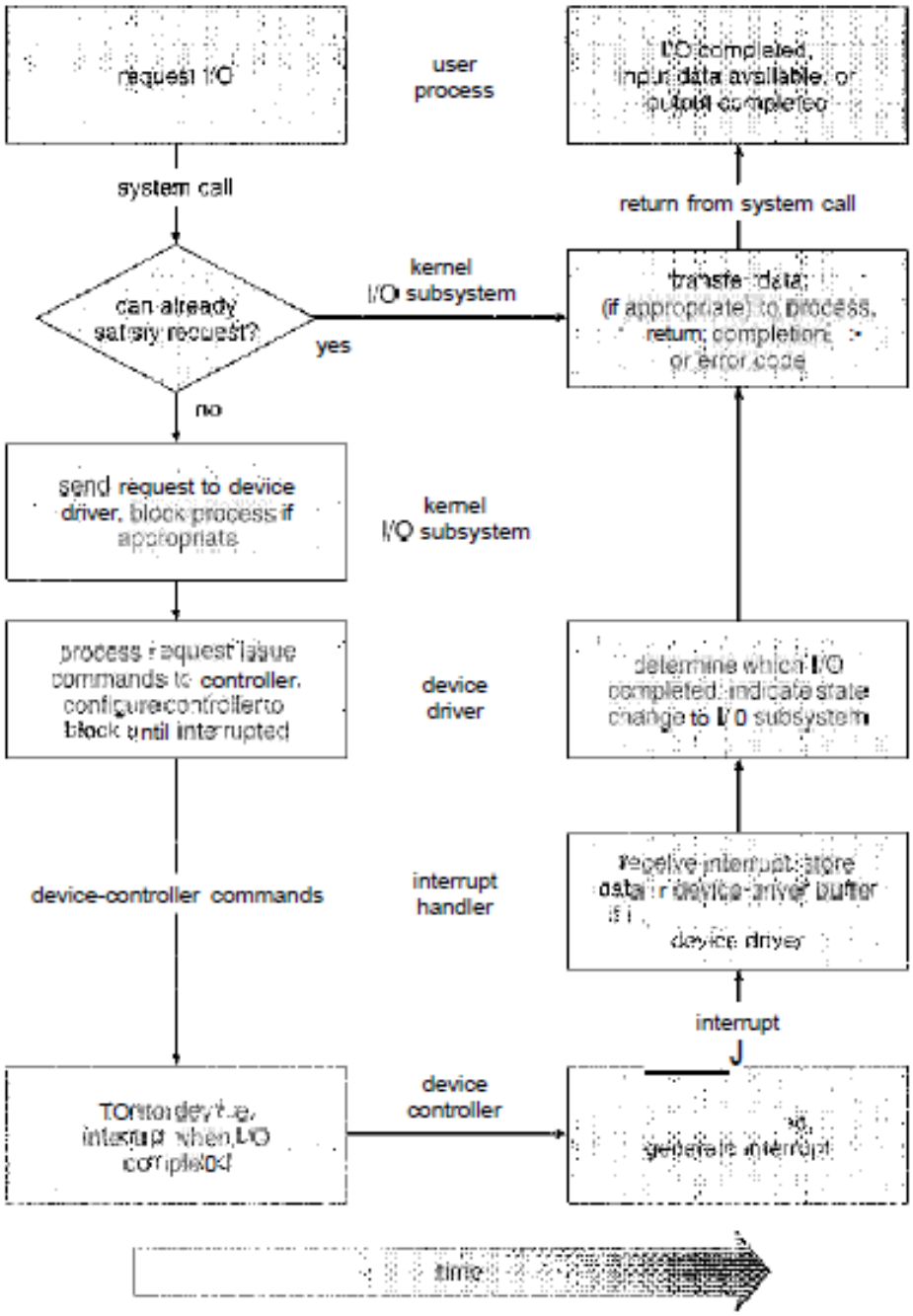


Figure 13.13 The life cycle of an I/O request.