UNIT - IV FILE SYSTEMS AND I/O SYSTEMS

Mass Storage system – Overview of Mass Storage Structure, Disk Structure, Disk Scheduling and Management, swap space management; File-System Interface - File concept, Access methods, Directory Structure, Directory organization, File system mounting, File Sharing and Protection; File System Implementation- File System Structure, Directory implementation, Allocation Methods, Free Space Management, Efficiency and Performance, Recovery; I/O Systems – I/O Hardware, Application I/O interface, Kernel I/O subsystem, Streams, Performance.

MASS STORAGE STRUCTURE- OVERVIEW

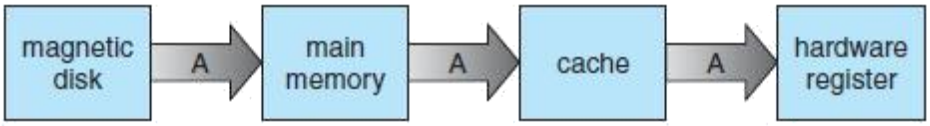
Main memory is usually too small to store all needed programs and data permanently.

Main memory is a **volatile** storage device that loses its contents when power is turned off or otherwise lost.

Thus, most computer systems provide **secondary storage** as an extension of main memory. The main requirement for secondary storage is that it be able to hold large quantities of data permanently.

The most common secondary-storage device is a **magnetic disk**, which provides storage for both programs and data.

Most of the secondary storage devices are internal to the computer such as the hard disk drive, the tape disk drive and even the compact disk drive and floppy disk drive.



Magnetic Disks

Magnetic disks provide the bulk of secondary storage for modern computer systems.

Each disk platter has a flat circular shape, like a CD. Common platter diameters range from 1.8 to 3.5 inches.

The two surfaces of a platter are covered with a magnetic material. We store information by recording it magnetically on the platters.

A read–write head ―flies‖ just above each surface of every platter. The heads are attached to a disk arm that moves all the heads as a unit.

The surface of a platter is logically divided into circular **tracks**, which are subdivided into **sectors**.

CYLINDER: The set of tracks that are at one arm position makes up a **cylinder**.

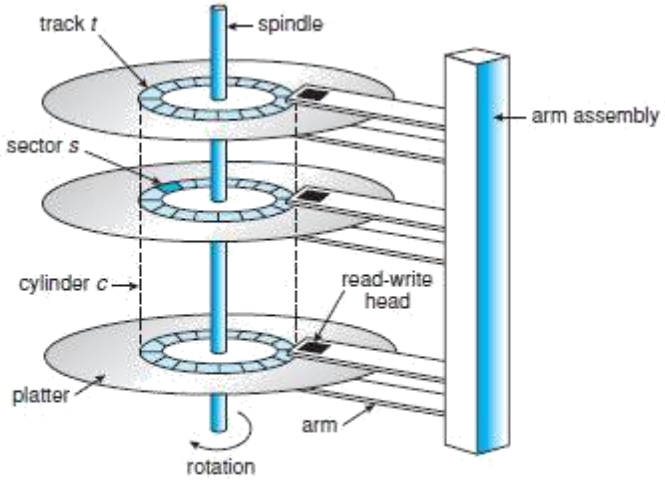


Figure 10.1 Moving-head disk mechanism.

The storage capacity of common disk drives is measured in gigabytes. When the disk is in use, a drive motor spins it at high speed. Most drives rotate 60 to 250 times per second, specified in terms of rotations per minute.

Disk speed has two parts.

The **transfer rate** is the rate at which data flow between the drive and the computer.

The **positioning time**, or **random-access time**

SEEK TIME: The time necessary to move the disk arm to the desired cylinder, is called the **seek time**.

ROTATIONAL LATENCY: The time necessary for the desired sector to rotate to the disk head, called the **rotational latency**.

HEAD CRASH:

The disk read write head has a danger that the head will make contact with the disk surface. Although the disk platters are coated with a thin protective layer, the head will sometimes damage the magnetic surface. This accident is called a **head crash**.

A disk drive is attached to a computer by a set of wires called an **I/O bus**. Several kinds of buses are available, including **advanced technology attachment (ATA)**, **serial ATA (SATA)**, **eSATA universal serial bus (USB)**, and **fibre channel (FC)**.

The data transfers on a bus are carried out by special electronic processors called **controllers**. The **host controller** is the controller at the computer end of the bus.

A **disk controller** is built into each disk drive. To perform a disk I/O operation, the computer places a command into the host controller, typically using memory-mapped I/O ports

Magnetic Tapes:

Magnetic tape was used as an early secondary-storage medium.

It is relatively permanent and can hold large quantities of data.

Its access time is slow compared with that of main memory and magnetic disk.

In addition, random access to magnetic tape is about a thousand times slower than random access to magnetic disk, so tapes are not very useful for secondary storage.

Tapes are used mainly for backup, for storage of infrequently used information, and as a medium for transferring information from one system to another.

Disk Structure

Magnetic disk drives are addressed as large one-dimensional arrays of **logical blocks**, where the logical block is the smallest unit of transfer.

The size of a logical block is usually 512 bytes, although some disks can be **low-level formatted** to have a different logical block size, such as 1,024 bytes.

The one-dimensional array of logical blocks is mapped onto the sectors of the disk sequentially. Sector 0 is the first sector of the first track on the outermost cylinder.

The number of sectors per track is not constant on some drives.

For the disks that use **constant linear velocity (CLV)**, the density of bits per track is uniform.

In **constant angular velocity (CAV)** the density of bits decreases from inner tracks to outer tracks to keep the data rate constant.

Disk Attachment

Computers access disk storage in the following ways

- Host attached Storage
- Network Attached Storage
- Storage Area Network.

HOST ATTACHED STORAGE: Host-attached storage is storage accessed through local I/O ports. The typical desktop PC uses an I/O bus architecture called IDE or ATA.

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

STORAGE AREA NETWORK: A storage-area network (SAN) is a private network connecting servers and storage units.

DISK SCHEDULING:

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

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

Disk Scheduling: If the desired disk drive and controller are available, the request can be serviced immediately. If the drive or controller is busy, any new requests for service will be placed in the queue of pending requests for that drive. When one request is completed, the operating system chooses which pending request to service next. This is called as Disk Scheduling.

Disk Components:

The two major components of the hard disk are Seek time and Rotational Latency.

Seek time: The **seek time** is the time for the disk arm to move the heads to the cylinder containing the desired sector.

Rotational latency: The **rotational latency** is the additional time for the disk to rotate the desired sector to the disk head.

Disk bandwidth: 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.

Disk Scheduling Algorithms:

First Come First Serve
Shortest Seek Time First
Scan Algorithm
Circular Scan Algorithm
Look Algorithm
Circular Look Algorithm

FCFS Scheduling:

The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm.

This algorithm is easy to implement but it generally does not provide the fastest service.

Example: Consider, for example, Given a disk with 200 cylinders and a disk queue with requests 98, 183, 37, 122, 14, 124, 65, 67, for I/O to blocks on cylinders. Disk head is initially at 53.

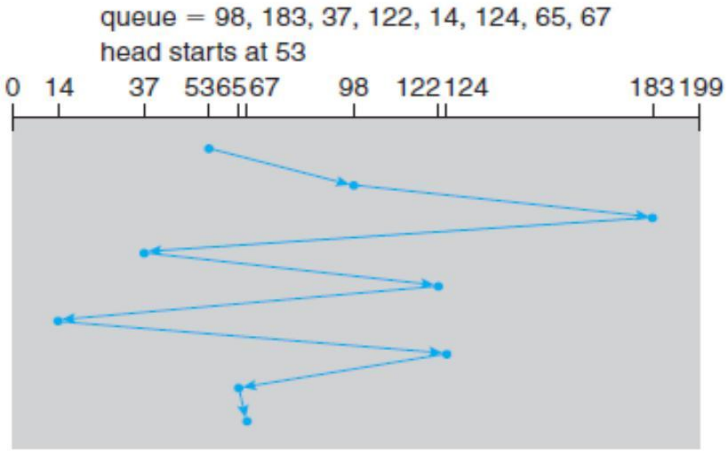


Figure 10.4 FCFS disk scheduling.

If the disk head is initially at cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 24, 65, and finally to 67.

The total head movements is Head Movements = (53-98)+(98-183)+((183-37)+(122-14)+(14-124)+(124-65)+(65-67) = 640 Head Movements.

Disadvantage:

The request from 122 to 14 and then back to 124 increases the total head movements.

If the requests for cylinders 37 and 14 could be serviced together, before or after the requests for 122 and 124, the total head movement could be decreased and performance could be thereby improved.

SSTF Scheduling:

The **shortest-seek-time-first (SSTF) algorithm** selects the request with the least seek time from the current head position. It chooses the pending request closest to the current head position.

Example: Consider, for example, Given a disk with 200 cylinders and a disk queue with requests 98, 183, 37, 122, 14, 124, 65, 67, for I/O to blocks on cylinders. Disk head is initially at 53.

The closest request to the initial head position (53) is at cylinder 65.

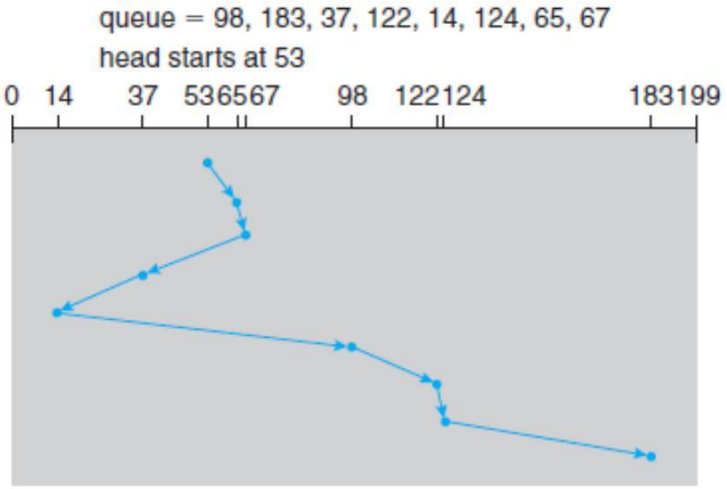
Once we are at cylinder 65, the next closest request is at cylinder 67.

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.

This scheduling method results in a total head movement of only 236 cylinders

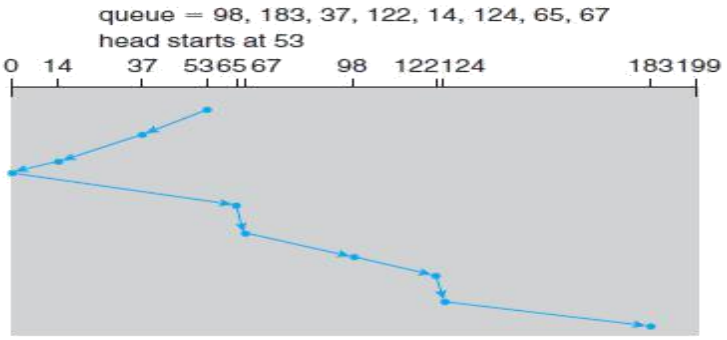
SSTF algorithm gives a substantial improvement in performance.



Disadvantage:
SSTF may cause starvation of some requests.

STARVATION: Suppose that we have two requests in the queue, for cylinders 14 and 186, and while the request from 14 is being serviced, 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 cause the request for cylinder 186 to wait indefinitely.

3) SCAN Scheduling:
In the **SCAN algorithm**, the disk arm starts at one end of the disk and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk
At the other end, the direction of head movement is reversed, and servicing continues.
The head continuously scans back and forth across the disk.
The SCAN algorithm is sometimes called the **elevator algorithm.**
Example: Consider, for example, Given a disk with 200 cylinders and a disk queue with requests 98, 183, 37, 122, 14, 124, 65, 67, for I/O to blocks on cylinders. Disk head is initially at 53.
Assuming that the disk arm is moving toward 0 the head will next service 37 and then 14.



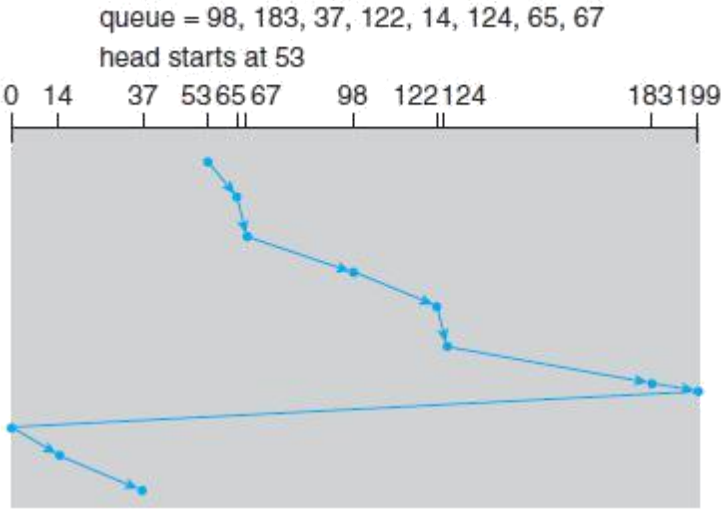
At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the requests at 65, 67, 98, 122, 124, and 183.
If a request arrives in the queue just in front of the head, it will be serviced almost immediately; a request arriving just behind the head will have to wait until the arm moves to the end of the disk, reverses direction, and comes back.

Circular SCAN Algorithm:
Circular SCAN (C-SCAN) scheduling is a variant of SCAN designed to provide a more uniform wait time.

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.

Example: Consider, for example, Given a disk with 200 cylinders and a disk queue with requests 98, 183, 37, 122, 14, 124, 65, 67, for I/O to blocks on cylinders. Disk head is initially at 53.



LOOK scheduling:

The **LOOK** algorithm is the same as the **SCAN** algorithm in that it also services the requests on both directions of the disk head, but it ―Looks" ahead to see if there are any requests pending in the direction of head movement.

If no requests are pending in the direction of head movement, then the disk head traversal will be reversed to the opposite direction and requests on the other direction can be served.

In LOOK scheduling, the arm goes only as far as final requests in each direction and then reverses direction without going all the way to the end.

Consider an example, given a disk with 200 cylinders (0-199), suppose we have 8 pending requests: 98, 183, 37, 122, 14, 124, 65, 67 and that the read/write head is currently at cylinder 53. In order to complete these requests, the arm will move in the increasing order first and then will move in decreasing order after reaching the end. So, the order in which it will execute is 65, 67, 98, 122, 124, 183, 37, and 14.

Note :(Draw the Diagram and calculate the head movements for the previous example)

C-LOOK Scheduling:

This is just an enhanced version of C-SCAN.

Arm only goes as far as the last request in each direction, then reverses direction immediately, without servicing all the way to the end of the disk and then turns the next direction to provide the service.

Note :(Draw the Diagram and calculate the head movements for the previous example)

DISK MANAGEMENT:

The operating system is responsible for disk management.

The Major Responsibility includes

- Disk Formatting,
- Booting from disk
- Bad-block recovery.

1. Disk Formatting:

The Disk can be formatted in two ways,
Physical or Low level Formatting,
Logical Or High Level Formatting

Physical or Low level Formatting:

Before a disk can store data, it must be divided into sectors that the disk controller can read and write. This process is called **low-level formatting**, or **physical formatting**.
Low-level formatting fills the disk with a special data structure for each sector.
The data structure for a sector typically consists of a header, a data area (usually 512 bytes in size), and a trailer.
The Header contains the Sector Number and the Trailer contains the Error correction code.
When the controller writes a sector of data during normal I/O, the ECC is updated with a value calculated from all the bytes in the data area.
When the sector is read, the ECC is recalculated and compared with the stored value. If the stored and calculated numbers are different, this mismatch indicates that the data area of the sector has become corrupted and that the disk sector may be bad. It then reports a recoverable **soft error**.

Logical Formatting Or High Level Formatting:

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

CLUSTERS: To increase efficiency, most file systems group blocks together into larger chunks, frequently called **clusters**.

RAW DISK: Some operating systems give special programs the ability to use a disk partition as a large sequential array of logical blocks, without any file-system data structures. This array is sometimes called the **raw disk,** and I/O to this array is termed **raw I/O.**

Boot Block

The Process of Starting a computer System by loading the Operating system in the main memory is called as System Booting.
This is done by a initial program called as Bootstrap program initializes all aspects of the system, from CPU registers to device controllers and the contents of main memory, and then starts the operating system.
The bootstrap is stored in **read-only memory (ROM)**.The Problem here is that changing this bootstrap code requires changing the ROM hardware chips.
To overcome this most systems store a tiny bootstrap loader program in the boot ROM whose only job is to bring in a full bootstrap program from disk.
The full bootstrap program can be changed easily: a new version is simply written onto the disk.
BOOT DISK: The full bootstrap program is stored in the ―boot blocks‖ at a fixed location on the disk. A disk that has a boot partition is called a **boot disk** or **system disk**.
The code in the boot ROM instructs the disk controller to read the boot blocks into memory and then starts executing that code which in turn loads the entire Operating System.

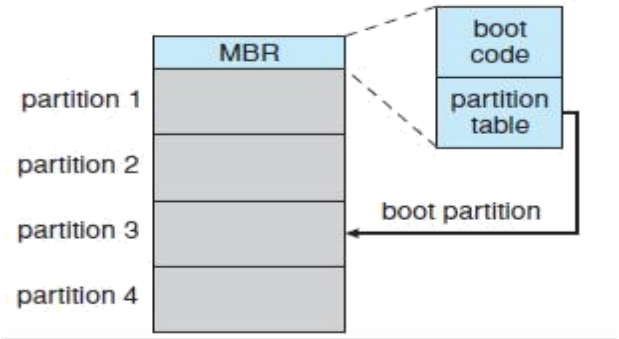
EXAMPLE: Boot Process in Windows.

BOOT PARTITION: Windows allows a hard disk to be divided into partitions, and one partition called as the **boot partition** contains the operating system and device drivers.

The Windows system places its boot code in the first sector on the hard disk, which it terms the **master boot record**, or **MBR**.

Booting begins by running code that is resident in the system's ROM memory. This code directs the system to read the boot code from the MBR.

Once the system identifies the boot partition, it reads the first sector from that partition and continues with the remainder of the boot process, which includes loading the various subsystems and system services.



Bad Blocks

A bad block is a damaged area of magnetic storage media that cannot reliably be used to store and retrieve data. Depending on the disk and controller in use, these blocks are handled in a variety of ways.

One strategy is to scan the disk to find bad blocks while the disk is being formatted. Any bad blocks that are discovered are flagged as unusable so that the file system does not allocate them.

In Some systems the controller maintains a list of bad blocks on the disk. This can be handled in two ways

Sector Sparing

Sector Slipping

SECTOR SPARING: Low-level formatting also sets aside spare sectors not visible to the operating system. The controller can be told to replace each bad sector logically with one of the spare sectors. This scheme is known as **sector sparing** or **forwarding**.

Example:

The operating system tries to read logical block 87.

The controller calculates the ECC and finds that the sector is bad. It reports this finding to the operating system.

The next time the system is rebooted, a special command is run to tell the controller to replace the bad sector with a spare.

After that, whenever the system requests logical block 87, the request is translated into the replacement sector's address by the controller.

SECTOR SLIPPING: The Process of moving all the sectors down one position from the bad sector is called as sector slipping.

Example: If the logical block 17 becomes defective and the first available spare follows sector 202. Sector slipping then remaps all the sectors from 17 to 202, moving them all down one spot. That is, sector 202 is copied into the spare, then sector 201 into 202, then 200 into 201, and so on, until sector 18 is copied into sector 19. Slipping the sectors in this way frees up the space of sector 18 so that sector 17 can be mapped to it.

FILE SYSTEM STORAGE:

The File System provides the mechanism for on-line storage of and access to both data and programs of the operating system and all the users of the computer system.

The file system consists of two distinct parts:

a collection of files, each storing related data,

a directory structure, which organizes and provides information about all the files in the system.

FILE CONCEPTS:

A file is defined as a named collection of related information that is stored on secondary storage device.
Many different types of information may be stored in a file such as source or executable programs, numeric or text data, photos, music, video, and so on.
A file has a certain defined structure, which depends on its type.
Types of Files: A **text file** is a sequence of characters organized into lines.
A **source file** is a sequence of functions, each of which is further organized as declarations followed by executable statements.
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 file name is the information kept in human readable form.
Identifier. This unique tag, usually a number, identifies the file within the file system
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)
Protection. Access-control information determines who can do reading, writing, executing
Time, date, and user identification. This information may be kept for creation, last modification, and last use.

FILE OPERATIONS:

A file is an abstract data type.
The operating system can provide system calls to create, write, read, reposition, delete, and truncate files. The Basic Operations on a file includes

Creating a File
Writing a File
Reading a File
Repositioning within a File
Deleting a file
Truncating a file

Creating a file: OS first finds space in the file system for the file. Second, an entry for the new file must be made in the directory.

Writing a file. To write a file, we make a system specifying both the name of the file and the information to be written to the file. The system must keep a **write pointer** to the location in the file where the next write is to take place.

Reading a file. To read from a file, we use a system call that specifies the name of the file. The system needs to keep a **read pointer** to the location in the file where the next read is to take place.

Current File Position Pointer: 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.**

Repositioning within a file. The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value. This file operation is also known as files seek.

Deleting a file. To delete a file, we search the directory for the named file.

Truncating a file. The user may want to erase the contents of a file but keep its attributes

Open File Table:

Most of the file operations involve searching the directory for the entry associated with the named file.
The operating system keeps a table, called the **open-file table**, containing information about all open files.
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 being actively used, it is closed by the process, and the operating system removes its entry from the open-file table.

The open-file table also has an **open count** associated with each file to indicate how many processes have the file open Each close() decreases this open 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.

When several processes may open the file , the operating system uses two levels of internal tables:

A per-process table

A system-wide table.

The per process table tracks all files that a process has open. It information regarding the process's use of the file such as the current file pointer, access rights.

The system-wide table contains process-independent information, such as the location of the file on disk, access dates, and file size.

FILE LOCKS:

File locks allow one process to lock a file and prevent other processes from gaining access to it. File locks are useful for files that are shared by several processes.

Shared Lock

Exclusive Lock

A **shared lock** is similar to a reader lock in that several processes can acquire the lock concurrently.

An **exclusive lock** behaves like a writer lock; only one process at a time can acquire such a lock.

Operating systems may provide either **mandatory** or **advisory** file-locking mechanisms.

If a lock is mandatory, then once a process acquires an exclusive lock, the operating system will prevent any other process from accessing the locked file.

If a lock is advisory then the Operating System will allow the process to access the locked file.

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 or a dot.

The system uses the extension to indicate the type of the file and the type of operations that can be done on that file.

Example: Java compilers expect source files to have a .java extension, and the Microsoft Word word processor expects its files to end with a .doc or .docx extension.

file type	usual extension	Function
Executable	exe, com, bin or none	ready-to-run machine-language program
Object	obj, o	Compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
Batch	bat, sh	Commands to the command interpreter
Markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
Library	lib, a , so, dll	Libraries of routines for programmers
point or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
Archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
Multimedia	mpeg, mov, mp3, mp4,avi	binary file containing audio or A/V information

FILE 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 Method

Direct Access method

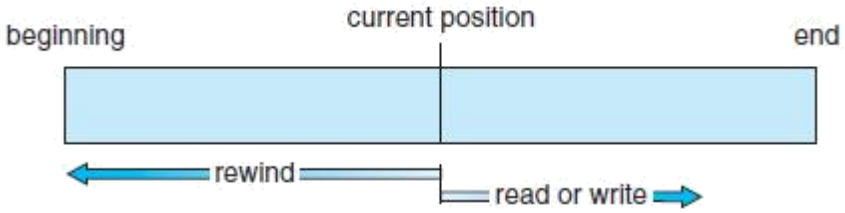
Indexed access method

Sequential access Method: The simplest access method is **sequential access**. Information in the file is processed in order, one record after the other.

Example: Editors and compilers usually access files in Sequential manner.

A read operation—read next ()—reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location.

Similarly, the write operation—write next ()—appends to the end of the file and advances to the end of the newly written material (the new end of file).



Direct Access Method:

A file is made up of fixed-length **logical records** that allow programs to read and write records rapidly in no particular order.

The file is viewed as a numbered sequence of blocks or records.

Direct-access files are of great use for immediate access to large amounts of information.

EXAMPLE: 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, write (n) rather than write next ().

RELATIVE BLOCK NUMBER: The block number provided by the user to the operating system is normally a **relative block number**

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

Indexed Access Methods:

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.

With large files, the index file 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 contains pointers to secondary index files, which point to the actual data items

FILE SYSTEM MOUNTING:

File System Mounting is defined as the process of attaching an additional file system to the currently accessible file system of a computer. **A file system** is a hierarchy of directories that is used to organize files on a computer or storage media.

The operating system is given the name of the device and the **mount point**

Mount Point: It is the location within the file structure where the file system is to be attached. A mount point is an empty directory.

Example: A file system containing a user's home directories might be mounted as /home. To access the directory structure within that file system, we could precede the directory names with /home, as in /home/Jane. Mounting that file system under /users would result in the path name /users/Jane, which we could use to reach the same directory.

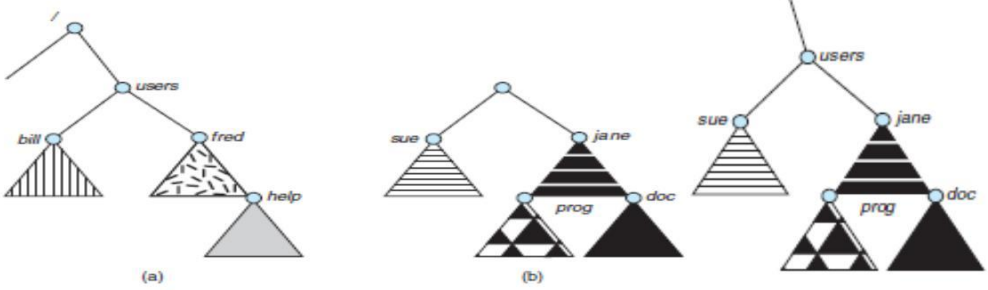


Figure 11.14 File system. (a) Existing system. (b) Unmounted volume.

Figure 11.15 Mount point.

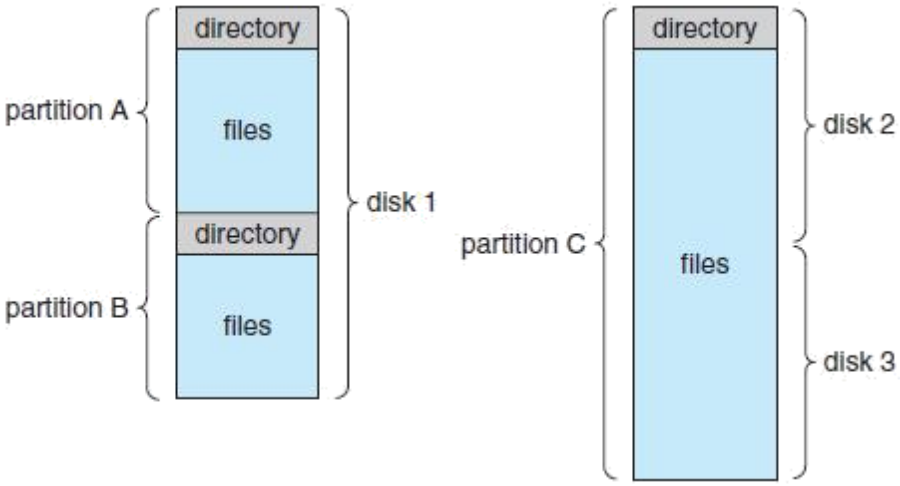
DIRECTORY AND DISK STRUCTURE

The Operating System divides the hard disk into various partitions and stores its File System in each Partition.

Any entity containing a file system is generally known as a **volume**.

Each volume that contains a file system must also contain information about the files in the system. This information is kept in entries in a **device directory** or **volume table of contents**.

The device directory records information—such as name, location, size, and type—for all files on that volume.



The directory translates file names into their directory entries. Operations that are to be performed on a directory includes

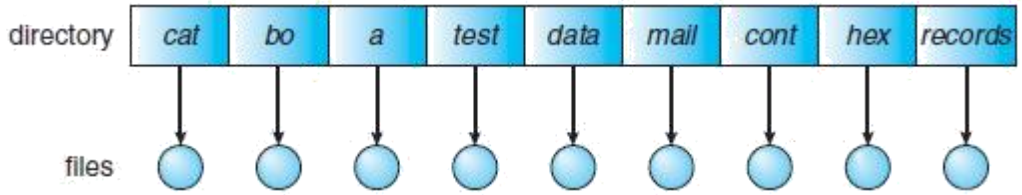
Search for a file
Create a File
Delete a file
List a Directory
Rename a File
List a directory
Traverse the file system.

Single level Directory
Two Level Directory
Tree Structured Directory

Acyclic Graph directory
General Graph directory.

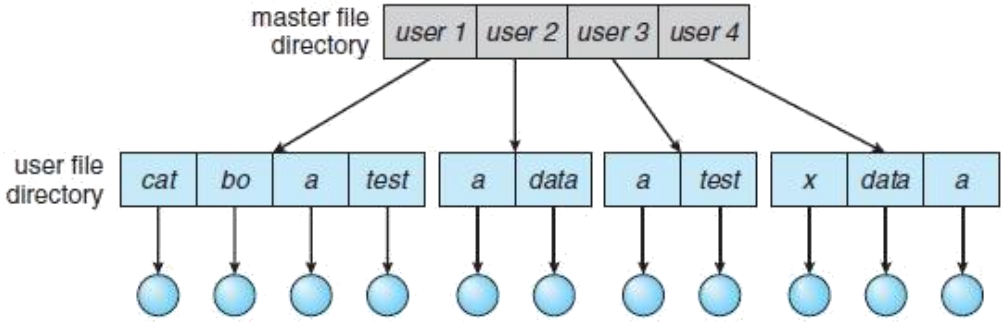
Single level Directory:

The simplest directory structure is the single-level directory. All files are contained in the same directory.
When the system has more than one user, the files created by different users should have unique names because all the files will be stored under a single directory
When the number of files increases the user may find it difficult to remember the names of all the files



Two Level directory:

In the two-level directory structure, each user has his own directory called **user file directory.**
The UFDs have similar structures, but each lists only the files of a single user.
When a user job starts or a user logs in, the system's **master file directory (MFD)** is searched.
The MFD is indexed by user name or account number, and each entry points to the UFD for that user.
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 check whether another file of that name exists.
To delete a file, the operating system searches the local UFD; thus, it cannot accidentally delete another user's file that has the same name.

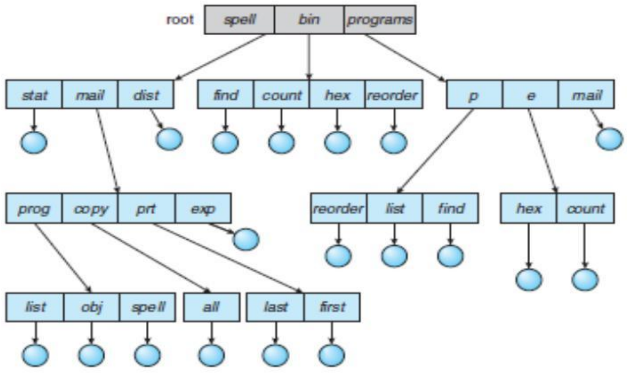


Advantage: The two-level directory structure solves the name-collision problem.
Disadvantage: It isolates one user from another user. It's a major disadvantage when the users want to cooperate on some task and to access one another's files.

Some systems simply do not allow local user files to be accessed by other users.
If access is to be permitted, one user must have the ability to name a file in another user's directory.
PATH NAME: A user name and a file name define a **path name**.
A two-level directory can be similar to that of a tree structure of height 2. The root of the tree is the MFD. Its direct Childs are the UFDs. The Childs of the UFDs are the files themselves. The files are the leaves of the tree.

Tree structured directory:

Tree structured directory is an expansion of two level directory extended to a tree of arbitrary height.
It allows users to create their own subdirectories and to organize their files.
The tree has a root directory, and every file in the system has a unique path name.
The **current directory** should contain most of the files that are of current need to the process.



Path names can be of two types:
Absolute path name
Relative path name
An **absolute path name** begins at the root and follows a path down to the specified file.
A **relative path name** defines a path from the current directory.
Users can be allowed to access the files of other users by specifying the path name.
A directory (or subdirectory) contains a set of files or subdirectories.
A bit in each directory entry defines the entry as a file (0) or as a subdirectory (1).

FILE DELETION: If a directory is empty, its entry in the directory that contains it can simply be deleted.
If the directory to be deleted is not empty and contains several files or subdirectories. One of two approaches can be taken.
Some systems will not delete a directory unless it is empty. Thus, to delete a directory, the user must first delete all the files in that directory.
If when a request is made to delete a directory, all that directory's files and subdirectories are also to be deleted.

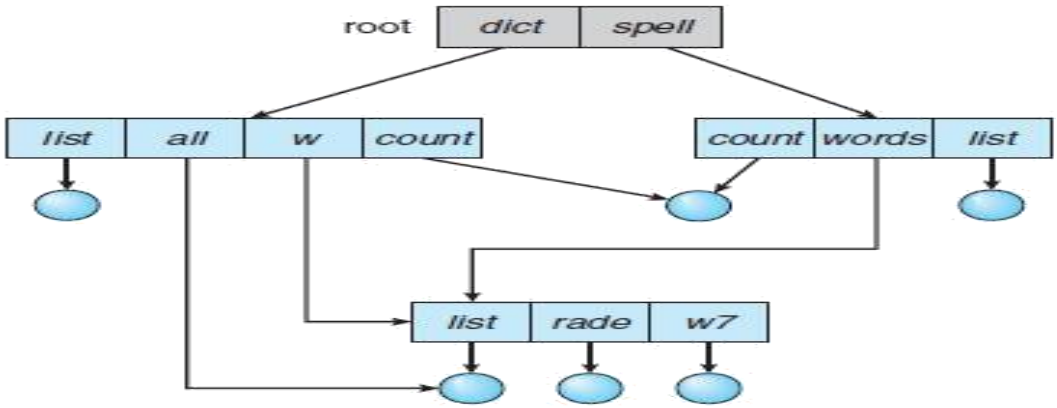
Acyclic Graph Directories:

An **acyclic graph is** a graph with no cycles.
It allows directories to share subdirectories and files
The same file or subdirectory may be in two different directories.
With a shared file, only one actual file exists, so any changes made by one person are immediately visible to the other.
All the files the user wants to share can be put into one directory. The UFD of each team member will contain this directory of shared files as a subdirectory.

SHARED FILE IMPLEMENTATION:

Shared files and subdirectories can be implemented in several ways.
The first way is to create a new directory entry called a link. A **link** is effectively a pointer to another file or subdirectory.
When a reference to a file is made, we search the directory. If the directory entry is marked as a link, then the name of the real file is included in the link information.
We **resolve** the link by using that path name to locate the real file.
Another common approach to implementing shared files is simply to duplicate all information about them in both sharing directories.

A major problem with duplicate directory entries is maintaining consistency when a file is mod



FILE DELETION:

One method is to remove the file whenever anyone deletes it, but this action may leave dangling pointers to the now-nonexistent file.

The second method is the deletion of a link. It does not affect the original file; only the link is removed.

Another approach to deletion is to preserve the file until all references to it are deleted.

We could keep a list of all references to a file.

When a link or a copy of the directory entry is established, a new entry is added to the file-reference list.

When a link or directory entry is deleted, we remove its entry on the list. The file is deleted when its file-reference list is empty.

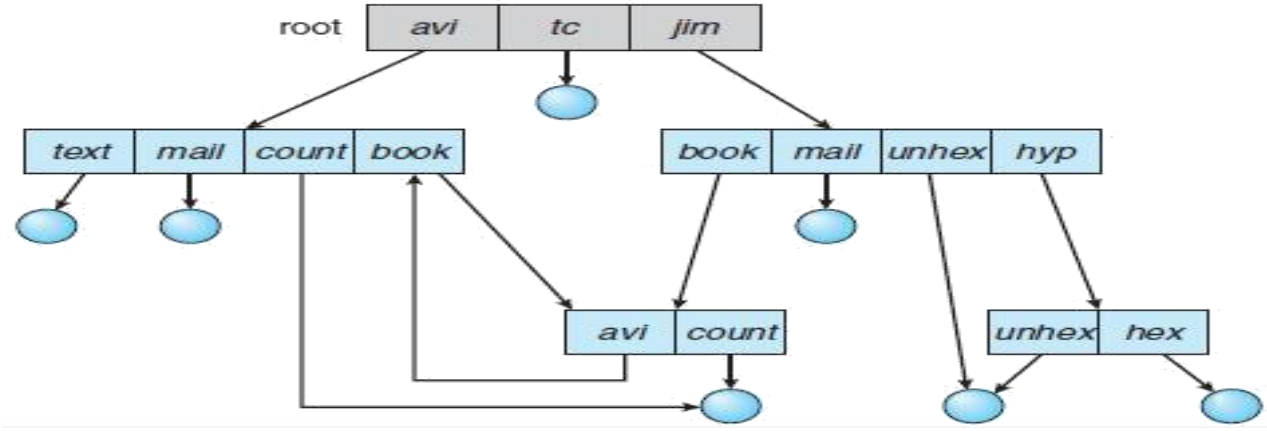
General Graph Directory:

If we start with a two-level directory and allow users to create subdirectories, a tree-structured directory results.

Adding new files and subdirectories to an existing tree-structured directory preserves the tree-structured nature.

When we add links, the tree structure is destroyed, resulting in a simple graph structure.

If cycles are allowed to exist in the directory, we want to avoid searching any component twice, to improve the performance.



With acyclic-graph directory structures, a value of 0 in the reference count means that there are no more references to the file or directory, and the file can be deleted.

However, when cycles exist, the reference count may not be 0 even when it is no longer possible to refer to a directory or file. In this case we use Garbage collection.

Garbage collection:

This scheme is used to determine when the last reference has been deleted and the disk space can be reallocated.

Garbage collection involves traversing the entire file system, marking everything that can be accessed.
Then, a second pass collects everything that is not marked onto a list of free space.

FILE SHARING AND PROTECTION:

File Sharing:

File sharing is very important for users who want to cooperate their files with each other and to reduce the effort required to achieve a computing goal.

Multiple users share files. When multiple users are allowed to share files, then there is a need to extend sharing to
multiple file systems, including remote file systems.

File sharing includes
- Multiple users
- Remote File Systems
  - Client server model
  - Distributed Information systems
  - Failure Modes
- Consistency semantics
  - Unix Semantics
  - Session Semantics
  - Immutable Shared File Semantics

Multiple Users:

The system with multiple users can either allow a user to access the files of other users by default or require that a user specifically grant access to the files.

To implement sharing and protection, the system must maintain more file and directory attributes than are needed on a single-user system.

The systems uses the concepts of file **owner** (or **user**) and **group for File sharing.**

The owner is the user who can change attributes and grant access and who has the most control over the file.

The group attribute defines a subset of users who can share access to the file.

The owner and group IDs of a given file are stored with the other file attributes.

When a user requests an operation on a file, the user ID can be compared with the owner attribute to determine if the requesting user is the owner of the file.

If he is not the owner of the file, the group IDs can be compared. The result indicates which permissions are applicable.

The system then applies those permissions to the requested operation and allows or denies it.

Remote File Systems:

Networking allows the sharing of resources across a campus or even around the world.

The first implemented method for remote file systems involves manually transferring files between machines via programs like FTP.

The second major method uses a **distributed file system (DFS)** in which remote directories is visible from a local machine.

The third method is the **World Wide Web** where the browser is needed to gain access to the remote files.

Client Server Model:

Remote file systems allow a computer to mount one or more file systems from one or more remote machines.

The machine containing the files is the **server**, and the machine seeking access to the files is the **client.**

The server declares that a resource is available to clients and specifies exactly which resource is shared by which clients.

A server can serve multiple clients, and a client can use multiple servers.

A client can be specified by a network name or other identifier, such as an IP address but these can be **spoofed**, or imitated.

As a result of spoofing, an unauthorized client could be allowed access to the server.

In the case of UNIX and its network file system (NFS), authentication takes place via the client networking information.

The user's IDs on the client and server must match. If they do not, the server will be unable to determine access rights to files.

Distributed Information Systems

Distributed information systems, also known as **distributed naming services**, provide unified access to the information needed for remote computing.

The **domain name system (DNS)** provides host-name-to-network-address translations for the entire Internet.

Distributed information systems provide **user name/password/user ID/group ID** space for a distributed facility.

In the case of Microsoft's **common Internet file system (CIFS)**, network information is used in conjunction with user authentication to create a network login that the server uses to decide whether to allow or deny access to a requested file system.

Microsoft uses **active directory** as a distributed naming structure to provide a single name space for users. Once established, the distributed naming facility is used by all clients and servers to authenticate users.

LDAP: **lightweight directory-access protocol (LDAP)** is a secure distributed naming mechanism.

Failure Modes:

Local file systems can fail for a variety of reasons that includes

- Failure of the disk containing the file system,
- Corruption of the directory structure or other disk-management information
- Disk-controller failure,
- Cable failure,
- Host-adapter failure
- User or system-administrator failure

Remote file systems have more failure modes because of the complexity of network systems and the required interactions between remote machines.

The failure semantics are defined and implemented as part of the remote-file-system protocol.

Termination of all operations can result in users' losing data.

To implement the recovery from failure, some kind of **state information** may be maintained on both the client and the server.

If both server and client maintain knowledge of their current activities and open files, then they can recover from a failure.

Consistency Semantics:

Consistency semantics represent a criterion for evaluating any file system that supports file sharing.

The semantics specify how multiple users of a system are to access a shared file simultaneously.

They specify when modifications of data by one user will be observable by other users. These semantics are typically implemented as code with the file system.

Consistency semantics are directly related to the process synchronization algorithms.

A series of file accesses attempted by a user to the same file is always enclosed between the open() and close() operations.

The series of accesses between the open() and close() operations makes up a **file session**

The Examples Of Consistency semantics includes

- Unix Semantics
- Session Semantics
- Immutable shared File Semantics

i) 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 who have this file open.

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

ii) Session Semantics:

The Andrew file system 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.

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.

Immutable Shared File Semantics:

The Immutable Shared file system uses the following consistency semantics

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

An immutable file signifies that the contents of the file are fixed.

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.

File Protection is also defined as the process of protecting the file of a user from unauthorized access or any other physical damage.

Goals of Protection:

To prevent malicious misuse of the system by users or programs. See chapter 15 for a more thorough coverage of this goal.

To ensure that each shared resource is used only in accordance with system policies, which may be set either by system designers or by system administrators.

To ensure that errant programs cause the minimal amount of damage possible.

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 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.

These higher-level functions may be implemented by a system program that makes lower-level system calls.

Access control:

The most common approach to the protection problem is to make access dependent on the identity of the user.

Different users may need different types of access to a file or directory.

The general scheme to implement identity dependent access is to associate with each file and directory an **access-control list (ACL)** specifying user names and the types of access allowed for each user.

When a user requests access to a particular file, the operating system checks the access list associated with that file.

If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs, and the user job is denied access to the file.

The main problem with access lists is their length.

This technique has two undesirable consequences:

Constructing such a list may be a tedious and unrewarding task, especially if we do not know in advance the list of users in the system.

The directory entry, previously of fixed size, now must be of variable size, resulting in more complicated space management

To reduce 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

Universe. All other users in the system constitute the universe

EXAMPLE: consider a person, Sara, who is writing a new book. She has hired three graduate students (Jim, Dawn, and Jill) to help with the project. The text of the book is kept in a file named book.tex

The protection associated with this file is as follows:

Sara should be able to invoke all operations on the file.

Jim, Dawn, and Jill should be able only to read and write the file; they should not be allowed to delete the file.

All other users should be able to read, but not write, the file.

In the UNIX system, groups can be created and modified only by the manager of the facility

Three fields are needed to define protection. Each field is a collection of bits, and each bit either allows or prevents the access associated with it.

The UNIX system defines three fields of 3 bits each—rwx where r controls read access, w controls write access, and x controls execution.

A separate field is kept for the file owner, for the file's group, and for all other users.

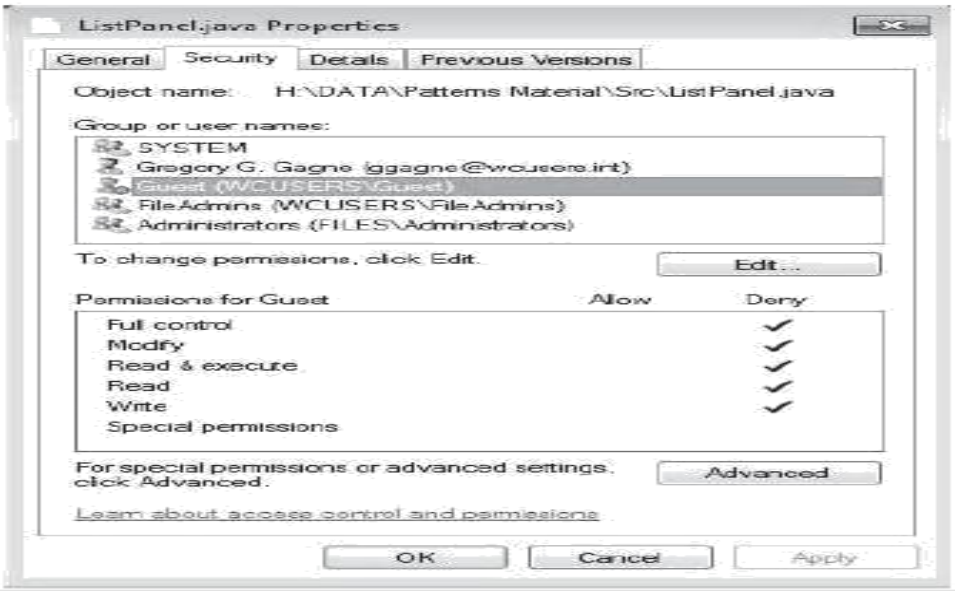
In this scheme, 9 bits per file are needed to record protection information.

Thus, for our example, the protection fields for the file book.tex are as follows

For the owner Sara, all bits are set;

For the group text, the r and w bits are set;

For the universe, only the r bit is set.



Windows users typically manage access-control lists via the Graphical User Interface .

Password Protection:

Another approach to the protection problem is to associate a password with each file.

Access to each file can be controlled with the help of passwords.

If the passwords are chosen randomly and changed often, this scheme may be effective in limiting access to a file.

Disadvantages: The number of passwords that a user needs to remember may become large.

If only one password is used for all the files, then once it is discovered, all files are accessible; protection is on an all-or-none basis.

In a multilevel directory structure, we need to protect not only individual files but also collections of files in subdirectories;

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.

FILE SYSTEM STRUCTURE:

The file system provides the mechanism for on-line storage and access to file contents, including data and programs.

The file system resides permanently on secondary storage, which is designed to hold a large amount of data permanently.

Disks provide most of the secondary storage on which file systems are maintained. Two characteristics make them convenient for this purpose:

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.

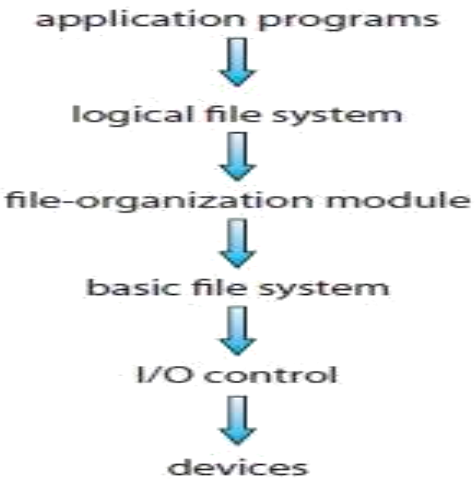
A disk can access directly any block of information it contains. Thus, it is simple to access any file either sequentially or randomly, and switching from one file to another requires only moving the read– write heads and waiting for the disk to rotate.

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 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 .



I/O Control:

The **I/O control** level consists of device drivers and interrupts handlers to transfer information between the main memory and the disk 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.

Basic File System:

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.

This layer also manages the memory buffers and caches that hold various file-system, directory, and data blocks

A block in the buffer is allocated before the transfer of a disk block can occur

Caches are used to hold frequently used file-system metadata to improve performance

File Organization Module:

The **file-organization module** knows about files and their logical blocks, as well as physical blocks.

The file-organization module can translate logical block addresses to physical block addresses for the basic file system to transfer.

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.

Logical File System:

The **logical file system** manages metadata information

The logical file system manages the directory structure to provide the file-organization module with the information it needs.

It maintains file structure via file-control blocks

A **file control block (FCB)** (an **inode** in UNIX file systems) contains information about the file, including ownership, permissions, and location of the file contents.

Advantages of Layered File system :

When a layered structure is used for file-system implementation, duplication of code is minimized

Each file system can then have its own logical file-system and file-organization modules

Disadvantages: The use of layering, including the decision about how many layers to use and what each layer should do, is a major challenge in designing new systems.

EXAMPLE FILE SYSTEMS:

UNIX uses the **UNIX file system (UFS)**, which is based on the Berkeley Fast File System (FFS).

Windows supports disk file-system formats of FAT, FAT32, and NTFS (or Windows NT File System), as well as CD-ROM and DVD file-system formats.

Although Linux supports over forty different file systems, the standard Linux file system is known as the **extended file system**, with the most common versions being ext3 and ext4.

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

The On Disk Structure of File system Provides the details such as

Boot Control Block

Volume Control Block

Directory Structure

File Control Block

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.

In UFS (Unix File System), it is called the **boot block**. In NTFS (New Technology File System), it is the **partition boot sector**.

Volume Control Block:

volume control block (per volume) contains volume (or partition) details, such as the number of blocks in the partition, the size of the blocks, a free-block count and free-block pointers, and a free-FCB count and FCB pointers.

In UFS, this is called a **superblock**. In NTFS, it is stored in the **master file table**.

Directory Structure:

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.

File Control Block:

A per-file FCB contains many details about the file.
It has a unique identifier number to allow association with a directory entry.
In NTFS, this information is actually stored within the master file table, which uses a relational database structure, with a row per file.

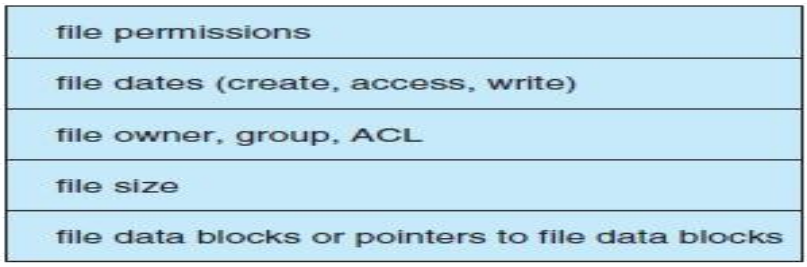


Figure 12.2 A typical file-control block.

The in memory Structure of File system provides the details such as

- An in-memory mount table
- An in-memory directory-structure cache
- The system-wide open-file table
- The per-process open-file table
- Buffers hold file-system blocks

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.
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.
Buffers hold file-system blocks when they are being read from disk or written to disk.

To create a new file, an application program calls the logical file system. To create a new file, it allocates a new FCB.

The system then reads the appropriate directory into memory, updates it with the new file name and FCB, and writes it back to the disk.

Example: A file has been created, and it can be used for I/O.It must be opened to perform I/O.

The open() call passes a file name to the logical file system. The open () system call first searches the system-wide open-file table to see if the file is already in use by another process.
If it is, a per-process open-file table entry is created pointing to the existing system-wide open-file table.
If the file is not already open, the directory structure is searched for the given file name.
Once the file is found, the FCB is copied into a system-wide open-file table in memory.
Next, an entry is made in the per-process open-file table, with a pointer to the entry in the system-wide open-file table and some other fields.

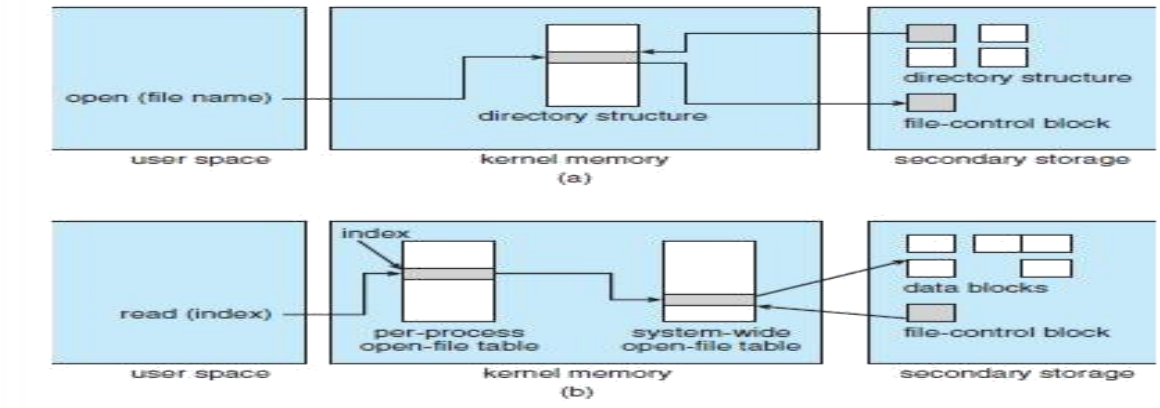
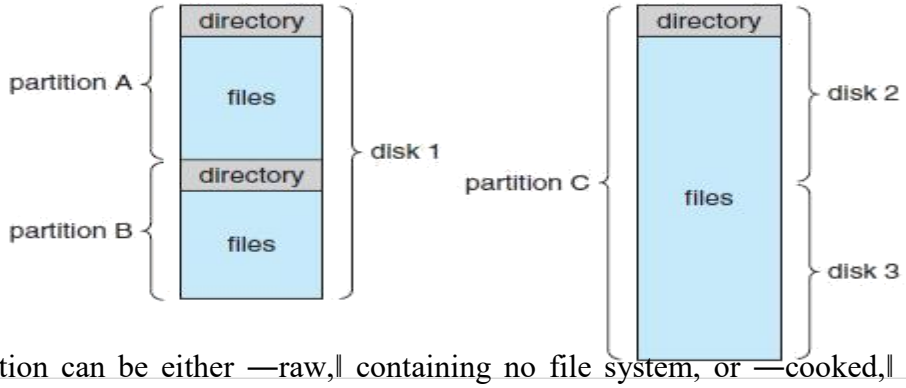


Figure 12.3 In-memory file-system structures. (a) File open. (b) File read.

When a process closes the file, the per-process table entry is removed, and the system-wide entry's open count is decremented.

PARTITIONING AND MOUNTING:

A disk can be sliced into multiple partitions, or a volume can span multiple partitions on multiple disks



Each partition can be either ―raw,‖ containing no file system, or ―cooked,‖ containing a file system.

Raw disk is used where no file system is appropriate.

Boot information can be stored in a separate partition,

The **boot loader** in turn knows enough about the file-system structure to be able to find and load the kernel and start it executing.

Many systems can be **dual-booted**, allowing us to install multiple operating systems on a single system.

The **root partition**, which contains the operating-system kernel and sometimes other system files, is mounted at boot time.

Microsoft Windows–based systems mount each volume in a separate name space, denoted by a letter and a colon.

VIRTUAL FILE SYSTEMS:

Modern operating systems must concurrently support multiple types of file systems. Virtual File Systems allow multiple types of file systems to be integrated into a directory structure.

Implementing multiple types of file systems requires writing directory and file routines for each type.

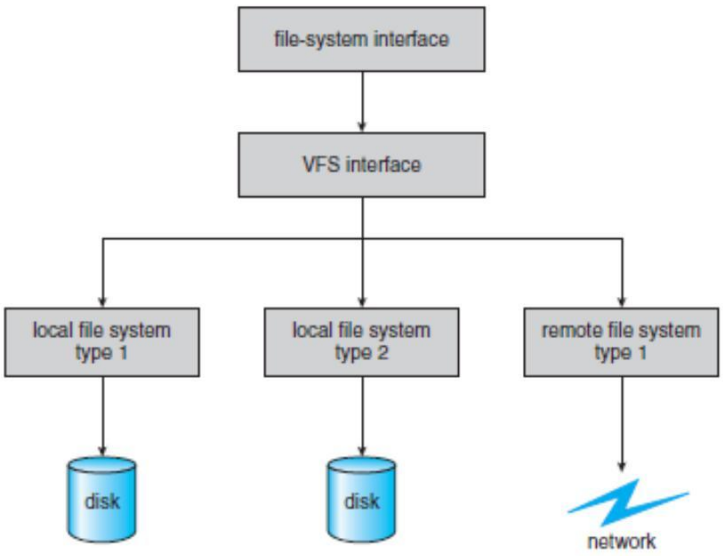
Users can access files contained within multiple file systems on the local disk or even on file systems available across the network.

Thus, the file-system implementation consists of three major layers

File System Interface

Virtual File System

Remote File System Protocol



The first layer is the file-system interface, based on the open(), read(), write(), and close() calls and on file descriptors

The second layer is called the **virtual file system (VFS)** layer. The VFS layer serves two important functions:

It separates file-system-generic operations from their implementation by defining a clean VFS interface.

It provides a mechanism for uniquely representing a file throughout a network. The VFS is based on a file-representation structure, called a **vnode** that contains a numerical designator for a network-wide unique file.

Thus, the VFS distinguishes local files from remote ones, and local files are further distinguished according to their file-system types.

The layer implementing the file-system type or the remote-file-system protocol is the third layer of the architecture.

EXAMPLE: VFS architecture in Linux

The four main object types defined by the Linux VFS are:

The **inode object**, which represents an individual file

The **file object**, which represents an open file

The **superblock object**, which represents an entire file system

The **dentry object**, which represents an individual directory entry

For each of these four object types, the VFS defines a set of operations that may be implemented.

Every object of one of these types contains a pointer to a function table. The function table lists the addresses of the actual functions that implement the defined operations for that particular object.

int open(. . .)—Open a file.

int close(...)—Close an already-open file.

ssize t read(. . .)—Read from a file.

ssize t write(. . .)—Write to a file.

int mmap(. . .)—Memory-map a file.

Thus, the VFS software layer can perform an operation on one of these objects by calling the appropriate function from the object's function table, without having to know in advance exactly what kind of object it is dealing with.

FILE / DISK ALLOCATION TECHNIQUES:

Many files are stored on the same disk. The File System allocates 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,

Contiguous Allocation

Linked Allocation

Indexed Allocation

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.
It supports both direct and sequential access. For direct access to block i of a file that starts at block b, we can immediately access block b + i.
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.

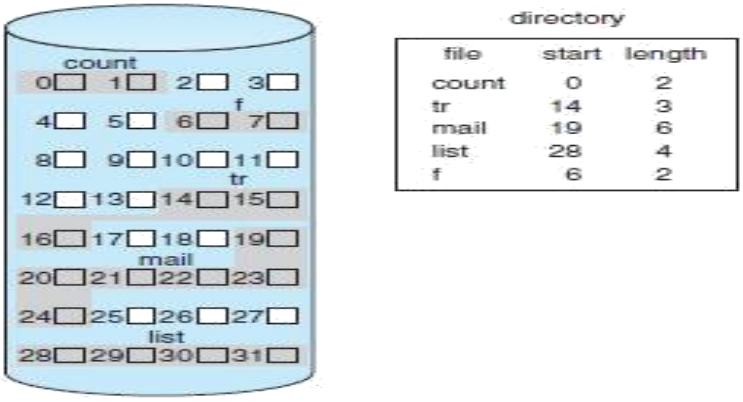


Figure 12.5 Contiguous allocation of disk space.

Advantages:

The number of disk seeks required for accessing contiguously allocated files is minimal.
Contiguous allocation of a file is defined by the disk address and length of the first block.
Accessing a file that has been allocated contiguously is easy.
One difficulty is finding space for a new file.
Contiguous memory allocation suffers from the problem of external fragmentation.

EXTERNAL FRAGMENTATION: As files are allocated and deleted, the free disk space is broken into little pieces. The total available space may not be enough to satisfy a request. Storage is fragmented into a number of holes, none of which is large enough to store the data.

SOLUTION: One strategy for preventing external fragmentation is to copy an entire file system onto another disk. The original disk is then freed completely, creating one large contiguous free space .We then copy the files back onto the original disk by allocating contiguous space from this one large hole. This is called as **Compaction.**

Another Problem with contiguous allocation is determining how much space is needed for a file.
Even if the total amount of space needed for a file is known in advance, preallocation may be inefficient.
To minimize these drawbacks, some operating systems use a contiguous chunk of space that is allocated initially.
If that amount proves not to be large enough, another chunk of contiguous space, known as an **extent**, is added.

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. 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.
Each directory entry has a pointer to the first disk block of the file.

A write to the file causes the free-space management system to find a free block, and this new block is written to and is linked to the end of the file.
To read a file, we simply read blocks by following the pointers from block to block.

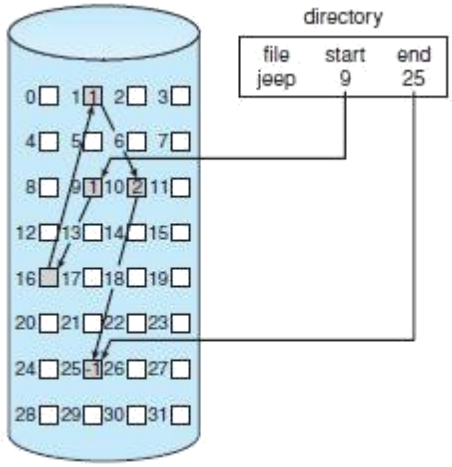


Figure 12.6 Linked allocation of disk space.

Advantages:

There is no external fragmentation with linked allocation, and any free block on the free-space list can be used to satisfy a request.
The size of a file need not be declared when the file is created.
A file can continue to grow as long as free blocks are available
It is inefficient to support a direct-access capability for linked-allocation files.
It requires more disk space for storing the pointers.
Solution: The usual solution to this problem is to collect blocks into multiples, called **clusters**, and to allocate clusters rather than blocks.
An important variation on linked allocation is the use of a **file-allocation table (FAT)**.
A section of disk at the beginning of each volume contains the table.
The table has one entry for each disk block and is indexed by block number.
The FAT is used in much the same way as a linked list. The directory entry contains the block number of the first block of the file.
The table entry indexed by that block number contains the block number of the next block in the file. This chain continues until it reaches the last block.

Indexed Allocation

In Indexed allocation each file has its own index block, which is an array of disk-block addresses.
The directory contains the address of the index block
The ith entry in the index block points to the ith block of the file.
To find and read the ith block, we use the pointer in the ith index-block entry.
When the file is created, all pointers in the index block are set to null.
When the ith block is first written, a block is obtained from the free-space manager, and its address is put in

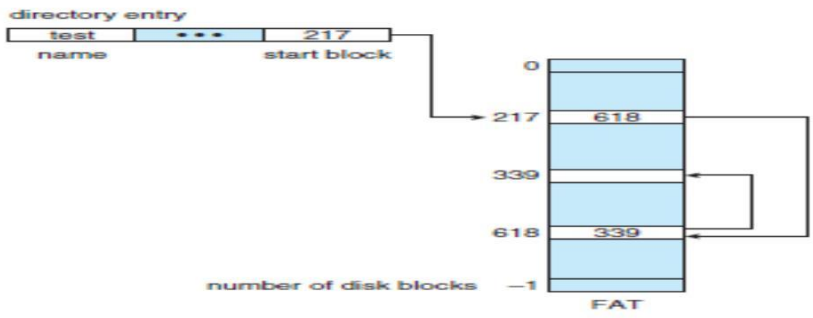


Figure 12.7 File-allocation table.

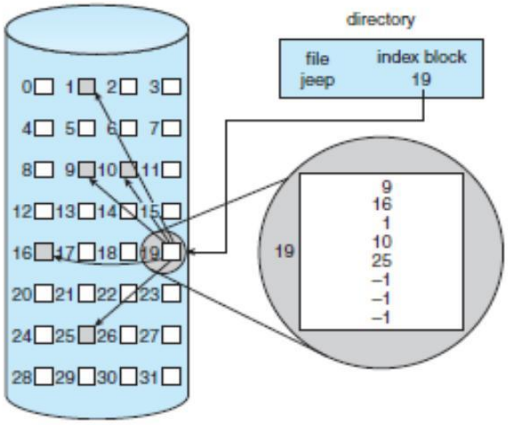


Figure 12.8 Indexed allocation of disk space.

Advantages:

Indexed allocation supports direct access

It does not suffer from External fragmentation. because any free block on the disk can satisfy a request for more Space

Indexed allocation does suffer from wasted space

The Pointer overhead of the index block is generally greater than the pointer overhead of linked allocation

Every file must have an index block, so we want the index block to be as small as possible.

Mechanisms for implementing the index block includes:

Linked scheme.

Multilevel index

Combined scheme.

An index block is normally one disk block. Thus, it can be read and written directly by itself.

To allow for large files, we can link together several index blocks.

Example: An index block might contain a small header giving the name of the file and a set of the first 100 disk-block addresses.

The next address is a pointer to another index block.

Multilevel index representation uses a first-level index block to point to a set of second-level index blocks, which in turn point to the file blocks.

To access a block, the operating system uses the first-level index to find a second-level index block and then uses that block to find the desired data block.

This approach could be continued to a third or fourth level, depending on the desired maximum file size.

Combined scheme:

Combined scheme is a combination of both linked and multilevel implementation.

EXAMPLE: Consider a UNIX-based file systems, which keeps 15 pointers of the index block in the file's inode(FAT).

The first 12 of these pointers point to **direct blocks**; that is, they contain addresses of blocks that contain data of the file.

The next three pointers point to **indirect blocks**.

The first points to a **single indirect block**, which is an index block containing not data but the addresses of blocks that do contain data.

The second points to a **double indirect block**, which contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks.

The last pointer contains the address of a **triple indirect block**.

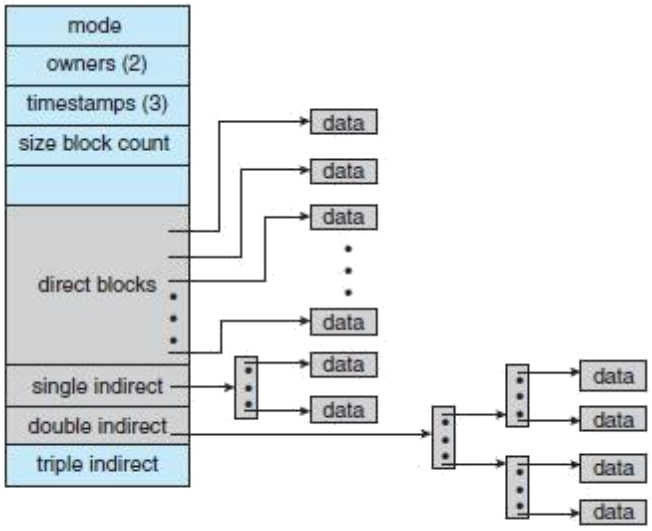


Figure 12.9 The UNIX inode.

FREE SPACE MANAGEMENT:

A number of files can be created and deleted by the user in a secondary storage device.

Since disk space is limited, we need to reuse the space from deleted files for new files.

FREE SPACE LIST:

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 can be implemented in the following ways

- Bit Vector or Bit Map
- Linked list
- Groping
- Counting
- Space maps

Bit Vector:

The free-space list is implemented as a **bit map** or **bit vector**.

Each block is represented by 1 bit. If the block is free, the bit is 1; if the block is allocated, the bit is 0.

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.

Free-space bit map:

01111001111110001100000011100000...

One technique for finding the first free block on a system that uses a bit-vector is to sequentially check each word in the bit map to see whether that value is not 0.

The first non-0 word is scanned for the first 1 bit, which is the location of the first free block. The calculation of the block number is **(number of bits per word) × (number of 0-value words) + offset of first 1 bit.**

Advantages:

The main advantage of this approach is its relative simplicity and its efficiency in finding the first free block or n consecutive free blocks on the disk.

Bit vectors are inefficient unless the entire vector is kept in main memory.

If the disk size constantly increases, the problem with bit vectors will continue to increase.

Linked list:

Linked list implementation link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory.

This first block contains a pointer to the next free disk block, and so on

Example: Blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 in the disk were free and the rest of the blocks were allocated.

We would keep a pointer to block 2 as the first free block. Block 2 would contain a pointer to block 3, which would point to block 4, which would point to block 5, which would point to block 8, and so on

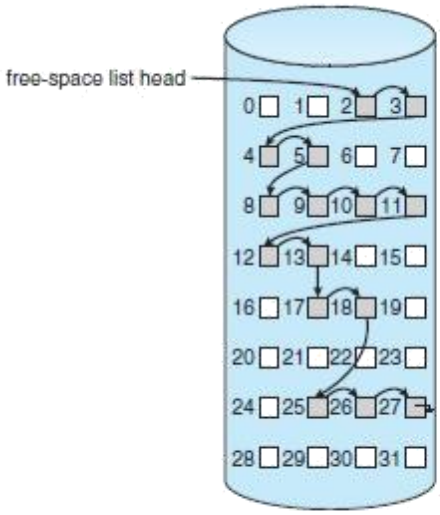


Figure 12.10 Linked free-space list on disk.

Disadvantages:

This scheme is not efficient; to traverse the list, we must read each block, which requires substantial I/O time.

Grouping:

A modification of the free-list approach stores the addresses of n free blocks in the first free block.

The first n−1 of these blocks are actually free. The last block contains the addresses of other n free blocks, and so on.

The addresses of a large number of free blocks can now be found quickly.

Counting:

Free space list keep the address of the first free block and the number (n) of free contiguous blocks that follow the first block.

Each entry in the free-space list then consists of a disk address and a count.

This method of tracking free space is similar to the extent method of allocating blocks.

These entries can be stored in a balanced tree, rather than a linked list, for efficient lookup, insertion, and deletion.

Space maps:

Oracle's **ZFS** file system creates **Meta slabs** to divide the space on the device into chunks of manageable size.

A given volume may contain hundreds of Meta slabs. Each Meta slab has an associated space map.

ZFS uses the counting algorithm to store information about free blocks.

A space map uses log-structured file-system techniques to record the information about the free blocks.

The space map is a log of all block activity (allocating and freeing), in time order, in counting format.

When ZFS decides to allocate or free space from a Meta slab, it loads the associated space map into memory in a balanced-tree structure, indexed by offset, and replays the log into that structure.

The in-memory space map is then an accurate representation of the allocated and free space in the Meta slab.

I/O SYSTEMS:

The role of the operating system in computer I/O is to manage and control I/O operations and I/O devices. This is met by a combination of hardware device controllers and software device driver techniques**.**

Computers operate a great many kinds of devices. These devices are grouped under various categories that includes

Storage devices – Disks and Tapes

Transmission devices – Networks cards, modems, Bluetooth

Human Interface devices – Monitor, Keyboard, Mouse, Printer.

A device communicates with a computer system by sending signals over a cable.

PORT: The device communicates with the machine via a connection point called a port.

BUS: If devices share 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.

DAISY CHAIN: 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.

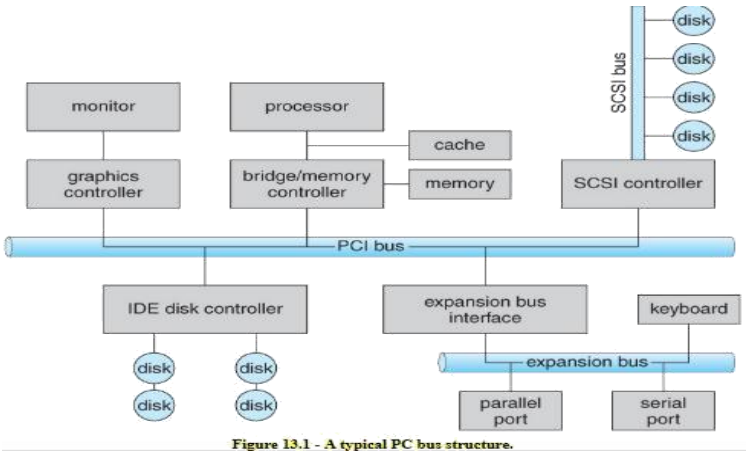


Figure 13.1 - A typical PC bus structure.

PCI – Peripheral Component Interconnect

SCSI – Small Computer Systems Interface

IDE – Integrated Drive Electronics

A PCI bus (the common PC system bus) connects the processor–memory subsystem to fast devices

An expansion bus connects relatively slow devices, such as the keyboard and serial and USB ports.

A Device controller is a collection of electronics that can operate a port, a bus, or a device.

The processor gives commands and data to a controller to accomplish an I/O transfer.

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.

MEMORY MAPPED I/O: The CPU executes I/O requests using the standard data-transfer instructions to read and write the device-control registers at their mapped locations in physical memory.

An I/O port typically consists of four registers, called the status, control, data-in, and 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 that 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.

POLLING:

HANDSHAKING: It is a protocol for interaction between the host and a controller.

The controller indicates its state through the busy bit in the status register.

The controller sets the busy bit when it is busy working and clears the busy bit when it is ready to accept the next command.

The host signals its wishes via the command-ready bit in the command register.

The host sets the command-ready bit when a command is available for the controller to execute.

POLLING: The host repeatedly reads the busy bit until that bit becomes clear. This process is called as polling or busy waiting.

The host sets the write bit in the command register and writes a byte into the data-out register.

The host sets the command-ready bit.

When the controller notices that the command-ready bit is set, it sets the busy bit.

The controller reads the command register and sees the write command. It reads the data-out register to get the byte and does the I/O to the device.

The controller clears the command-ready bit, clears the error bit in the status register to indicate that the device I/O succeeded, and clears the busy bit to indicate that it is finished.

INTERUPPTS:

An interrupt is a signal to the kernel (i.e., the core of the operating system) that an event has occurred, and this result in changes in the sequence of instructions that is executed by the CPU.

The CPU hardware has a wire called the **interrupt-request line** that the CPU senses after executing every instruction.

When the CPU detects that a controller has asserted a signal on the interrupt-request line, the CPU performs a state save and jumps to the **interrupt-handler routine** at a fixed address in memory.

The device controller **raises** an interrupt by asserting a signal on the interrupt request line, the CPU **catches** the interrupt and **dispatches** it to the interrupt handler, and the handler **clears** the interrupt by servicing the device.

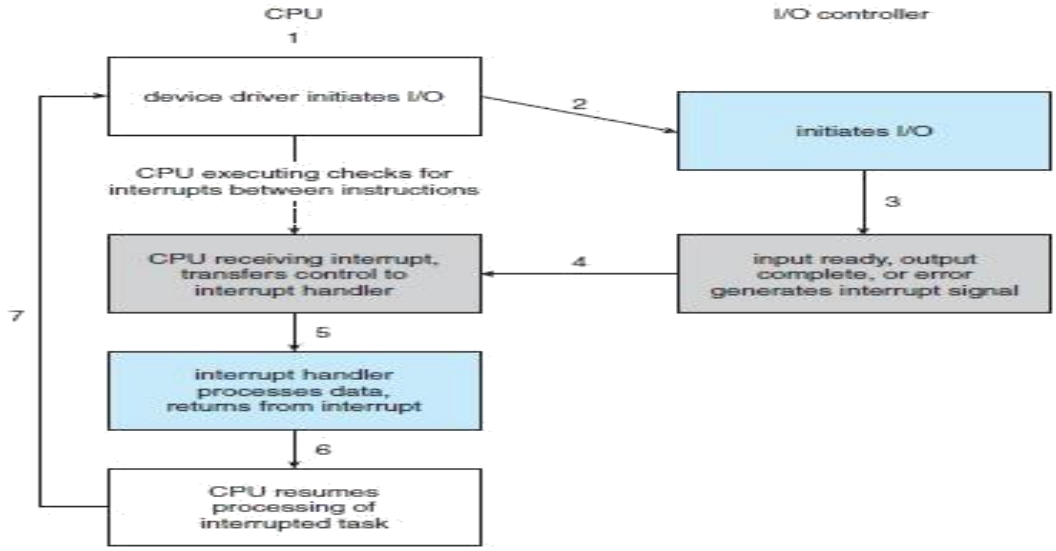


Figure 13.3 Interrupt-driven I/O cycle.

In a modern operating system, we need more sophisticated interrupt-handling features

We need the ability to suspend interrupt handling during critical processing.

We need an efficient way to dispatch to the proper interrupt handler for a device without first polling all the devices to see which one raised the interrupt.

We need multilevel interrupts, so that the operating system can distinguish between high- and low-priority interrupts and can respond with the appropriate degree of urgency.

Most CPUs have two interrupt request lines.

Non Maskable Interrupt

Maskable Interrupt

The Non maskable interrupts are those that cannot be ignored by the processor. It is reserved for events such as unrecoverable memory errors.

The Maskable interrupts are those that can be ignored by the processor. The maskable interrupt is used by device controllers to request service.

Interrupt vector: The interrupt mechanism accepts an **address**—a number that selects a specific interrupt-handling routine from a small set. This address is an offset in a table called the **interrupt vector**.

The purpose of a vectored interrupt mechanism is to reduce the need for a single interrupt handler to search all possible sources of interrupts to determine which one needs service.

Interrupt Chaining: A common way to solve this problem is to use **interrupt chaining**, in which each element in the interrupt vector points to the head of a list of interrupt handlers.

The interrupt mechanism also implements a system of **interrupt priority levels**.

These levels enable the CPU to suspend the handling of low-priority interrupts without masking all interrupts and makes it possible for a high priority interrupt to preempt the execution of a low-priority interrupt.

The interrupt mechanism is also used to handle a wide variety of **exceptions**, such as dividing by 0, accessing a protected or nonexistent memory address, or attempting to execute a privileged instruction from user mode.

EXAMPLE: A page fault is an exception that raises an interrupt. The interrupt suspends the current process and jumps to the page-fault handler in the kernel.

This handler saves the state of the process, moves the process to the wait queue, performs page-cache management, schedules an I/O operation to fetch the page, schedules another process to resume execution, and then returns from the interrupt.

Interrupts can also be used to manage the flow of control within the kernel.

Thus interrupts are used throughout modern operating systems to handle asynchronous events and to trap to supervisor-mode routines in the kernel.

DIRECT MEMORY ACCESS:

In Programmed I/O, when the processor is executing a program and encounters an instruction relating to I/O, it executes that instruction by issuing a command to the appropriate I/O module. The I/O module performs the requested action and takes no action to alert the processor and it does not interrupt the processor. The processor periodically checks the status of the I/O module until it finds that the operation is complete. This burdens the CPU.

Many computers avoid burdening the main CPU with Programmed I/O by offloading some of this work to a special-purpose processor called a **direct-memory-access (DMA)** controller.

To initiate a DMA transfer, the host writes a DMA command block into memory.

This block contains a pointer to

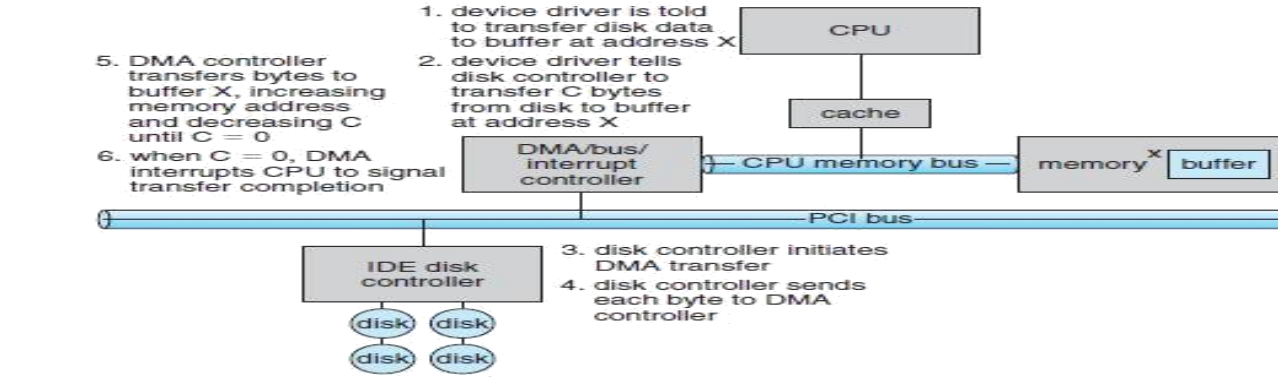
The source of a transfer,

A pointer to the destination of the transfer,

A count of the number of bytes to be transferred.

The CPU writes the address of this command block to the DMA controller, then goes on with other work.

The DMA controller proceeds to operate the memory bus directly, placing addresses on the bus to perform transfers without the help of the main CPU.



Handshaking between the DMA controller and the device controller:

Handshaking between the DMA controller and the device controller is performed via a pair of wires called

DMA-request and **DMA-acknowledge**.

The device controller places a signal on the DMA-request wire when a word of data is available for transfer.

When the device controller receives the DMA-acknowledge signal, it transfers the word of data to memory and removes the DMA-request signal.

When the entire transfer is finished, the DMA controller interrupts the CPU.

CYCLE STEALING: When the DMA controller seizes the memory bus, the CPU is prevented from accessing main memory, although it can still access data items in its primary and secondary caches.This is called as Cycle stealing.

This **cycle stealing** can slow down the CPU computation, offloading the data-transfer work to a DMA controller generally improves the total system performance.

DIRECT VIRTUAL MEMORY ADDRESS: Some computer architectures use physical memory addresses for DMA, but others perform **direct virtual memory access (DVMA)**, using virtual addresses that undergo translation to physical addresses.

DVMA can perform a transfer between two memory-mapped devices without the intervention of the CPU or the use of main memory.

On protected-mode kernels, the operating system generally prevents processes from issuing device commands directly.

This direct access can be used to achieve high performance, since it can avoid kernel communication, context switches, and layers of kernel software.

APPLICATION I/O INTERFACE:

The operating system acts as an interface between the application programs and the I/O Devices.
The interface involves abstraction, encapsulation, and software layering.
The kernel modules called device drivers are internally custom-tailored to specific devices.
The purpose of the device-driver layer is to hide the differences among device controllers from the I/O subsystem of the kernel
The kernels allows the existing I/O Subsystem for new devices to be compatible with an existing host controller interface or they write device drivers to interface the new hardware to popular operating systems.

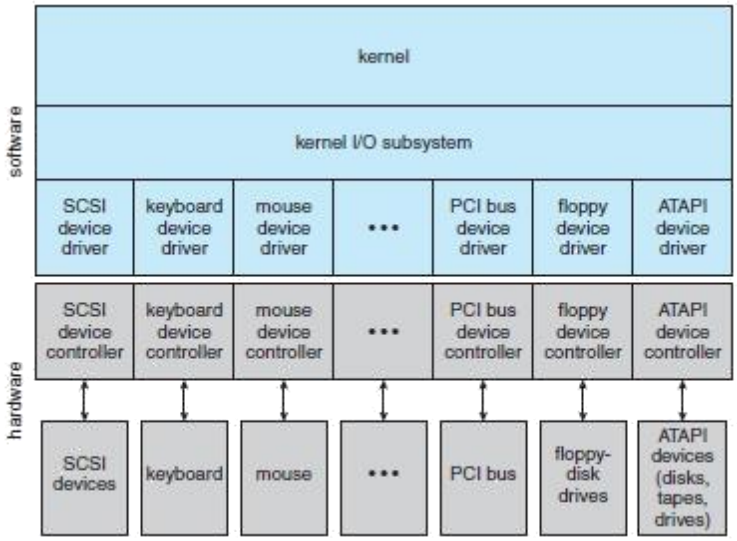


Figure 13.6 A kernel I/O structure.

Devices vary on many dimensions such as

- Character-stream or block
- Sequential or random access
- Synchronous or asynchronous
- Sharable or dedicated
- Speed of operation
- Read–write, read only, or write only
- Character-stream or block

Character-stream or block: A character-stream device transfers bytes one by one, whereas a block device transfers a block of bytes as a unit.
Sequential or random access: A sequential device transfers data in a fixed order determined by the device, whereas the user of a random-access device can instruct the device to seek to any of the available data storage locations.
Synchronous or asynchronous. A synchronous device performs data transfers with predictable response times and an asynchronous device exhibits irregular or unpredictable response times.
Sharable or dedicated. A sharable device can be used concurrently by several processes or threads; a dedicated device cannot.
Speed of operation. Device speeds range from a few bytes per second to a few gigabytes per second.
Read–write, read only, or write only. Some devices perform both input and output, but others support only one data transfer direction.
For the application programs, many of these differences are hidden by the operating system, and the devices are grouped into a few conventional types.

Operating systems also provide special system calls to access a few additional devices, such as a time-of-day clock and a timer.
Some operating systems provide a set of system calls for graphical display, video, and audio devices.
ESCAPE: Most operating systems also have an **escape** (or **back door**) that transparently passes arbitrary commands from an application to a device driver.

Block and Character Devices:

The **block-device interface** captures all the aspects necessary for accessing disk drives and other block-oriented devices.

The device is expected to understand commands such as read () and write (). If it is a random-access device, it is also expected to have a seek () command to specify which block to transfer next.

The read (), write (), and seek () system calls are the essential behaviors of block-storage devices, so that applications are insulated from the low-level differences among those devices.

RAW I/O: The operating system itself, as well as special applications such as database management

Systems may prefer to access a block device as a simple linear array of blocks. This mode of access is called **raw I/O**.

DIRECT I/O: operating system allows a mode of operation on a file that disables buffering and locking. In the UNIX, this is called **direct I/O**.

A keyboard is an example of a device that is accessed through a **character stream interface**. The basic system calls in this interface enable an application to get () or put () one character.

Network devices:

Most operating systems provide a network I/O interface that is different from the read()–write()–seek() interface used for disks, Because the performance and addressing characteristics of network I/O differ significantly from those of disk I/O.

One interface available in many operating systems, including UNIX and Windows, is the network **socket** interface.

Many other approaches to interprocess communication and network communication have also been implemented

EXAMPLE: Windows provides one interface to the network interface card and a second interface to the network protocols.

Clocks and Timers:

Most computers have hardware clocks and timers that provide three basic functions:

Give the current time.

Give the elapsed time.

Set a timer to trigger operation X at time T.

PROGRAMMABLE INTERVAL TIMER: The hardware to measure elapsed time and to trigger operations is called a programmable interval timer.

It can be set to wait a certain amount of time and then generate an interrupt, and it can be set to do this once or to repeat the process to generate periodic interrupts.

EXAMPLE: The scheduler uses this mechanism to generate an interrupt that will preempt a process at the end of its time slice.

Nonblocking and Asynchronous I/O:

When an application issues a **blocking** system call, the execution of the application is suspended.

The application is moved from the operating system's run queue to a wait queue. After the system call completes, the application is moved back to the run queue, where it is eligible to resume execution.

The physical actions performed by I/O devices are generally asynchronous.

Some user-level processes need nonblocking I/O. One example is a user interface that receives keyboard and mouse input while processing and displaying data on the screen.

An alternative to a nonblocking system call is an asynchronous system call. An asynchronous call returns immediately, without waiting for the I/O to complete.

The difference between nonblocking and asynchronous system calls is that a nonblocking read() returns immediately with whatever data are available and an asynchronous read() call requests a transfer that will be performed in its entirety but will complete at some future time.

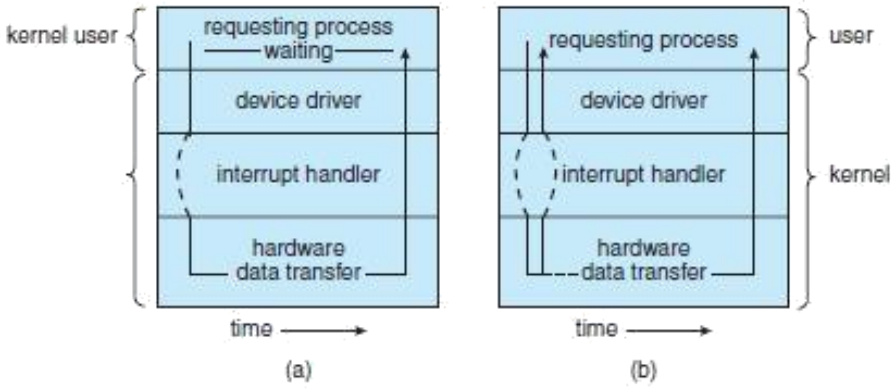


Figure 13.8 Two I/O methods: (a) synchronous and (b) asynchronous.

Vectored I/O:

Vectored I/O allows one system call to perform multiple I/O operations involving multiple locations.

EXAMPLE:

SCATTER – GATHER METHOD: A system call in UNIX accepts a vector of multiple buffers and either reads from a source to that vector or writes from that vector to a destination**.** The same transfer could be caused by several individual invocations of system calls.

KERNEL I/O SUBSYSTEM:

Kernels provide many services related to I/O. The I/O subsystem is also responsible for protecting itself from errant processes and malicious users. The Services include

I/O Scheduling
Buffering
Caching
Spooling and device reservation
Error Handling
I/O Protection
Kernel Data Structures

I/O Scheduling:

Scheduling can improve overall system performance, can share device access among processes, and can reduce the average waiting time for I/O to complete.

EXAMPLE: 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.

The OS must be able to keep track of many I/O requests at the same time. For this purpose, the operating system might attach the wait queue to a device-status table.

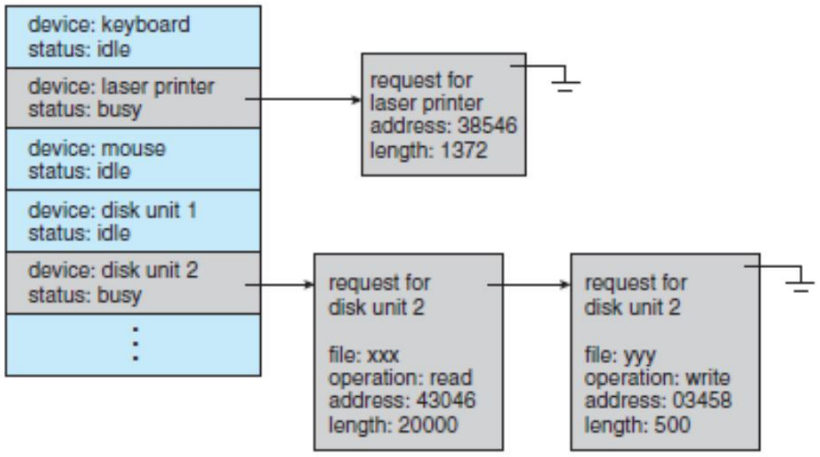


Figure 13.9 Device-status table.

Buffering:

A buffer, of course, is a memory area that stores data being 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

A second use of buffering is to provide adaptations for devices that have different data-transfer sizes

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

EXAMPLE: Consider a file that is being received via modem for storage on the hard disk. 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.

DOUBLE BUFFERING: 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 is called as double buffering.

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

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, holds a copy on faster storage of an item that resides.

A **spool** is a buffer that holds output for a device, such as a printer, that cannot accept interleaved data streams. Spooling is a way of operating systems to coordinate concurrent output

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.

The spool can be managed in two ways

System daemon process

In-kernel thread.

Some operating systems (including VMS) provide support for exclusive device access by enabling a process to allocate an idle device and to deallocate that device when it is no longer needed.

An operating system that uses protected memory can guard against many kinds of hardware and application errors.

Devices and I/O transfers can fail when a network becomes overloaded, or when a disk controller becomes defective. Operating systems can often compensate effectively for transient failures.

A disk read() failure results in a read() retry, and a network send() error results in a resend(), if the protocol so specifies.

An I/O system call will return one bit of information about the status of the call, signifying either success or failure.

In the UNIX operating system, an additional integer variable named errno is used to return an error code indicating the general nature of the failure.

EXAMPLE: A failure of a SCSI device is reported by the SCSI protocol in three levels of detail:

Sense Key - Identifies the general nature of the failure

Additional sense code - states the category of failure

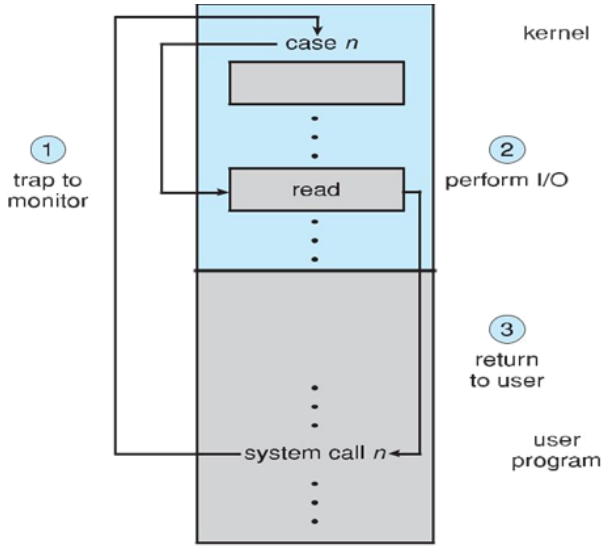
Additional sense-code qualifier -Specifies which command parameter was in error or which hardware subsystem failed its self-test.

A user process may attempt to disrupt the normal operation of a system by attempting to issue illegal I/O instructions.

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.

To do I/O, a user program executes a system call to request that the operating system perform I/O on its behalf. 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.

Any memory-mapped and I/O port memory locations must be protected from user access by the memory-protection system.



Kernel Data Structures:

The kernel needs to keep state information about the use of I/O components which is done through a variety of in-kernel data structures such as open file table, per process open file table, etc..

The kernel uses many similar structures to track network connections, character-device communications, and other I/O activities.

To read a user file, the kernel needs to probe the buffer cache before deciding whether to perform a disk I/O.

To read a raw disk, the kernel needs to ensure that the request size is a multiple of the disk sector size and is aligned on a sector boundary.

An I/O request is converted into a message that is sent through the kernel to the I/O manager and then to the device driver, each of which may change the message contents. For output, the message contains the data to be written. For input, the message contains a buffer to receive the data