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 system: Overview of Mass Storage Structure

Mass Storage system:

Mass Storage refers to systems meant to store large amounts of data. Mass storage system is where the operating system is stored, where all our PC programs are kept and where we keep the stuff we create and collect.

Magnetic Disks

Magnetic disk provides bulk of secondary storage for modern computer systems.

Traditional magnetic disks have the following basic structure:

One or more ***platters*** in the form of disks covered with magnetic media. ***Hard disk*** platters are made of rigid metal, while "***floppy***" disks are made of more flexible plastic.Common platter diameters range from **1.8 to 5.25 inches.**

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

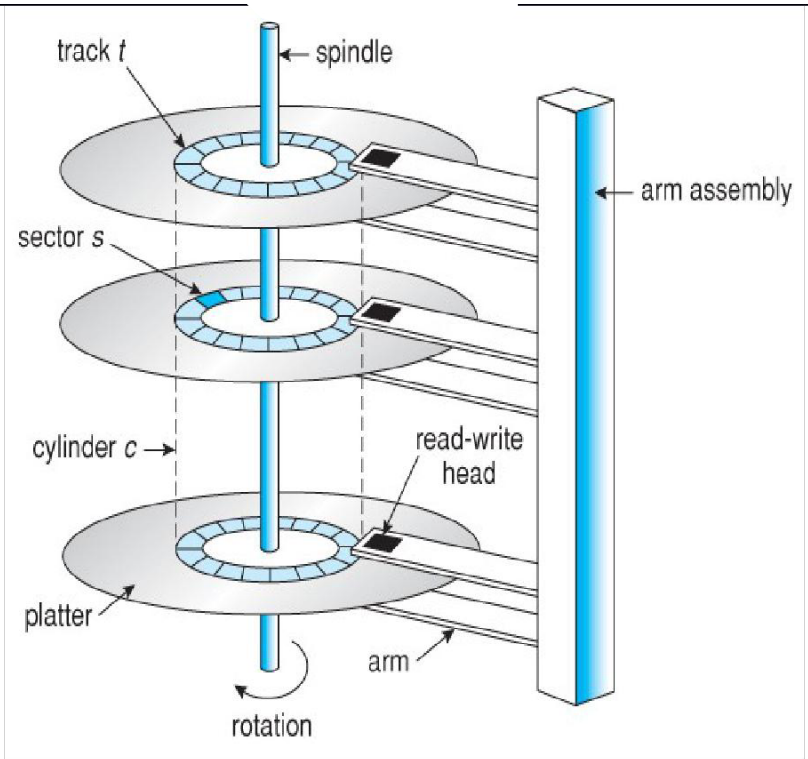
Each platter has two working ***surfaces.*** Older hard disk drives would sometimes not use the very top or bottom surface of a stack of platters, as these surfaces were more susceptible to potential damage.

Each working surface is divided into a number of concentric rings called ***tracks.*** The collection of all tracks that are the same distance from the edge of the platter, (i.e. all tracks immediately above one another in the following diagram) is called a ***cylinder***.

Each track is further divided into ***sectors,*** traditionally containing **512 bytes** of data each, although some modern disks occasionally use larger sector sizes. The data on a hard drive is read by read-write ***heads.*** The standard configuration (shown below) uses one head per surface, each on a separate ***arm***, and controlled by a common ***arm assembly*** which moves all heads simultaneously from one cylinder to another.

The storage capacity of a traditional disk drive is equal to the number of heads(i.e. the number of working surfaces), times the number of tracks per surface, times the number of sectors per track, times the number of bytes per sector.

In operation the disk rotates at high speed, such as 7200 rpm (120 revolutions per second.) The rate at which data can be transferred from the disk to the computer is composed of several steps:



Moving-head disk mechanism.

The ***positioning time***, the ***seek time*** or ***random access time*** is the time required to move the heads from one cylinder to another, and for the heads to settle down after the move. This is typically the slowest step in the process and the predominant bottleneck to overall transfer rates.

The ***rotational latency*** is the amount of time required for the desired sector to rotate around and come under the read-write head. This can range anywhere from zero to one full revolution, and on the average will equal one-half revolution. The ***transfer rate***, which is the time required to move the data electronically from the disk to the computer. Disk heads "fly" over the surface on a very thin cushion of air. If they should accidentally contact the disk, then a ***head crash*** occurs, which may or may not permanently damage the disk or even destroy it completely.

Floppy disks are normally ***removable***.Hard drives can also be removable, and some are even ***hot-swappable***, meaning they can be removed while the computer is running, and a new hard drive inserted in their place.

Disk drives are connected to the computer via a cable known as the ***I/O Bus.*** Some of the common interface formats include Enhanced Integrated Drive Electronics, EIDE; Advanced Technology Attachment, ATA; Serial ATA, SATA, Universal Serial Bus, USB; Fiber Channel, FC, and Small Computer Systems Interface, SCSI. The ***host controller*** is at the computer end of the I/O bus, and the ***disk controller*** is built into the disk itself.

The CPU issues commands to the host controller via I/O ports. Data is transferred between the magnetic surface and onboard ***cache*** by the disk controller, and then the data is transferred from that cache to the host controller and the motherboard memory at electronic speeds.

Solid-State Disks

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

Magnetic Tapes

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

Tapes are used mainly for backup, for storage of infrequently used information, and as a medium for transferring information from one system to another. Some tapes have built-in compressions that can more than double the effective storage. Tapes and their drivers are usually categorized by width, including **4, 8, and 19 millimeters and 1/4 and 1/2 inch.** Some are named according to technology, such as LTO-5 and SDLT.

Disk Structure:

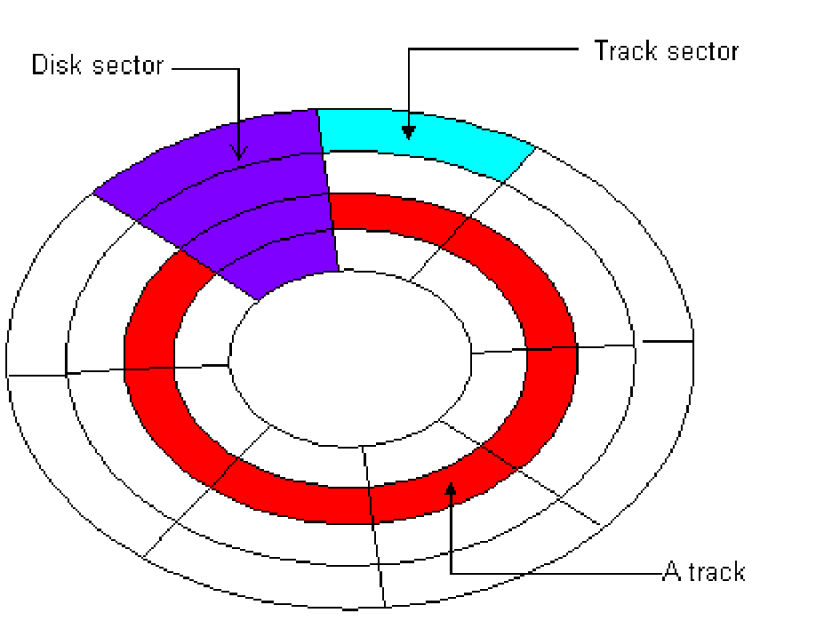
A hard disk is a memory storage device which looks like this:

The disk is divided into **tracks**. Each track is further divided into **sectors**. The point to be noted here is that outer tracks are bigger in size than the inner tracks but they contain the same number of sectors and have equal storage capacity.

This is because the storage density is high in sectors of the inner tracks whereas the bits are sparsely arranged in sectors of the outer tracks.

Some space of every sector is used for formatting. So, the actual capacity of a sector is less than the given capacity. Read-Write(R-W) head moves over the rotating hard disk.

It is this Read-Write head that performs all the read and write operations on the disk and hence, position of the R-W head is a major concern. To perform a read or write operation on a memory location, we need to place the R-W head over that position.



Some important terms must be noted here:

1. **Seek time –** The time taken by the R-W head to reach the desired track from it's current position.
2. **Rotational latency –** Time taken by the sector to come under the R-W head.
3. **Data transfer time –** Time taken to transfer the required amount of data. It depends upon the rotational speed.
4. **Controller time –** The processing time taken by the controller.
5. **Average Access time –** seek time + Average Rotational latency + data transfer time + controller time.

In questions, if the seek time and controller time is not mentioned, take them to be zero. If the amount of data to be transferred is not given, assume that no data is being transferred. Otherwise, calculate the time taken to transfer the given amount of data.

The average of rotational latency is taken when the current position of R-W head is not given. Because, the R-W may be already present at the desired position or it might take a whole rotation to get the desired sector under the R-W head. But, if the current position of the R-W head is given then the rotational latency must be calculated.

Example –
Consider a hard disk with:

4 surfaces
64 tracks/surface
128 sectors/track
256 bytes/sector

1. **What is the capacity of the hard disk?**
   Disk capacity = surfaces * tracks/surface * sectors/track * bytes/sector
   Disk capacity = 4 * 64 * 128 * 256
   Disk capacity = 8 MB

2. **The disk is rotating at 3600 RPM, what is the data transfer rate?**
   60 sec -> 3600 rotations
   1 sec -> 60 rotations
   Data transfer rate = number of rotations per second * track capacity * number of surfaces (since 1 R-W head is used for each surface)
   Data transfer rate = 60 * 128 * 256 *
   4 Data transfer rate = 7.5 MB/sec

3. **The disk is rotating at 3600 RPM, what is the average access time?**
   Since, seek time, controller time and the amount of data to be transferred is not given, we consider all the three terms as 0.
   Therefore, Average Access time = Average rotational
   delay Rotational latency => 60 sec -> 3600 rotations 1 sec
   -> 60 rotations
   Rotational latency = (1/60) sec = 16.67 msec.
   Average Rotational latency = (16.67)/2
   = 8.33 msec.
   Average Access time = 8.33 msec.

Disk Scheduling and Management

Disk scheduling is done by operating systems to schedule I/O requests arriving for disk. Disk scheduling is also known as I/O scheduling.
Disk scheduling is important because:

Multiple I/O requests may arrive by different processes and only one I/O request can be served at a time by disk controller. Thus other I/O requests need to wait in waiting queue and need to be scheduled. Two or more request may be far from each other so can result in greater disk arm movement.
Hard drives are one of the slowest parts of computer system and thus need to be accessed in an efficient manner.

There are many Disk Scheduling Algorithms but before discussing them let's have a quick look at some of the important terms:

Seek Time: Seek time is the time taken to locate the disk arm to a specified track where the data is to be read or write. So the disk scheduling algorithm that gives minimum average seek time is better.

Rotational Latency: Rotational Latency is the time taken by the desired sector of disk to rotate into a position so that it can access the read/write heads. So the disk scheduling algorithm that gives minimum rotational latency is better.

Transfer Time: Transfer time is the time to transfer the data. It depends on the rotating speed of the disk and number of bytes to be transferred.

Disk Access Time: Disk Access Time is:

Disk Access Time = Seek Time + Rotational Latency + Transfer Time

Disk Response Time: Response Time is the average of time spent by a request waiting to perform its I/O operation. *Average Response time* is the response time of the all requests. *Variance Response Time* is measure of how individual request are serviced with respect to average response time. So the disk scheduling algorithm that gives minimum variance response time is better.

Disk Scheduling Algorithms

1. **FCFS:** FCFS is the simplest of all the Disk Scheduling Algorithms. In FCFS, the requests are addressed in the order they arrive in the disk queue.

Advantages:

Every request gets a fair chance
No indefinite postponement

Disadvantages:

Does not try to optimize seek time
May not provide the best possible service

2. **SSTF:** In SSTF (Shortest Seek Time First), requests having shortest seek time are executed first. So, the seek time of every request is calculated in advance in queue and then they are scheduled according to their calculated seek time. As a result, the request near the disk arm will get executed first. SSTF is certainly an improvement over FCFS as it decreases the average response time and increases the throughput of system.

Advantages:

Average Response Time
Decreases Throughput increases

Disadvantages:

Overhead to calculate seek time in advance
Can cause Starvation for a request if it has higher seek time as compared to incoming requests High variance of response time as SSTF favours only some requests

3. **SCAN:** In SCAN algorithm the disk arm moves into a particular direction and services the requests coming in its path and after reaching the end of disk, it reverses its direction and again services the request arriving in its path. So, this algorithm works like an elevator and hence also known as **elevator algorithm.** As a result, the requests at the midrange are serviced more and those arriving behind the disk arm will have to wait.

Advantages:

High throughput

Low variance of response time Average response time

Disadvantages:

Long waiting time for requests for locations just visited by disk arm

4. **CSCAN**: In SCAN algorithm, the disk arm again scans the path that has been scanned, after reversing its direction. So, it may be possible that too many requests are waiting at the other end or there may be zero or few requests pending at the scanned area.

These situations are avoided in *CSAN* algorithm in which the disk arm instead of reversing its direction goes to the other end of the disk and starts servicing the requests from there. So, the disk arm moves in a circular fashion and this algorithm is also similar to SCAN algorithm and hence it is known as C-SCAN (Circular SCAN).

Advantages:

Provides more uniform wait time compared to SCAN

5. **LOOK:** It is similar to the SCAN disk scheduling algorithm except the difference that the disk arm in spite of going to the end of the disk goes only to the last request to be serviced in front of the head and then reverses its direction from there only. Thus it prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

6. **CLOOK:** As LOOK is similar to SCAN algorithm, in similar way, CLOOK is similar to CSCAN disk scheduling algorithm. In CLOOK, the disk arm inspite of going to the end goes only to the last request to be serviced in front of the head and then from there goes to the other end's last request. Thus, it also prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

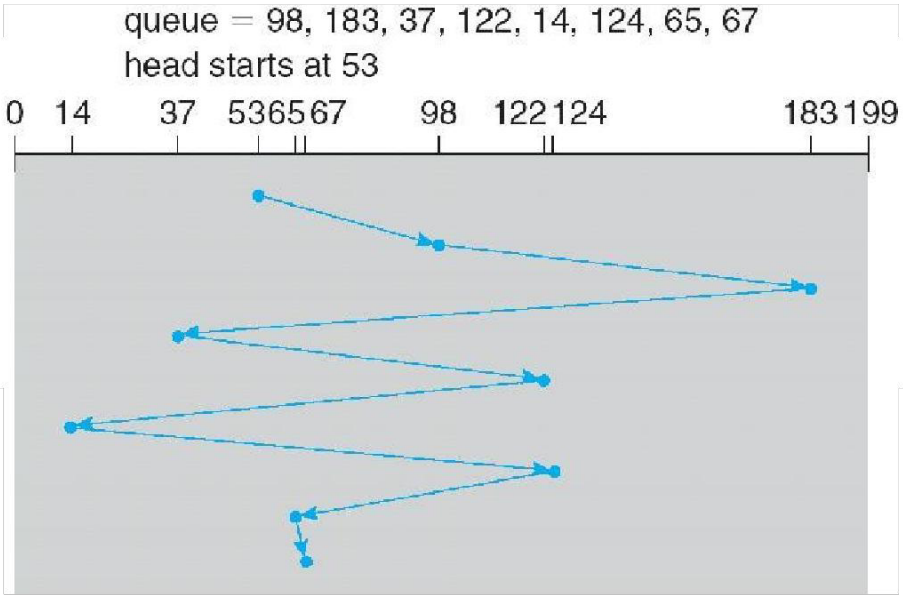
1. **FCFS Scheduling:**

The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service. Consider, for

example, a disk queue with requests for I/O to blocks on cylinders

I/O to blocks on cylinders

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

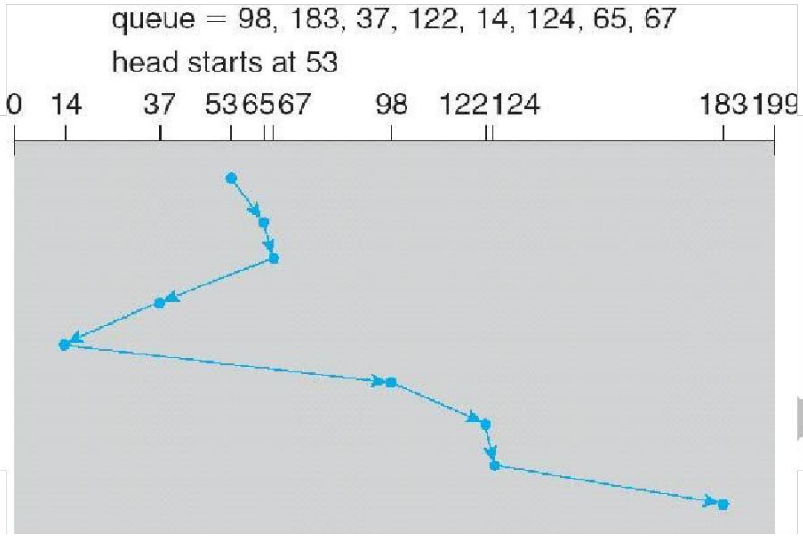


FCFS disk scheduling.

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

2. SSTF(shortest-seek-time-first)Scheduling

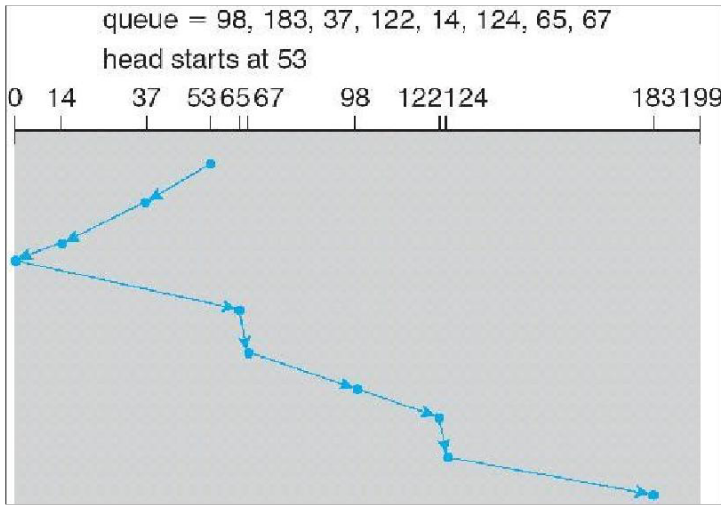
Service all the requests close to the current head position, before moving the head far away to service other requests. That is selects the request with the minimum seek time from the current head position.



Total head movement = 236 cylinders

3. SCAN Scheduling

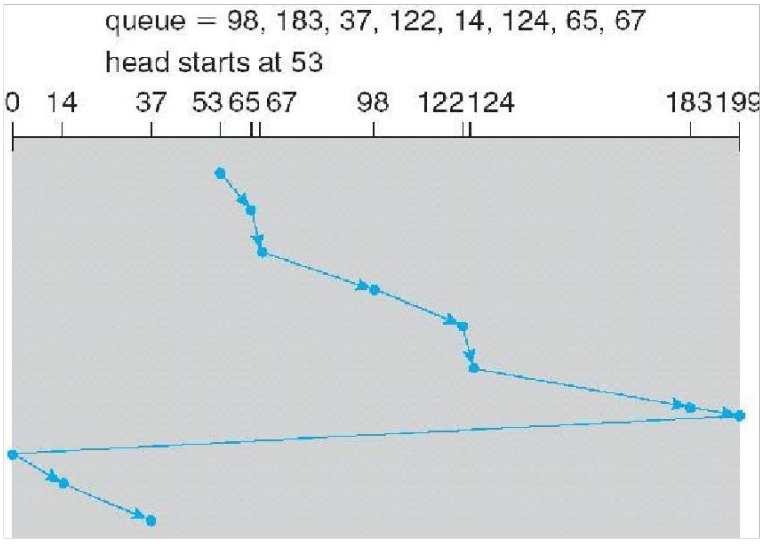
The disk head starts at one end of the disk, and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues.



SCAN disk scheduling.

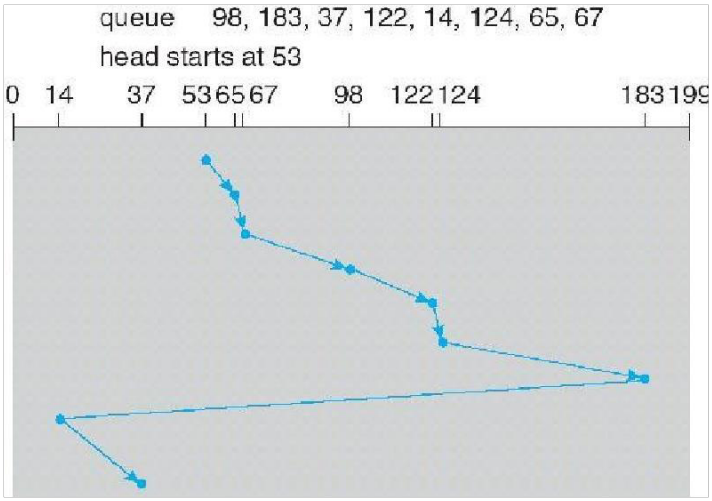
4. C-SCAN Scheduling

Variant of SCAN designed to provide a more uniform wait time. It 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.



5. LOOK Scheduling

Both SCAN and C-SCAN move the disk arm across the full width of the disk. In this, the arm goes only as far as the final request in each direction. Then, it reverses direction immediately, without going all the way to the end of the disk. LOOK scheduling improves upon SCAN by looking ahead at the queue of pending requests, and not moving the heads any farther towards the end of the disk than is necessary. The following diagram illustrates the circular form of LOOK:



C-LOOK disk scheduling.

Disk Management

1. Disk Formatting:

Before a disk can store data, the sector is divided into various partitions. This process is called low- level formatting or physical formatting. It fills the disk with a special data structure for each sector. The data structure for a sector consists of

✓ Header,

✓ Data area (usually 512 bytes in size),and

✓ Trailer.

Error-Correcting Code (ECC).

This formatting enables the manufacturer to

1. Test the disk and

2. To initialize the mapping from logical block numbers

To use a disk to hold files, the operating system still needs to record its own data structures on the disk. It does so in two steps.

(a) The first step is **Partition** the disk into one or more groups of cylinders. Among the partitions, one partition can hold a copy of the OS‘s executable code, while another holds user files.

(b) The second step is **logical formatting** .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.

2. Boot Block:

For a computer to start running-for instance, when it is powered up or rebooted-it needs to have an initial program to run. This initial program is called bootstrap program & it should be simple. It initializes all aspects of the system, from CPU registers to device controllers and the contents of main memory, and then starts the operating system.

To do its job, the bootstrap program

1. Finds the operating system kernel on disk,

2. Loads that kernel into memory, and

3. Jumps to an initial address to begin the operating system execution.

The bootstrap is stored in read-only memory **(ROM).**

Advantages:

1. ROM needs no initialization.
2. It is at a fixed location that the processor can start executing when powered up or reset.
3. It cannot be infected by a computer virus. Since, ROM is read only.

The full bootstrap program is stored in a partition called 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. **Bootstrap loader**: load the entire operating system from a non-fixed location on disk, and to start the operating system running.

3. Bad Blocks:

The disk with defected sector is called as bad block.

Depending on the disk and controller in use, these blocks are handled in a variety of ways;

Method 1: "Handled manually''

If blocks go bad during normal operation, a **special program** must be run manually to search for the bad blocks and to lock them away as before. Data that resided on the bad blocks usually are lost.

Method 2: "sector sparing or forwarding"

The controller maintains a list of bad blocks on the disk. Then the controller can be told to replace each bad sector logically with one of the spare sectors. This scheme is known as sector sparing or forwarding.

A typical bad-sector transaction might be as follows:

1. The operating system tries to read logical block87.
2. The controller calculates the ECC and finds that the sector is bad.
3. It reports this finding to the operating system.
4. The next time that the system is rebooted, a special command is run to tell the controller to replace the bad sector with a spare.
5. After that, whenever the system requests logical block 87, the request is translated into the replacement sector's address by the controller.

Method 3: "sector slipping"

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

Swap Space Management:

Modern systems typically swap out pages as needed, rather than swapping out entire processes. Hence the swapping system is part of the virtual memory management system.

Managing swap space is obviously an important task for modern OS.

Swap-Space Use

The amount of swap space needed by an OS varies greatly according to how it is used. Some systems require an amount equal to physical RAM; some want a multiple of that; some want an amount equal to the amount by which virtual memory exceeds physical RAM, and some systems use little or none at all!

Some systems support multiple swap spaces on separate disks in order to speed up the virtual memory system.

The interchange of data between virtual memory and real memory is called as swapping and **space** on disk as "**swap space**".

Swap-Space Location

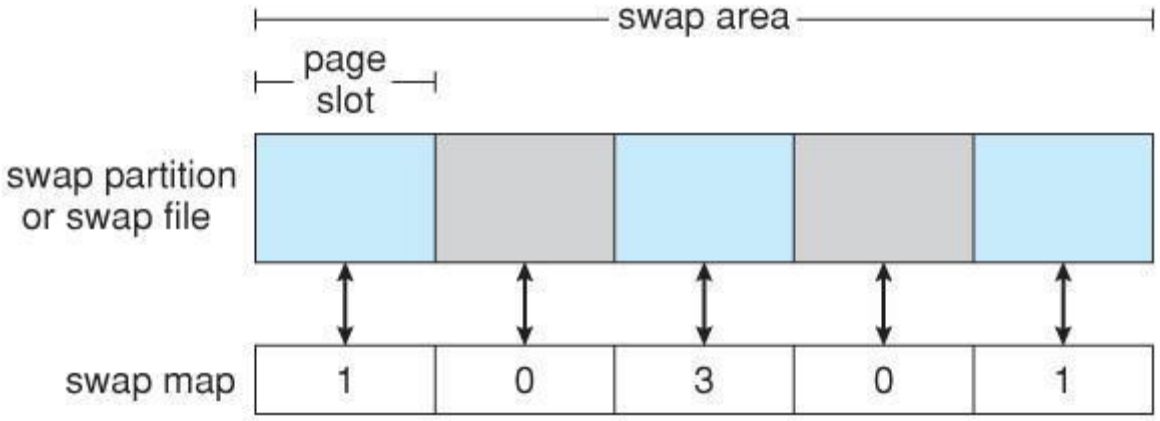
Swap space can be physically located in one of two locations:

As a large file which is part of the regular file system. This is easy to implement, but inefficient. Not only must the swap space be accessed through the directory system, the file is also subject to fragmentation issues. Caching the block location helps in finding the physical blocks, but that is not a complete fix.

As a raw partition, possibly on a separate or little-used disk. This allows the OS more control over swap space management, which is usually faster and more efficient. Fragmentation of swap space is generally not a big issue, as the space is re-initialized every time the system is rebooted. The downside of keeping swap space on a raw partition is that it can only be grown by repartitioning the hard drive.

Swap-Space Management: An Example

Historically OS swapped out entire processes as needed. Modern systems swap out only individual pages, and only as needed. In the mapping system shown below for Linux systems, a map of swap space is kept in memory, where each entry corresponds to a 4K block in the swap space. Zeros indicate free slots and non-zeros refer to how many processes have a mapping to that particular block (>1 for shared pages only.)



The data structures for swapping on Linux systems.

File-System Interface – File concept

File: A file is a named collection of related information that is recorded on secondary storage such as magnetic disks, magnetic tapes and optical disks. In general, a file is a sequence of bits, bytes, lines or records whose meaning is defined by the files creator and user.

File Attributes

Different OS keep track of different file attributes, including:

Name - Some systems give special significance to names, and particularly extensions (.exe, .txt, etc.), and some do not. Some extensions may be of significance to the OS (.exe), and others only to certain applications (.jpg)
Identifier
Type - Text, executable, other binary, etc.
Location - on the hard drive.
Size
Protection
Time & Date
User ID

File Operations

The file **ADT** supports many common operations:

- **Creating a file**
- **Writing a file**
- **Reading a file**

- **Repositioning within a file**
- **Deleting a file**
- **Truncating a file.**

Some systems provide support for *file locking.*

- A ***shared lock*** is for reading only.
- A ***exclusive lock*** is for writing as well as reading.
- An ***advisory lock*** is informational only, and not enforced. (A "Keep Out" sign, which may be ignored.)
- A ***mandatory lock*** is enforced. (A truly locked door.)
- UNIX used advisory locks, and Windows uses mandatory locks.

File Types

Windows (and some other systems) use special file extensions to indicate the type of each file:

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

File Structure

Some files contain an internal structure, which may or may not be known to the OS.

For the OS to support particular file formats increases the size and complexity of the OS.

UNIX treats all files as sequences of bytes, with no further consideration of the internal structure. (With the exception of executable binary programs, which it must know how to load and find the first executable statement, etc.)

Macintosh files have **two *forks* - a *resource fork*, and a *data fork***. The resource fork contains information relating to the UI, such as icons and button images, and can be modified independently of the data fork, which contains the code or data as appropriate.

A File Structure should be according to a required format that the operating system can understand.

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

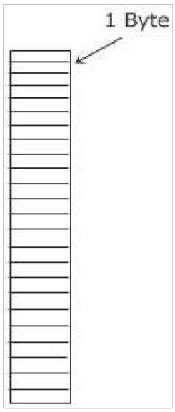
text file is a sequence of characters organized into lines. A

source file is a sequence of procedures and functions.

An **object file** is a sequence of bytes organized into blocks that are understandable by the machine.

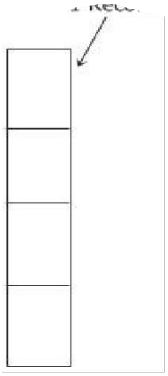
Files can be structured in several ways in which three common structures are given in this tutorial with their short description one by one.

File Structure 1



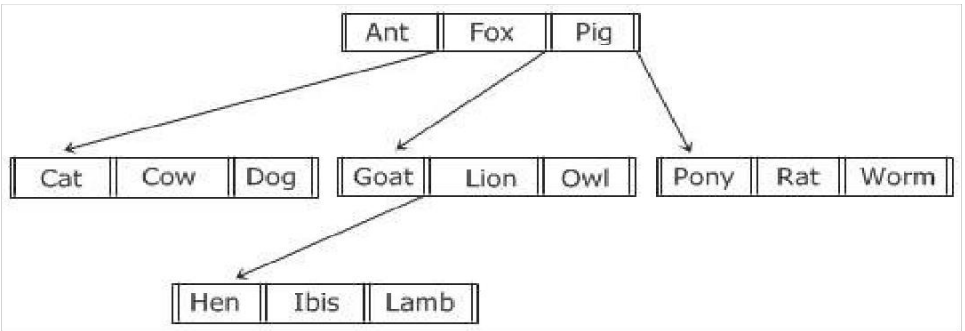
Here, as you can see from the above figure, the file is an unstructured sequence of bytes. Therefore, the OS doesn't care about what is in the file, as all it sees are bytes.

File Structure 2



Now, as you can see from the above figure that shows the second structure of a file, where a file is a sequence of fixed-length records where each with some internal structure. Central to the idea about a file being a sequence of records is the idea that read operation returns a record and write operation just appends a record.

File Structure 3



Now in the last structure of a file that you can see in the above figure, a file basically consists of a tree of records, not necessarily all the same length, each containing a key field in a fixed position in the record. The tree is stored on the field, just to allow the rapid searching for a specific key.

Access Methods

File access mechanism refers to the manner in which the records of a file may be accessed. There are several ways to access files −

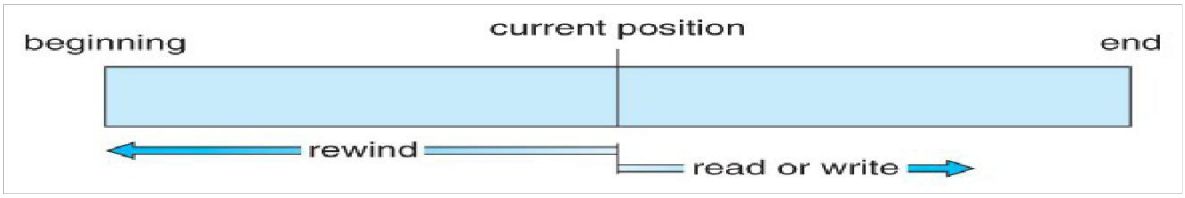
Sequential access

Direct/Random access

Indexed sequential access

Sequential access

A sequential access is that in which the records are accessed in some sequence, i.e., the information in the file is processed in order, one record after the other. This access method is the most primitive one. Example: Compilers usually access files in this fashion.



Sequential-access file.

Direct/Random access

Random access file organization provides, accessing the records directly.

Each record has its own address on the file with by the help of which it can be directly accessed for reading or writing.

The records need not be in any sequence within the file and they need not be in adjacent locations on the storage medium.

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

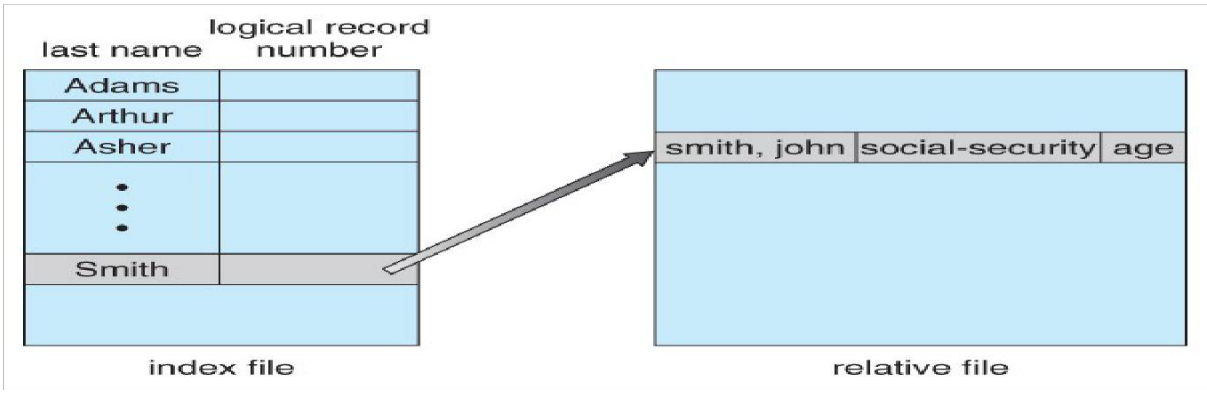
Simulation of sequential access on a direct-access file.

Indexed sequential access

This mechanism is built up on base of sequential access.

An index is created for each file which contains pointers to various blocks.

Index is searched sequentially and its pointer is used to access the file directly.



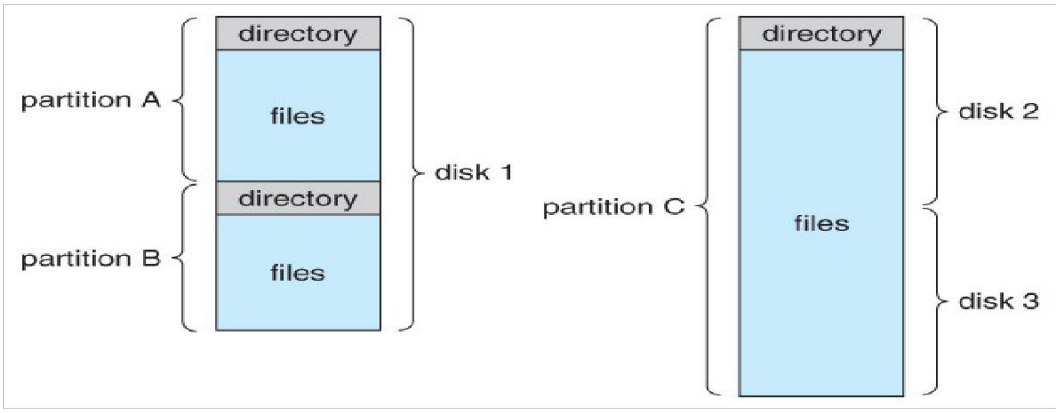
Example of index and relative files.

Directory Structure:

A directory is a container that is used to contain folders and file. It organizes files and folders into hierarchical manner.

Storage Structure

A disk can be used in its entirety for a file system. Alternatively a physical disk can be broken up into multiple ***partitions, slices, or mini-disks***, each of which becomes a virtual disk and can have its own file system(or be used for raw storage, swap space, etc.)Or, multiple physical disks can be combined into one *volume*, i.e. a larger virtual disk, with its own file system spanning the physical disks.



A typical file-system organization.

Directory Overview

Directory operations to be supported include:

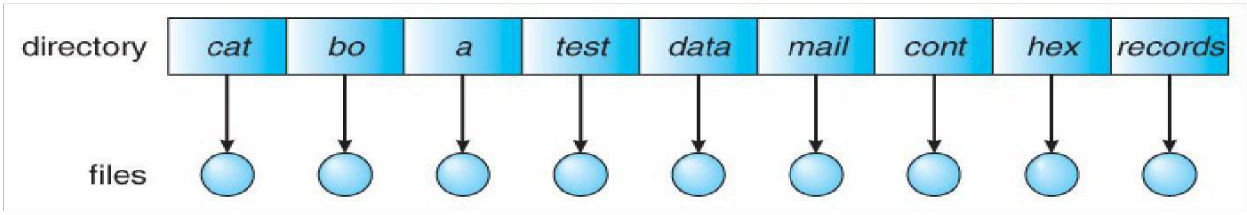
- Search for a file
- Create a file - add to the directory
- Delete a file - erase from the directory

- List a directory - possibly ordered in different ways.
- Rename a file - may change sorting order
- Traverse the file system.

Directory organization

Single-Level Directory

Simple to implement, but each file must have a unique name. The simplest method is to have one big list of all the files on the disk. The entire system will contain only one **directory** which is supposed to mention all the files present in the file system. The **directory** contains one entry per each file present on the file system.



Single-level directory.

Two-Level Directory

Each user gets their own directory space.

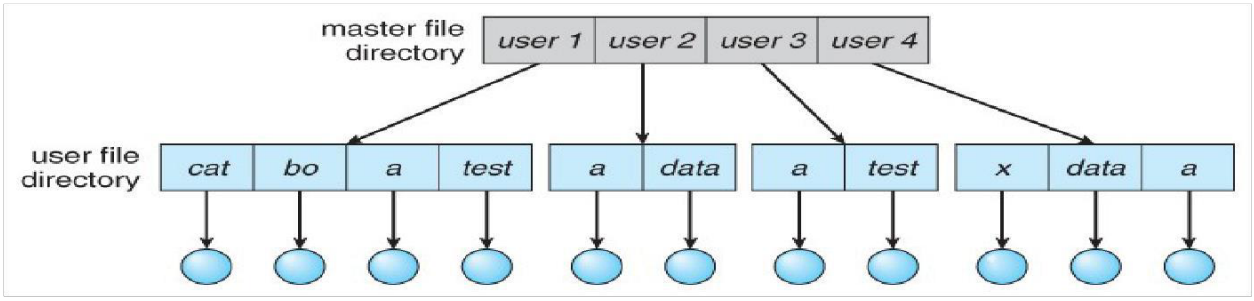
File names only need to be unique within a given user's directory.

A master file directory is used to keep track of each users directory, and must be maintained when users are added to or removed from the system.

A separate directory is generally needed for system (executable) files.

Systems may or may not allow users to access other directories besides their own

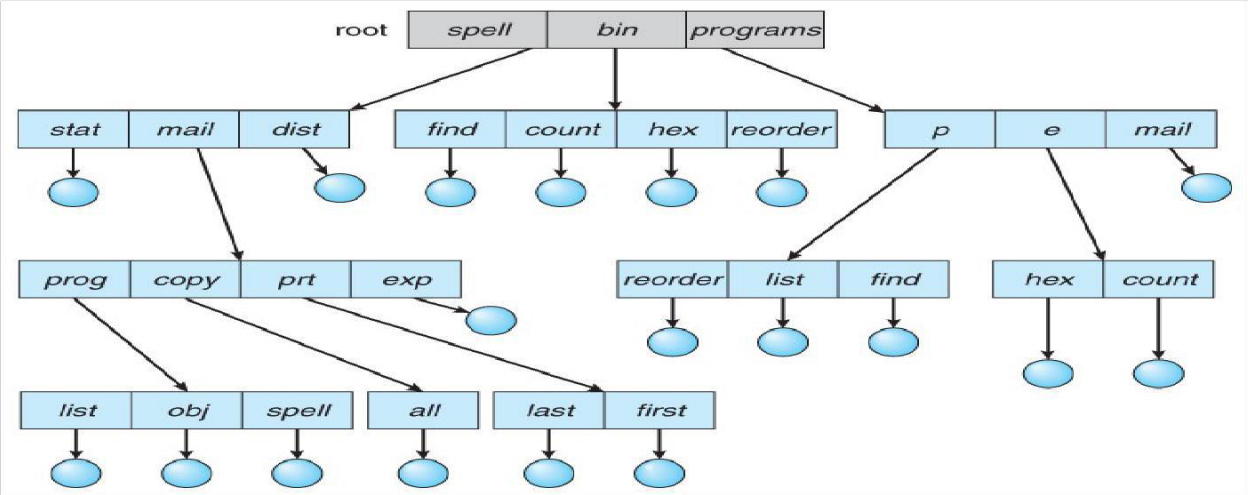
- If access to other directories is allowed, then provision must be made to specify the directory being accessed.
- If access is denied, then special consideration must be made for users to run programs located in system directories. A *search path* is the list of directories in which to search for executable programs, and can be set uniquely for each user.



Two-level directory structure.

Tree-Structured Directories

An obvious extension to the two-tiered directory structure, and the one with which we are all most familiar. Each user / process has the concept of a ***current directory*** from which all (relative) searches take place. Files may be accessed using either absolute pathnames (relative to the root of the tree) or relative pathnames (relative to the current directory.)Directories are stored the same as any other file in the system, except there is a bit that identifies them as directories, and they have some special structure that the OS understands. One question for consideration is whether or not to allow the removal of directories that are not empty - Windows requires that directories be emptied first, and UNIX provides an option for deleting entire sub-trees.



Tree-structured directory structure.

Acyclic-Graph Directories

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

UNIX provides two types of *links* for implementing the acyclic-graph structure. (See "man ln" for more details.)

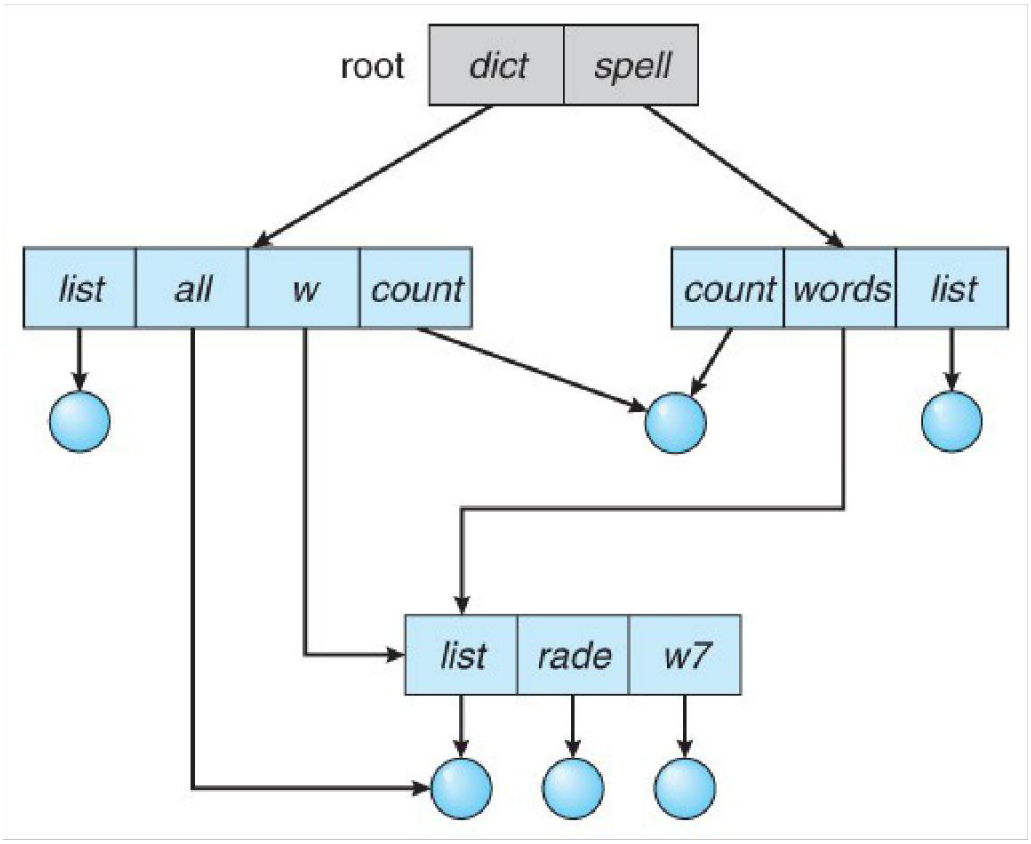
- **A *hard link*** (usually just called a link) involves multiple directory entries that both refer to the same file. Hard links are only valid for ordinary files in the same file system.
- **A *symbolic link*** that involves a special file, containing information about where to find the linked file. Symbolic links may be used to link directories and/or files in other file systems, as well as ordinary files in the current file system.

Windows only supports symbolic links, termed *shortcuts.*

Hard links require a *reference count*, or *link count* for each file, keeping track of how many directory entries are currently referring to this file. Whenever one of the references is removed the link count is reduced, and when it reaches zero, the disk space can be reclaimed.

For symbolic links there is some question as to what to do with the symbolic links when the original file is moved or deleted:

- One option is to find all the symbolic links and adjust them also.
- Another is to leave the symbolic links dangling, and discover that they are no longer valid the next time they are used.
- What if the original file is removed, and replaced with another file having the same name before the symbolic link is next used?



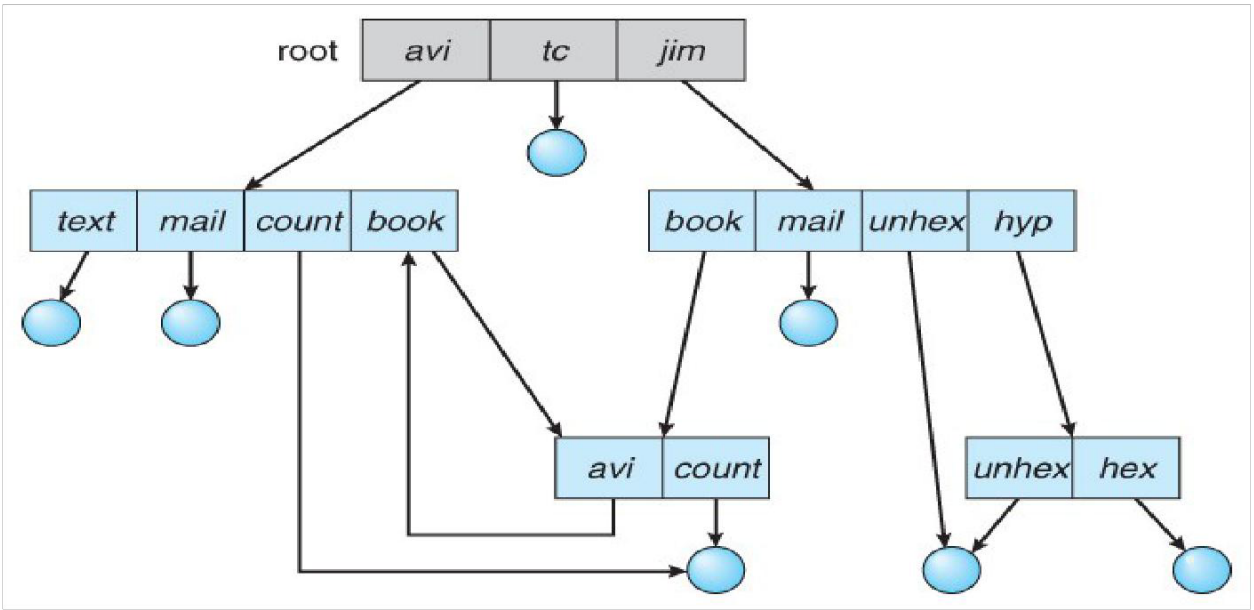
Acyclic-graph directory structure.

General Graph Directory

If cycles are allowed in the graphs, then several problems can arise:

- Search algorithms can go into infinite loops. One solution is to not follow links in search algorithms. (Or not to follow symbolic links, and to only allow symbolic links to refer to directories.)

- Sub-trees can become disconnected from the rest of the tree and still not have their reference counts reduced to zero. Periodic garbage collection is required to detect and resolve this problem.



General graph directory.

File system mounting:

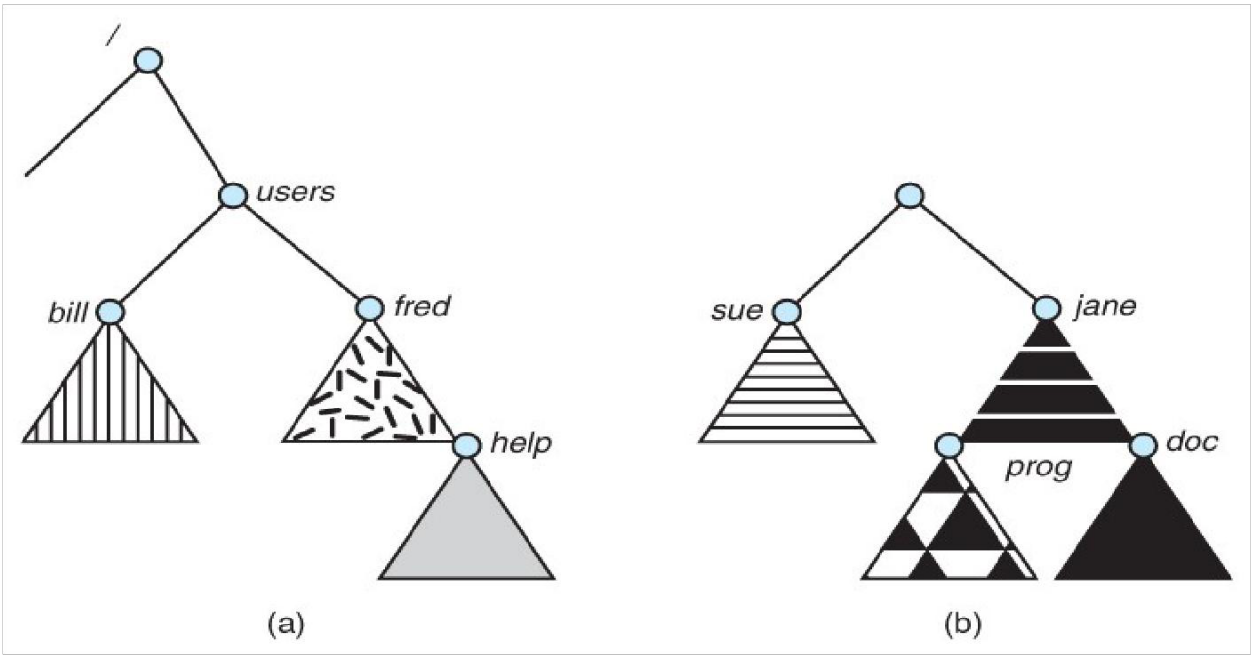
- The basic idea behind mounting file systems is to combine multiple file systems into one large tree structure.
- The mount command is given a file system to mount and a ***mount point*** (directory) on which to attach it.
- Once a file system is mounted onto a mount point, any further references to that directory actually refer to the root of the mounted file system.

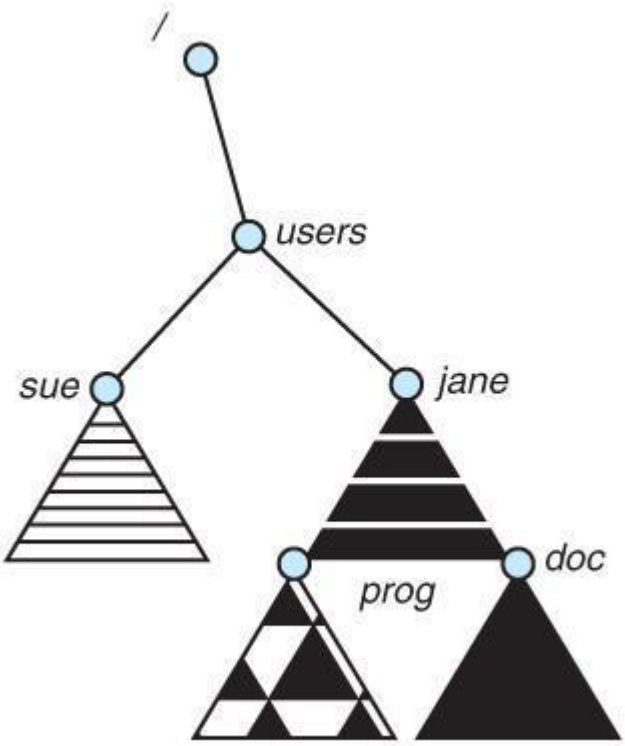
Any files (or sub-directories) that had been stored in the mount point directory prior to mounting the new file system are now hidden by the mounted file system, and are no longer available. For this reason some systems only allow mounting onto empty directories.

File systems can only be mounted by root, unless root has previously configured certain filesystems to be mountable onto certain pre-determined mount points. (E.g. root may allow users to mount floppy filesystems to /mnt or something like it.) Anyone can run the mount command to see what file systems is currently mounted.

Filesystems may be mounted read-only, or have other restrictions imposed.

(a) Existing System **(b) Unmounted Volume**



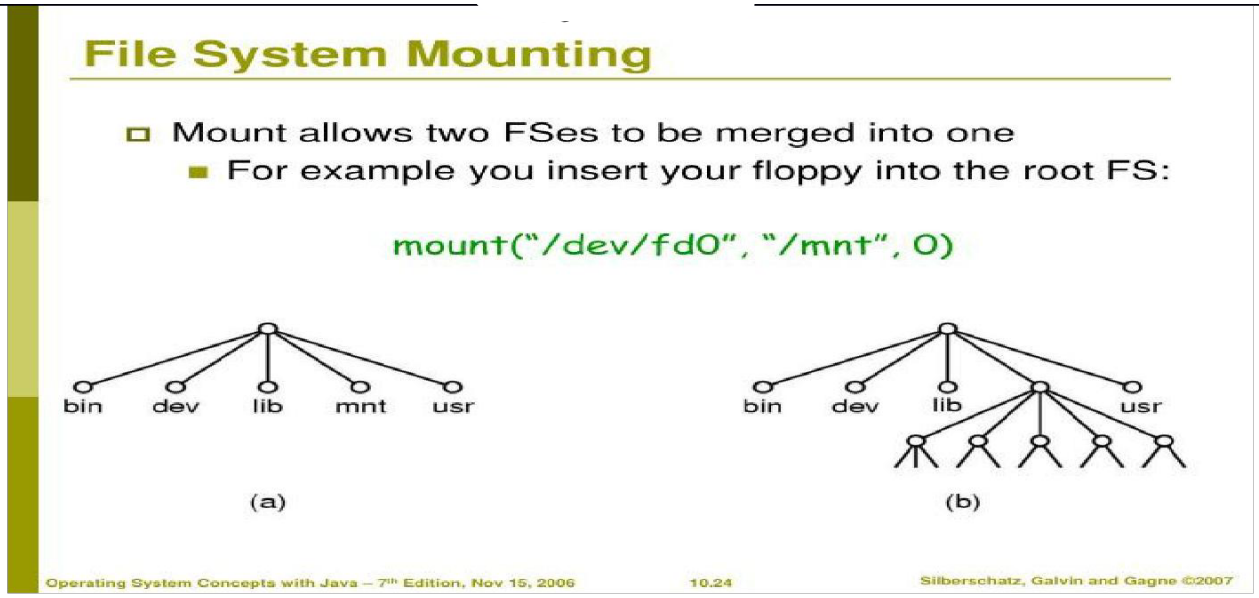


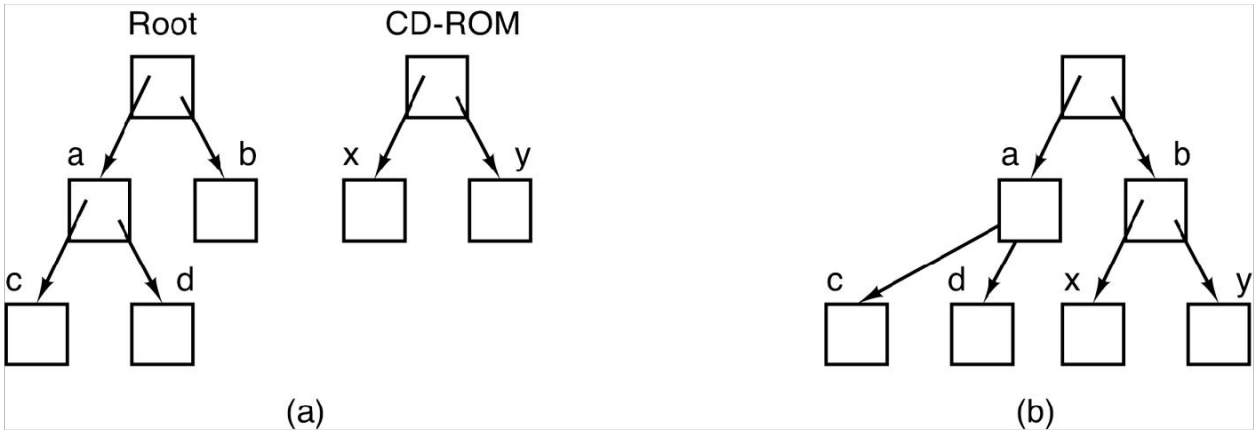
Mount point.

The traditional Windows OS runs an extended two-tier directory structure, where the first tier of the structure separates volumes by drive letters, and a tree structure is implemented below that level.

Macintosh runs a similar system, where each new volume that is found is automatically mounted and added to the desktop when it is found.

More recent Windows systems allow filesystems to be mounted to any directory in the filesystem, much like UNIX.





File Sharing and Protection:

File Sharing

Multiple Users

On a multi-user system, more information needs to be stored for each file:

- The owner (user) who owns the file, and who can control its access.
- The group of other user IDs that may have some special access to the file.
- What access rights are afforded to the owner (User), the Group, and to the rest of the world (the universe, a.k.a. Others.)

- Some systems have more complicated access control, allowing or denying specific accesses to specifically named users or groups.

Remote File Systems

The advent of the Internet introduces issues for accessing files stored on remote computers

- The original method was ftp, allowing individual files to be transported across systems as needed. Ftp can be either account or password controlled, or *anonymous*, not requiring any user name or password.
- Various forms of ***distributed file systems*** allow remote file systems to be mounted onto a local directory structure, and accessed using normal file access commands. (The actual files are still transported across the network as needed, possibly using ftp as the underlying transport mechanism.)
- The WWW has made it easy once again to access files on remote systems without mounting their filesystems, generally using (anonymous) ftp as the underlying file transport mechanism.

The Client-Server Model

When one computer system remotely mounts a file system that is physically located on another system, the system which physically owns the files acts as a *server*, and the system which mounts them is the *client.*

User IDs and group IDs must be consistent across both systems for the system to work properly. (I.e. this is most applicable across multiple computers managed by the same organization, shared by a common group of users.)

The same computer can be both a client and a server. (E.g. cross-linked file systems.

) There are a number of security concerns involved in this model:

- Servers commonly restrict mount permission to certain trusted systems only. Spoofing (a computer pretending to be a different computer) is a potential security risk.
- Servers may restrict remote access to read-only.
- Servers restrict which filesystems may be remotely mounted. Generally the information within those subsystems is limited, relatively public, and protected by frequent backups.

The NFS (Network File System) is a classic example of such a system.

Distributed Information Systems

The ***Domain Name System****, DNS,* provides for a unique naming system across all of the Internet.

Domain names are maintained by the ***Network Information System, NIS***, which unfortunately has several security issues. NIS+ is a more secure version, but has not yet gained the same widespread acceptance as NIS.

Microsoft's ***Common Internet File System, CIFS,*** establishes a *network login* for each user on a networked system with shared file access. Older Windows systems used *domains*, and newer systems (XP, 2000), use *active directories.* User names must match across the network for this system to be valid.

A newer approach is the ***Lightweight Directory-Access Protocol, LDAP,*** which provides a *secure single sign-on* for all users to access all resources on a network. This is a secure system which is gaining in popularity, and which has the maintenance advantage of combining authorization information in one central location.

Failure Modes

When a local disk file is unavailable, the result is generally known immediately, and is generally non-recoverable. The only reasonable response is for the response to fail.

However when a remote file is unavailable, there are many possible reasons, and whether or not it is unrecoverable is not readily apparent. Hence most remote access systems allow for blocking or delayed response, in the hopes that the remote system (or the network) will come back up eventually.

Consistency Semantics

Consistency Semantics deals with the consistency between the views of shared files on a networked system. When one user changes the file, when do other users see the changes?

At first glance this appears to have all of the synchronization issues discussed in Chapter 6. Unfortunately the long delays involved in network operations prohibit the use of atomic operations as discussed in that chapter.

UNIX Semantics

The UNIX file system uses the following semantics:

- Writes to an open file are immediately visible to any other user who has the file open.
- One implementation uses a shared location pointer, which is adjusted for all sharing users.

The file is associated with a single exclusive physical resource, which may delay some accesses.

Session Semantics

The Andrew File System, AFS uses the following semantics:

- Writes to an open file are not immediately visible to other users.

- When a file is closed, any changes made become available only to users who open the file at a later time.

According to these semantics, a file can be associated with multiple (possibly different) views. Almost no constraints are imposed on scheduling accesses. No user is delayed in reading or writing their personal copy of the file.

AFS file systems may be accessible by systems around the world. Access control is maintained through (somewhat) complicated access control lists, which may grant access to the entire world (literally) or to specifically named users accessing the files from specifically named remote environments.

Immutable-Shared-Files Semantics

Under this system, when a file is declared as *shared* by its creator, it becomes immutable and the name cannot be re-used for any other resource. Hence it becomes read-only, and shared access is simple.

Protection

The processes in an operating system must be **protected** from one another's activities. To provide such **protection**, we can use various mechanisms to ensure that only processes that have gained proper authorization from the operating system can operate on the **files**, memory segments, CPU, and other resources of a system.

Goals of Protection

Obviously 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.

Note that protection systems only provide the *mechanisms* for enforcing policies and ensuring reliable systems. It is up to administrators and users to implement those mechanisms effectively.

Principles of Protection

The ***principle of least privilege*** dictates that programs, users, and systems be given just enough privileges to perform their tasks.

This ensures that failures do the least amount of harm and allow the least of harm to be done.

For example, if a program needs special privileges to perform a task, it is better to make it a SGID program with group ownership of "network" or "backup" or some other pseudo group, rather than SUID with root ownership. This limits the amount of damage that can occur if something goes wrong.

Typically each user is given their own account, and has only enough privilege to modify their own files.

The root account should not be used for normal day to day activities - The System Administrator should also have an ordinary account, and reserve use of the root account for only those tasks which need the root privileges.

Domain of Protection

A computer can be viewed as a collection of *processes* and *objects* (both HW & SW).

The ***need to know principle*** states that a process should only have access to those objects it needs to accomplish its task, and furthermore only in the modes for which it needs access and only during the time frame when it needs access.

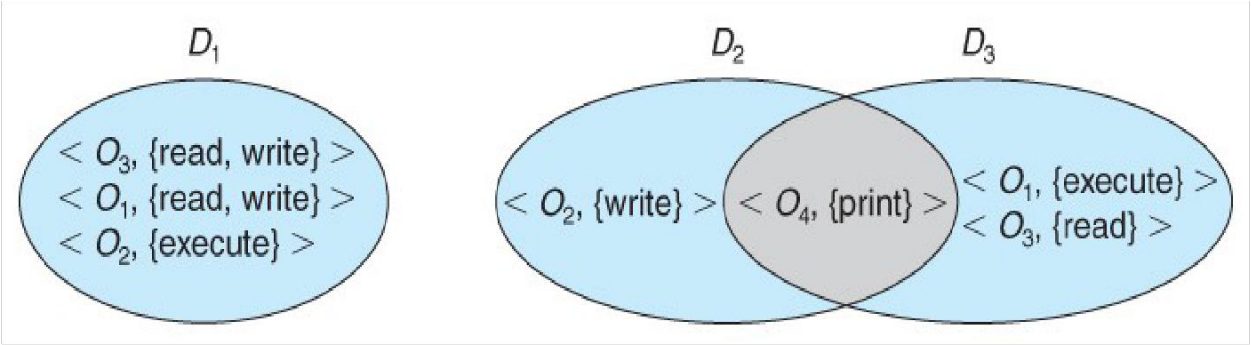
The modes available for a particular object may depend upon its type.

Domain Structure

A ***protection domain*** specifies the resources that a process may access.

Each domain defines a set of objects and the types of operations that may be invoked on each object. An ***access right*** is the ability to execute an operation on an object.

A domain is defined as a set of < object, { access right set } > pairs, as shown below. Note that some domains may be disjoint while others overlap.



System with three protection domains.

The association between a process and a domain may be ***static* or *dynamic.***

If the association is static, then the need-to-know principle requires a way of changing the contents of the domain dynamically. If the association is dynamic, then there needs to be a mechanism for ***domain switching.*** Domains may be realized in different fashions - as users, or as processes, or as procedures. E.g. if each user corresponds to a domain, then that domain defines the access of that user, and changing domains involves changing user ID. The model of protection that we have been discussing can be viewed

as an ***access matrix,*** in which columns represent different system resources and rows represent different protection domains. Entries within the matrix indicate what access that domain has to that resource.

object / domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Access matrix.

Domain switching can be easily supported under this model, simply by providing "switch" access to other domains:

object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Access matrix of above Figure with domains as objects.

Types of Access

The following low-level operations are often controlled
Read - View the contents of the file
Write - Change the contents of the file.
Execute - Load the file onto the CPU and follow the instructions contained therein.
Append - Add to the end of an existing file.
Delete - Remove a file from the system.
List -View the name and other attributes of files on the system.
Higher-level operations, such as copy, can generally be performed through combinations of the above.

Access Control

One approach is to have complicated ***Access Control Lists, ACL,*** which specify exactly what access is allowed or denied for specific users or groups.

The AFS uses this system for distributed access.

Control is very finely adjustable, but may be complicated, particularly when the specific users involved are unknown. (AFS allows some wild cards, so for example all users on a certain remote system may be trusted, or a given username may be trusted when accessing from any remote system.)

UNIX uses a set of 9 access control bits, in three groups of three. These correspond to R, W, and X permissions for each of the Owner, Group, and Others. (See "man chmod" for full details.) The RWX bits control the following privileges for ordinary files and directories:

bit	Files	Directories
R	Read (view) file contents.	Read directory contents. Required to get a listing of the directory.
W	Write (change) file contents.	Change directory contents. Required to create or delete files.
X	Execute file contents as a program.	Access detailed directory information. Required to get a long listing, or to access any specific file in the directory. Note that if a user has X but not R permissions on a directory, they can still access specific files, but only if they already know the name of the file they are trying to access.

File System Structure

Hard disks have two important properties that make them suitable for secondary storage of files in file systems: (1) Blocks of data can be rewritten in place, and (2) they are direct access, allowing any block of data to be accessed with only (relatively) minor movements of the disk heads and rotational latency.

Disks are usually accessed in physical blocks, rather than a byte at a time. Block sizes may range from **512 bytes to 4K or larger.**

File systems organize storage on disk drives, and can be viewed as a layered design:

- At the lowest layer are the physical devices, consisting of the magnetic media, motors & controls, and the electronics connected to them and controlling them. Modern disk put more and more of the electronic controls directly on the disk drive itself, leaving relatively little work for the disk controller card to perform.

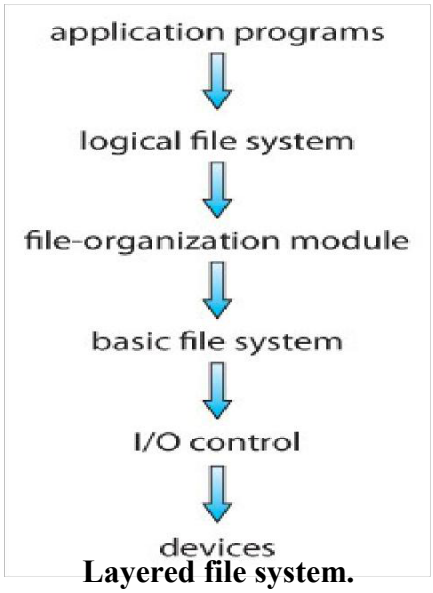
- ***I/O Control*** consists of ***device drivers***, special software programs (often written in assembly) which communicate with the devices by reading and writing special codes directly to and from memory addresses corresponding to the controller card's registers. Each controller card (device) on a system has a different set of addresses (registers, a.k.a. ***ports***) that it listens to, and a unique set of command codes and results codes that it understands.

- The ***basic file system*** level works directly with the device drivers in terms of retrieving and storing raw blocks of data, without any consideration for what is in each block. Depending on the system, blocks may be referred to with a single block number, (e.g. block # 234234), or with head-sector-cylinder combinations.

- The ***file organization module*** knows about files and their logical blocks, and how they map to physical blocks on the disk. In addition to translating from logical to physical blocks, the file organization module also maintains the list of free blocks, and allocates free blocks to files as needed.

- The ***logical file system*** deals with all of the meta data associated with a file (UID, GID, mode, dates, etc), i.e. everything about the file except the data itself. This level manages the directory structure and the mapping of file names to ***file control blocks, FCBs***, which contain all of the meta data as well as block number information for finding the data on the disk.

The layered approach to file systems means that much of the code can be used uniformly for a wide variety of different file systems, and only certain layers need to be file system specific. Common file systems in use include the UNIX file system, UFS, the Berkeley Fast File System, FFS, Windows systems FAT, FAT32, NTFS, CD-ROM systems ISO 9660, and for Linux the extended file systems ext2 and ext3 (among 40 others supported.)



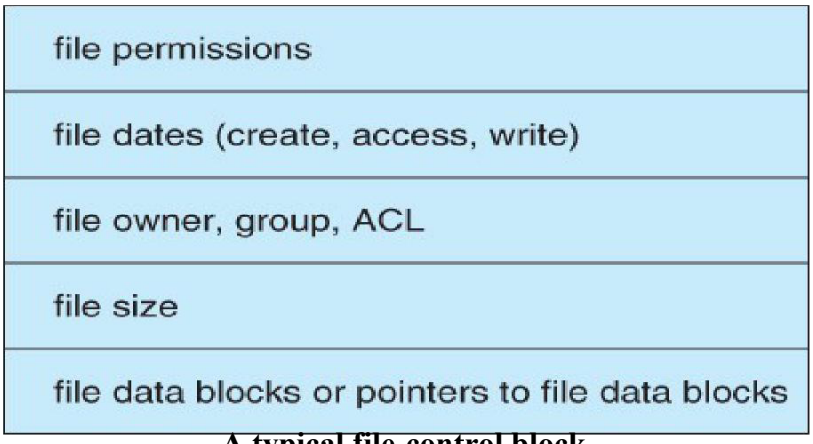
Layered file system.

File System Implementation:

Overview

File systems store several important data structures on the disk:

- A ***boot-control bloc*k**, (per volume) a.k.a. the *boot block* in UNIX or the *partition boot sector* in Windows contains information about how to boot the system off of this disk. This will generally be the first sector of the volume if there is a bootable system loaded on that volume, or the block will be left vacant otherwise.
- A ***volume control block****,* (per volume) a.k.a. the *master file table* in UNIX or the *superblock* in Windows, which contains information such as the partition table, number of blocks on each file system, and pointers to free blocks and free FCB blocks.
- A directory structure (per file system), containing file names and pointers to corresponding FCBs. UNIX uses inode numbers, and NTFS uses a *master file table.*
- The ***File Control Block****, FCB,* (per file) containing details about ownership, size, permissions, dates, etc. UNIX stores this information in inodes, and NTFS in the master file table as a relational database structure.



A typical file-control block.

There are also several key data structures stored in memory:

- An in-memory mount table.
- An in-memory directory cache of recently accessed directory information.
- *A* ***system-wide open file table***, containing a copy of the FCB for every currently open file in the system, as well as some other related information.

- *A **per-process open file table**,* containing a pointer to the system open file table as well as some other information. (For example the current file position pointer may be either here or in the system file table, depending on the implementation and whether the file is being shared or not.)

Figure illustrates some of the interactions of file system components when files are created and/or used:

- When a new file is created, a new FCB is allocated and filled out with important information regarding the new file. The appropriate directory is modified with the new file name and FCB information.
- When a file is accessed during a program, the open ( ) system call reads in the FCB information from disk, and stores it in the system-wide open file table. An entry is added to the per-process open file table referencing the system-wide table, and an index into the per-process table is returned by the open( ) system call. UNIX refers to this index as a *file descriptor*, and Windows refers to it as a *file handle*.
- If another process already has a file open when a new request comes in for the same file, and it is sharable, then a counter in the system-wide table is incremented and the per-process table is adjusted to point to the existing entry in the system-wide table.
- When a file is closed, the per-process table entry is freed, and the counter in the system-wide table is decremented. If that counter reaches zero, then the system wide table is also freed. Any data currently stored in memory cache for this file is written out to disk if necessary.

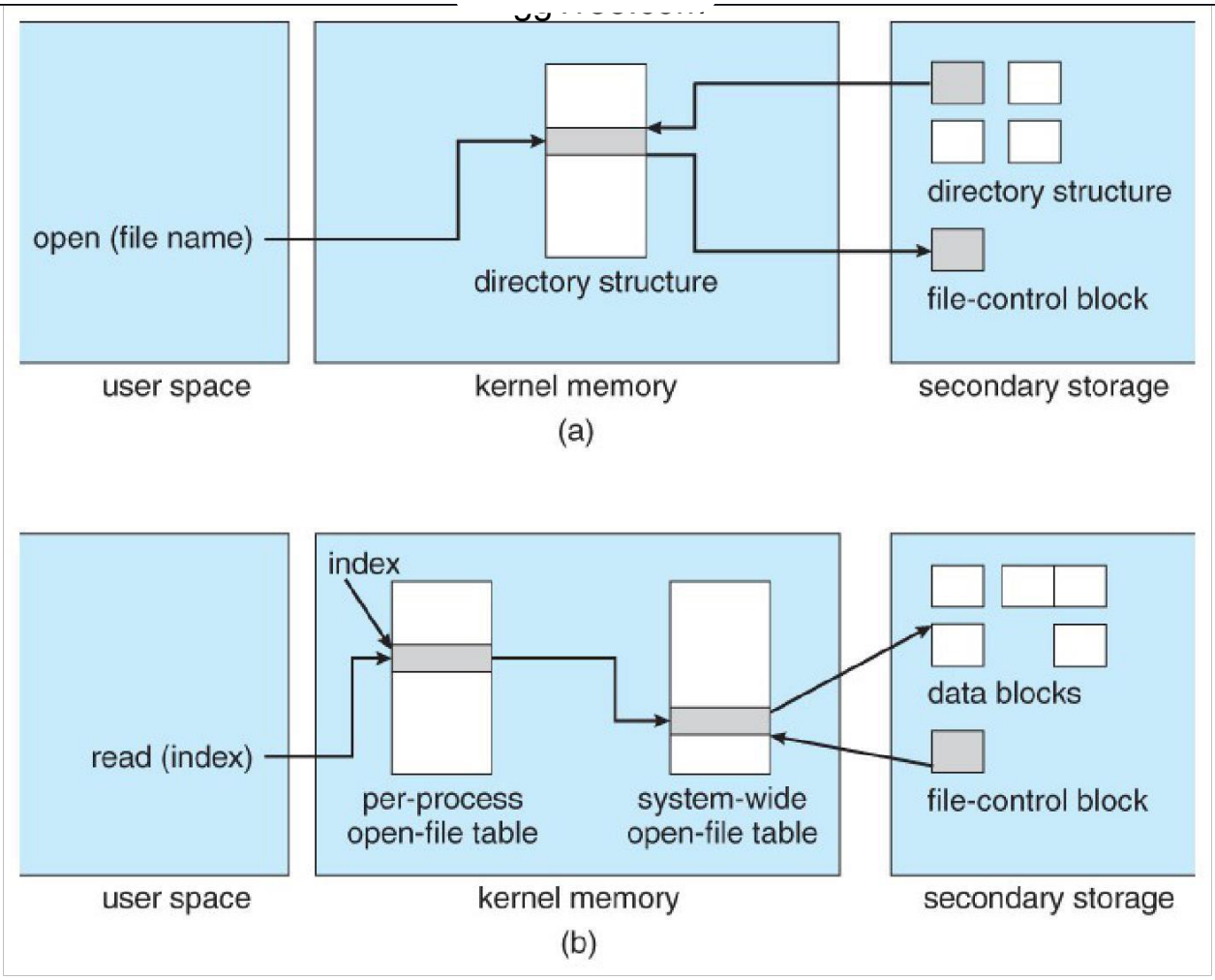


Figure 12.3 - In-memory file-system structures. (a) File open. (b) File read.

12.2.2 Partitions and Mounting

Physical disks are commonly divided into smaller units called partitions. They can also be combined into larger units, but that is most commonly done for RAID installations and is left for later chapters.

Partitions can either be used as raw devices (with no structure imposed upon them), or they can be formatted to hold a file system (i.e. populated with FCBs and initial directory structures as appropriate.) Raw partitions are generally used for swap space, and may also be used for certain programs such as databases that choose to manage their own disk storage system. Partitions containing filesystems can generally only be accessed using the file system structure by ordinary users, but can often be accessed as a raw device also by root.

The boot block is accessed as part of a raw partition, by the boot program prior to any operating system being loaded. The ***root partition*** contains the OS kernel and at least the key portions of the OS needed to complete the boot process. At boot time the root partition is mounted, and control is transferred from the boot program to the kernel found there. (Older systems required that the root partition lie completely within the first 1024 cylinders of the disk, because that was as far as the boot program could reach. Once the kernel had control, then it could access partitions beyond the 1024 cylinder boundary.)

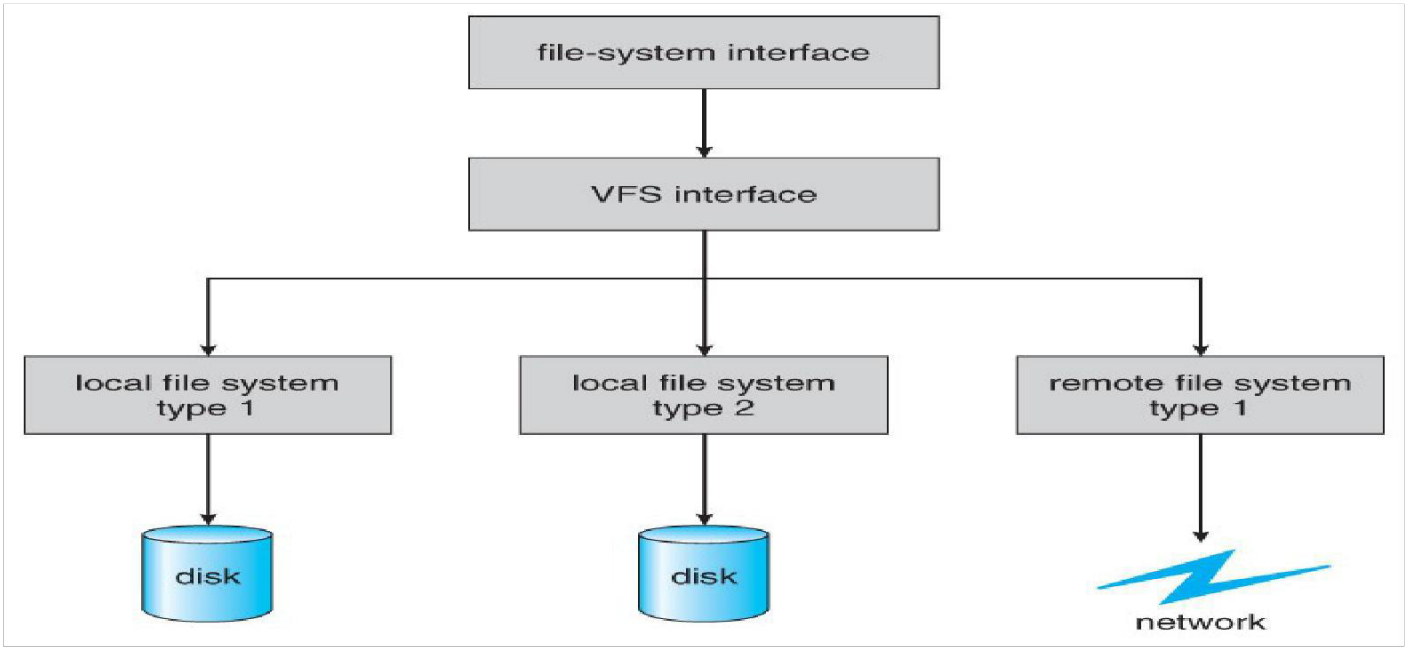
Virtual File Systems

***Virtual File Systems, VFS*,** provide a common interface to multiple different file system types. In addition, it provides for a unique identifier (vnode) for files across the entire space, including across all file systems of different types. (UNIX inodes are unique only across a single file system, and certainly do not carry across networked file systems)

The VFS in Linux is based upon four key object types:

- The *inode* object, representing an individual file
- The *file* object, representing an open file.
- The *superblock* object, representing a file system.
- The *dentry* object, representing a directory entry.

Linux VFS provides a set of common functionalities for each file system, using function pointers accessed through a table. The same functionality is accessed through the same table position for all file system types, though the actual functions pointed to by the pointers may be file system-specific. Common operations provided include open( ), read( ), write( ), and mmap( ).



Schematic view of a virtual file system.

Directory implementation:

Directories need to be fast to search, insert, and delete, with a minimum of wasted disk space.

Linear List

A linear list is the simplest and easiest directory structure to set up, but it does have some drawbacks.

Finding a file (or verifying one does not already exist upon creation) requires a linear search.

Deletions can be done by moving all entries, flagging an entry as deleted, or by moving the last entry into the newly vacant position.

Sorting the list makes searches faster, at the expense of more complex insertions and deletions.

A linked list makes insertions and deletions into a sorted list easier, with overhead for the links.

More complex data structures, such as B-trees, could also be considered.

Hash Table

A hash table can also be used to speed up searches.

Hash tables are generally implemented *in addition to* a linear or other structure

Allocation Methods:

The allocation methods define how the files are stored in the disk blocks. There are three main disk space or file allocation methods.

Contiguous Allocation

Linked Allocation

Indexed Allocation

The main idea behind these methods is to provide:

Efficient disk space utilization.

Fast access to the file blocks.

All the **three methods** have their own advantages and disadvantages as discussed below:

Contiguous Allocation

Contiguous Allocation requires that all blocks of a file be kept together contiguously.

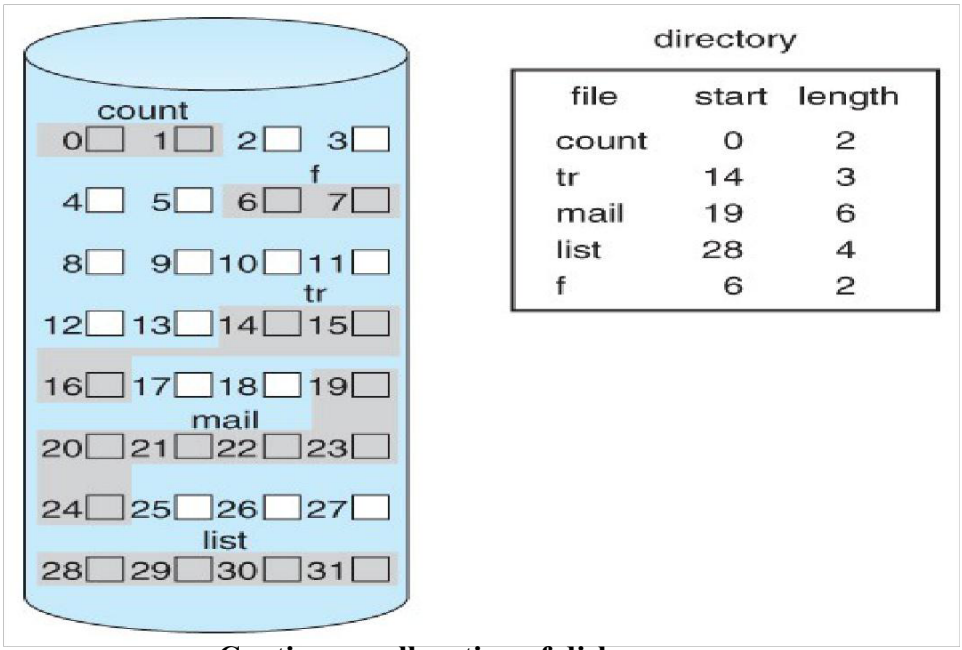
Performance is very fast, because reading successive blocks of the same file generally requires no movement of the disk heads, or at most one small step to the next adjacent cylinder.

Storage allocation involves the same issues discussed earlier for the allocation of contiguous blocks of memory (first fit, best fit, fragmentation problems, etc.) The distinction is that the high time penalty required for moving the disk heads from spot to spot may now justify the benefits of keeping files contiguously when possible. In this scheme, each file occupies a contiguous set of blocks on the disk. For example, if a file requires n blocks and is given a block b as the starting location, then the blocks assigned to the file will be: b, b+1, b+2,……b+n-1.This means that given the starting block address and the length of the file (in terms of blocks required), we can determine the blocks occupied by the file.

The directory entry for a file with contiguous allocation contains

Address of starting block
Length of the allocated portion.

The file 'mail' in the following figure starts from the block 19 with length = 6 blocks. Therefore, it occupies 19, 20, 21, 22, 23, 24 blocks.



Contiguous allocation of disk space.

Advantages:

Both the Sequential and Direct Accesses are supported by this. For direct access, the address of the kth block of the file which starts at block b can easily be obtained as (b+k).

This is extremely fast since the number of seeks are minimal because of contiguous allocation of file blocks.

Disadvantages:

This method suffers from both internal and external fragmentation. This makes it inefficient in terms of memory utilization.

Increasing file size is difficult because it depends on the availability of contiguous memory at a particular instance.

Linked Allocation

Disk files can be stored as linked lists, with the expense of the storage space consumed by each link. (E.g. a block may be 508 bytes instead of 512.)

Linked allocation involves no external fragmentation, does not require pre-known file sizes, and allows files to grow dynamically at any time.

Unfortunately linked allocation is only efficient for sequential access files, as random access requires starting at the beginning of the list for each new location access.

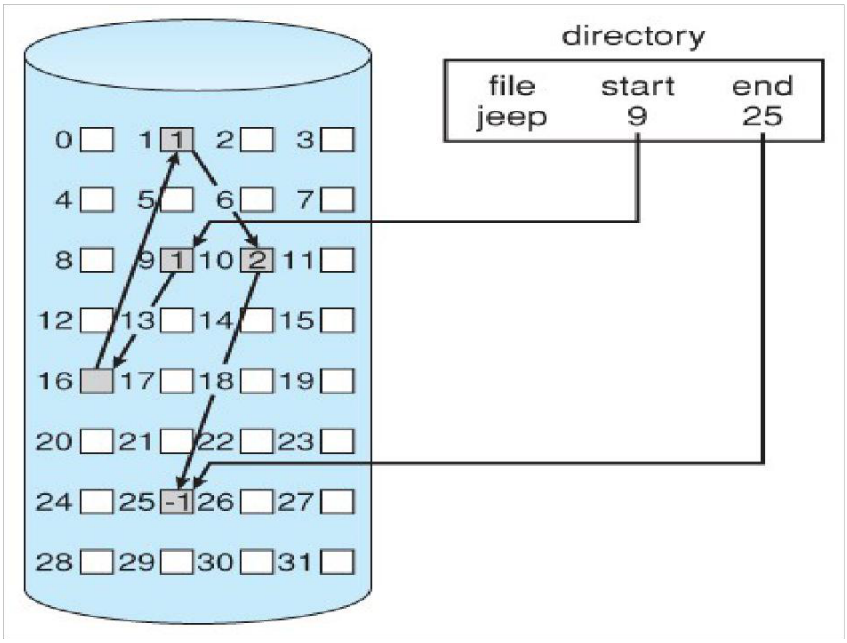
Allocating *clusters* of blocks reduces the space wasted by pointers, at the cost of internal fragmentation.

Another big problem with linked allocation is reliability if a pointer is lost or damaged. Doubly linked lists provide some protection, at the cost of additional overhead and wasted space.

In this scheme, each file is a linked list of disk blocks which need not be contiguous. The disk blocks can be scattered anywhere on the disk.

The directory entry contains a pointer to the starting and the ending file block. Each block contains a pointer to the next block occupied by the file.

The file 'jeep' in following image shows how the blocks are randomly distributed. The last block (25) contains -1 indicating a null pointer and does not point to any other block.



Linked allocation of disk space.

Advantages:

This is very flexible in terms of file size. File size can be increased easily since the system does not have to look for a contiguous chunk of memory.

This method does not suffer from external fragmentation. This makes it relatively better in terms of memory utilization.

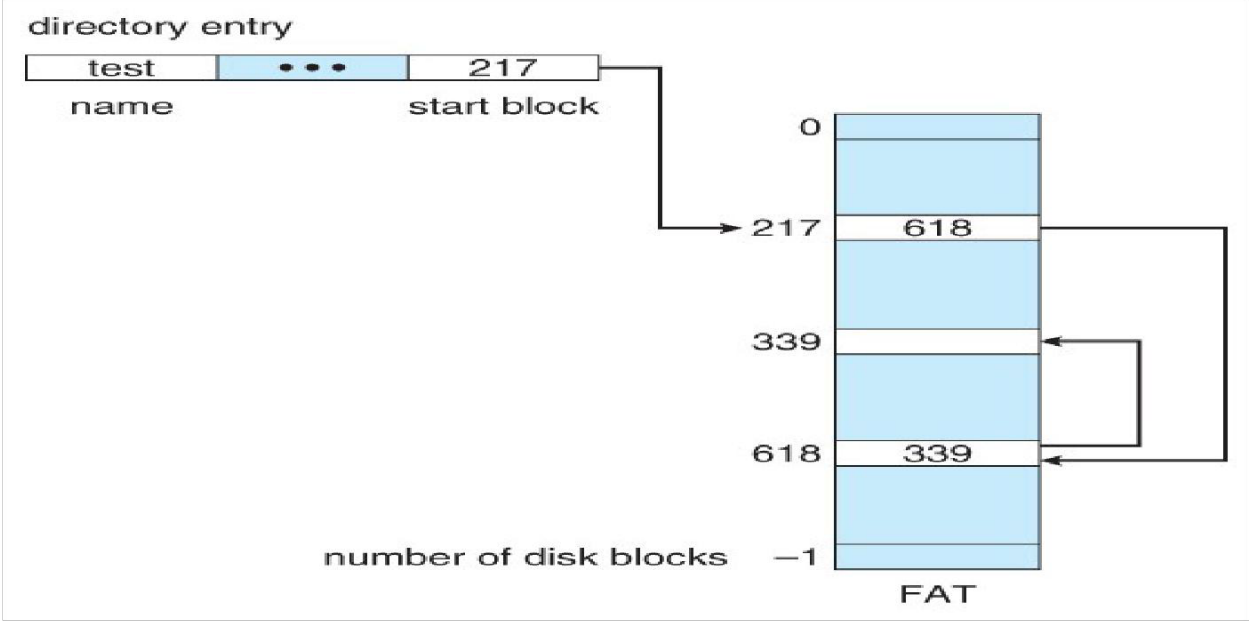
Disadvantages:

Because the file blocks are distributed randomly on the disk, a large number of seeks are needed to access every block individually. This makes linked allocation slower.

It does not support random or direct access. We cannot directly access the blocks of a file. A block k of a file can be accessed by traversing k blocks sequentially (sequential access) from the starting block of the file via block pointers.

Pointers required in the linked allocation incur some extra overhead.

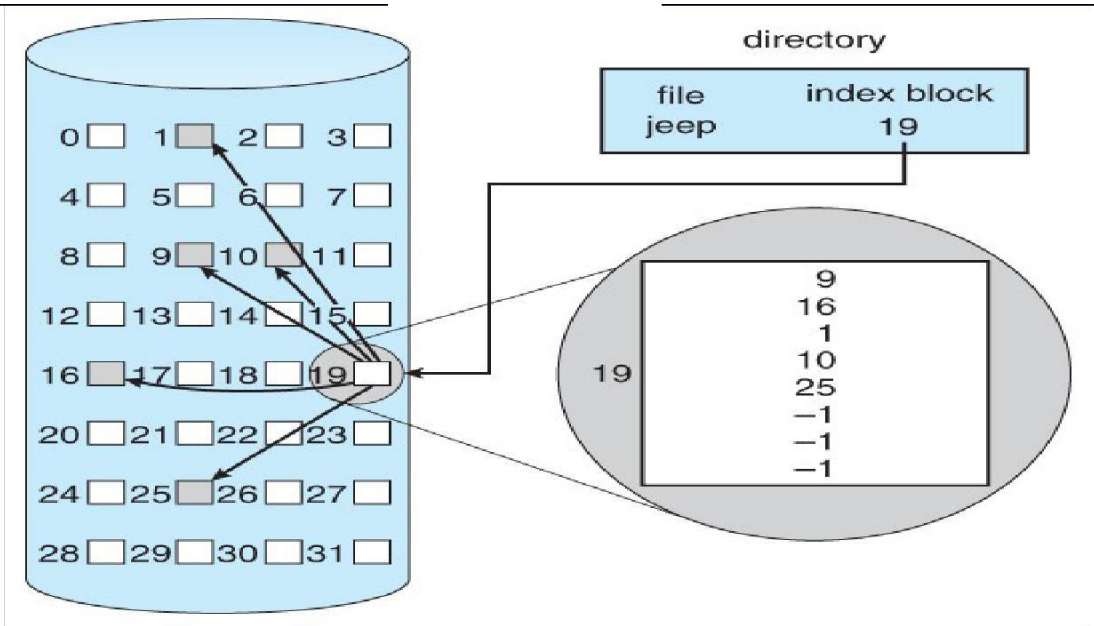
The ***File Allocation Table, FAT,*** used by DOS is a variation of linked allocation, where all the links are stored in a separate table at the beginning of the disk. The benefit of this approach is that the FAT table can be cached in memory, greatly improving random access speeds.



File-allocation table.

Indexed Allocation

Indexed Allocation combines all of the indexes for accessing each file into a common block (for that file), as opposed to spreading them all over the disk or storing them in a FAT table.



Indexed allocation of disk space.

Advantages:

This supports direct access to the blocks occupied by the file and therefore provides fast access to the file blocks.

It overcomes the problem of external fragmentation.

Disadvantages:

The pointer overhead for indexed allocation is greater than linked allocation.

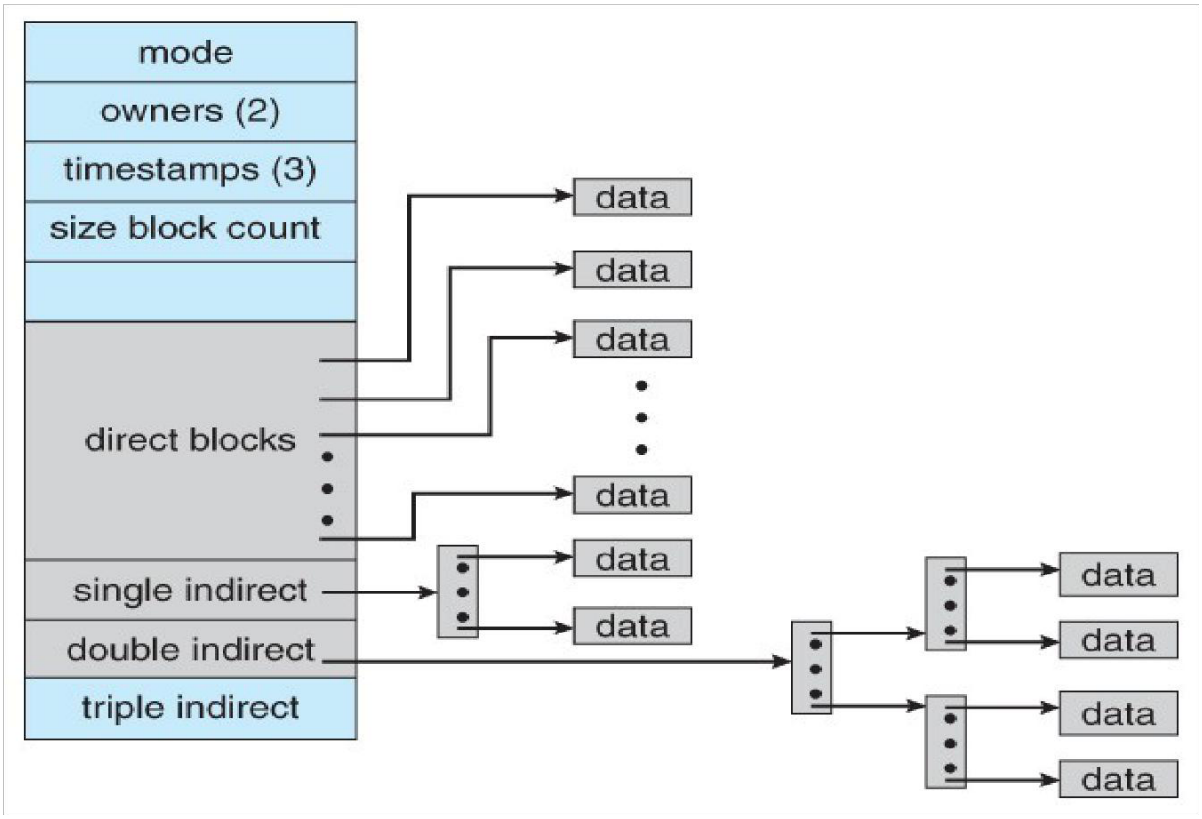
For very small files, say files that expand only 2-3 blocks, the indexed allocation would keep one entire block (index block) for the pointers which is inefficient in terms of memory utilization. However, in linked allocation we lose the space of only 1 pointer per block.

Some disk space is wasted (relative to linked lists or FAT tables) because an entire index block must be allocated for each file, regardless of how many data blocks the file contains. This leads to questions of how big the index block should be, and how it should be implemented. There are several approaches:

Linked Scheme - An index block is one disk block, which can be read and written in a single disk operation. The first index block contains some header information, the first N block addresses, and if necessary a pointer to additional linked index blocks.

Multi-Level Index - The first index block contains a set of pointers to secondary index blocks, which in turn contain pointers to the actual data blocks.

Combined Scheme - This is the scheme used in UNIX inodes, in which the first 12 or so data block pointers are stored directly in the inode, and then singly, doubly, and triply indirect pointers provide access to more data blocks as needed. (See below) The advantage of this scheme is that for small files (which many are), the data blocks are readily accessible (up to 48K with 4K block sizes); files up to about 4144K (using 4K blocks) are accessible with only a single indirect block (which can be cached), and huge files are still accessible using a relatively small number of disk accesses (larger in theory than can be addressed by a 32-bit address, which is why some systems have moved to 64-bit file pointers.)



The UNIX inode.

Performance

The optimal allocation method is different for sequential access files than for random access files, and is also different for small files than for large files.

Some systems support more than one allocation method, which may require specifying how the file is to be used (sequential or random access) at the time it is allocated. Such systems also provide conversion utilities.

Some systems have been known to use contiguous access for small files, and automatically switch to an indexed scheme when file sizes surpass a certain threshold.

And of course some systems adjust their allocation schemes (e.g. block sizes) to best match the characteristics of the hardware for optimum performance.

Free Space Management:

Another important aspect of disk management is keeping track of and allocating free space.

Bit Vector

One simple approach is to use a *bit vector*, in which each bit represents a disk block, set to 1 if free or 0 if allocated.

Fast algorithms exist for quickly finding contiguous blocks of a given size

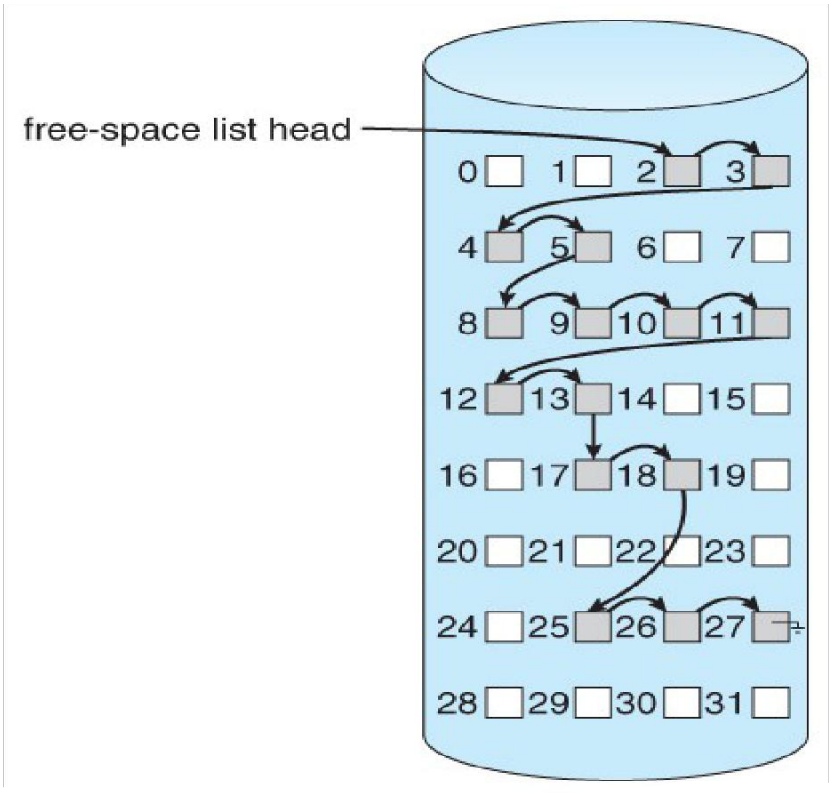
The down side is that a 40GB disk requires over 5MB just to store the bitmap. (For example.)

Linked List

A linked list can also be used to keep track of all free blocks.

Traversing the list and/or finding a contiguous block of a given size are not easy, but fortunately are not frequently needed operations. Generally the system just adds and removes single blocks from the beginning of the list.

The **FAT** table keeps track of the free list as just one more linked list on the table.



Linked free-space list on disk.

Grouping

A variation on linked list free lists is to use links of blocks of indices of free blocks. If a block holds up to N addresses, then the first block in the linked-list contains up to N-1 addresses of free blocks and a pointer to the next block of free addresses.

Counting

When there are multiple contiguous blocks of free space then the system can keep track of the starting address of the group and the number of contiguous free blocks. As long as the average length of a contiguous group of free blocks is greater than two this offers a savings in space needed for the free list. (Similar to compression techniques used for graphics images when a group of pixels all the same color is encountered.)

Space Maps (New)

Sun's ZFS file system was designed for HUGE numbers and sizes of files, directories, and even file systems.

The resulting data structures could be VERY inefficient if not implemented carefully. For example, freeing up a 1 GB file on a 1 TB file system could involve updating thousands of blocks of free list bit maps if the file was spread across the disk.

ZFS uses a combination of techniques, starting with dividing the disk up into (hundreds of) *metaslabs* of a manageable size, each having their own space map.

Free blocks are managed using the counting technique, but rather than write the information to a table, it is recorded in a log-structured transaction record. Adjacent free blocks are also coalesced into a larger single free block.

An in-memory space map is constructed using a balanced tree data structure, constructed from the log data.

The combination of the in-memory tree and the on-disk log provide for very fast and efficient management of these very large files and free blocks.

Efficiency and Performance:

Efficiency

UNIX pre-allocates inodes, which occupies space even before any files are created.

UNIX also distributes inodes across the disk, and tries to store data files near their inode, to reduce the distance of disk seeks between the inodes and the data.

Some systems use variable size clusters depending on the file size.

The more data that is stored in a directory (e.g. last access time), the more often the directory blocks have

to be re-written.

As technology advances, addressing schemes have had to grow as well.

Sun's ZFS file system uses 128-bit pointers, which should theoretically never need to be expanded. (The mass required to store 2^128 bytes with atomic storage would be at least 272 trillion kilograms!) Kernel table sizes used to be fixed, and could only be changed by rebuilding the kernels. Modern tables are dynamically allocated, but that requires more complicated algorithms for accessing them.

Performance

Disk controllers generally include on-board caching. When a seek is requested, the heads are moved into place, and then an entire track is read, starting from whatever sector is currently under the heads (reducing latency.) The requested sector is returned and the unrequested portion of the track is cached in the disk's electronics.

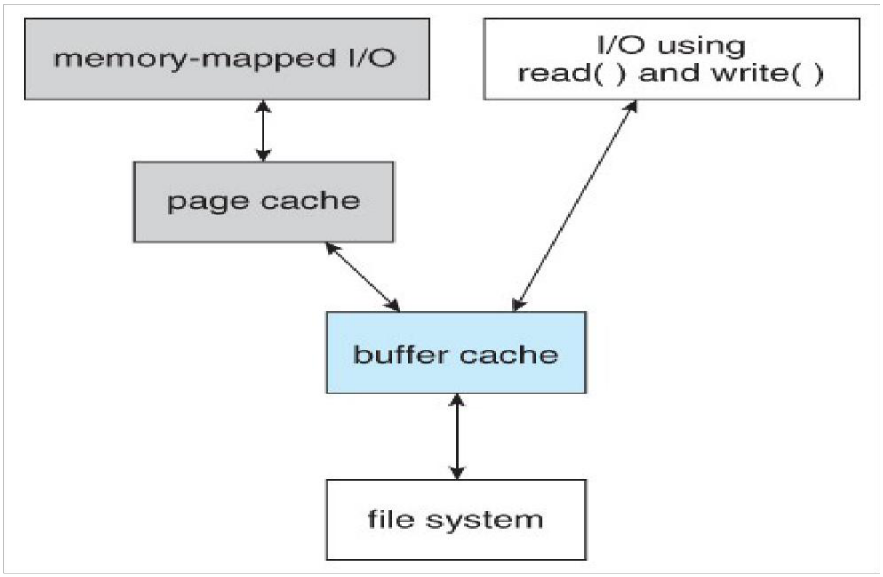
Some OSes cache disk blocks they expect to need again in a ***buffer cache.***

A ***page cache*** connected to the virtual memory system is actually more efficient as memory addresses do not need to be converted to disk block addresses and back again.

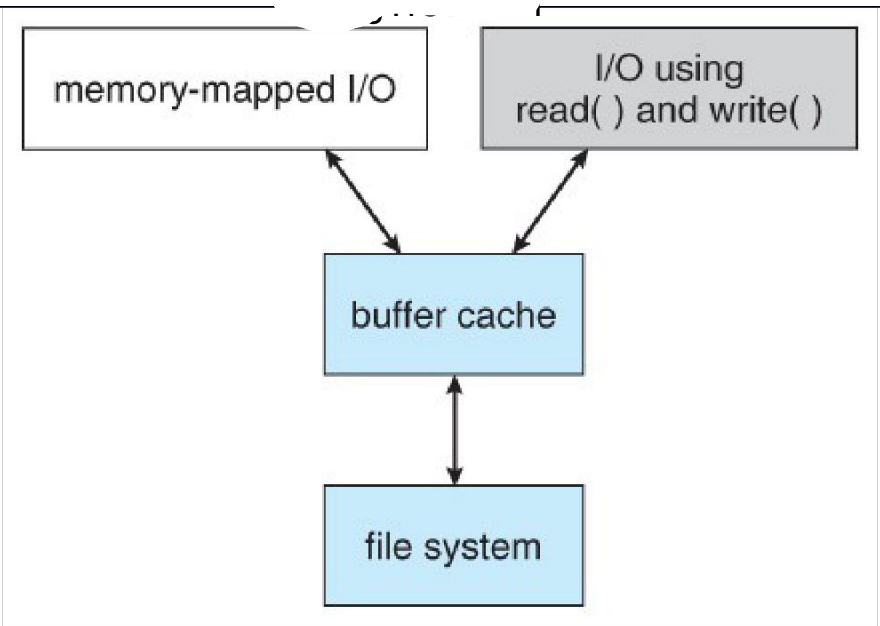
Some systems (Solaris, Linux, Windows 2000, NT, XP) use page caching for both process pages and file data in a ***unified virtual memory.***

Figures show the advantages of the *unified buffer cache* found in some versions of UNIX and Linux

- Data does not need to be stored twice, and problems of inconsistent buffer information are avoided.



I/O without a unified buffer cache.



I/O using a unified buffer cache.

Page replacement strategies can be complicated with a unified cache, as one needs to decide whether to replace process or file pages, and how many pages to guarantee to each category of pages. Solaris, for example, has gone through many variations, resulting in *priority paging* giving process pages priority over file I/O pages, and setting limits so that neither can knock the other completely out of memory.

Another issue affecting performance is the question of whether to implement *synchronous writes* or *asynchronous writes.* Synchronous writes occur in the order in which the disk subsystem receives them, without caching; Asynchronous writes are cached, allowing the disk subsystem to schedule writes in a more efficient order

The type of file access can also have an impact on optimal page replacement policies. For example, LRU is not necessarily a good policy for sequential access files. For these types of files progression normally goes in a forward direction only, and the most recently used page will not be needed again until after the file has been rewound and re-read from the beginning, (if it is ever needed at all.)

On the other hand, we can expect to need the next page in the file fairly soon. For this reason sequential access files often take advantage of two special policies:

- ***Free-behind*** frees up a page as soon as the next page in the file is requested, with the assumption that we are now done with the old page and won't need it again for a long time.
- ***Read-ahead*** reads the requested page and several subsequent pages at the same time, with the assumption that those pages will be needed in the near future. This is similar to the track caching that is already performed by the disk controller, except it saves the future latency of transferring data from the disk controller memory into motherboard main memory.

The caching system and asynchronous writes speed up disk writes considerably, because the disk subsystem can schedule physical writes to the disk to minimize head movement and disk seek times.

Reads, on the other hand, must be done more synchronously in spite of the caching system, with the result that disk writes can counter-intuitively be much faster on average than disk reads.

Recovery:

Consistency Checking

The storing of certain data structures (e.g. directories and inodes) in memory and the caching of disk operations can speed up performance, but what happens in the result of a system crash? All volatile memory structures are lost, and the information stored on the hard drive may be left in an inconsistent state.

A ***Consistency Checker*** is often run at boot time or mount time, particularly if a filesystem was not closed down properly. Some of the problems that these tools look for include:

Disk blocks allocated to files and also listed on the free list.
Disk blocks neither allocated to files nor on the free list.
Disk blocks allocated to more than one file.
The number of disk blocks allocated to a file inconsistent with the file's stated size. Properly allocated files / inodes which do not appear in any directory entry.

Log-Structured File Systems

Log-based transaction-oriented filesystems borrow techniques developed for databases, guaranteeing that any given transaction either completes successfully or can be rolled back to a safe state before the transaction commenced:

All metadata changes are written sequentially to a log.
A set of changes for performing a specific task (e.g. moving a file) is a *transaction*.
As changes are written to the log they are said to be *committed,* allowing the system to return to its work. In the meantime, the changes from the log are carried out on the actual filesystem, and a pointer keeps track of which changes in the log have been completed and which have not yet been completed.
When all changes corresponding to a particular transaction have been completed, that transaction can be safely removed from the log.
At any given time, the log will contain information pertaining to uncompleted transactions only, e.g. actions that were committed but for which the entire transaction has not yet been completed. From the log, the remaining transactions can be completed,or if the transaction was aborted, then the partially completed changes can be undone.

Other Solutions (New)

Sun's ZFS and Network Appliance's WAFL file systems take a different approach to file system consistency.
No blocks of data are ever over-written in place. Rather the new data is written into fresh new blocks, and

after the transaction is complete, the metadata (data block pointers) is updated to point to the new blocks. The old blocks can then be freed up for future use.

Alternatively, if the old blocks and old metadata are saved, then a *snapshot* of the system in its original state is preserved. This approach is taken by WAFL.

ZFS combines this with check-summing of all metadata and data blocks, and RAID, to ensure that no inconsistencies are possible, and therefore ZFS does not incorporate a consistency checker.

Backup and Restore

In order to recover lost data in the event of a disk crash, it is important to conduct backups regularly.

Files should be copied to some removable medium, such as magnetic tapes, CDs, DVDs, or external removable hard drives.

A full backup copies every file on a file system.

Incremental backups copy only files which have changed since some previous time.

A combination of full and incremental backups can offer a compromise between full recoverability, the number and size of backup tapes needed, and the number of tapes that need to be used to do a full restore. For example, one strategy might be:

- At the beginning of the month do a full backup.
- At the end of the first and again at the end of the second week, backup all files which have changed since the beginning of the month.
- At the end of the third week, backup all files that have changed since the end of the second week.
- Every day of the month not listed above, do an incremental backup of all files that have changed since the most recent of the weekly backups described above.

Backup tapes are often reused, particularly for daily backups, but there are limits to how many times the same tape can be used.

Every so often a full backup should be made that is kept "forever" and not overwritten.

Backup tapes should be tested, to ensure that they are readable!

For optimal security, backup tapes should be kept off-premises, so that a fire or burglary cannot destroy both the system and the backups. There are companies (e.g. Iron Mountain) that specialize in the secure off-site storage of critical backup information.

Keep your backup tapes secure - The easiest way for a thief to steal all your data is to simply pocket your backup tapes!

Storing important files on more than one computer can be an alternate though less reliable form of backup.

Note that incremental backups can also help users to get back a previous version of a file that they have since changed in some way.

Beware that backups can help forensic investigators recover e-mails and other files that users had though they had deleted!

I/O Systems:

Overview

Management of I/O devices is a very important part of the operating system - so important and so varied that entire I/O subsystems are devoted to its operation. (Consider the range of devices on a modern computer, from mice, keyboards, disk drives, display adapters, USB devices, network connections, audio I/O, printers, special devices for the handicapped, and many special-purpose peripherals.)

I/O Subsystems must contend with two (conflicting?) trends: (1) The gravitation towards standard interfaces for a wide range of devices, making it easier to add newly developed devices to existing systems, and (2) the development of entirely new types of devices, for which the existing standard interfaces are not always easy to apply.

Device drivers are modules that can be plugged into an OS to handle a particular device or category of similar devices.

I/O Hardware:

I/O devices can be roughly categorized as storage, communications, user-interface, and

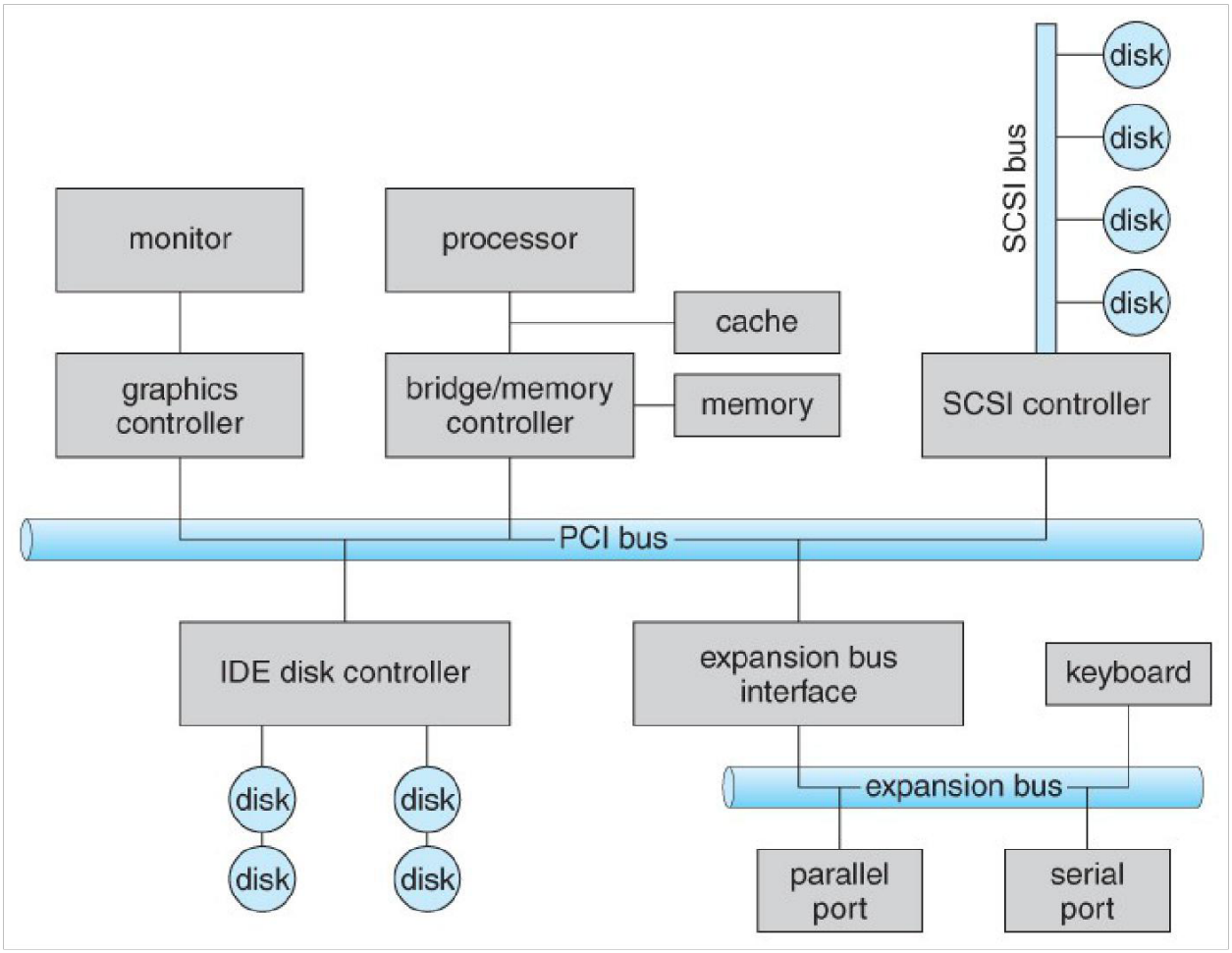
other Devices communicate with the computer via signals sent over wires or through the air.

Devices connect with the computer via ***ports***, e.g. a serial or parallel port. A

common set of wires connecting multiple devices is termed a ***bus.***

- Buses include rigid protocols for the types of messages that can be sent across the bus and the procedures for resolving contention issues.
- Figure below illustrates three of the four bus types commonly found in a modern PC:

1. The ***PCI bus*** connects high-speed high-bandwidth devices to the memory subsystem (and the CPU.)
2. The ***expansion bus*** connects slower low-bandwidth devices, which typically deliver data one character at a time (with buffering.)
3. The ***SCSI bus*** connects a number of SCSI devices to a common SCSI controller.
4. A ***daisy-chain bus,*** (not shown) is when a string of devices is connected to each other like beads on a chain, and only one of the devices is directly connected to the host.



A typical PC bus structure.

One way of communicating with devices is through ***registers*** associated with each port. Registers may be one to four bytes in size, and may typically include (a subset of) the following four:

1. The ***data-in register*** is read by the host to get input from the device.
2. The ***data-out register*** is written by the host to send output.
3. The ***status register*** has bits read by the host to ascertain the status of the device, such as idle, ready for input, busy, error, transaction complete, etc.
4. The ***control register*** has bits written by the host to issue commands or to change settings of the device such as parity checking, word length, or full- versus half-duplex operation.

Figure shows some of the most common I/O port address ranges.

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

Device I/O port locations on PCs (partial).

Another technique for communicating with devices is ***memory-mapped I/O.***

- In this case a certain portion of the processor's address space is mapped to the device, and communications occur by reading and writing directly to/from those memory areas.
- Memory-mapped I/O is suitable for devices which must move large quantities of data quickly, such as graphics cards.
- Memory-mapped I/O can be used either instead of or more often in combination with traditional registers. For example, graphics cards still use registers for control information such as setting the video mode.

- A potential problem exists with memory-mapped I/O, if a process is allowed to write directly to the address space used by a memory-mapped I/O device.
- (Note: Memory-mapped I/O is not the same thing as direct memory access, DMA.)

Polling

One simple means of device *handshaking* involves polling:

1. The host repeatedly checks the *busy bit* on the device until it becomes clear.
2. The host writes a byte of data into the data-out register, and sets the *write bit* in the command register (in either order.)
3. The host sets the *command ready bit* in the command register to notify the device of the pending command.
4. When the device controller sees the command-ready bit set, it first sets the busy bit.
5. Then the device controller reads the command register, sees the write bit set, reads the byte of data from the data-out register, and outputs the byte of data.
6. The device controller then clears the *error bit* in the status register, the command-ready bit, and finally clears the busy bit, signalling the completion of the operation.

Polling can be very fast and efficient, if both the device and the controller are fast and if there is significant data to transfer. It becomes inefficient, however, if the host must wait a long time in the busy loop waiting for the device, or if frequent checks need to be made for data that is infrequently there.

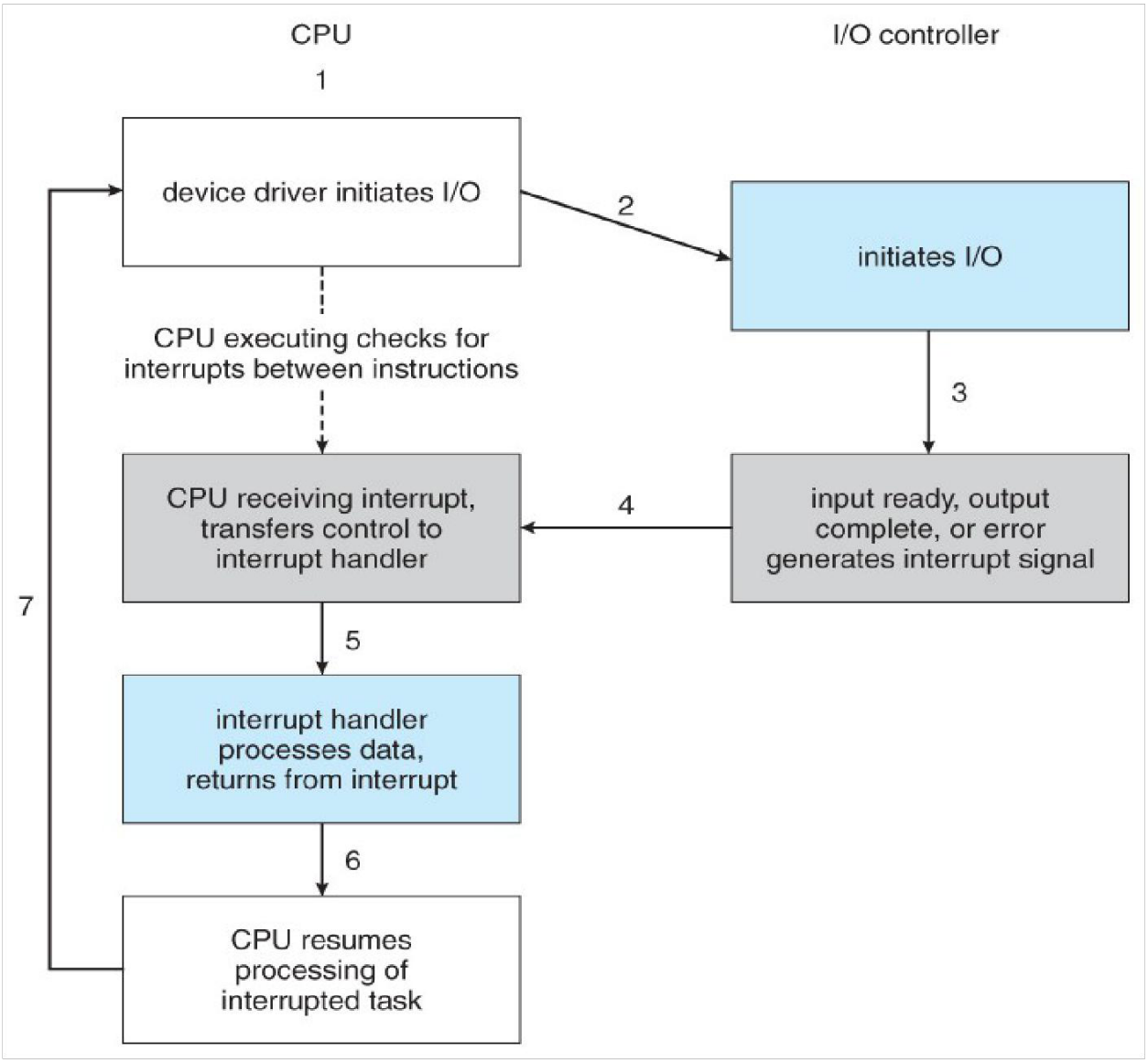
Interrupts

Interrupts allow devices to notify the CPU when they have data to transfer or when an operation is complete, allowing the CPU to perform other duties when no I/O transfers need its immediate attention.

The CPU has an *interrupt-request line* that is sensed after every instruction.

- A device's controller *raises* an interrupt by asserting a signal on the interrupt request line.
- The CPU then performs a state save, and transfers control to the *interrupt handler* routine at a fixed address in memory. (The CPU *catches* the interrupt and *dispatches* the interrupt handler.)
- The interrupt handler determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a *return from interrupt* instruction to return control to the CPU. (The interrupt handler *clears* the interrupt by servicing the device.)

(Note that the state restored does not need to be the same state as the one that was saved when the interrupt went off. See below for an example involving time-slicing.)



Interrupt-driven I/O procedure

Interrupt-driven I/O cycle.

The above description is adequate for simple interrupt-driven I/O, but there are three needs in modern computing which complicate the picture:

1. The need to defer interrupt handling during critical processing,
2. The need to determine *which* interrupt handler to invoke, without having to poll all devices to see which one needs attention, and
3. The need for multi-level interrupts, so the system can differentiate between high- and low-priority interrupts for proper response.

These issues are handled in modern computer architectures with *interrupt-controller* hardware.

- Most CPUs now have two interrupt-request lines: One that is *non-maskable* for critical error conditions and one that is *maskable,* that the CPU can temporarily ignore during critical processing.
- The interrupt mechanism accepts an *address,* which is usually one of a small set of numbers for an offset into a table called the *interrupt vector.* This table (usually located at physical address zero ?) holds the addresses of routines prepared to process specific interrupts.
- The number of possible interrupt handlers still exceeds the range of defined interrupt numbers, so multiple handlers can be *interrupt chained*. Effectively the addresses held in the interrupt vectors are the head pointers for linked-lists of interrupt handlers.
- Figure shows the Intel Pentium interrupt vector. Interrupts 0 to 31 are non-maskable and reserved for serious hardware and other errors. Maskable interrupts, including normal device I/O interrupts begin at interrupt 32.Modern interrupt hardware also supports *interrupt priority levels*, allowing systems to mask off only lower-priority interrupts while servicing a high-priority interrupt, or conversely to allow a high-priority signal to interrupt the processing of a low-priority one.

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Intel Pentium processor event-vector table.

At boot time the system determines which devices are present, and loads the appropriate handler addresses into the interrupt table.

During operation, devices signal errors or the completion of commands via interrupts.

Exceptions, such as dividing by zero, invalid memory accesses, or attempts to access kernel mode instructions can be signalled via interrupts.

Time slicing and context switches can also be implemented using the interrupt mechanism.

- The scheduler sets a hardware timer before transferring control over to a user process.
- When the timer raises the interrupt request line, the CPU performs a state-save, and transfers control over to the proper interrupt handler, which in turn runs the scheduler.
- The scheduler does a state-restore of a *different* process before resetting the timer and issuing the return-from-interrupt instruction.

A similar example involves the paging system for virtual memory - A page fault causes an interrupt, which in turn issues an I/O request and a context switch as described above, moving the interrupted process into the wait queue and selecting a different process to run. When the I/O request has completed (i.e. when the requested page has been loaded up into physical memory), then the device interrupts, and the interrupt handler moves the process from the wait queue into the ready queue, (or depending on scheduling algorithms and policies, may go ahead and context switch it back onto the CPU.)

System calls are implemented via *software interrupts,* a.k.a. *traps.* When a (library) program needs work performed in kernel mode, it sets command information and possibly data addresses in certain registers, and then raises a software interrupt. (E.g. 21 hex in DOS.) The system does a state save and then calls on the proper interrupt handler to process the request in kernel mode. Software interrupts generally have low priority, as they are not as urgent as devices with limited buffering space.

Interrupts are also used to control kernel operations, and to schedule activities for optimal performance. For example, the completion of a disk read operation involves two interrupts:

- A high-priority interrupt acknowledges the device completion, and issues the next disk request so that the hardware does not sit idle.
- A lower-priority interrupt transfers the data from the kernel memory space to the user space, and then transfers the process from the waiting queue to the ready queue.

The Solaris OS uses a multi-threaded kernel and priority threads to assign different threads to different interrupt handlers. This allows for the "simultaneous" handling of multiple interrupts, and the assurance that high-priority interrupts will take precedence over low-priority ones and over user processes.

Direct Memory Access

For devices that transfer large quantities of data (such as disk controllers), it is wasteful to tie up the CPU transferring data in and out of registers one byte at a time.

Instead this work can be off-loaded to a special processor, known as the *Direct Memory Access, DMA, Controller.*

The host issues a command to the DMA controller, indicating the location where the data is located, the location where the data is to be transferred to, and the number of bytes of data to transfer. The DMA controller handles the data transfer, and then interrupts the CPU when the transfer is complete.

A simple DMA controller is a standard component in modern PCs, and many *bus-mastering* I/O cards contain their own DMA hardware.

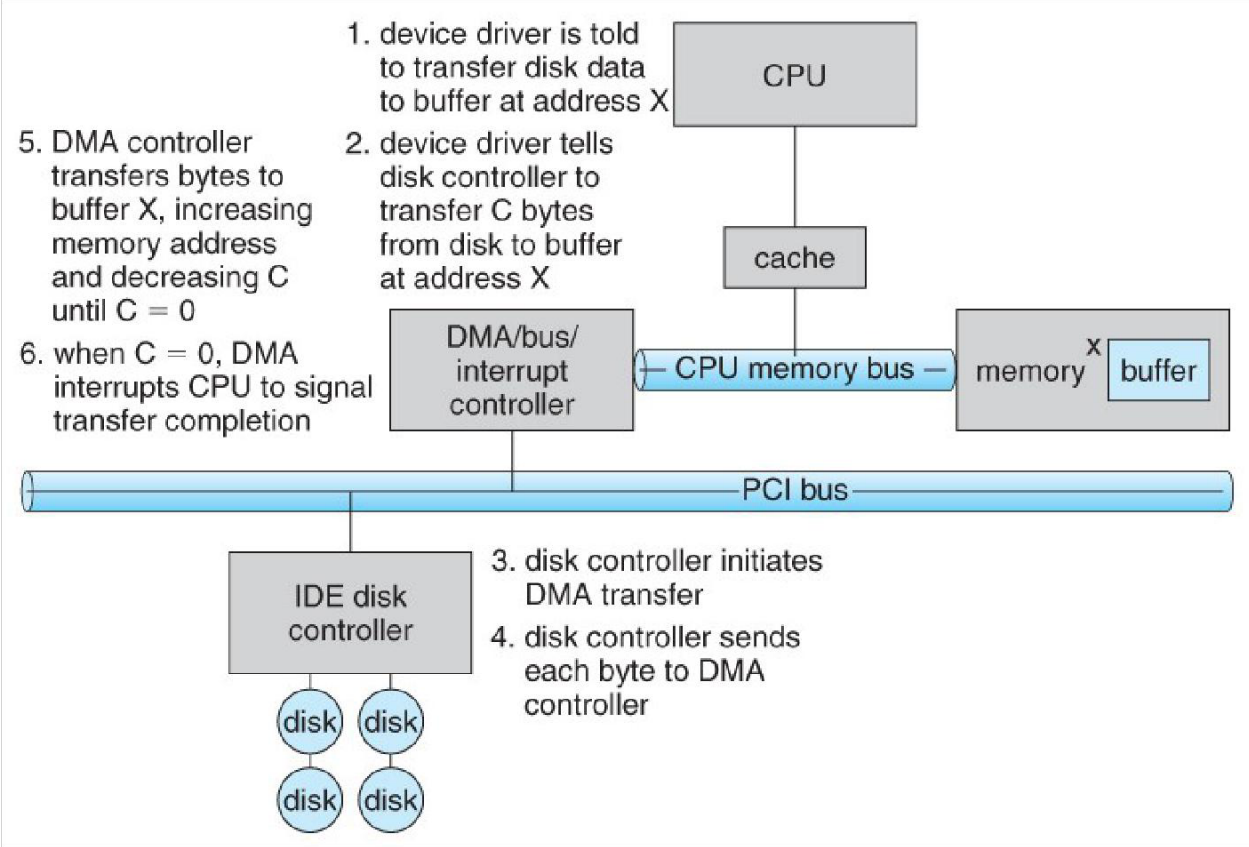
Handshaking between DMA controllers and their devices is accomplished through two wires called the DMA-request and DMA-acknowledge wires.

While the DMA transfer is going on the CPU does not have access to the PCI bus (including main memory), but it does have access to its internal registers and primary and secondary caches.

DMA can be done in terms of either physical addresses or virtual addresses that are mapped to physical addresses. The latter approach is known as ***Direct Virtual Memory Access, DVMA***, and allows direct data transfer from one memory-mapped device to another without using the main memory chips.

Direct DMA access by user processes can speed up operations, but is generally forbidden by modern systems for security and protection reasons. (i.e. DMA is a kernel-mode operation.)

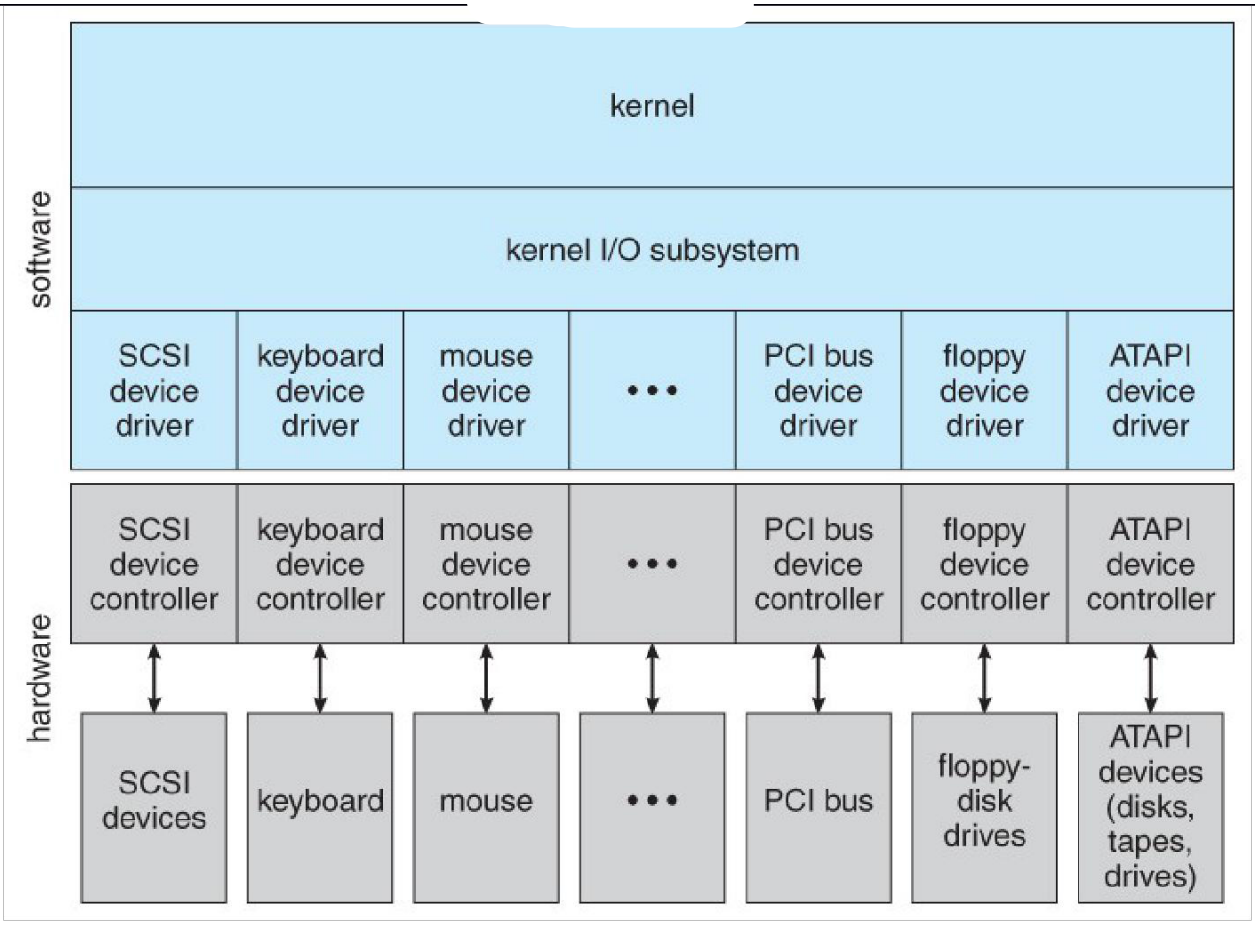
Below illustrates the **DMA** process.



Steps in a DMA transfer

Application I/O interface:

User application access to a wide variety of different devices is accomplished through layering, and through encapsulating all of the device-specific code into ***device drivers***, while application layers are presented with a common interface for all (or at least large general categories of) devices.



A kernel I/O structure.

Devices differ on many different dimensions, as outlined

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

Characteristics of I/O devices.

Block and Character Devices

Block devices are accessed a block at a time, and are indicated by a "b" as the first character in a long listing on UNIX systems. Operations supported include read( ), write( ), and seek( ).

- Accessing blocks on a hard drive directly (without going through the filesystem structure) is called ***raw I/O***, and can speed up certain operations by bypassing the buffering and locking normally conducted by the OS. (It then becomes the application's responsibility to manage those issues.)
- A new alternative is ***direct I/O,*** which uses the normal filesystem access, but which disables buffering and locking operations.

Character devices are accessed one byte at a time, and are indicated by a "c" in UNIX long listings. Supported operations include get( ) and put( ), with more advanced functionality such as reading an entire line supported by higher-level library routines.

Network Devices

Because network access is inherently different from local disk access, most systems provide a separate interface for network devices.

One common and popular interface is the ***socket*** interface, which acts like a cable or pipeline connecting two networked entities. Data can be put into the socket at one end, and read out sequentially at the other end. Sockets are normally full-duplex, allowing for bi-directional data transfer.

The select( ) system call allows servers (or other applications) to identify sockets which have data waiting, without having to poll all available sockets.

Clocks and Timers

Three types of time services are commonly needed in modern

systems:
- Get the current time of day.
- Get the elapsed time (system or wall clock) since a previous event.
- Set a timer to trigger event X at time T.

Unfortunately time operations are not standard across all systems.

A ***programmable interrupt timer, PIT*** can be used to trigger operations and to measure elapsed time. It can be set to trigger an interrupt at a specific future time, or to trigger interrupts periodically on a regular basis.

- The scheduler uses a PIT to trigger interrupts for ending time slices.
- The disk system may use a PIT to schedule periodic maintenance cleanup, such as flushing buffers to disk.
- Networks use PIT to abort or repeat operations that are taking too long to complete. I.e. resending packets if an acknowledgement is not received before the timer goes off.
- More timers than actually exist can be simulated by maintaining an ordered list of timer events, and setting the physical timer to go off when the next scheduled event should occur.

On most systems the system clock is implemented by counting interrupts generated by the PIT. Unfortunately this is limited in its resolution to the interrupt frequency of the PIT, and may be subject to some drift over time. An alternate approach is to provide direct access to a high frequency hardware counter, which provides much higher resolution and accuracy, but which does not support interrupts.

Blocking and Non-blocking I/O

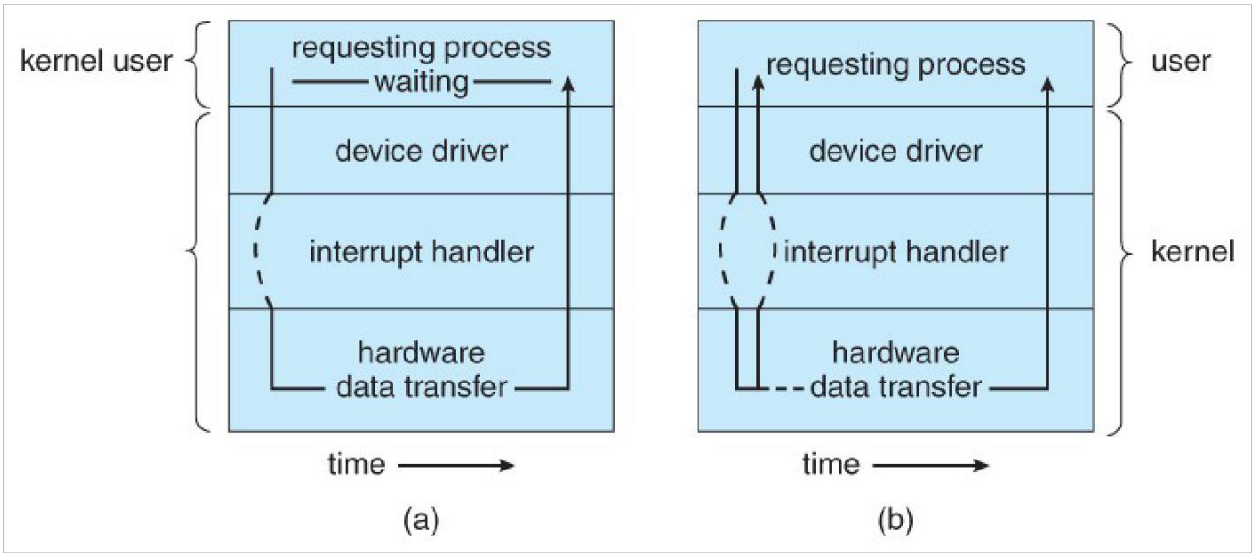
With ***blocking I/O*** a process is moved to the wait queue when an I/O request is made, and moved back to the ready queue when the request completes, allowing other processes to run in the meantime.

With ***non-blocking I/O*** the I/O request returns immediately, whether the requested I/O operation has (completely) occurred or not. This allows the process to check for available data without getting hung completely if it is not there.

One approach for programmers to implement non-blocking I/O is to have a multi-threaded application, in which one thread makes blocking I/O calls (say to read a keyboard or mouse), while other threads continue to update the screen or perform other tasks.

A subtle variation of the non-blocking I/O is the ***asynchronous I/O,*** in which the I/O request returns immediately allowing the process to continue on with other tasks, and then the process is notified (via changing a process variable, or a software interrupt, or a callback function) when the I/O operation has

completed and the data is available for use. (The regular non-blocking I/O returns immediately with whatever results are available, but does not complete the operation and notify the process later.)



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

Kernel I/O subsystem:

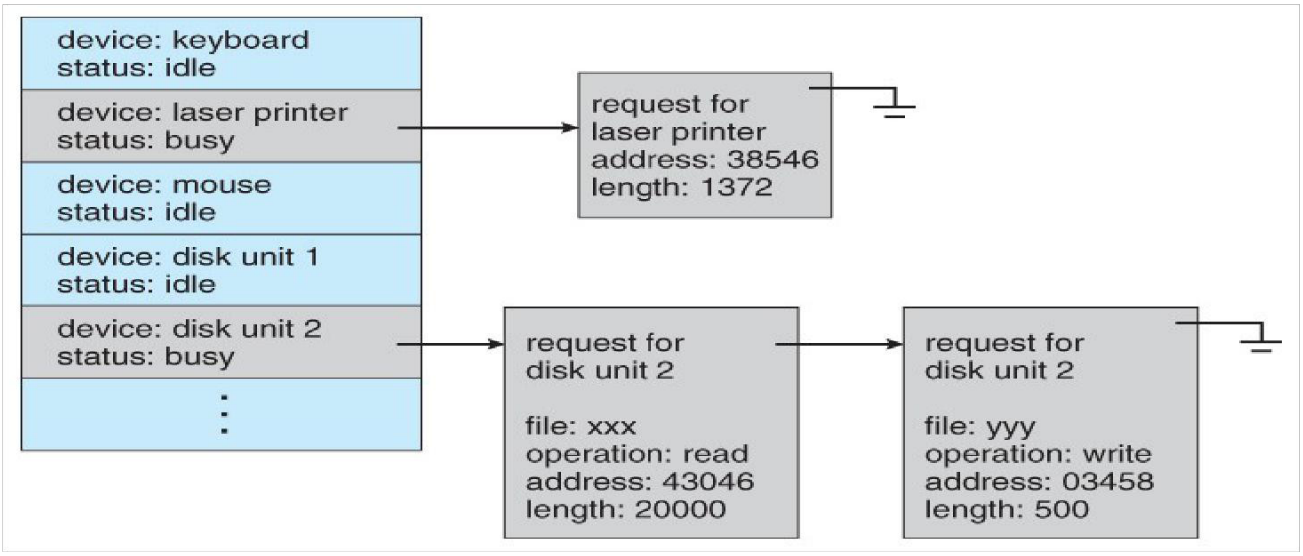
I/O Scheduling

Scheduling I/O requests can greatly improve overall efficiency. Priorities can also play a part in request scheduling.

The classic example is the scheduling of disk accesses, as discussed in detail.

Buffering and caching can also help, and can allow for more flexible scheduling options.

On systems with many devices, separate request queues are often kept for each device:



Device-status table.

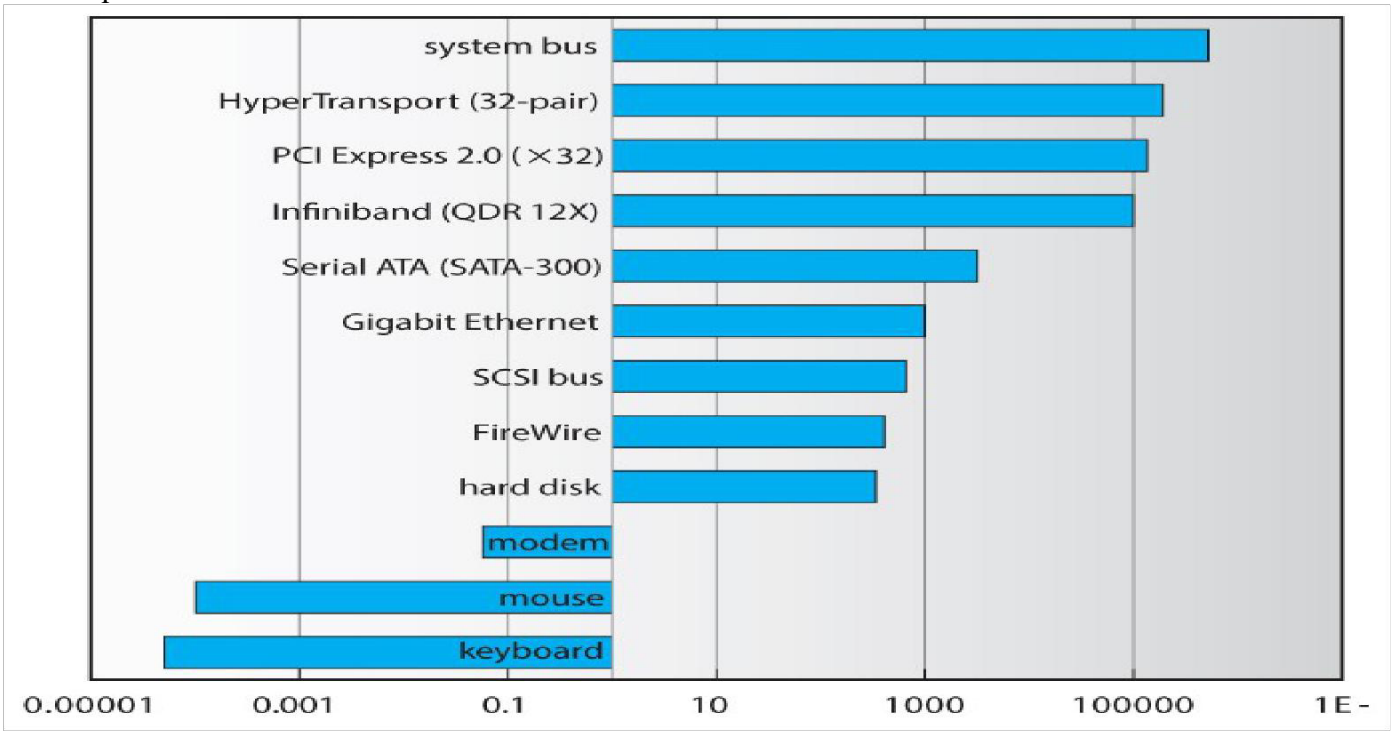
Buffering

Buffering of I/O is performed for (at least) 3 major reasons:

Speed differences between two devices. (See Figure below.) A slow device may write data into a buffer, and when the buffer is full, the entire buffer is sent to the fast device all at once. So that the slow device still has somewhere to write while this is going on, a second buffer is used, and the two buffers alternate as each becomes full. This is known as *double buffering.* (Double buffering is often used in (animated) graphics, so that one screen image can be generated in a buffer while the other (completed) buffer is displayed on the screen. This prevents the user from ever seeing any half-finished screen images.)

Data transfer size differences. Buffers are used in particular in networking systems to break messages up into smaller packets for transfer, and then for re-assembly at the receiving side.

To support ***copy semantics***. For example, when an application makes a request for a disk write, the data is copied from the user's memory area into a kernel buffer. Now the application can change their copy of the data, but the data which eventually gets written out to disk is the version of the data at the time the write request was made.



Sun Enterprise 6000 device-transfer rates (logarithmic).

Caching

Caching involves keeping a copy of data in a faster-access location than where the data is normally stored.

Buffering and caching are very similar, except that a buffer may hold the only copy of a given data item, whereas a cache is just a duplicate copy of some other data stored elsewhere.

Buffering and caching go hand-in-hand, and often the same storage space may be used for both purposes.

For example, after a buffer is written to disk, then the copy in memory can be used as a cached copy, (until that buffer is needed for other purposes)

Spooling and Device Reservation

A *spool (Simultaneous Peripheral Operations On-Line)* buffers data for (peripheral) devices such as printers that cannot support interleaved data streams.

If multiple processes want to print at the same time, they each send their print data to files stored in the spool directory. When each file is closed, then the application sees that print job as complete, and the print scheduler sends each file to the appropriate printer one at a time.

Support is provided for viewing the spool queues, removing jobs from the queues, moving jobs from one queue to another queue, and in some cases changing the priorities of jobs in the queues.

Spool queues can be general (any laser printer) or specific (printer number 42.)

Error Handling

I/O requests can fail for many reasons, either transient (buffers overflow) or permanent (disk crash). I/O requests usually return an error bit (or more) indicating the problem. UNIX systems also set the global variable *errno* to one of a hundred or so well-defined values to indicate the specific error that has occurred. (Seeerrno.h for a complete listing, or man errno.)

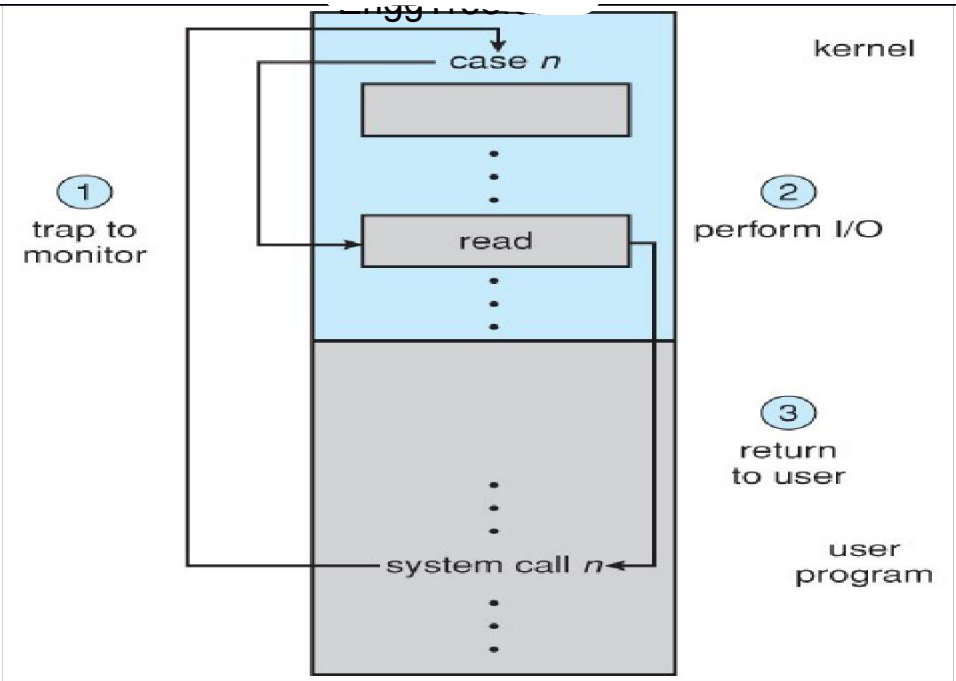
Some devices, such as SCSI devices, are capable of providing much more detailed information about errors, and even keep an on-board error log that can be requested by the host.

I/O Protection

The I/O system must protect against either accidental or deliberate erroneous I/O.

User applications are not allowed to perform I/O in user mode - All I/O requests are handled through system calls that must be performed in kernel mode.

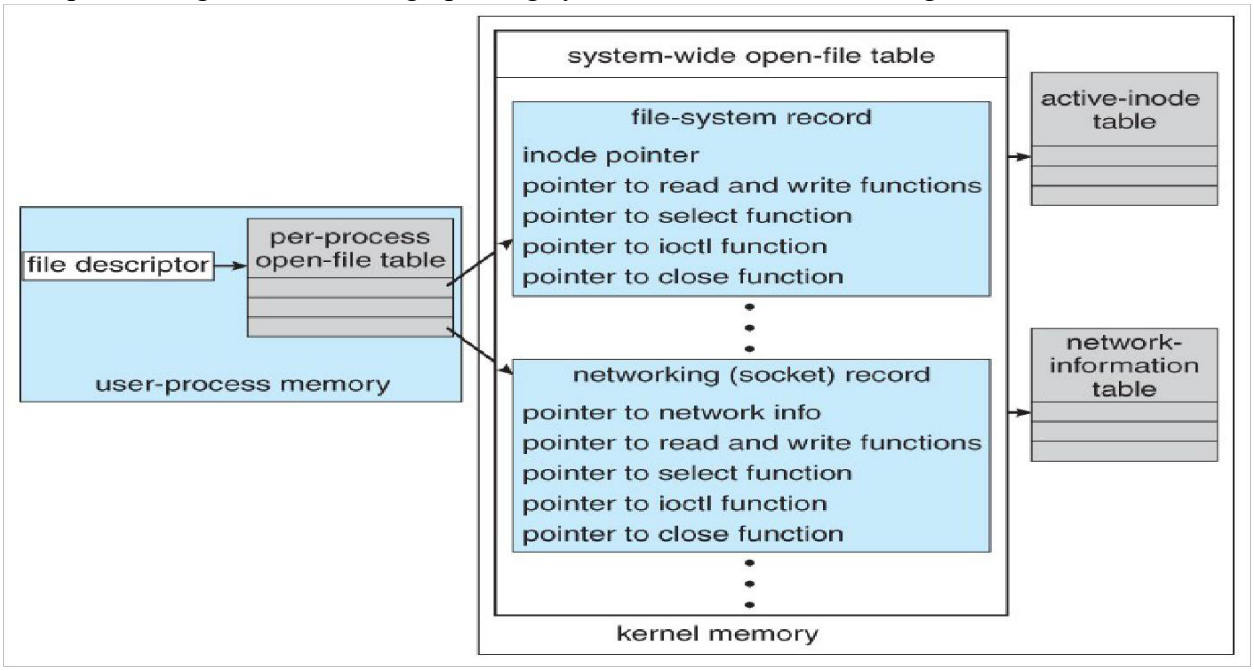
Memory mapped areas and I/O ports must be protected by the memory management system, but access to these areas cannot be totally denied to user programs. (Video games and some other applications need to be able to write directly to video memory for optimal performance for example.) Instead the memory protection system restricts access so that only one process at a time can access particular parts of memory, such as the portion of the screen memory corresponding to a particular window.



Use of a system call to perform I/O.

Kernel Data Structures

The kernel maintains a number of important data structures pertaining to the I/O system, such as the open file table. These structures are object-oriented, and flexible to allow access to a wide variety of I/O devices through a common interface. (See Figure below.)Windows NT carries the object-orientation one step further, implementing I/O as a message-passing system from the source through various intermediaries to the device.



UNIX I/O kernel structure.

Streams

The ***streams*** mechanism in UNIX provides a bi-directional pipeline between a user process and a device driver, onto which additional modules can be added.

The user process interacts with the ***stream head.***

The device driver interacts with the ***device end.***

Zero or more ***stream modules*** can be pushed onto the stream, using ioctl( ). These modules may filter and/or modify the data as it passes through the stream.

Each module has a ***read queue*** and a ***write queue.***

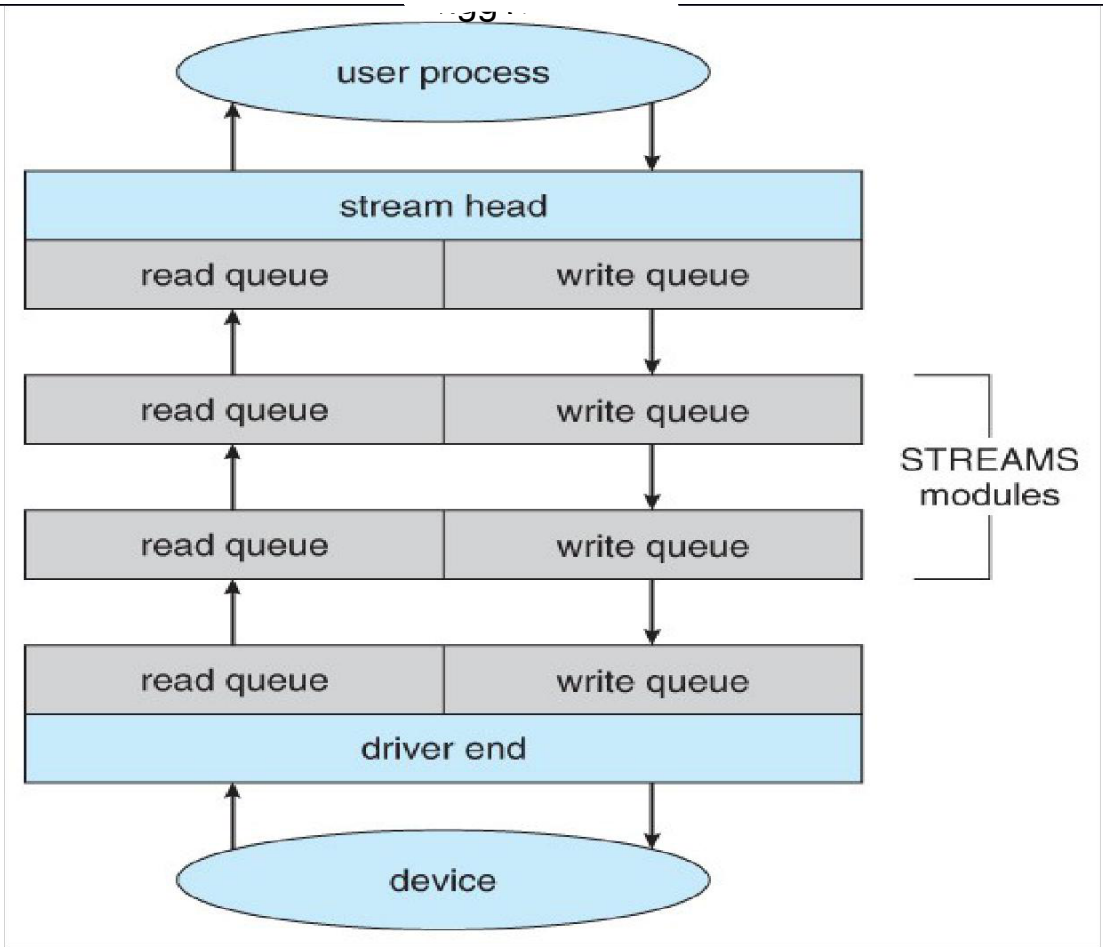
Flow control can be optionally supported, in which case each module will buffer data until the adjacent module is ready to receive it. Without flow control, data is passed along as soon as it is ready.

User processes communicate with the stream head using either read( ) and write( ) (or putmsg( ) and getmsg( ) for message passing.)

Streams I/O is asynchronous (non-blocking), except for the interface between the user process and the stream head.

The device driver **must** respond to interrupts from its device - If the adjacent module is not prepared to accept data and the device driver's buffers are all full, and then data is typically dropped.

Streams are widely used in UNIX, and are the preferred approach for device drivers. For example, UNIX implements sockets using streams.



The SREAMS structure.

Performance:

The I/O system is a major factor in overall system performance, and can place heavy loads on other major components of the system (interrupt handling, process switching, memory access, bus contention, and CPU load for device drivers just to name a few.)

Interrupt handling can be relatively expensive (slow), which causes programmed I/O to be faster than interrupt-driven I/O when the time spent busy waiting is not excessive.

Network traffic can also put a heavy load on the system. Consider for example the sequence of events that occur when a single character is typed in a telnet session, as shown in figure (And the fact that a similar set of events must happen in reverse to echo back the character that was typed) Sun uses in-kernel threads for the telnet daemon, increasing the supportable number of simultaneous telnet sessions from the hundreds to the thousands.

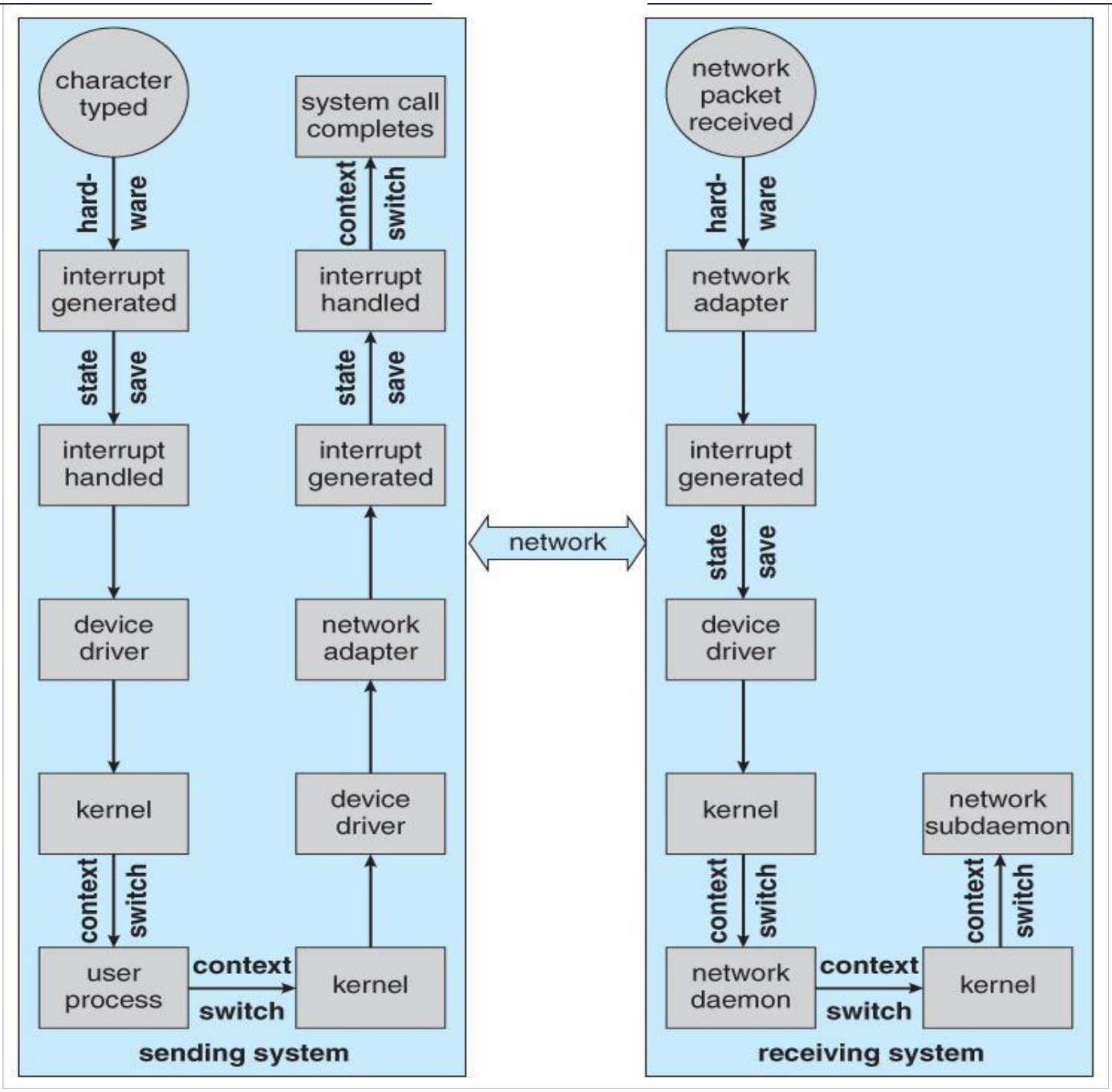


Figure Intercomputer communications.

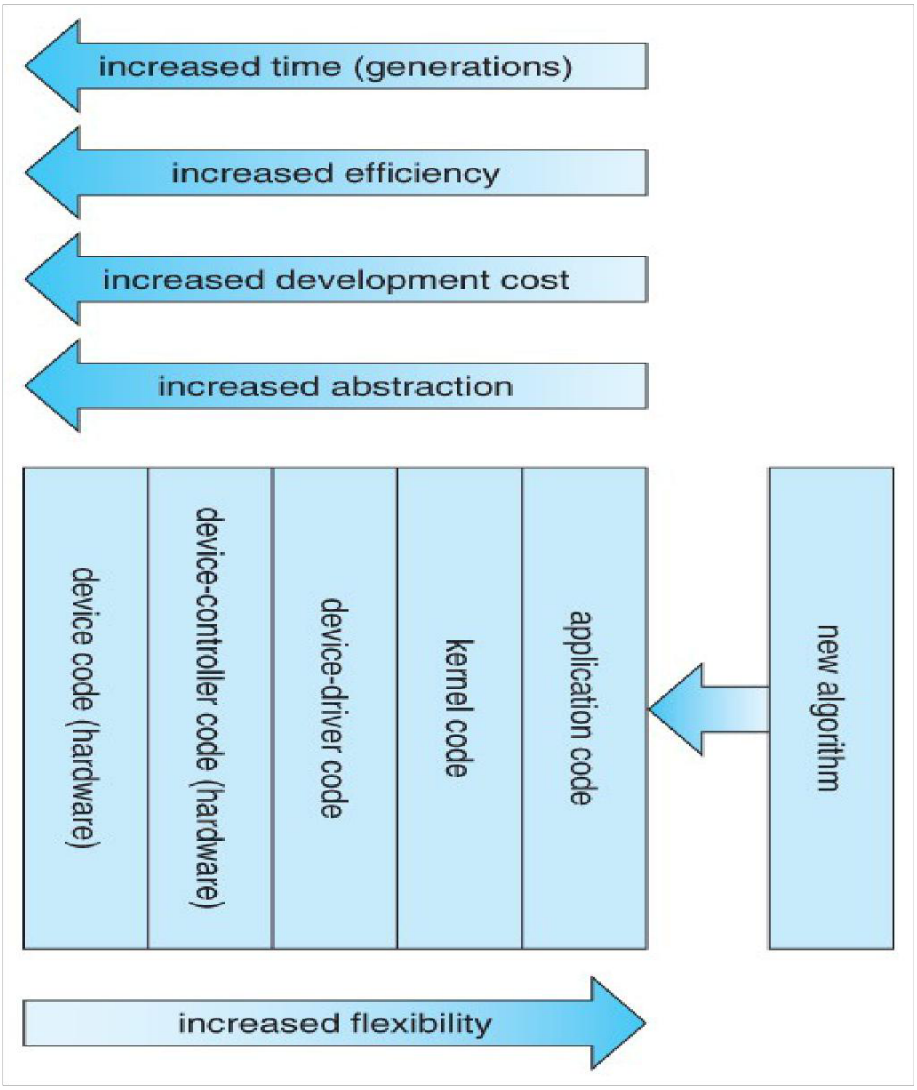
Other systems use ***front-end processors*** to off-load some of the work of I/O processing from the CPU. For example a ***terminal concentrator*** can multiplex with hundreds of terminals on a single port on a large computer.

Several principles can be employed to increase the overall efficiency of I/O processing:

1. Reduce the number of context switches.
2. Reduce the number of times data must be copied.

3. Reduce interrupt frequency, using large transfers, buffering, and polling where appropriate.
4. Increase concurrency using DMA.
5. Move processing primitives into hardware, allowing their operation to be concurrent with CPU and bus operations.
6. Balance CPU, memory, bus, and I/O operations, so a bottleneck in one does not idle all the others.

The development of new I/O algorithms often follows a progression from application level code to on-board hardware implementation, as shown in Figure 13.16. Lower-level implementations are faster and more efficient, but higher-level ones are more flexible and easier to modify. Hardware-level functionality may also be harder for higher-level authorities (e.g. the kernel) to control.



Device functionality progression.