CS8493-OPERATING SYSTEMS
UNIT-I COMPUTER SYSTEM OVERVIEW

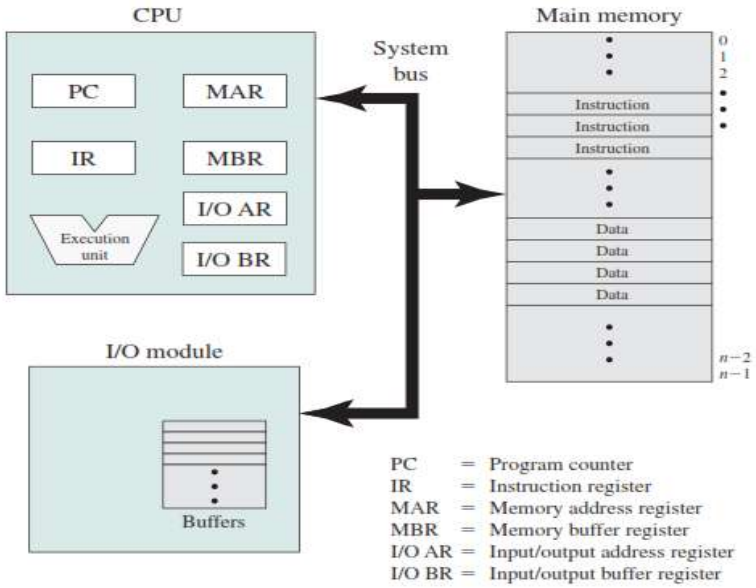
Computer System Overview-Basic Elements, Instruction Execution, Interrupts, Memory Hierarchy, Cache Memory, Direct Memory Access, Multiprocessor and Multicore Organization. Operating system overview-objectives and functions, Evolution of Operating System.- Computer System Organization-Operating System Structure and Operations- System Calls, System Programs, OS Generation and System Boot.

BASIC ELEMENTS OF A COMPUTER.

➔At a top level, a computer consists of processor, memory, and I/O components, with one or more modules of each type. These components are interconnected in some fashion to achieve the main function of the computer, which is to execute programs.

Thus, there are four main structural elements:

- **Processor:** Controls the operation of the computer and performs its data processing functions. When there is only one processor, it is often referred to as the central processing unit (CPU).
- **Main memory:** Stores data and programs. This memory is typically volatile; that is, when the computer is shut down, the contents of the memory are lost. In contrast, the contents of disk memory are retained even when the computer system is shut down. Main memory is also referred to as real memory or primary memory.
- **I/O modules:** Move data between the computer and its external environment. The external environment consists of a variety of devices, including secondary memory devices (e.g., disks), communications equipment, and terminals.
- **System bus:** Provides for communication among processors, main memory, and I/O modules.



Computer Components: Top-Level View

➔The figure depicts these top - level components. One of the processor's functions is to exchange data with memory. For this purpose, it typically makes use of two internal (to the processor) registers: a memory address register (MAR), which specifies the address in memory for the next read or write; and a memory buffer register (MBR), which contains the data to be written into memory or which receives the data read from memory.

➔Similarly, an I/O address register (I/OAR) specifies a particular I/O device. An I/O buffer register (I/OBR) is used for the exchange of data between an I/O module and the processor.

➔A memory module consists of a set of locations, defined by sequentially numbered addresses. Each location contains a bit pattern that can be interpreted as either an instruction or data. An I/O module transfers data from external devices to processor and memory, and vice versa. It contains internal buffers for temporarily holding data until they can be sent on.

INSTRUCTION EXECUTION WITH INSTRUCTION EXECUTION CYCLE.

➔A program to be executed by a processor consists of a set of instructions stored in memory. In its simplest form, instruction processing consists of two steps:

➔The processor reads (fetches) instructions from memory one at a time and executes each instruction. Program execution consists of repeating the process of instruction fetch and instruction execution.

➔The processing required for a single instruction is called an instruction cycle. Using a simplified two-step description, the instruction cycle is depicted in Figure.

The two steps are referred to as the

(i)Fetch stage (ii)Execution stage.

➔Program execution halts only if the processor is turned off, some sort of unrecoverable error occurs, or a program instruction that halts the processor is encountered.

➔The program counter (PC) holds the address of the next instruction to be fetched. Unless instructed otherwise, the processor always increments the PC after each instruction fetch so that it will fetch the next instruction in sequence.

➔For example, consider a simplified computer in which each instruction occupies one 16-bit word of memory. Assume that the program counter is set to location 300. The processor will next fetch the instruction at location 300. On succeeding instruction cycles, it will fetch instructions from locations 301, 302, 303, and so on. This sequence may be altered, as explained subsequently.

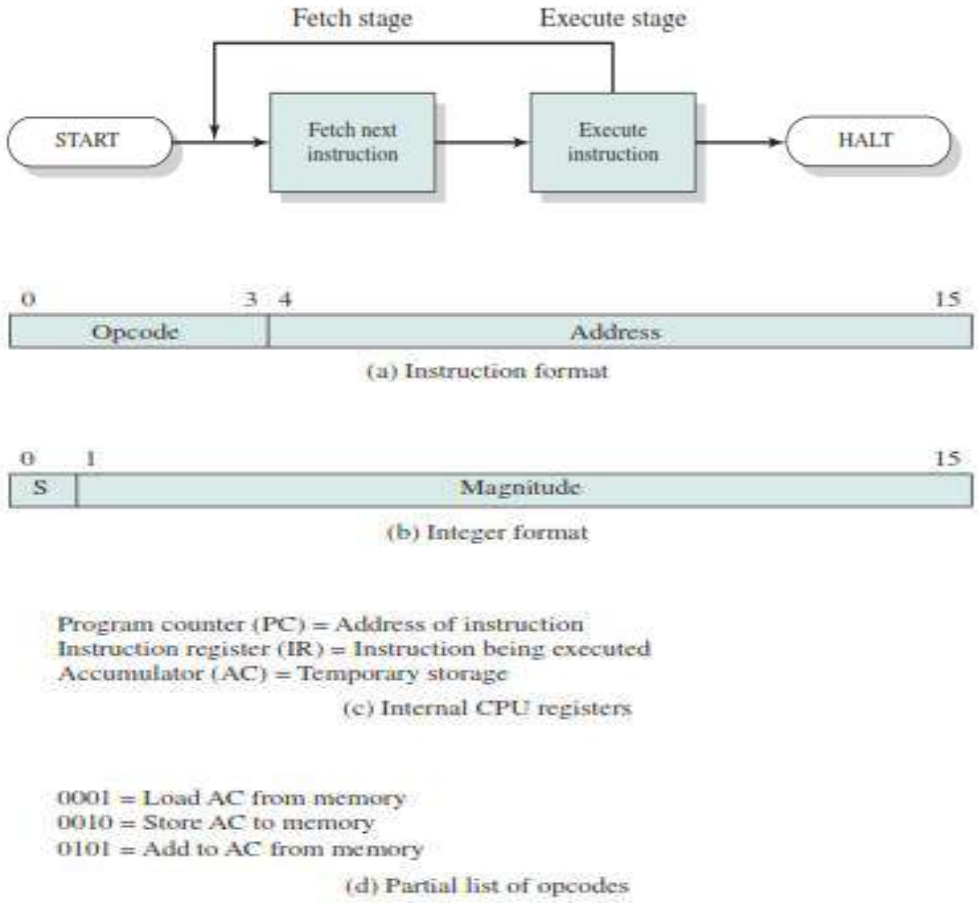
➔The fetched instruction is loaded into the instruction register (IR). The instruction contains bits that specify the action the processor is to take. The processor interprets the instruction and performs the required action.

➔In general, these actions fall into four categories:

- **Processor-memory**: Data may be transferred from processor to memory or from memory to processor.
- **Processor-I/O**: Data may be transferred to or from a peripheral device by transferring between the processor and an I/O module.
- **Data processing**: The processor may perform some arithmetic or logic operation on data.
- **Control:** An instruction may specify that the sequence of execution be altered. For example, the processor may fetch an instruction from location 149, which specifies that the next instruction will be from location 182. The processor sets the program counter to

182. Thus, on the next fetch stage, the instruction will be fetched from location 182 rather than 150.

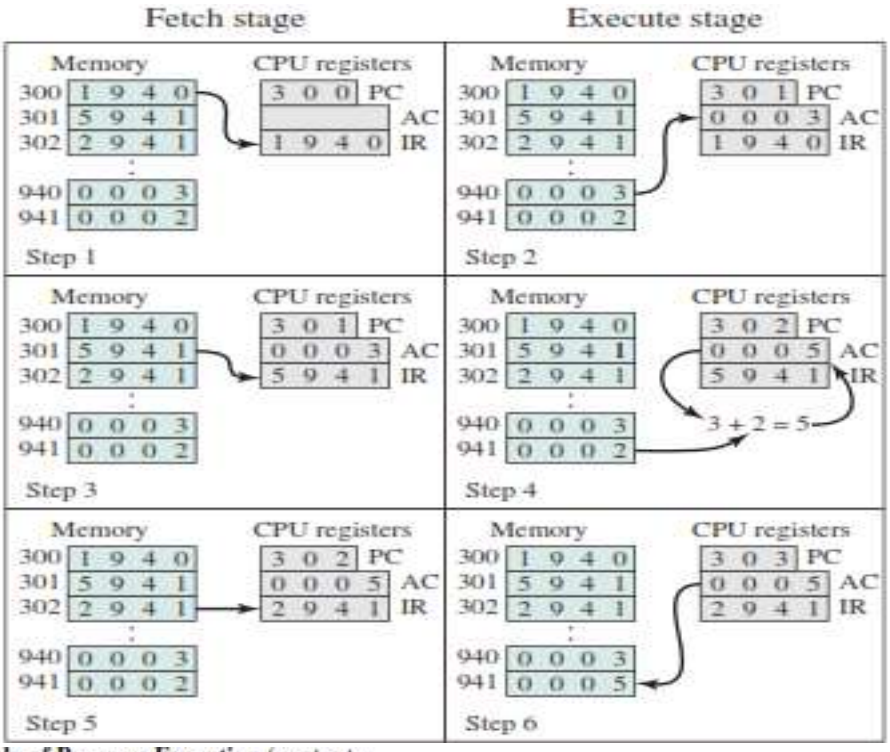
Basic Instruction Cycle



➔Figure shows a partial program execution, showing the relevant portions of memory and processor registers. The program fragment shown adds the contents of the memory word at address 940 to the contents of the memory word at address 941 and stores the result in the latter location.

Three instructions, which can be described as three fetch and three execute stages, are required:

1. The PC contains 300, the address of the first instruction. This instruction (the value 1940 in hexadecimal) is loaded into the IR and the PC is incremented. Note that this process involves the use of a memory address register (MAR) and a memory buffer register (MBR). For simplicity, these intermediate registers are not shown.
2. The first 4 bits (first hexadecimal digit) in the IR indicate that the AC is to be loaded from memory. The remaining 12 bits (three hexadecimal digits) specify the address, which is 940.



Example of Program Execution (contents of memory and registers in hexadecimal)

3. The next instruction (5941) is fetched from location 301 and the PC is incremented.
4. The old contents of the AC and the contents of location 941 are added and the result is stored in the AC.
5. The next instruction (2941) is fetched from location 302 and the PC is incremented.
6. The contents of the AC are stored in location 941.

In this example, three instruction cycles, each consisting of a fetch stage and an execute stage, are needed to add the contents of location 940 to the contents of 941. With a more complex set of instructions, fewer instruction cycles would be needed.

INTERRUPT PROCESSING.

➔Virtually all computers provide a mechanism by which other modules (I/O , memory) may interrupt the normal sequence of the processor. Interrupts are provided primarily as a way to improve processor utilization.

Four Classes of Interrupts are

1. **Program** Generated by some condition that occurs as a result of an instruction execution, such as arithmetic overflow, division by zero, attempt to execute an illegal machine instruction, and reference outside a user's allowed memory space.
2. **Timer** Generated by a timer within the processor. This allows the operating system to perform certain functions on a regular basis.
3. **I/O** Generated by an I/O controller, to signal normal completion of an operation or to signal a variety of error conditions.
4. **Hardware failure** Generated by a failure, such as power failure or memory parity error.

➔The user program performs a series of WRITE calls interleaved with processing. The solid vertical lines represent segments of code in a program. Code segments 1, 2, and 3 refer to sequences of instructions that do not involve I/O. The WRITE calls are to an I/O routine that is a system utility and that will perform the actual I/O operation.

The I/O program consists of three sections:

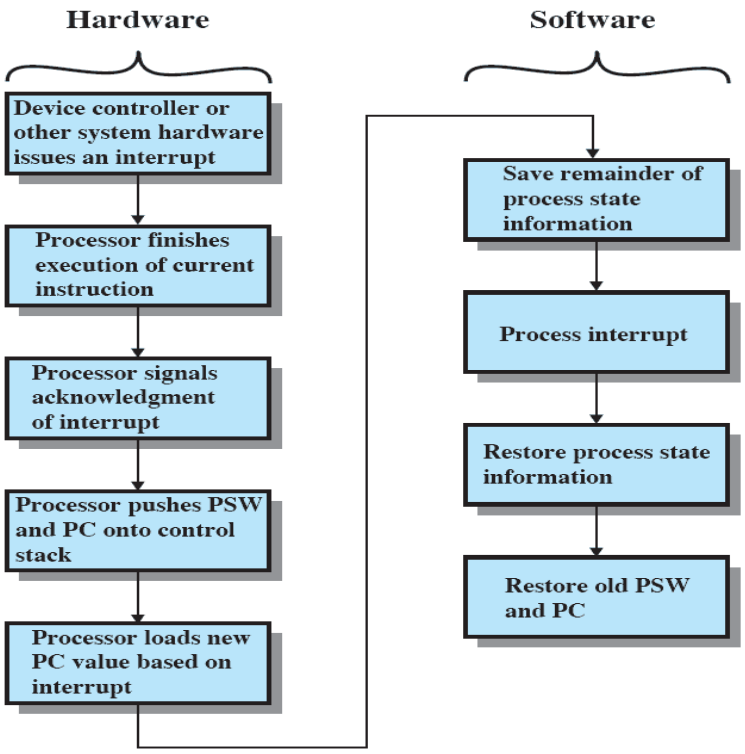
• A sequence of instructions, labeled 4 in the figure, to prepare for the actual I/O operation. This may include copying the data to be output into a special buffer and preparing the parameters for a device command.

• The actual I/O command. Without the use of interrupts, once this command is issued, the program must wait for the I/O device to perform the requested function (or periodically check the status, or poll, the I/O device). The program might wait by simply repeatedly performing a test operation to determine if the I/O operation is done.

• A sequence of instructions, labeled 5 in the figure, to complete the operation. This may include setting a flag indicating the success or failure of the operation.

INTERRUPT PROCESSING

The following gives the detailed interrupt processing procedure:



➔An interrupt triggers a number of events, both in the processor hardware and in software.

This figure shows a typical sequence. When an I/O device completes an I/O operation, the following sequence of hardware events occurs:

1. The device issues an interrupt signal to the processor.
2. The processor finishes execution of the current instruction before responding to the interrupt.
3. The processor tests for a pending interrupt request, determines that there is one, and sends an acknowledgment signal to the device that issued the interrupt. The acknowledgment allows the device to remove its interrupt signal.
4. The processor next needs to prepare to transfer control to the interrupt routine.
5. The processor then loads the program counter with the entry location of the interrupt- handling routine that will respond to this interrupt.
6. At this point, the program counter and PSW relating to the interrupted program have been saved on the control stack.
7. The interrupt handler may now proceed to process the interrupt.
8. The saved register values are retrieved from the stack and restored to the registers
9. The final act is to restore the PSW and program counter values from the stack.
It is important to save all of the state information about the interrupted program for later resumption.

Because the interrupt is not a routine called from the program.
Rather, the interrupt can occur at any time and therefore at any point in the execution of a user program.
Its occurrence is unpredictable.

MULTIPLE INTERRUPTS

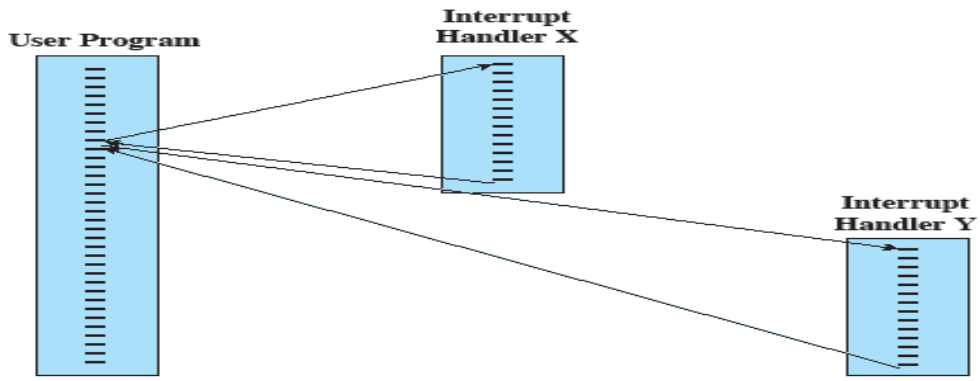
➔The above only discussed the case in which a single interrupt happens. Actually, in a computer system, there are multiple interrupt signal sources, so more than one interrupt requests may happen at the same time or during a same period.

➔The typical two approaches are: sequential interrupt processing - by disabling interrupt request while an interrupt is being processed, all interrupts will be processed sequentially (usually PSW contains a bit for this purpose); nested interrupt processing - all the interrupts may be assigned different priorities, so that whenever an interrupt occurs while an interrupt handler is running, their priorities will be compared first, and the further action will be determined according to the result. These two approaches are illustrated by the following figures:

a) Sequential Interrupt Processing

➔Two approaches can be taken to dealing with multiple interrupts. The first is to disable interrupts while an interrupt is being processed. A *disabled interrupt* simply means that the processor ignores any new interrupt request signal. If an interrupt occurs during this time, it generally remains pending and will be checked by the processor after the processor has re-enabled interrupts.

Thus if an interrupt occurs when a user program is executing, then interrupts are disabled immediately. After the interrupt-handler routine completes, interrupts are re-enabled before resuming the user program and the processor checks to see if additional interrupts have occurred. This approach is simple, as interrupts are handled in strict sequential order

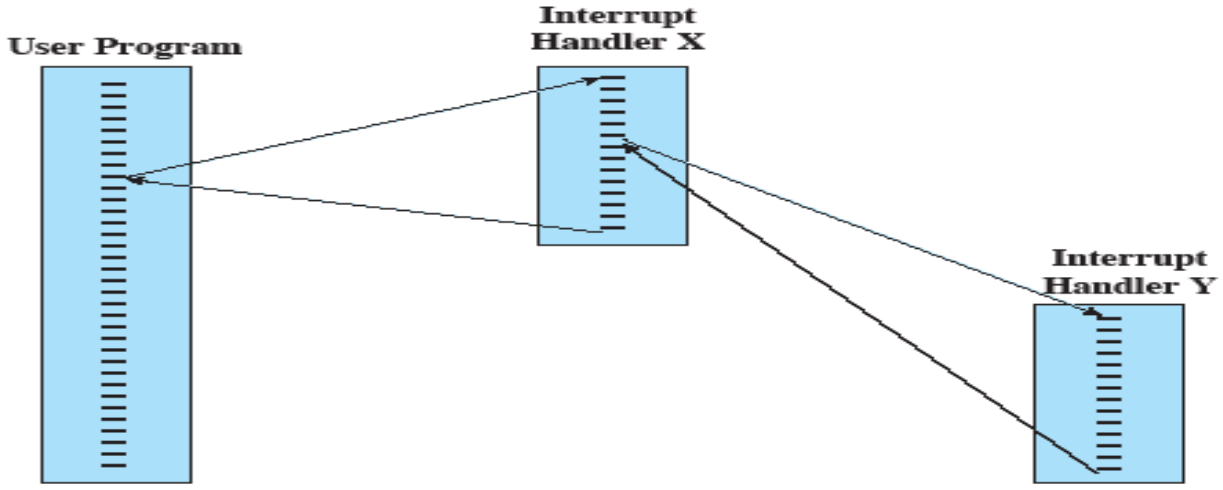


(a) Sequential interrupt processing

The drawback of sequential approach is that it does not take into account relative priority or time-critical needs.

b) Nested Interrupt Processing

A second approach is to define priorities for interrupts and to allow an interrupt of higher priority to cause a lower-priority interrupt handler to be interrupted.



(b) Nested interrupt processing

As an example of this second approach, consider a system with three I/O devices:

- a printer (priority 2),
- a disk (priority 4), and
- a communications line (priority 5).

This figure illustrates a possible sequence.

1. A user program begins at *t*= 0.
2. At *t*=10, a printer interrupt occurs;

   user information is placed on the control stack and execution continues at the printer interrupt service routine (ISR).

3. While this routine is still executing, at t =15 a communications interrupt occurs.

   Because the communications line has higher priority than the printer, the interrupt request is honored.

4. The printer ISR is interrupted, its state is pushed onto the stack, and execution continues at the communications ISR.

5. While this routine is executing, a disk interrupt occurs (t=20).

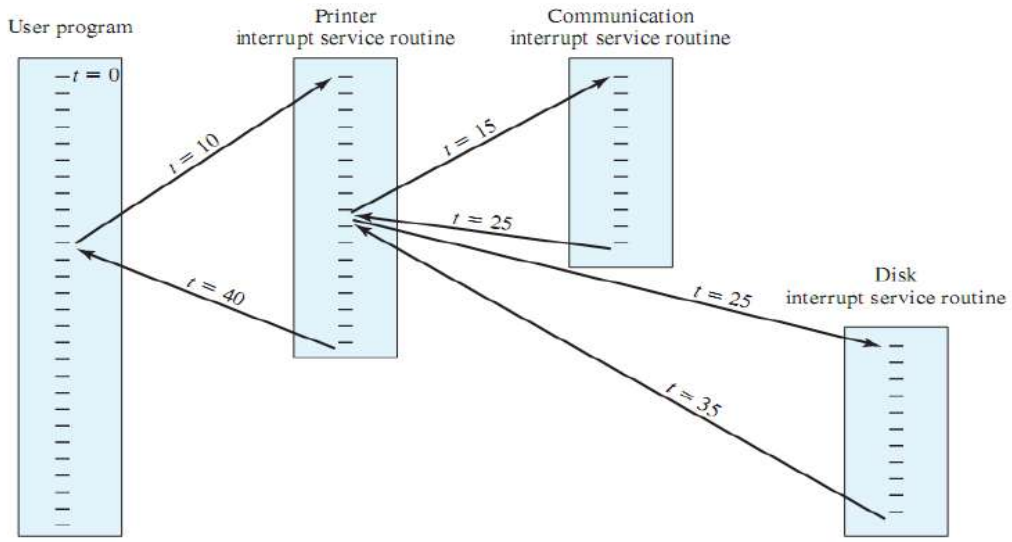
   Because this interrupt is of lower priority, it is simply held, and the communications ISR runs to completion.

6. When the communications ISR is complete (t=25), the previous processor state is restored, which is the execution of the printer ISR.

7. However, before even a single instruction in that routine can be executed, the processor honors the higher-priority disk interrupt and transfers control to the disk ISR.

8. Only when that routine is complete (t=35) is the printer ISR resumed.

9. When that routine completes (t=40), control finally returns to the user program.



MEMORY HIERARCHY.

Memory Hierarchy

➔The memory unit is an essential component in any digital computer since it is needed for storing programs and data.

➔Not all accumulated information is needed by the CPU at the same time. Therefore, it is more economical to use low-cost storage devices to serve as a backup for storing the information that is not currently used by CPU.

➔Computer Memory Hierarchy is a pyramid structure that is commonly used to illustrate the significant differences among memory types.

➔The memory unit that directly communicates with CPU is called the main memory. Devices that provide backup storage is called auxiliary memory.

➔The memory hierarchy system consists of all storage devices employed in a computer system from the slow by high-capacity auxiliary memory to a relatively faster main memory, to an even smaller and faster cache memory

Performance

Access time —Time between presenting the address and getting the valid data

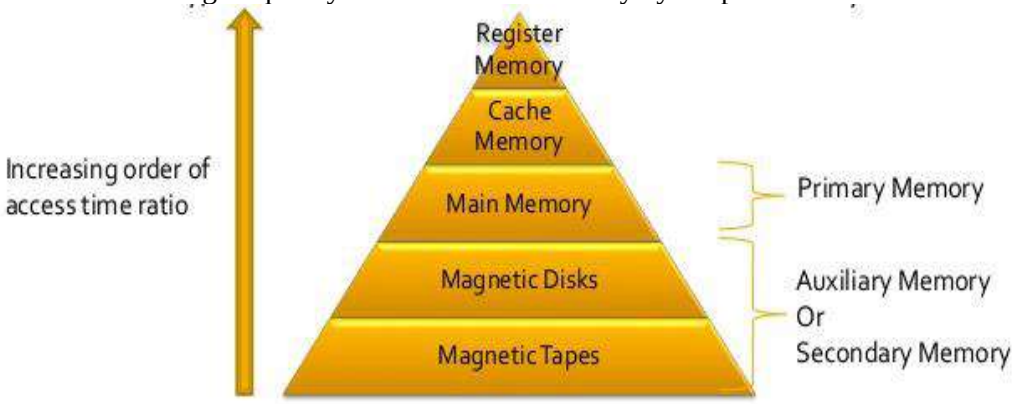
Memory Cycle time —Time may be required for the memory to "recover" before next access

—Cycle time is access + recovery

Transfer Rate —Rate at which data can be moved

Going down the hierarchy

– Decreasing cost per bit
– Increasing capacity
– Increasing access time
– Decreasing frequency of access to the memory by the processor



Main Memory

➔Most of the main memory in a general purpose computer is made up of RAM integrated circuits chips, but a portion of the memory may be constructed with ROM chips

1. RAM– Random Access memory
2. ROM– Read Only memory

RAM

A RAM chip is better suited for communication with the CPU if it has one or more control inputs that select the chip when needed.

Key features

RAM is packaged as a chip.
Basic storage unit is a cell (one bit per cell).
Multiple RAM chips form a memory.

Static RAM (SRAM)

Each cell stores bit with a six-transistor circuit.
Retains value indefinitely, as long as it is kept powered.
Relatively insensitive to disturbances such as electrical noise.
Faster and more expensive than DRAM.

Dynamic RAM (DRAM)

Each cell stores bit with a capacitor and transistor.
Value must be refreshed every 10-100 ms.

Sensitive to disturbances.
Slower and cheaper than SRAM.

ROM

➔ROM is used for storing programs that are **PERMENTLY** resident in the computer and for tables of constants that do not change in value once the production of the computer is completed.

➔The ROM portion of main memory is needed for storing an initial program called *bootstrap loader*, which is to start the computer software operating when power is turned off.

➔Data is programmed into the chip using an external ROM programmer
The programmed chip is used as a component into the circuit
The circuit doesn't change the content of the ROM

Auxiliary Memory

➔Auxiliary memory, also known as auxiliary storage, secondary storage, secondary memory or external memory, is a non-volatile memory (does not lose stored data when the device is powered down) that is not directly accessible by the CPU, because it is not accessed via the input/output channels (it is an external device).

➔Some examples of auxiliary memory would be disks, external hard drives, USB drives, etc.

Cache Memory

➔Cache memory, also called CPU memory, is random access memory (RAM) that a computer microprocessor can access more quickly than it can access regular RAM. This memory is typically integrated directly with the CPU chip or placed on a separate chip that has a separate bus interconnect with the CPU.

➔The basic purpose of cache memory is to store program instructions that are frequently re- referenced by software during operation. Fast access to these instructions increases the overall speed of the software program.

➔As the microprocessor processes data, it looks first in the cache memory; if it finds the instructions there (from a previous reading of data), it does not have to do a more time-consuming reading of data from larger memory or other data storage devices.

Tertiary Storage

➔Tertiary Storage, also known as tertiary memory, consists of anywhere from one to several storage drives. It is a comprehensive computer storage system that is usually very slow, so it is usually used to archive data that is not accessed frequently. A computer can access tertiary storage without being told to do so, which is unlike off-line storage.

➔This type of computer storage device is not as popular as the other two storage device types. Its main use is for storing data at a very large-scale. This includes optical jukeboxes and tape libraries. Tertiary storage devices require a database to organize the data that are stored in them, and the computer needs to go through the database to access those data.

Memory hierarchy is just like the real world situation where -

1. A train fare is cheaper and it can carry a lot people at a time but it takes long time
2. The air fare of professional flights is more than the train, it can carry lesser number of people but it is much faster than the train
3. The air fare for personal jet is further high, it can carry further lesser number of people but it is fastest of the three.

So, depending upon the price and the urgency to reach destination, you will use combination of these in different situations.

The memory hierarchy is exactly the same. Here, the situation is-

1. We need a lot of memory which is cheap and could be slow (secondary memory, Hard Disk)
2. We also need some memory which could be smaller than secondary memory but should be faster than it (primary memory, RAM)
3. We also need another kind of memory which could be smaller than the primary memory but it should be much faster than it (cache memory).

That's why we need memory hierarchy.

CACHE MEMORY

Concept

Small amount of fastest memory.

Sits between normal main memory and CPU.

May be located on CPU chip or module.

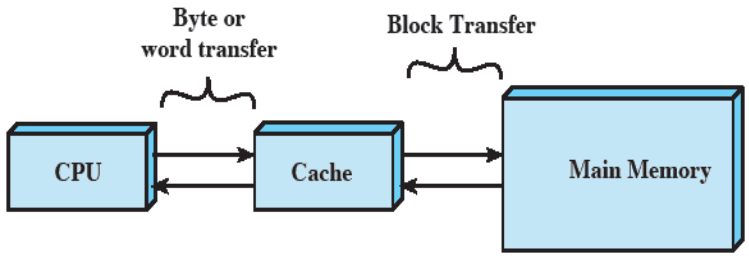
Cache Principles

Contains copy of a portion of main memory

Processor first checks cache

If desired data item not found, relevant block of memory read into cache

Because of locality of reference, it is likely that future memory references are in that block.



Cache Operation

CPU requests contents of memory location.

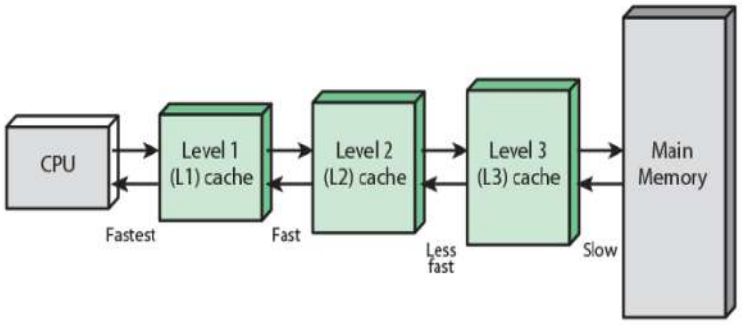
Check cache for this data.

If present, get from cache (fast).

If not present, read required block from main memory to cache.

Then deliver from cache to CPU.

Cache includes tags to identify which block of main memory is in each cache slot.



Three Level Cache Memory Hierarchy

The L3 cache is usually built onto the motherboard between the main memory (RAM) and the L1 and L2 caches of the processor module.

This serves as another bridge to park information like processor commands and frequently used data in order to prevent bottlenecks resulting from the fetching of these data from the main memory.

In short, the L3 cache of today is what the L2 cache was before it got built-in within the processor module itself.

The CPU checks for information it needs from L1 to the L3 cache. If it does not find this info in L1 it looks to L2 then to L3, the biggest yet slowest in the group.

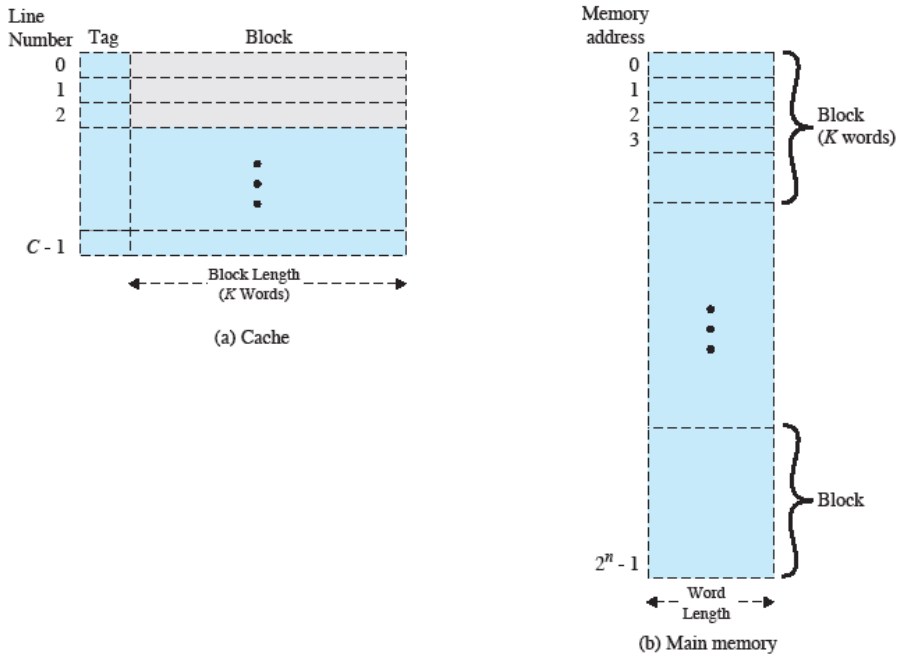
The purpose of the L3 differs depending on the design of the CPU. In some cases the L3 holds copies of instructions frequently used by multiple cores that share it.

Most modern CPUs have built-in L1 and L2 caches per core and share a single L3 cache on the motherboard, while other designs have the L3 on the CPU die itself.

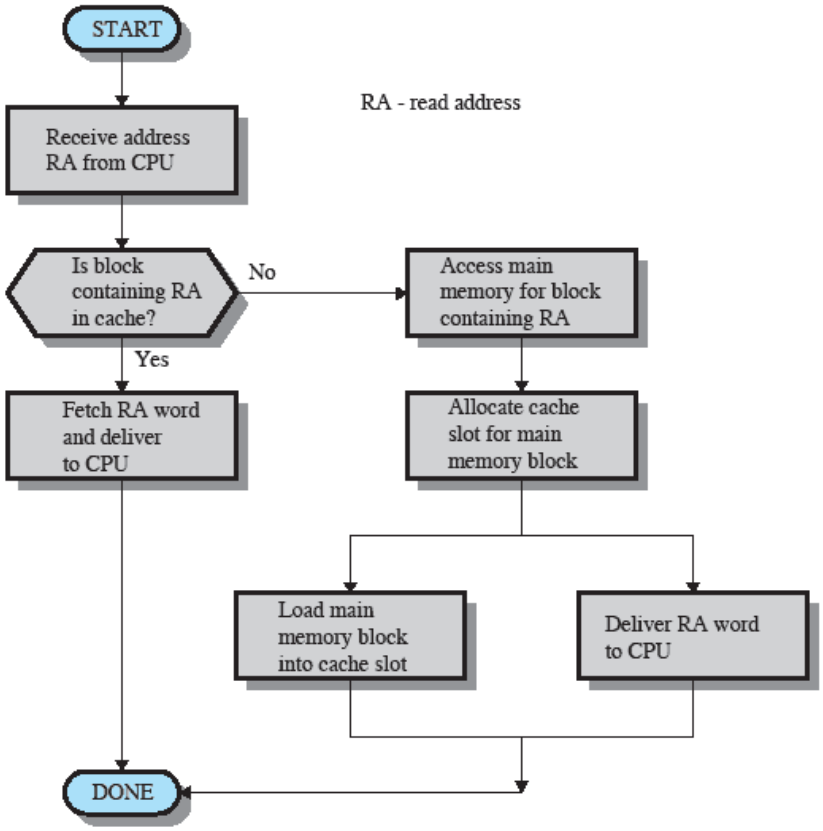
Cache Memory Structure

➔ N address lines => 2n words of memory
➔ Cache stores fixed length blocks of K words
➔ Cache views memory as an array of M blocks where M = 2n/K
➔ A block of memory in cache is referred to as a line. K is the line size
➔ Cache size of C blocks where C < M
(considerably)
➔ Each line includes a tag that identifies the block being stored
➔ Tag is usually upper portion of memory address

As a simple example, suppose that we have a 6-bit address and a 2-bit tag.
The tag 01 refers to the block of locations with the following addresses: 010000, 010001, 010010, 010011, 010100, 010101, 010110, 010111, 011000, 011001, 011010, 011011, 011100, 011101, 011110, and 011111.



Cache Read Operation



The processor generates the address, RA, of a word to be read. If the word is contained in the cache, it is delivered to the processor. Otherwise, the block containing that word is loaded into the cache and the word is delivered to the processor.

Cache Design

Elements of Cache Design

- Addresses (logical or physical)
- Size
- Mapping Function (direct, associative, set associative)
- Replacement Algorithm (LRU, LFU, FIFO, random)
- Write Policy (write through, write back, write once)
- Line Size
- Number of Caches (how many levels, unified or split)

Cache size

Even small caches have significant impact on performance

Block size

The unit of data exchanged between cache and main memory

Larger block size yields more hits until probability of using newly fetched data becomes less than the probability of reusing data that have to be moved out of cache.

Mapping function

Determines which cache location the block will occupy

Replacement algorithm

Chooses which block to replace

Least-recently-used (LRU) algorithm

Write policy

- Dictates when the memory write operation takes place
- Can occur every time the block is updated
- Can occur when the block is replaced

Minimize write operations

Leave main memory in an obsolete state

DIRECT MEMORY ACCESS (DMA)

➔Three techniques are possible for I/O operations: programmed I/O, interrupt-driven I/O, and direct memory access (DMA). Before discussing DMA, we briefly define the other two techniques; see Appendix C for more detail. When the processor is executing a program and encounters an instruction relating to I/O, it executes that instruction by issuing a command to the appropriate I/O module.

➔In the case of **programmed I/O** , the I/O module performs the requested action and then sets the appropriate bits in the I/O status register but takes no further action to alert the processor. In particular, it does not interrupt the processor. Thus, after the I/O instruction is invoked, the processor must take some active role in determining when the I/O instruction is completed. For this purpose, the processor periodically checks the status of the I/O module until it finds that the
Operation is complete.

➔With programmed I/O, the processor has to wait a long time for the I/O module of concern to be ready for either reception or transmission of more data. The processor, while waiting, must repeatedly interrogate the status of the I/O module.

➔As a result, the performance level of the entire system is severely degraded. An alternative, known as **interrupt-driven I/O** , is for the processor to issue an I/O command to a module and then go on to do some other useful work.

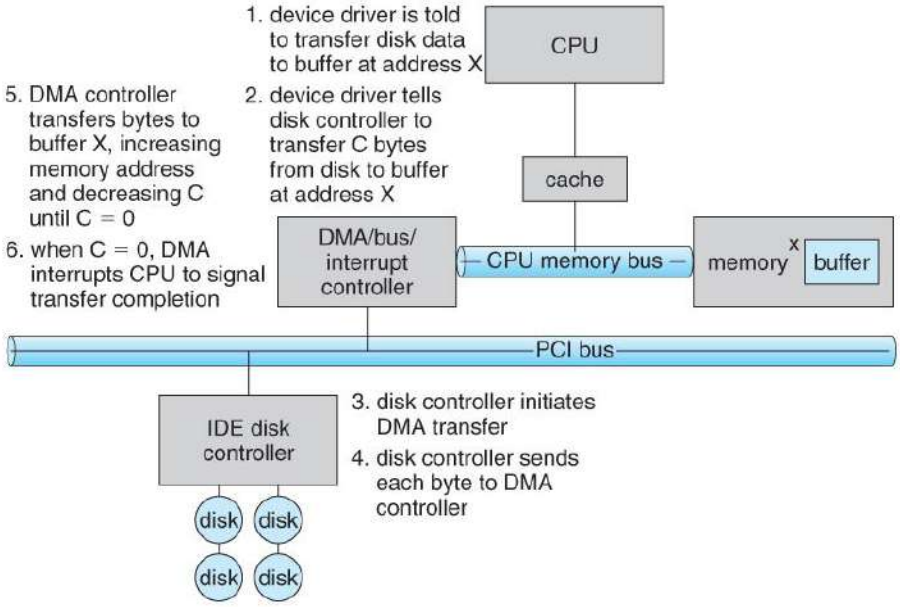
➔The I/O module will then interrupt the processor to request service when it is ready to exchange data with the processor. The processor then executes the data transfer, as before, and then resumes its former processing.

➔When large volumes of data are to be moved, a more efficient technique is required: **direct memory access (DMA).** The DMA function can be performed by a separate module on the system bus or it can be incorporated into an I/O module. In either case, the technique works as follows. When the processor wishes to read or write a block of data, it issues a command to the DMA module, by sending to the DMA module the following information:

- Whether a read or write is requested
- The address of the I/O device involved
- The starting location in memory to read data from or write data to
- The number of words to be read or written

➔The processor then continues with other work. It has delegated this I/O operation to the DMA module, and that module will take care of it. The DMA module transfers the entire block of data, one word at a time, directly to or from memory without going through the processor. When the transfer is complete, the DMA module sends an interrupt signal to the processor. Thus, the processor is involved only at the beginning and end of the transfer.

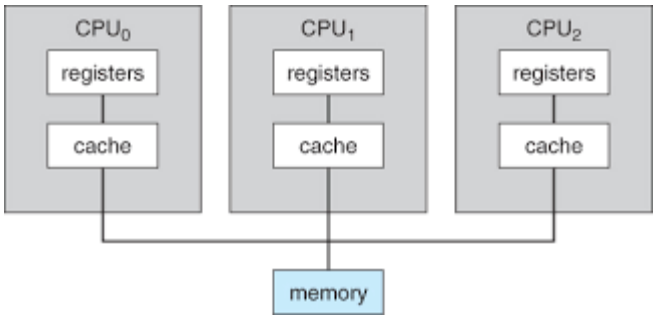
The Figure below illustrates the DMA process.



MULTIPROCESSOR AND MULTICORE ORGANIZATION

MULTIPROCESSING

➔Multiprocessing is the use of two or more central processing units (CPUs) within a single computer system. The term also refers to the ability of a system to support more than one processor and/or the ability to allocate tasks between them.



➔There are multiple processors, each of which contains its own control unit, arithmetic logic unit, and registers. Each processor has access to a shared main memory and the I/O devices through some form of interconnection mechanism; a shared bus is a common facility. The processors can communicate with each other through memory (messages and status information left in shared address spaces). It may also be possible for processors to exchange signals directly. The memory is often organized so that multiple simultaneous accesses to separate blocks of memory are possible.

➔Multiprocessor systems have three main advantages.

1. Increased throughput.
2. Economy of scale.
3. Increased reliability.

The most common multiple-processor systems now use **symmetric multiprocessing** (SMP), in which each processor runs an identical copy of the operating system, and these copies communicate with one another as needed.

Some systems use **asymmetric multiprocessing**, in which each processor is assigned a specific task. A master processor controls the system; the other processors either look to the master for instruction or have predefined tasks. This scheme defines a master-slave relationship. The master processor schedules and allocates work to the slave processors.

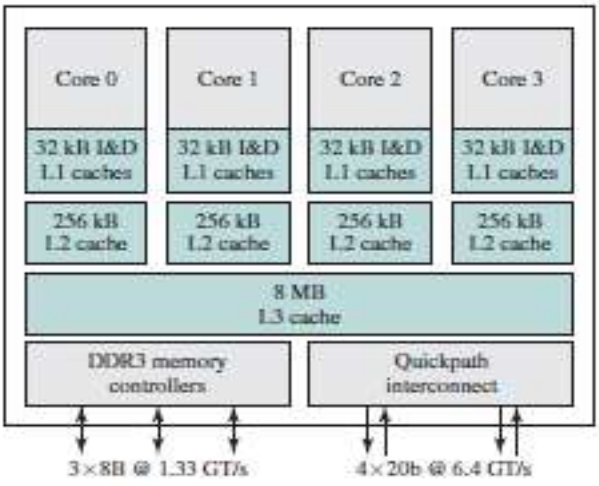
An SMP organization has a number of potential advantages over a uni-processor organization, including the following:

- **Performance:** If the work to be done by a computer can be organized so that some portions of the work can be done in parallel, then a system with multiple processors will yield greater performance than one with a single processor of the same type.
- **Availability:** In a symmetric multiprocessor, because all processors can perform the same functions, the failure of a single processor does not halt the machine. Instead, the system can continue to function at reduced performance.
- **Incremental growth:** A user can enhance the performance of a system by adding an additional processor.
- **Scaling:** Vendors can offer a range of products with different price and performance characteristics based on the number of processors configured in the system.

➔**MULTICORE COMPUTERS**

➔A **multicore** computer, also known as a **chip multiprocessor,** combines two or more processors (called cores) on a single piece of silicon (called a die). Typically, each core consists of all of the components of an independent processor, such as registers, ALU, pipeline hardware, and control unit, plus L1 instruction and data caches. In addition to the multiple cores, contemporary multicore chips also include L2 cache and, in some cases, L3 cache. The motivation for the development of multicore computers can be summed up as follows.

➔For decades, microprocessor systems have experienced a steady, usually exponential, increase in performance. This is partly due to hardware trends, such as an increase in clock frequency and the ability to put cache memory closer to the processor because of the increasing miniaturization of microcomputer components. Performance has also been improved by the increased complexity of processor design to exploit parallelism in instruction execution and memory access.



➔In brief, designers have come up against practical limits in the ability to achieve greater performance by means of more complex processors. Designers have found that the best way to improve performance to take advantage of advances in hardware is to put multiple processors and a substantial amount of cache memory on a single chip

➔An example of a multicore system is the Intel Core i7, which includes four x86 processors, each with a dedicated L2 cache, and with a shared L3 cache. One mechanism Intel uses to make its caches more effective is prefetching, in which the hardware examines memory access patterns and attempts to fill the caches speculatively with data that's likely to be requested soon.

COMPONENTS OF OPERATING SYSTEM.

There are eight major operating system components.

They are :
- Process management
- Main-memory management
- File management
- I/O-system management
- Secondary-storage management
- Networking
- Protection system
- Command-interpreter system

(i) Process Management

- A process can be thought of as a program in execution. A batch job is a process. A time shared user program is a process.
- A process needs certain resources-including CPU time, memory, files, and I/O devices-to accomplish its task.
- A program by itself is not a process; a program is a passive entity, such as the contents of a file stored on disk, whereas a process is an active entity, with a program counter specifying the next instruction to execute.
- A process is the unit of work in a system.
- The operating system is responsible for the following activities in connection with process management:
  - Creating and deleting both user and system processes
  - Suspending and resuming processes
  - Providing mechanisms for process synchronization
  - Providing mechanisms for process communication
  - Providing mechanisms for deadlock handling

(ii) Main – Memory Management

- Main memory is a large array of words or bytes, ranging in size from hundreds of thousands to billions. Each word or byte has its own address.
- Main memory is a repository of quickly accessible data shared by the CPUand I/O devices.
- To improve both the utilization of the CPU and the speed of the computer's response to its users, we must keep several programs in memory.
- The operating system is responsible for the following activities in connection with memory management:
  - Keeping track of which parts of memory are currently being used and by whom.
  - Deciding which processes are to be loaded into memory when memory space becomes available
  - Allocating and deallocating memory space as needed.

(iii) File Management

- File management is one of the most visible components of an operating system.
- The operating system is responsible for the following activities in connection with file management:
  - Creating and deleting files
  - Creating and deleting directories
  - Supporting primitives for manipulating files and directories
  - Mapping files onto secondary storage
  - Backing up files on stable (nonvolatile) storage media

(iv) I/O System management

- One of the purposes of an operating system is to hide the peculiarities of specific hardware devices from the user. This is done using the I/O subsystem.
- The I/O subsystem consists of
  - A memory-management component that includes buffering, caching, and spooling
  - A general device-driver interface
  - Drivers for specific hardware devices

(v) Secondary storage management

- Because main memory is too small to accommodate all data and programs, and because the data that it holds are lost when power is lst the computer system must provide secondary storage to back up main memory.
- The operating system is responsible for the following activities in connection with disk management:
  - Free-space management
  - Storage allocation
  - Disk scheduling

(vi) Networking

- A distributed system is a collection of processors that do not share memory, peripheral devices, or a clock.
- Instead, each processor has its own local memory and clock, and the processors communicate with one another through various communication lines, such as high-speed buses or networks.
- The processors in the system are connected through a communication network, which can be configured in a number of different ways.

(vii) Protection System

- Various processes must be protected from one another's activities. For that purpose, mechanisms ensure that the files, memory segments, CPU, and other resources can be operated on by only those processes that have gained proper authorization from the operating system.
- Protection is any mechanism for controlling the access of programs, processes, or users to the resources defined by a computer system.
- Protection can improve reliability by detecting latent errors at the interfaces between component subsystems.

(viii) Command-Interpreter System

- One of the most important systems programs for an operating system is the command interpreter.
- It is the interface between the user and the operating system.
- Some operating systems include the command interpreter in the kernel. Other operating systems, such as MS-DOS and UNIX, treat the

command interpreter as a special program that is running when a job is initiated, or when a user first logs on (on time-sharing systems).

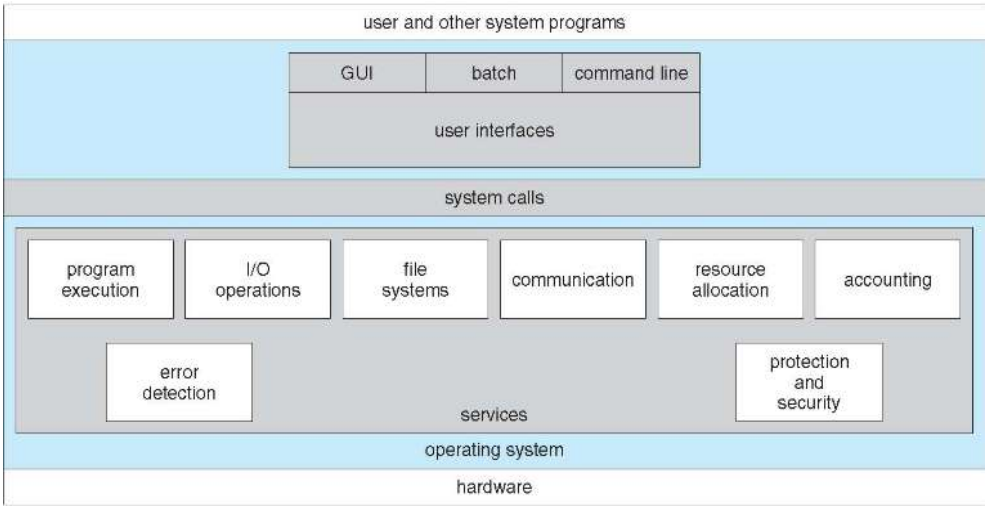
- Many commands are given to the operating system by control statements.
- When a new job is started in a batch system, or when a user logs on to a time-shared system, a program that reads and interprets control statements is executed automatically.
- This program is sometimes called the control-card interpreter or the Command-line interpreter, and is often known as the shell.

SERVICES OF OPERATING SYSTEM

➔An operating system provides services to programs and to the users of those programs. It provided by one environment for the execution of programs.

➔The services provided by one operating system is difficult than other operating system. Operating system makes the programming task easier. The common service provided by the operating system is listed below.

1.Program execution
2.I/O operation
3.File system manipulation
4.Communications
5.Error detection



The OS provides certain services to programs and to the users of those programs.

1. **Program execution:** The system must be able to load a program into memory and to run that program. The program must be able to end its execution, either normally or abnormally (indicating error).
2. **I/O operations:** A running program may require I/O. This I/O may involve a file or an I/O device.
3. **File-system manipulation:** The program needs to read, write, create, delete files.
4. **Communications:** In many circumstances, one process needs to exchange Information with another process. Such communication can occur in two major ways. The first takes place between processes that are executing on the same computer; the second takes place between processes that are executing on different computer systems that are tied together by a computer network.
5. **Error detection:** The operating system constantly needs to be aware of possible errors. Errors may occur in the CPU and memory hardware (such as a memory error or a power failure),

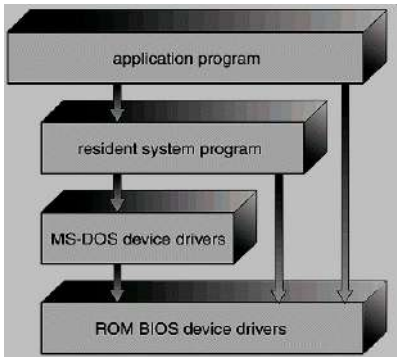
in I/O devices (such as a parity error on tape, a connection failure on a network, or lack of paper in the printer), and in the user program (such as an arithmetic overflow, an attempt to access an illegal memory location, or a too-great use of CPU time). For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing.

6. **Resource allocation:** Different types of resources are managed by the Os.When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them.
7. **Accounting:** We want to keep track of which users use how many and which kinds of computer resources. This record keeping may be used for accounting or simply for accumulating usage statistics.
8. **Protection:** The owners of information stored in a multiuser computer system may want to control use of that information. Security of the system is also important.

OPERATING SYSTEM STRUCTURES

SIMPLE STRUCTURE:

➔In MS-DOS, application programs are able to access the basic I/O routines to write directly to the display and disk drives. Such freedom leaves MS-DOS vulnerable to errant (or malicious) programs, causing entire system to crash when user programs fail.



MS-DOS LAYER STRUCTURE:

➔ UNIX operating system. It consists of two separable parts, the kernel and the system programs. The kernel is further separated into a series of interfaces and device drivers. We can view the traditional UNIX operating system as being layered. Everything below the system call interface and above the physical hardware is the kernel.

(the users)		
shells and commands compilers and interpreters system libraries		
system-call interface to the kernel		
signals terminal handling character I/O system terminal drivers	file system swapping block I/O system disk and tape drivers	CPU scheduling page replacement demand paging virtual memory
kernel interface to the hardware		
terminal controllers terminals	device controllers disks and tapes	memory controllers physical memory

The kernel provides the file system, CPU scheduling, memory management, and other operating system functions through system calls. There is number of functionality to be combined into one level. This monolithic structure was difficult to implement and maintain.

LAYERED APPROACH:

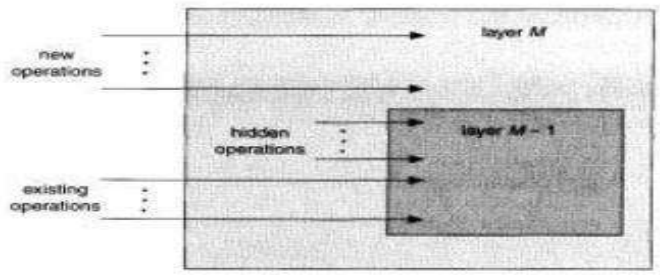
➔The operating system is broken into a number of layers (levels). The bottom layer (layer 0) is the hardware; the highest (layer M) is the user interface.

➔The main advantage of the layered approach is simplicity of construction and debugging. The layers are selected so that each uses functions (operations) and services of only lower-level layers. This approach simplifies debugging and system verification. The first layer can be debugged without any concern for the rest of the system, because, by definition, it uses only the basic hardware to implement its functions.

➔Once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and so on. If an error is found during the debugging of a particular layer, the error must be on that layer, because the layers below it are already debugged. Each layer hides the existence of certain data structures, operations, and hardware from higher-level layers.

➔The major difficulty with the layered approach involves appropriately defining the various layers as a layer can use only lower-level layers. Another problem with layered implementations is they tend to be less efficient than other types. Each layer adds overhead to the system call; the net result is a system call that takes longer than a non- layered system.

Example of Layered Approach

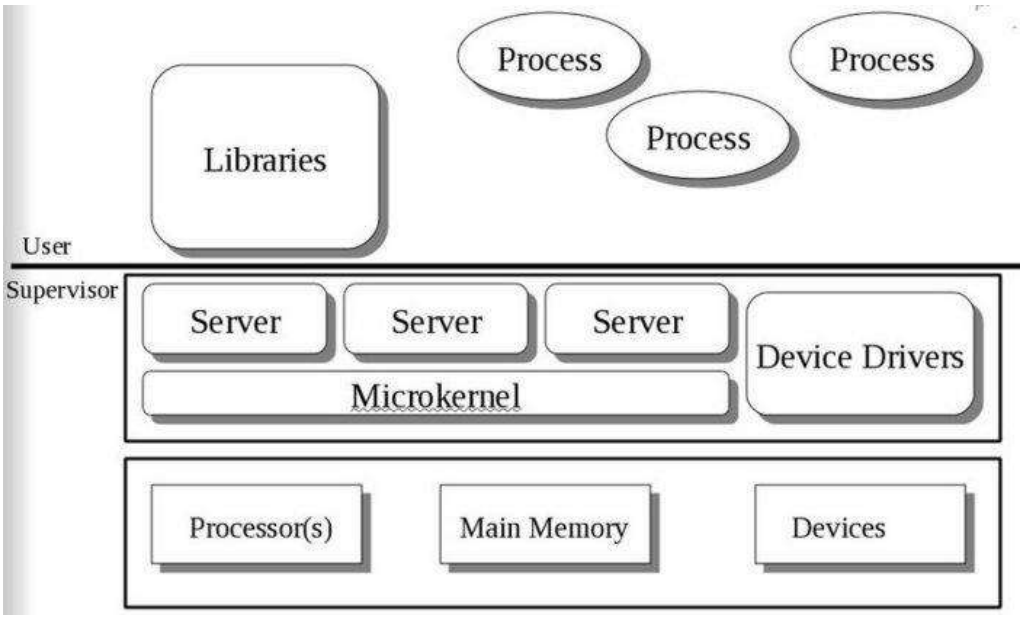


MICROKERNEL APPROACH:

➔In the mid-1980s, researchers at Carnegie Mellon University developed an operating system called Mach that modularized the kernel using the microkernel approach. Microkernel's provide minimal process and memory management, in addition to a communication facility.

➔The main function of the micro kernel is to provide a communication facility between the client program and the various services running in user space. One benefit of the microkernel approach is ease of extending the operating system. All new services are added to user space and consequently do not require modification of the kernel. The microkernel also provides more security and reliability, since most services are running as user, rather than kernel-processes.

➔Microkernel's can suffer from decreased performance due to increased system function overhead.



MODULES:

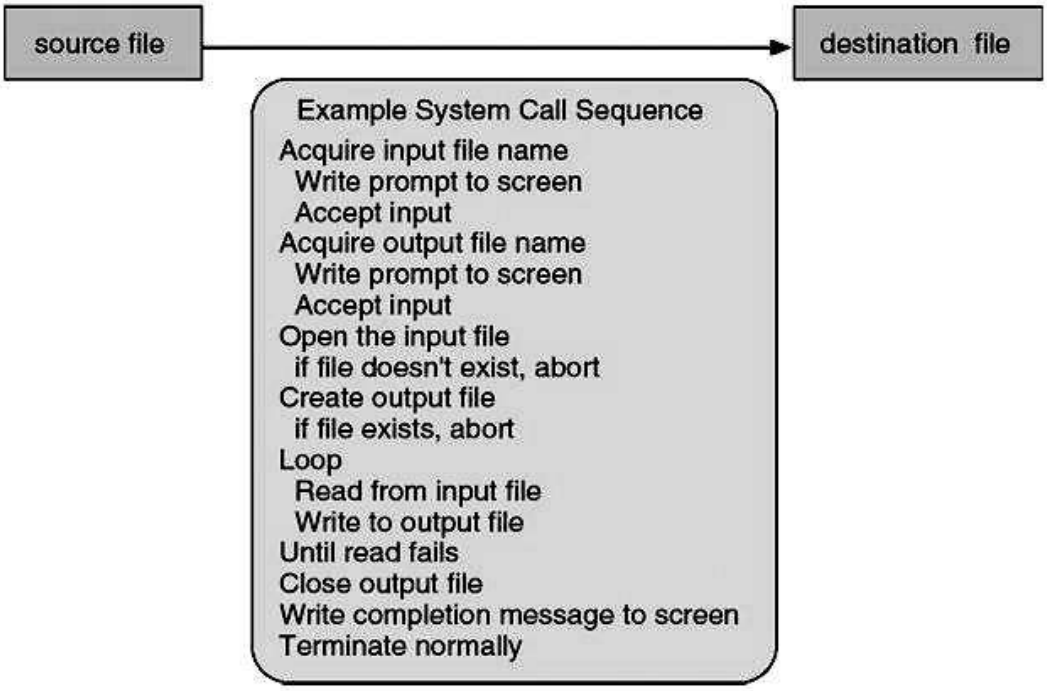
➔The current methodology for operating-system design involves using object-oriented programming techniques to create a modular kernel. Here, the kernel has a set of core components and links in additional services either during boot time or during run time. Such a strategy uses dynamically loadable modules.

➔Such a design allows the kernel to provide core services yet also allows certain features to be implemented dynamically.

SYSTEM CALLS

➔A system call is a request that a program makes to the kernel through a software interrupt. System calls provide the interface between a process and the operating system.

➔These calls are generally available as assembly-language instructions. Certain systems allow system calls to be made directly from a high-level language program, in which case the calls normally resemble predefined function or subroutine calls.



TYPES OF SYSTEM CALLS:

Traditionally, System Calls can be categorized in six groups, which are: Process Control, File Management, Device Management, Information Maintenance, Communications and Protection.

PROCESS CONTROL

- A running program needs to be able to stop execution either normally or abnormally.
- When execution is stopped abnormally, often a dump of memory is taken and can be examined with a debugger.

Following are functions of process control:

End, abort
Load, execute
Create process, terminate process
Get process attributes, set process attributes
Wait for time
Wait event, signal event
Allocate and free memory

FILE MANAGEMENT

- We first need to be able to create and delete files. Either system call requires the name of the file and perhaps some of the file's attributes.
- Once the file is created, we need to open it and to use it. We may also read, write, or reposition. Finally, we need to close the file, indicating that we are no longer using it.
- We may need these same sets of operations for directories if we have a directory structure for organizing files in the file system.
- In addition, for either files or directories, we need to be able to determine the values of various attributes and perhaps to reset them if necessary. File attributes include the file name, a file type, protection codes, accounting information, and so on

Functions:

Create, delete file
Open, close
Read, write, reposition
Get file attributes, set file attributes

DEVICE MANAGEMENT

- A process may need several resources to execute - main memory, disk drives, access to files, and so on. If the resources are available, they can be granted, and control can be returned to the user process. Otherwise, the process will have to wait until sufficient resources are available.
- The various resources controlled by the OS can be thought of as devices. Some of these devices are physical devices (for example, tapes), while others can be thought of as abstract or virtual devices (for example, files).
- Once the device has been requested (and allocated to us), we can read, write, and (possibly) reposition the device, just as we can with files.
- In fact, the similarity between I/O devices and files is so great that many OSs, including UNIX, merge the two into a combined file-device structure.
- A set of system calls is used on files and devices. Sometimes, 1/0 devices are identified by special file names, directory placement, or file attributes.

Functions:

Request device, release device
Read, write, reposition
Get device attributes, set device attributes
Logically attach or detach devices

INFORMATION MAINTENANCE

- Many system calls exist simply for the purpose of transferring information between the user program and the OS. For example, most systems have a system call to return the current time and date.
- Other system calls may return information about the system, such as the number of current users, the version number of the OS, the amount of free memory or disk space, and so on.
- In addition, the OS keeps information about all its processes, and system calls are used to access this information. Generally, calls are also used to reset the process information.

Functions:

Get time or date, set time or date
Get system data, set system data
Get process, file, or device attributes
Set process, file, or device attributes

COMMUNICATIONS

- There are two common models of interprocess communication: the message-passing model and the shared-memory model. In the message-passing model, the communicating processes exchange messages with one another to transfer information.
- In the shared-memory model, processes use shared memory creates and shared memory attaches system calls to create and gain access to regions of memory owned by other processes.
- Recall that, normally, the OS tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas.
- Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. It is also easier to implement than is shared memory for intercomputer communication.
- Shared memory allows maximum speed and convenience of communication, since it can be done at memory speeds when it takes place within a computer. Problems exist, however, in the areas of protection and synchronization between the processes sharing memory.

Functions:

Create, delete communication connection
Send, receive messages
Transfer status information
Attach or detach remote devices

PROTECTION

Get File Security, Set File Security

Get Security Group, Set Security Group

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

SYSTEM PROGRAMS

➔System programs provide a convenient environment for program development and execution. They can be divided into several categories:

1. **File management:** These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.
2. **Status information:** The status such as date, time, amount of available memory or diskspace, number of users or similar status information.
3. **File modification:** Several text editors may be available to create and modify the content of files stored on disk or tape.
4. **Programming-language support:** Compilers, assemblers, and interpreters for common programming languages are often provided to the user with the operating system.
5. **Program loading and execution:** The system may provide absolute loaders, relocatable loaders,

linkage editors, and overlay loaders.

6. **Communications:** These programs provide the mechanism for creating virtual connections among processes, users, and different computer systems. (email, FTP, Remote log in)
7. **Application programs:** Programs that are useful to solve common problems, or to perform common operations.

Eg. Web browsers, database systems.

GENERATION AND SYSTEM BOOT.

Operating-System Generation

➔ It is possible to design, code, and implement an operating system specifically for one machine at one site. More commonly, however, operating systems are designed to run on any of a class of machines at a variety of sites with a variety of peripheral configurations. The system must then be configured or generated for each specific computer site, a process sometimes known as system generation (SYSGEN).

➔The operating system is normally distributed on disk or CD-ROM. To generate a system, we use a special program. The SYSGEN program reads from a given file, or asks the operator of the system for information concerning the specific configuration of the hardware system, or probes the hardware directly to determine what components are there. The following kinds of information must be determined.

➔What CPU is to be used?

What options (extended instruction sets, floating-point arithmetic, and so on) are installed? For multiple CPU systems, each CPU must be described.

➔ How much memory is available?

Some systems will determine this value themselves by referencing memory location after memory location until an "illegal address" fault is generated. This procedure defines the final legal address and hence the amount of available memory.

➔ What devices are available?

The system will need to know how to address each device (the device number), the device interrupt number, the device's type and model, and any special device characteristics.

➔What operating-system options are desired, or what parameter values are to be used? These options or values might include how many buffers of which sizes should be used, what type of CPU-scheduling algorithm is desired, what the maximum number of processes to be supported is, and so on. Once this information is determined, it can be used in several ways.

System Boot

➔After an operating system is generated, it must be made available for use by the hardware. But how does the hardware know where the kernel is or how to load that kernel? The procedure of starting a computer by loading the kernel is known as booting the system.

➔On most computer systems, a small piece of code known as the **bootstrap program** or **bootstrap loader** locates the kernel, loads it into main memory, and starts its execution. Some computer systems, such as PCs, use a two-step process in which a simple bootstrap loader fetches a more complex boot program from disk, which in turn loads the kernel.

➔When a CPU receives a reset event—for instance, when it is powered up or rebooted—the instruction register is loaded with a predefined memory location, and execution starts there. At that location is the initial bootstrap program.

➔This program is in the form of read-only memory (ROM), because the RAM is in an unknown state at system startup. ROM is convenient because it needs no initialization and cannot be infected by a computer virus.

➔The bootstrap program can perform a variety of tasks. Usually, one task is to run diagnostics to determine the state of the machine. If the diagnostics pass, the program can continue with the booting steps. It can also initialize all aspects of the system, from CPU registers to device controllers and the contents of main memory.

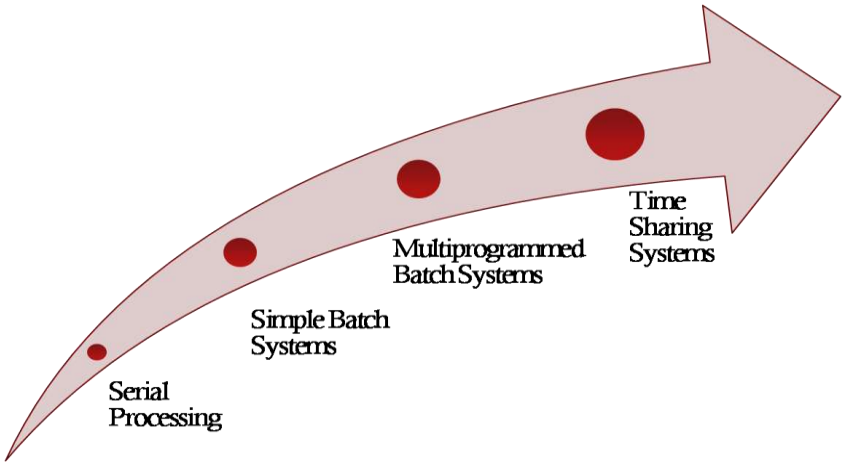
➔Sooner or later, it starts the operating system. Some systems—such as cellular phones, PDAs, and game consoles—store the entire operating system in ROM. Storing the operating system in ROM is suitable for small operating systems, simple supporting hardware, and rugged operation.

➔A problem with this approach is that changing the bootstrap code requires changing the ROM hardware chips. Some systems resolve this problem by using erasable programmable read-only memory (EPROM), which is read-only except when explicitly given a command to become writable.

➔All forms of ROM are also known as firmware, since their characteristics fall somewhere between those of hardware and those of software. A problem with firmware in general is that executing code there is slower than executing code in RAM. Some systems store the operating system in firmware and copy it to RAM for fast execution. A final issue with firmware is that it is relatively expensive, so usually only small amounts are available.

EVALUATION OF OPERATING SYSTEMS.

Stages of Evaluation



Serial Processing

➔Users access the computer in series. From the late 1940's to mid 1950's, the programmer interacted directly with computer hardware i.e., no operating system.

➔These machines were run with a console consisting of display lights, toggle switches, some form of input device and a printer. Programs in machine code are loaded with the input device like card reader.

➔If an error occur the program was halted and the error condition was indicated by lights. Programmers examine the registers and main memory to determine error. If the program is success, then output will appear on the printer.

➔Main problem here is the setup time. That is single program needs to load source program into memory, saving the compiled (object) program and then loading and linking together.

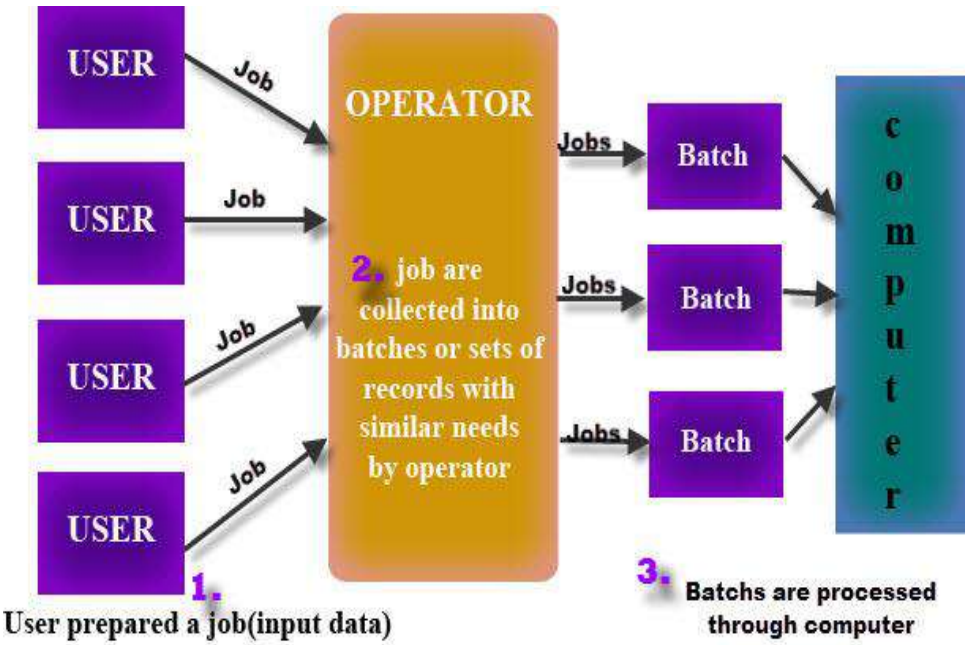
Simple Batch Systems

➔To speed up processing, jobs with similar needs are batched together and run as a group. Thus, the programmers will leave their programs with the operator. The operator will sort programs into batches with similar requirements.

The problems with Batch Systems are:

➔Lack of interaction between the user and job.CPU is often idle, because the speeds of the mechanical I/O devices are slower than CPU. For overcoming this problem use the Spooling

➔Technique. Spool is a buffer that holds output for a device, such as printer, that can not accept interleaved data streams. That is when the job requests the printer to output a line. That line is copied into a system buffer and is written to the disk. When the job is completed, the output is printed. Spooling technique can keep both the CPU and the I/O devices working at much higher rates.

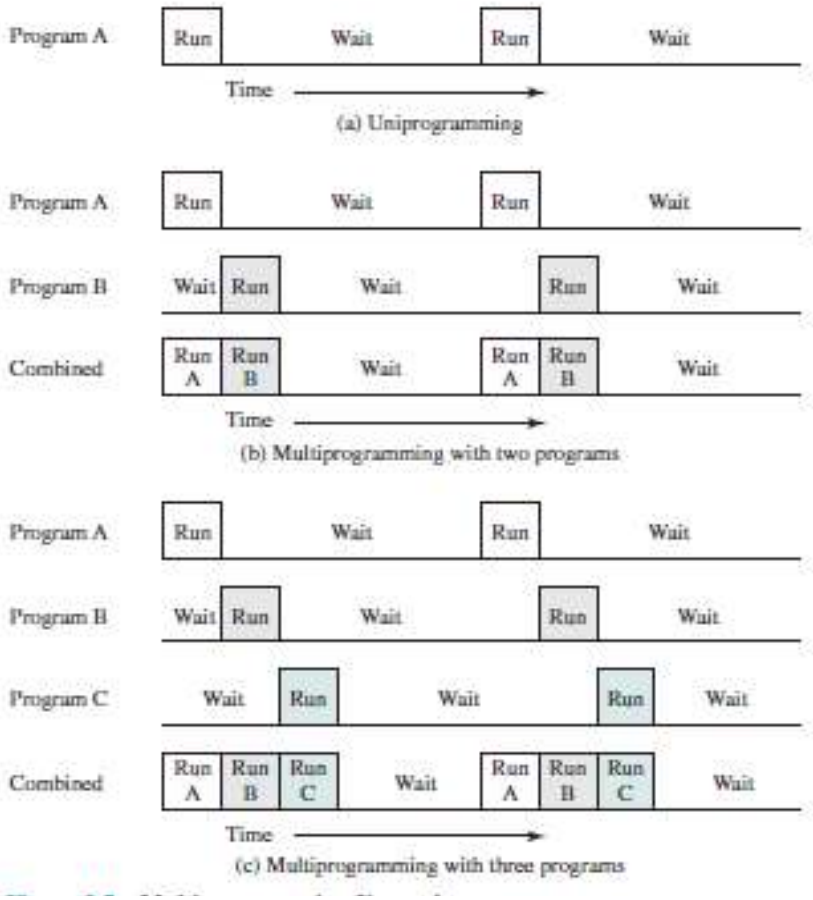


Multiprogrammed Batch Systems

➔Jobs must be run sequentially, on a first-come, first-served basis. However when several jobs are on a direct-access device like disk, job scheduling is possible. The main aspect of job scheduling is multiprogramming. Single user cannot keep the CPU or I/O devices busy at all times.

Thus multiprogramming increases CPU utilization.

➔In when one job needs to wait, the CPU is switched to another job, and so on. Eventually, the first job finishes waiting and gets the CPU back.

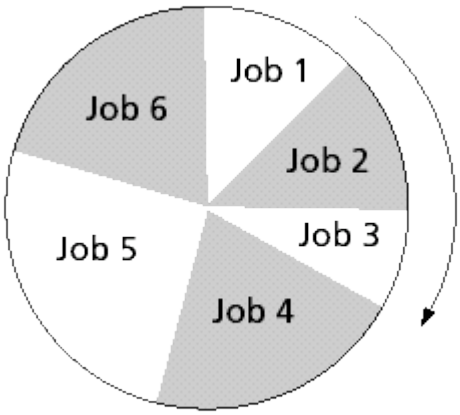


Time-Sharing Systems

➔Time-sharing systems are not available in 1960s. Time-sharing or multitasking is a logical extension of multiprogramming. That is processors time is shared among multiple users simultaneously is called time-sharing. The main difference between Multiprogrammed Batch Systems and Time-Sharing Systems is in multiprogrammed batch systems its objective is maximize processor use, whereas in Time-Sharing Systems its objective is minimize response time.

➔Multiple jobs are executed by the CPU by switching between them, but the switches occur so frequently. Thus, the user can receives an immediate response. For example, in a transaction processing, processor execute each user program in a short burst or quantum of computation. That is if n users are present, each user can get time quantum. When the user submits the command, the response time is seconds at most.

➔Operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time. Computer systems that were designed primarily as batch systems have been modified to time-sharing systems.
For example IBM's OS/360.
Time-sharing operating systems are even more complex than multi-programmed operating systems. As in multiprogramming, several jobs must be kept simultaneously in memory.



OBJECTIVES AND FUNCTIONS OF AN OPERATING SYSTEMS

An OS is a program that controls the execution of application programs and acts as an interface between applications and the computer hardware. It can be thought of as having three objectives:

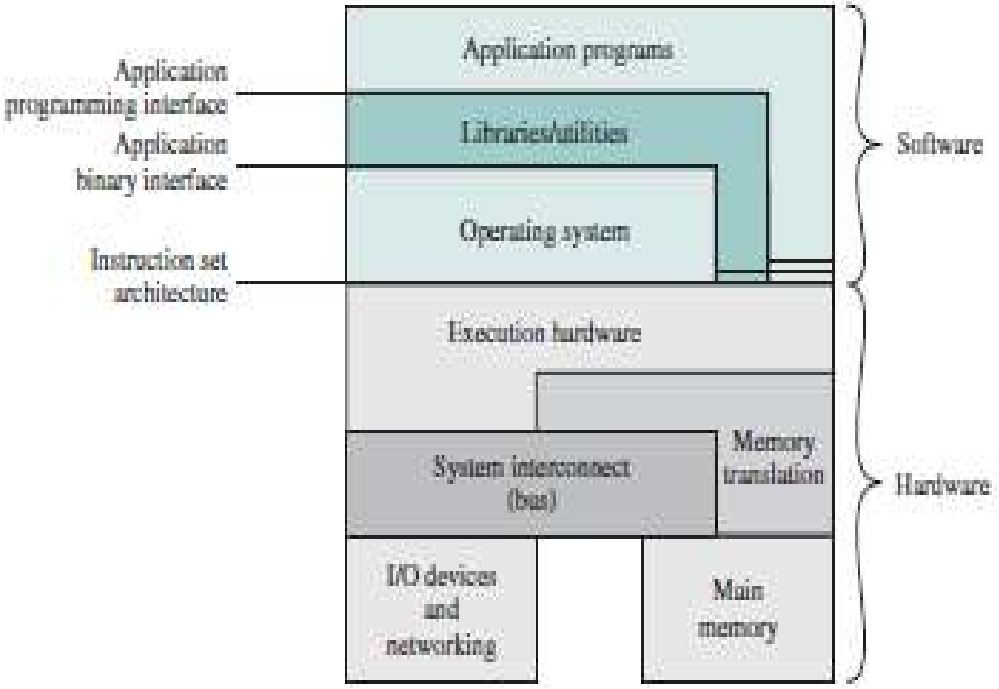
- **Convenience**: An OS makes a computer more convenient to use.
- **Efficiency**: An OS allows the computer system resources to be used in an efficient manner.
- **Ability to evolve**: An OS should be constructed in such a way as to permit the effective development, testing, and introduction of new system functions without interfering with service.

The Operating System as a User/Computer Interface

➔The hardware and software used in providing applications to a user can be viewed in a layered or hierarchical fashion. The user of those applications, the end user, generally is not concerned with the details of computer hardware. Thus, the end user views a computer system in terms of a set of applications.

➔An application can be expressed in a programming language and is developed by an application programmer. If one were to develop an application program as a set of machine instructions that is completely responsible for controlling the computer hardware, one would be faced with an overwhelmingly complex undertaking.

➔To ease this chore, a set of system programs is provided. Some of these programs are referred to as utilities, or library programs. These implement frequently used functions that assist in program creation, the management of files, and the control of I/O devices. A programmer will make use of these facilities in developing an application, and the application, while it is running, will invoke the utilities to perform certain functions.



Three key interfaces in a typical computer system:

· **Instruction set architecture (ISA)** : The ISA defines the repertoire of machine language instructions that a computer can follow. This interface is the boundary between hardware and software. Note that both application programs and utilities may access the ISA directly. For these programs, a subset of the instruction repertoire is available (user ISA). The OS has access to additional machine language instructions that deal with managing system resources (system ISA).

· **Application binary interface (ABI)** : The ABI defines a standard for binary portability across programs. The ABI defines the system call interface to the operating system and the hardware resources and services available in a system through the user ISA.

· **Application programming interface (API)** : The API gives a program access to the hardware resources and services available in a system through the user ISA supplemented with high-level language (HLL) library calls. Any system calls are usually performed through libraries. Using an API enables application software to be ported easily, through recompilation, to other systems that support the same API.

The Operating System as Resource Manager

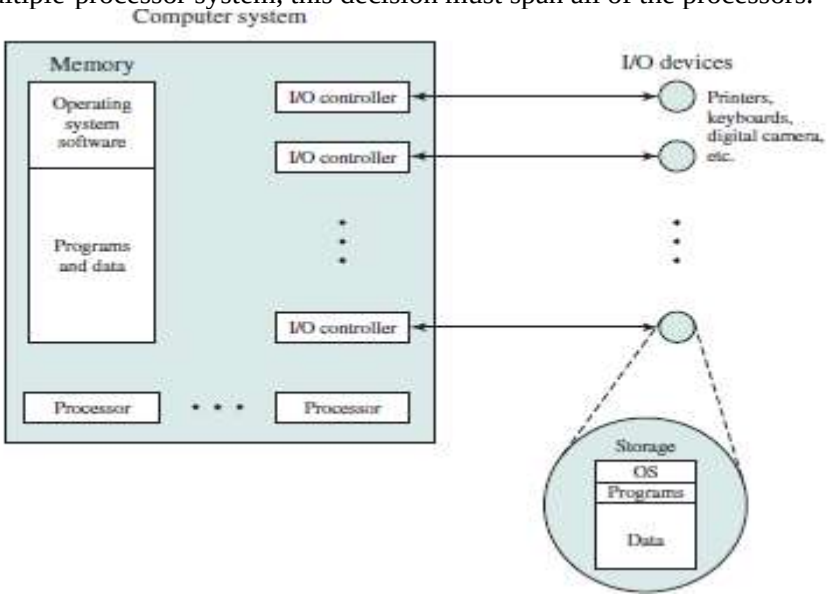
A computer is a set of resources for the movement, storage, and processing of data and for the control of these functions. The OS is responsible for managing these resources.

By managing the computer's resources, the OS is in control of the computer' s basic functions

The main resources that are managed by the OS. A portion of the OS is in main memory. This includes the **kernel** , or **nucleus** , which contains the most frequently used functions in the OS and, at a given time, other portions of the OS currently in use. The remainder of main memory contains user programs and data.

The memory management hardware in the processor and the OS jointly control the allocation of main memory, as we shall see. The OS decides when an I/O device can be used by a program in execution and

controls access to and use of files. The processor itself is a resource, and the OS must determine how much processor time is to be devoted to the execution of a particular user program. In the case of a multiple-processor system, this decision must span all of the processors.



Ease of Evolution of an Operating System

A major OS will evolve over time for a number of reasons:

- **Hardware upgrades plus new types of hardware**
- **New services**
- **Fixes**

OPERATING SYSTEM OPERATIONS

Modern operating systems are interrupt driven. If there are no processes to execute, OS will sit idle and wait for some event to happen. Interrupts could be hardware interrupts or software interrupts. The OS is designed to handle both. A trap (or an exception) is a software generated interrupt caused either by an error (e.g. divide by zero) or by a specific request from a user program. A separate code segment is written in the OS to handle different types of interrupts. These codes are known as interrupt handlers/ interrupt service routine. A properly designed OS ensures that an illegal program should not harm the execution of other programs. To ensure this, the OS operates in dual mode.

Dual mode of operation

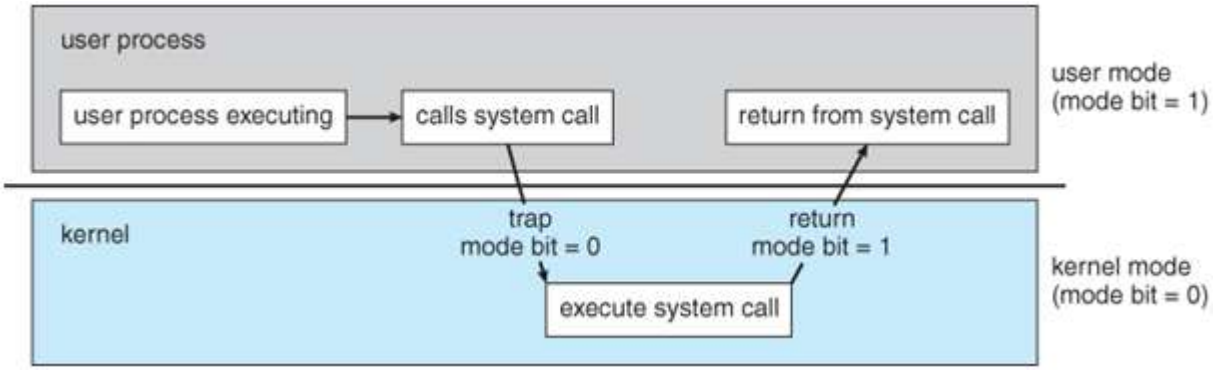
The OS is design in such a way that it is capable of differentiating between the execution of OS code and user defined code. To achieve this OS need two different modes of operations this is thereby controlled by mode bit added to hardware of computer system as shown in Table 4.

Mode Type	**Definition**	**Mode Bit**	**Examples**
User Mode	User Defined codes are executed	Mode Bit=1	Creation of word document or in general user using any application program
Kernel Mode	OS system codes are executed (also known as supervisor, system, or privileged mode)	Mode Bit=0	Handling interrupts-Transferring control of a process from CPU to I/O on request

User and Kernel Mode of Operating System

Transition from User to Kernel mode

When a user application is executing on the computer system OS is working in user mode. On signal of system call via user application, the OS transits from user mode to kernel mode to service that request as shown in Fig. 11.



Transition from user to kernel mode

When the user starts the system the hardware starts in monitor/ kernel mode and loads the operating system. OS has the initial control over the entire system, when instructions are executed in kernel mode. OS then starts the user processes in user mode and on occurrence of trap, interrupt or system call again switch to kernel mode and gains control of the system. System calls are designed for the user programs through which user can ask OS to perform tasks reserved for operating system. System calls usually take the form of the trap. Once the OS service the interrupt it transfers control back to user program hence user mode by setting mode bit=1.

Benefits of Dual Mode

The dual mode of operation protects the operating system from errant users, and errant users from one another by designating some of the machine instructions that may cause harm as privileged instructions. These instructions can execute only in kernel mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction, but rather treats the instruction as illegal and traps to the operating system. Examples of privileged instructions:

1. Switching to kernel mode
2. Managing I/O control
3. Timer Management
4. Interrupt Management

Timer

Since OS operates in dual mode it should maintain control over CPU. The system should not allow a user application:

1. To be stuck in an infinite loop
2. To fail to call system services
3. Never return control to the OS

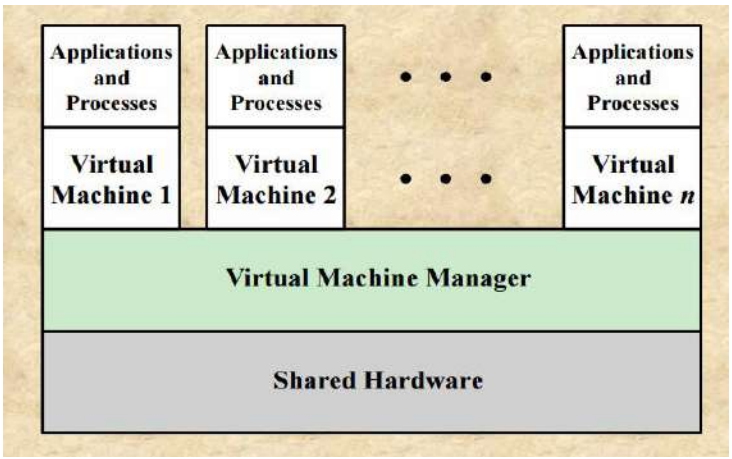
To achieve this goal, we can use timer. This timer control mechanism will interrupt the system at a specified period; thereby preventing user program from running too long. This can be implemented either as fixed timer or variable timer

Additional Topics

Virtual Machines (VM)

Virtualization technology enables a single PC or server to simultaneously run multiple operating systems or multiple sessions of a single OS

 A machine with virtualization software can host numerous applications, including those that run on different operating systems, on a single platform
 The host operating system can support a number of virtual machines, each of which has the characteristics of a particular OS
 The solution that enables virtualization is a virtual machine monitor (VMM), or hypervisor



A virtual machine takes the layered approach to its logical conclusion. It treats hardware and the operat

with its own (virtual) memory.

The resources of the physical computer are shared to create the virtual machines.

1. CPU scheduling can create the appearance that users have their own processor.
2. Spooling and a file system can provide virtual card readers and virtual line printers.
3. A normal user time-sharing terminal serves as the virtual machine operator's console.

Advantages/Disadvantages of Virtual Machines

The virtual-machine concept provides complete protection of system resources since each virtual machine is isolated from all other virtual machines.

This isolation, however, permits no direct sharing of resources.

A virtual-machine system is a perfect vehicle for operating-systems research and development. System development is done on the virtual machine, instead of on a physical machine and so does not disrupt normal system operation.

The virtual machine concept is difficult to implement due to the effort required to provide an exact duplicate to the underlying machine.

CS6401-OPERATING SYSTEMS

UNIT II PROCESS MANAGEMENT

Processes-Process Concept, Process Scheduling, Operations on Processes, Interprocess Communication; Threads- Overview, Multicore Programming, Multithreading Models; Windows 7 -Thread and SMP Management. Process Synchronization - Critical Section Problem, Mutex Locks, Semophores, Monitors; CPU Scheduling and Deadlocks.

PROCESS CONCEPTS

Process Concept

➔A process can be thought of as a program in execution.

➔A process is the unit of the unit of work in a modern time-sharing system.

A process is more than the program code, which is sometimes known as the **text section**. It also includes the current activity, as represented by the value of the **program counter** and the contents of the processor's registers.

A process generally also includes the process **stack**, which contains temporary data (such as function parameters, return addresses, and local variables), and a **data section**, which contains global variables. A process may also include a **heap**, which is memory that is dynamically allocated during process run time.

Difference between program and process

➔A program is a passive entity, such as the contents of a file stored on disk, whereas a process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources.

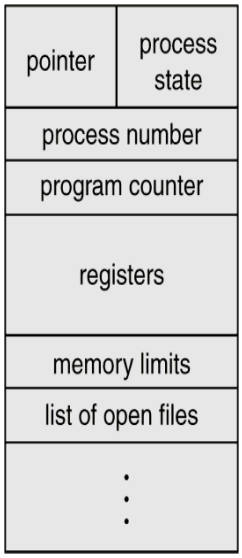
Process Control Block (PCB)

➔Each process is represented in the operating system by a process control block (PCB)-also called a task control block.

➔A PCB defines a process to the operating system.

➔It contains the entire information about a process.

Some of the information a PCB.



Process state: The state may be new, ready, running, and waiting, halted, and SO on.

Program counter: The counter indicates the address of the next instruction to be executed for this process.

CPU registers: The registers vary in number and type, depending on the computer architecture.

CPU-scheduling information: This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

Memory-management information: This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.

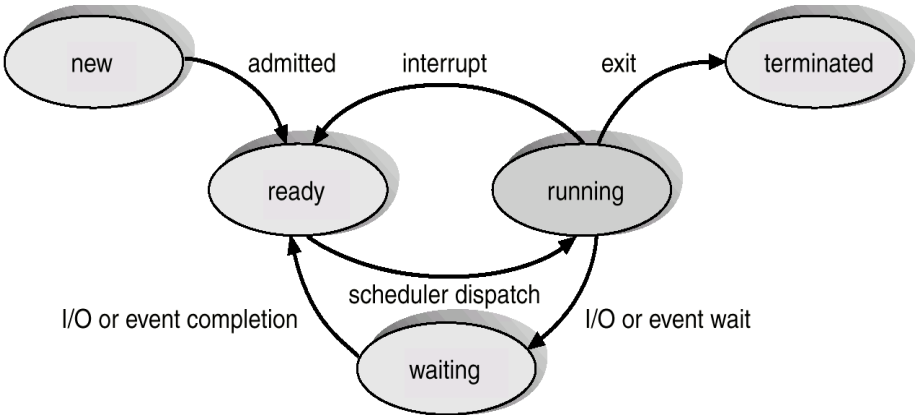
Accounting information: This information includes the amount

of CPU and real time used, time limits, account numbers, job or process numbers, and so on.

Status information: The information includes the list of I/Odevices allocated to this process, a list of open files, and so on.

Process States:

- As a process executes, it changes state.
- The state of a process is defined in part by the current activity of that process.
- Each process may be in one of the following states:
  - **New**: The process is being created.
  - **Running**: Instructions are being executed.
  - **Waiting**: The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
  - **Ready**: The process is waiting to be assigned to a processor.
  - **Terminated**: The process has finished execution.



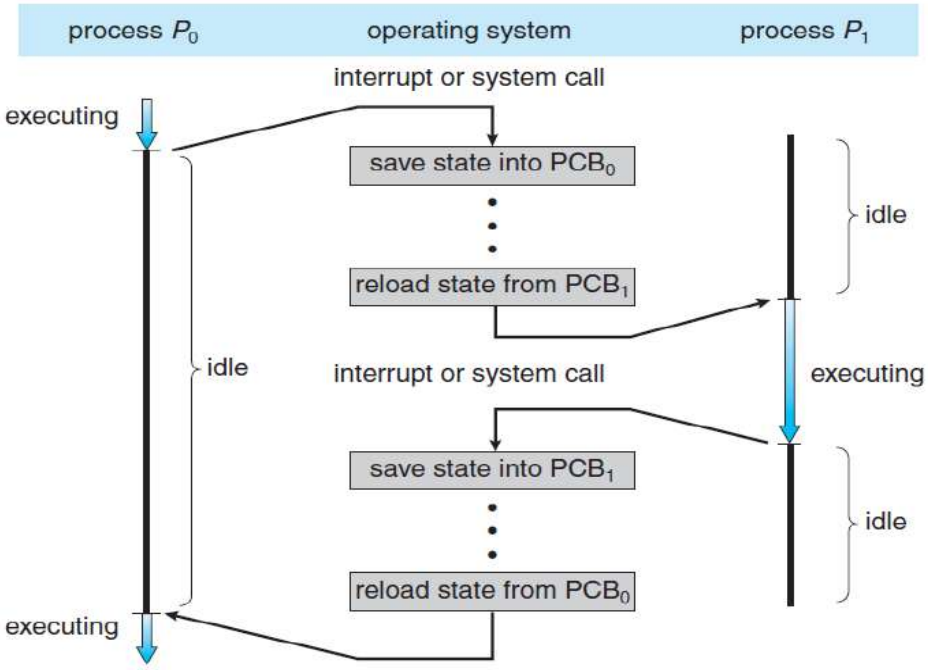


Diagram shows CPU switch from process to process.

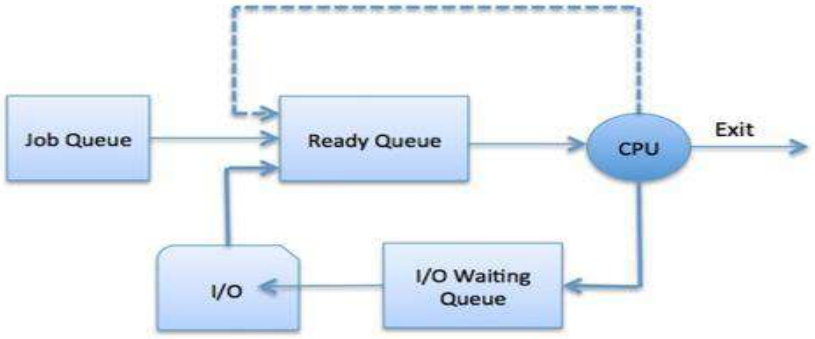
PROCESS SCHEDULING

- The objective of multiprogramming is to have some process running at all times, so as to maximize CPU utilization.

Scheduling Queues

There are 3 types of scheduling queues .They are :

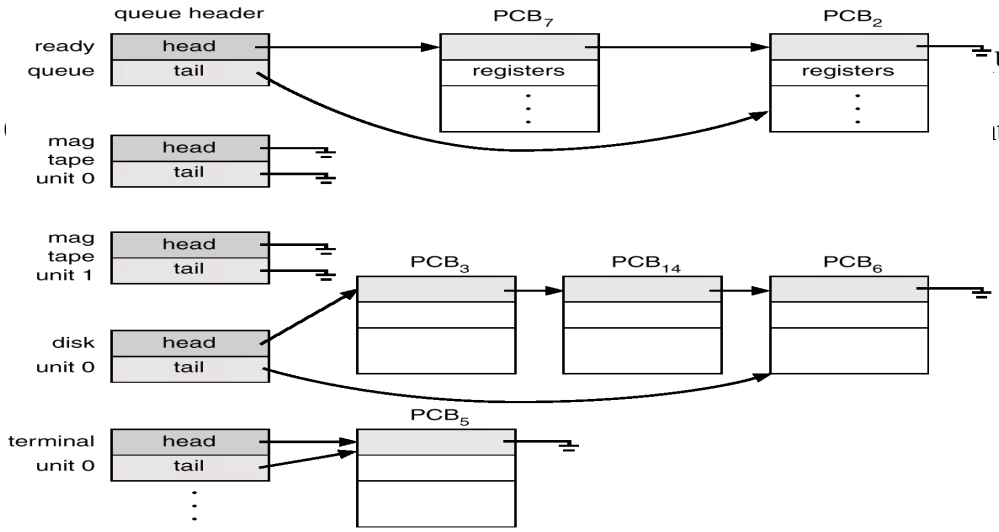
1. Job Queue
2. Ready Queue
3. Device Queue



As processes enter the system, they are put into a **job queue.**

The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**.

The list of processes waiting for an I/O device is kept in a **device queue** for that particular device.



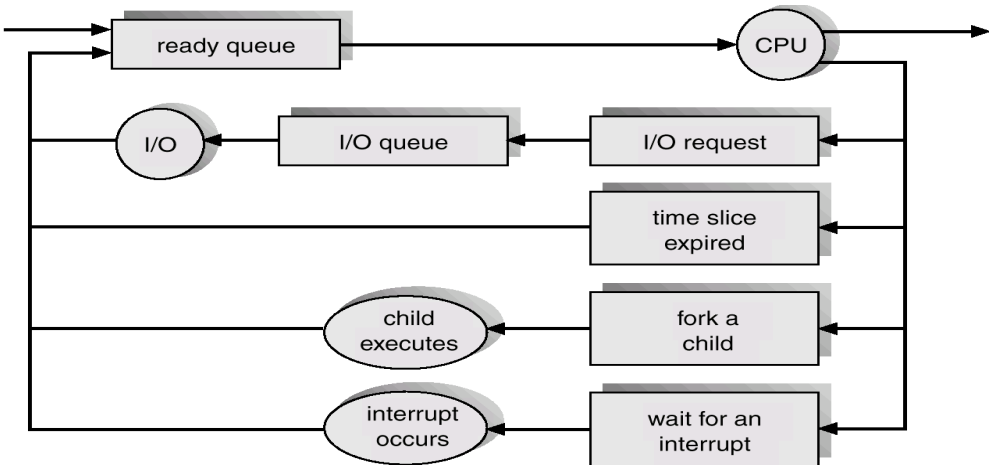
ueue until it is select d for

ts could occur:

- The process could issue an I/O request, and then be placed in an I/O queue.
- The process could create a new subprocess and wait for its

termination.

- The process could be removed forcibly from the CPU, as a result of aninterrupt, and be put back in the ready Queue.
- A common representation of process scheduling is a queueing diagram.



Schedulers

- The operating system must select, for scheduling purposes, processes from these queues in some order
- The selection process is carried out by the appropriate scheduler.

They are:

1. Long-term Scheduler or Job Scheduler
2. Short-term Scheduler or CPU Scheduler
3. Medium term Scheduler

Long-Term Scheduler

- The **long-term scheduler**, or **job scheduler**, selects processes from this pool and loads them into memory for execution. It is invoked very infrequently. It controls the degree of multiprogramming.

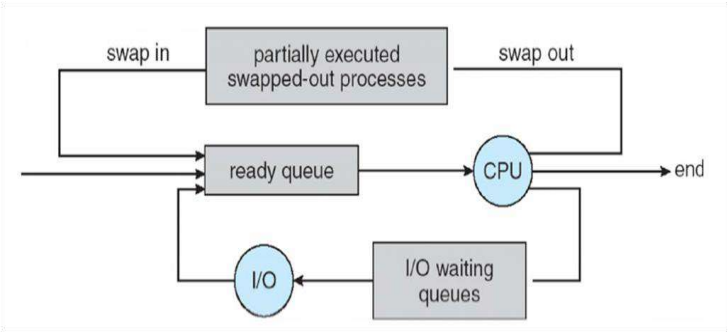
Short-Term Scheduler

- The **short-term scheduler**, or **CPU scheduler**, selects from among the processes that are ready to execute, and allocates the CPU to one of them. It is invoked very frequently.
- Processes can be described as either **I/O bound** or **CPU bound**.
- An **I\O-bound process** spends more of its time doing I/O than it spends doing computations.
- A **CPU-bound process**, on the other hand, generates I/O requests infrequently, using more of its time doing computation than an I/O-bound process uses.
- The system with the best performance will have a combination of CPU- bound and I/O-bound processes.

Medium Term Scheduler

- Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling.
- The key idea is medium-term scheduler, removes processes from memory and thus reduces the degree of multiprogramming.

- At some later time, the process can be reintroduced into memory and its execution can be continued where it left off. This scheme is called swapping.



Context Switching

- Switching the CPU to another process requires saving the state of the old process and loading the saved state for the new process.This task is known as a context switch.
- Context-switch time is pure overhead, because the system does no useful work while switching.
- Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions.

OPERATIONS ON PROCESS

1. Process Creation

A process may create several new processes, during execution.

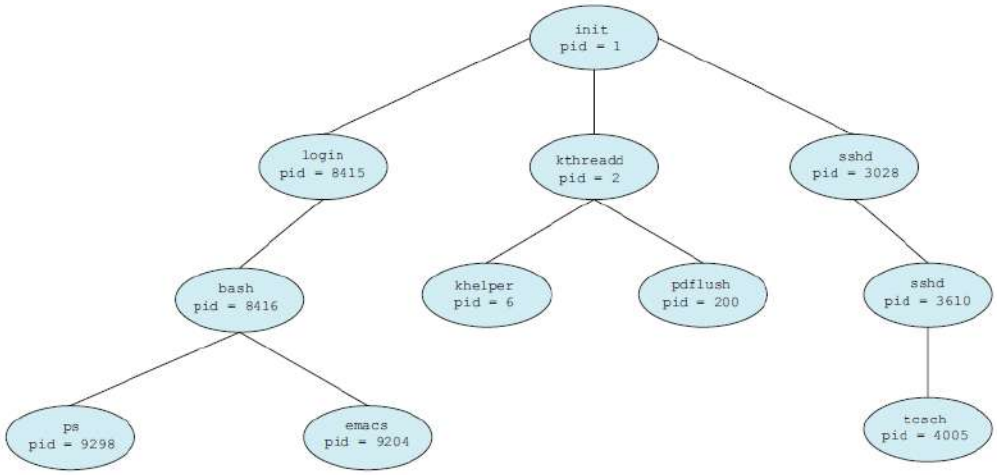
The creating process is called a **parent** process, whereas the new processes are called the **children** of that process.

When a process creates a new process, two possibilities exist **in terms of execution:**

1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

There are also two possibilities **in terms of the address space** of the new process:

1. The child process is a duplicate of the parent process.



2. The child process has a program loaded into it.

In UNIX, each process is identified by its process identifier, which is a unique integer. A new process is created by the **fork** system call.

A tree of processes on a typical Linux system.

we see two children of **init**—kthreadd and **sshd**.

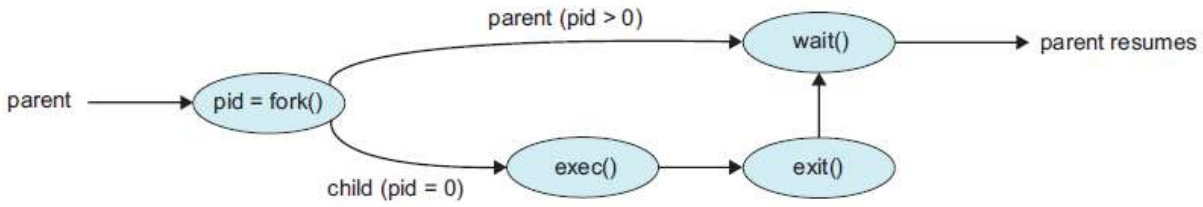
The kthreadd process is responsible for creating additional processes that perform tasks on behalf of the kernel (in this situation, khelper and pdflush).

The sshd process is responsible for managing clients that connect to the system by using ssh (which is short for *secure shell*). The login process is responsible for managing clients that directly log onto the system.

In general, when a process creates a child process, that child process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task.

A child process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process.

The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children. Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many child processes.



2. Process Termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the **exit** system call.

At that point, the process may return data (output) to its parent process (via the **wait** system call).

A process can cause the termination of another process via an appropriate system call.

A parent may terminate the execution of one of its children for a variety of reasons, such as these:

1. The child has exceeded its usage of some of the resources that it has Been allocated.

2. The task assigned to the child is no longer required.
3. The parent is exiting, and the operating system does not allow a child to continue if its parent terminates. On such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as cascading **termination**, is normally initiated by the operating system.

When a process terminates, its resources are de-allocated by the operating system.

A process that has terminated, but whose parent has not yet called wait(), is known as a zombie process.

Now consider what would happen if a parent did not invoke wait() and instead terminated, thereby leaving its child processes as orphans.

CO-OPERATING PROCESS

Processes executing concurrently in the operating system may be either **independent processes** or **cooperating processes**.

A process is independent if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent.

A process is cooperating if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

There are several reasons for providing an environment that allows process cooperation:

• **Information sharing.** Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.

• **Computation speedup**. If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing cores.

• **Modularity.** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.

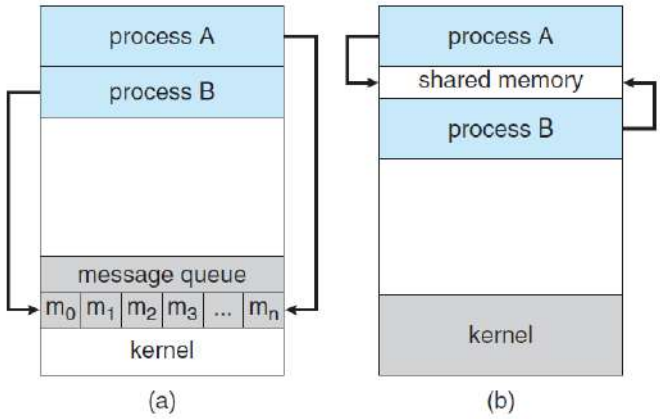
• **Convenience.** Even an individual user may work on many tasks at the same time. For instance, a user may be editing, listening to music, and compiling in parallel.

INTERPROCESS COMMUNICATION

Cooperating processes require an **Inter Process Communication (IPC)** mechanism that will allow them to exchange data and information. There are two fundamental models of interprocess communication: **shared memory** and **message passing**.

In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.

In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.



(a) Message passing. (b) Shared memory.

Shared-Memory Systems

Interprocess communication using shared memory requires communicating processes to establish a region of shared memory.

Other processes that wish to communicate using this shared-memory segment must attach it to their address space.

Message passing

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.

1. Basic Structure:

If processes P and Q want to communicate, they must send messages to and receive messages from each other; a communication link must exist between them.

Physical implementation of the link is done through a hardware bus , network etc,

There are several methods for logically implementing a link and the operations:

1. **Direct or indirect communication**
2. **Symmetric or asymmetric communication**
3. **Automatic or explicit buffering**
4. **Send by copy or send by reference**
5. **Fixed-sized or variable-sized messages**

2. Naming

Processes that want to communicate must have a way to refer to each other.

They can use either direct or indirect communication.

1. **Direct Communication**

Each process that wants to communicate must explicitly name the recipient or sender of the communication.

A communication link in this scheme has the following properties:

i. A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.

ii. A link is associated with exactly two processes.

iii. Exactly one link exists between each pair of processes.

There are two ways of addressing namely

Symmetry in addressing

Asymmetry in addressing

In symmetry in addressing, the send and receive primitives are defined as:

send(P, message) Send a message to process P

receive(Q, message) Receive a message from Q

In asymmetry in addressing , the send & receive primitives are defined as:

send (p, message) send a message to process p

receive(id, message) receive message from any process

2. Indirect Communication

With indirect communication, the messages are sent to and received from mailboxes, or ports.

The send and receive primitives are defined as follows:

send (A, message) Send a message to mailbox A.

receive (A, message) Receive a message from mailbox A.

A communication link has the following properties:

i. A link is established between a pair of processes only if both members of the pair have a shared mailbox.

ii. A link may be associated with more than two processes.

iii. A number of different links may exist between each pair of communicating processes, with each link corresponding to one mailbox.

3. Buffering

A link has some capacity that determines the number of message that can reside in it temporarily. This property can be viewed as a queue of messages attached to the link.

There are three ways that such a queue can be implemented.

Zero capacity : Queue length of maximum is 0. No message is waiting in a queue. The sender must wait until the recipient receives the message.

Bounded capacity: The queue has finite length n. Thus at most n messages can reside in it.

Unbounded capacity: The queue has potentially infinite length. Thus any number of messages can wait in it. The sender is never delayed

4. Synchronization

Message passing may be either blocking or non-blocking.

1. **Blocking Send** - The sender blocks itself till the message sent by it is received by the receiver.
2. **Non-blocking Send** - The sender does not block itself after sending the message but continues with its normal operation.
3. **Blocking Receive** - The receiver blocks itself until it receives the message.
4. **Non-blocking Receive** – The receiver does not block itself.

THREADS

Thread

A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack.

It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals. Traditional (or heavyweight) process has a single thread of control.

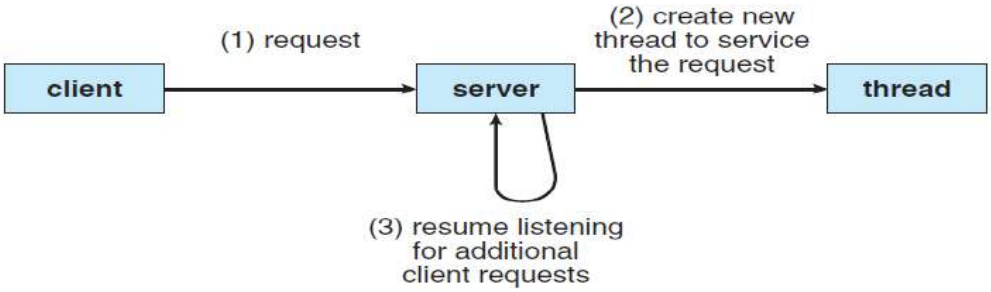
If a process has multiple threads of control, it can perform more than one task at a time.

Motivation

Most software applications that run on modern computers are multithreaded. An application typically is implemented as a separate process with several threads of control.

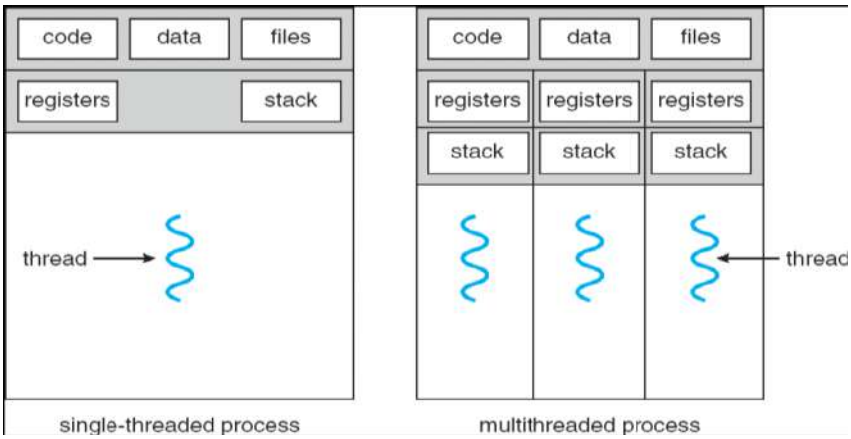
A web browser might have one thread display images or text while another thread retrieves data from the network.

A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.



MULTITHREADING

Multithreading is the ability of a program or an operating system process to manage its use by more than one user at a time and to even manage multiple requests by the same user without having to have multiple copies of the programming running in the computer.



Benefits

There are four major categories of benefits to multi-threading:

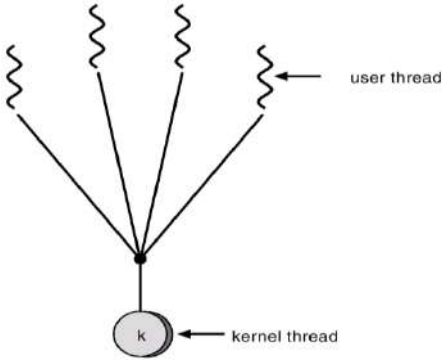
1. **Responsiveness** - One thread may provide rapid response while other threads are blocked or slowed down doing intensive calculations.
2. **Resource sharing** - By default threads share common code, data, and other resources, which allows multiple tasks to be performed simultaneously in a single address space.
3. **Economy -** Creating and managing threads (and context switches between them) is much faster than performing the same tasks for processes.
4. **Scalability**, i.e. Utilization of multiprocessor architectures - A single threaded process can only run on one CPU, no matter how many may be available, whereas the execution of a multi-threaded application may be split amongst available processors

Multithreading Models

1. Many-to-One
2. One-to-One
3. Many-to-Many

1. Many-to-One:

Many to one model maps many user level threads to one Kernel level thread. Thread management is done in user space. When thread makes a blocking system call, the entire process will be blocks. Only one thread can access the Kernel at a time,so multiple threads are unable to run in parallel on multiprocessors.



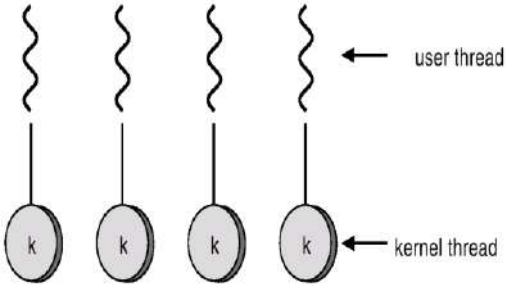
If the user level thread libraries are implemented in the operating system in such a way that system does not support them then Kernel threads use the many to one relationship modes.

2.One-to-One:

There is one to one relationship of user level thread to the kernel level thread.

This model provides more concurrency than the many to one model.

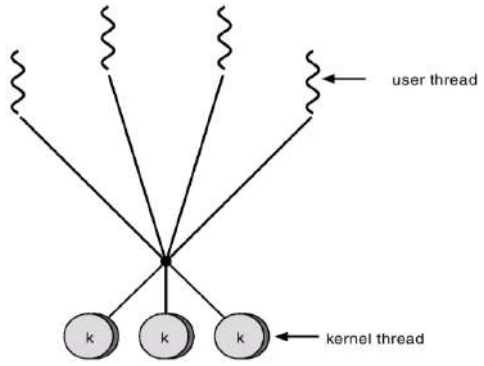
It also another thread to run when a thread



makes a blocking system call. It supports multiple thread to execute in parallel on microprocessors.

3.Many-to-Many Model:

In this model, many user level threads multiplexes to the Kernel thread of smaller or equal numbers.



The number of Kernel threads may be specific to either a particular application or a particular machine.

In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallels on a multiprocessor.

THREADING ISSUES:

1. **fork() and exec() system calls.**

   A fork () system call may duplicate all threads or duplicate only the thread that invoked fork().

   If a thread invoke exec() system call ,the program specified in the parameter to exec will replace the entire process.

2. **Thread cancellation.**

   It is the task of terminating a thread before it has completed . A thread that is to be cancelled is called a target thread.

   There are two types of cancellation namely

   1. **Asynchronous Cancellation** – One thread immediately terminates the target thread.
   2. **Deferred Cancellation** – The target thread can periodically check if it should terminate , and does so in an orderly fashion.

3. **Signal handling**

   1. A signal is a used to notify a process that a particular event has occurred.
   2. A generated signal is delivered to the process.
      a. Deliver the signal to the thread to which the signal applies. b. Deliver the signal to every thread in the process.
      c. Deliver the signal to certain threads in the process.
      d. Assign a specific thread to receive all signals for the process.
   3. Once delivered the signal must be handled. a. Signal is handled by
      i. A default signal handler
      ii. A user defined signal handler

4. **Thread pools**

   - Creation of unlimited threads exhaust system resources such as CPU time or memory. Hence we use a thread pool.
   - In a thread pool , a number of threads are created at process startup and placed in the pool.
   - When there is a need for a thread the process will pick a thread from the pool and assign it a task.

- After completion of the task,the thread is returned to the pool.

5. **Thread specific data**

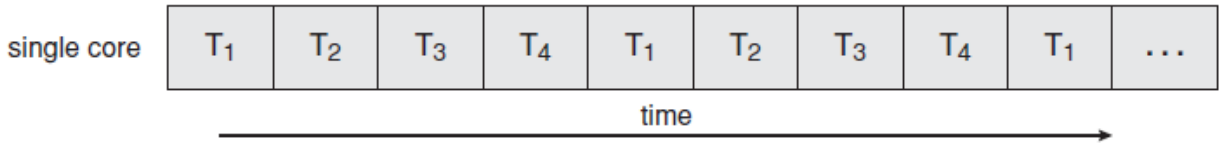
   Threads belonging to a process share the data of the process. However each thread might need its own copy of certain data known as thread-specific data.

MULTICORE PRORGAMMING

Single-CPU systems evolved into multi-CPU systems. A more recent, similar trend in system design is to place multiple computing cores on a single chip.

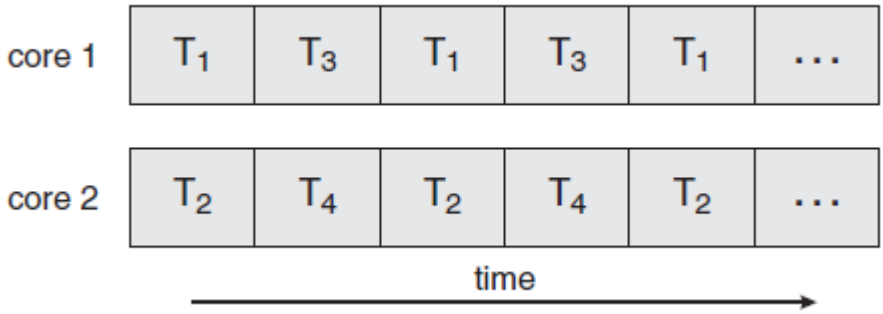
Each core appears as a separate processor to the operating system. Whether the cores appear across CPU chips or within CPU chips, we call these systems multicore or multiprocessor systems.

Multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency.



A system is parallel if it can perform more than one task simultaneously.

A concurrent system supports more than one task by allowing all the tasks to make progress. Thus, it is possible to have concurrency without parallelism.



In general, five areas present challenges in programming for multicore systems:

1. **Identifying tasks**. This involves examining applications to find areas that can be divided into separate, concurrent tasks.

2. **Balance.** While identifying tasks that can run in parallel, programmers must also ensure that the tasks perform equal work of equal value.

3. **Data splitting**. Just as applications are divided into separate tasks, the data accessed and manipulated by the tasks must be divided to run on separate cores.

4. **Data dependency**. The data accessed by the tasks must be examined for dependencies between two or more tasks. When one task depends on data from another, programmers must ensure that the execution of the tasks is synchronized to accommodate the data dependency.

5. **Testing and debugging**. When a program is running in parallel on multiple cores, many different execution paths are possible. Testing and debugging such concurrent programs is inherently more difficult than testing and debugging single-threaded applications.

Types of Parallelism

In general, there are two types of parallelism: **data parallelism and task parallelism**.

Data parallelism focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core.

Task parallelism involves distributing not data but tasks (threads) across multiple computing cores.

PROCESS SYNCHRONIZATION

 Concurrent access to shared data may result in data inconsistency.

 Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

 Shared-memory solution to bounded-butter problem allows at most $n - 1$ items in buffer at the same time. A solution, where all N buffers are used is not simple.

 Suppose that we modify the producer-consumer code by adding a variable *counter*, initialized to 0 and increment it each time a new item is added to the buffer

 Race condition: The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.

 To prevent race conditions, concurrent processes must be synchronized.

THE CRITICAL-SECTION PROBLEM

Definition: Each process has a segment of code, called a critical section (CS), in which the process may be changing common variables, updating a table, writing a file, and so on.

 The important feature of the system is that, when one process is executing in its CS, no other process is to be allowed to execute in its CS.

 That is, no two processes are executing in their CSs at the same time.

 Each process must request permission to enter its CS. The section of code implementing this request is the entry section.

 The CS may be followed by an exit section.

 The remaining code is the remainder section.

Requirements to be satisfied for a Solution to the Critical-Section Problem:

1. **Mutual Exclusion -** If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress -** If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
3. **Bounded Waiting -** A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

General structure of process P_i

```
{
        entry section
        critical section

        exit section

        remainder section

} while (1);
```

Two general approaches are used to handle critical sections in operating systems: preemptive kernels and nonpreemptive kernels.

 A preemptive kernel allows a process to be preempted while it is running in kernel mode.

 A non-preemptive kernel does not allow a process running in kernel mode to be preempted; a kernelmode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

MUTEX LOCKS

 Operating-systems designers build software tools to solve the critical-section problem. The simplest of these tools is the mutex lock.

 We use the mutex lock to protect critical regions and thus prevent race conditions.

 That is, a process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section.

 The acquire()function acquires the lock, and the release() function releases the lock.
Solution to the critical-section problem using mutex locks.

```
do {
        acquire lock
        critical section
        release lock
        remainder section
} while (true);
```

 A mutex lock has a boolean variable available whose value indicates if the lock is available or not.
 If the lock is available, a call to acquire() succeeds, and the lock is then considered unavailable.
 A process that attempts to acquire an unavailable lock is blocked until the lock is released.

 The definition of acquire() is as follows:

```
        acquire()
        {
                while (!available); /* busy wait */
                available = false;;
        }
```

 The definition of release() is as follows:

```
                release()
                {
                        available = true;
                }
```

 Calls to either acquire() or release() must be performed atomically. Thus, mutex locks are often implemented using one of the hardware mechanisms.

Disadvantage of the implementation given here is that it requires busy waiting.

 While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire().

 In fact, this type of mutex lock is also called a spinlock because the process "spins" while waiting for the lock to become available.

 This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively.

SEMAPHORES

 A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations:
wait() and
signal().
 The wait() operation was originally termed P (from the Dutch proberen, "to test"); signal() was originally called V (from verhogen, "to increment").

 The definition of wait() is as follows:

```
wait(S)
{
while (S <= 0); // busy wait
S--;
}
```

 The definition of signal() is as follows:

```
signal(S)
{
S++;
}
```

Semaphore Usage

 The value of a counting semaphore can range over an unrestricted domain. The value of a binary semaphore can range only between 0 and 1.

 Binary semaphores behave similarly to mutex locks.

 On systems that do not provide mutex locks, binary semaphores can be used instead for providing mutual exclusion.

 Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.

 The semaphore is initialized to the number of resources available.

 Each process that wishes to use a resource performs a **wait()** operation on the semaphore (thereby decrementing the count).

 When a process releases a resource, it performs a **signal()** operation (incrementing the count).

When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0

 We can also use semaphores to solve various synchronization problems.

 For example, consider two concurrently running processes: P1 with a statement S1 and P2 with a statement S2. Suppose we require that S2 be executed only after S1 has completed. We can implement this scheme readily by letting P1 and P2 share a common semaphore synch, initialized to 0. In process P1, we insert the statements

```
S1;
signal(synch);
```

In process P2, we insert the statements

```
wait(synch);
S2;
```

 Because synch is initialized to 0, P2 will execute S2 only after P1 has invoked signal(synch), which is after statement S1 has been executed.

Semaphore Implementation

 To overcome the need for busy waiting, we can modify the definition of the wait() and

signal() operations as follows: When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait.

 Rather than engaging in busy waiting, the process can block itself.

 The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state.

 Then control is transferred to the CPU scheduler, which selects another process to execute.
 A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation.

 The process is restarted by a wakeup() operation, which changes the process from the waiting state to the ready state.
 The process is then placed in the ready queue. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.)

 To implement semaphores under this definition, we define a semaphore as follows:

```
typedef struct
{
int value;
struct process *list;
} semaphore;
```

 Each semaphore has an integer value and a list of processes list.

 When a process must wait on a semaphore, it is added to the list of processes.
A signal() operation removes one process from the list of waiting processes and awakens that process.

 The wait() semaphore operation can be defined as

```
wait(semaphore *S)
{
S->value--;
if (S->value < 0)
{
add this process to S->list;
block();
}
}
```

 The signal() semaphore operation can be defined as

```
signal(semaphore *S)
{
S->value++;
if (S->value <= 0)
{
remove a process P from S->list;
wakeup(P);
}
```

}

 The block() operation suspends the process that invokes it.

 The wakeup(P) operation resumes the execution of a blocked process P.

Deadlocks and Starvation

 The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes

.

 When such a state is reached, these processes are said to be deadlocked

 To illustrate this, consider a system consisting of two processes, P0 and P1, each accessing two semaphores, S and Q, set to the value 1:

P0	P1
wait(S);	wait(Q);
wait(Q);	wait(S);
. .	..
. .	..
. .	..
signal(S);	signal(Q);
signal(Q);	signal(S);

 Suppose that P0 executes wait(S) and then P1 executes wait(Q).When P0 executes wait(Q), it must wait until P1 executes signal(Q).

 Similarly, when P1 executes wait(S), it must wait until P0 executes signal(S).

 Since these signal() operations cannot be executed, P0 and P1 are deadlocked.

 We say that a set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set.

 Another problem related to deadlocks is indefinite blocking or starvation, a situation in which processes wait indefinitely within the semaphore.

 Indefinite blocking may occur if we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

Priority Inversion

 A scheduling challenge arises when a higher-priority process needs to read or modify kernel data that are currently being accessed by a lower-priority process—or a chain of lower-priority processes.

 The kernel data are typically protected with a lock, the higher-priority process will have to wait for a lower-priority one to finish with the resource.

 The situation becomes more complicated if the lower-priority process is preempted in favor

of another process with a higher priority.

 This problem is known as priority inversion. It occurs only in systems with more than two priorities, so one solution is to have only two priorities.

 Typically these systems solve the problem by implementing a priority-inheritance protocol. According to this protocol, all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question.

 When they are finished, their priorities revert to their original values. In the example above, a priority-inheritance protocol would allow process L to temporarily inherit the priority of process.

CLASSIC PROBLEMS OF SYNCHRONIZATION

1. Bounded Buffer Problem
2. Reader Writer Problem
3. Dining Philosopher's Problem

The Bounded-Buffer Problem

 We assume that the pool consists of n buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1.

 The empty and full semaphores count the number of empty and full buffers.

 The semaphore empty is initialized to the value n.

 The semaphore full is initialized to the value 0.

The producer and consumer processes share the following data structures:

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

The structure of the producer process.

```
do {
. . .
/* produce an item in next produced */
. . .
wait(empty);
wait(mutex);
. . .
/* add next produced to the buffer */
. . .
signal(mutex);
signal(full);
```

```
} while (true);
```

The structure of the consumer process.

```
do {
wait(full);
wait(mutex);
. . .
/* remove an item from buffer to next consumed */
. . .
signal(mutex);
signal(empty);
. . .
/* consume the item in next consumed */
. . .
} while (true);
```

 We can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

Reader Writer Problem

The R-W problem is another classic problem for which design of synchronization and concurrency mechanisms can be tested. The producer/consumer is another such problem; the dining philosophers is another.

Definition

- There is a data area that is shared among a number of processes.
- Any number of readers may simultaneously write to the data area.
- Only one writer at a time may write to the data area.
- If a writer is writing to the data area, no reader may read it.
- If there is at least one reader reading the data area, no writer may write to it.
- Readers only read and writers only write
- A process that reads and writes to a data area must be considered a writer (consider producer or consumer)

In the solution to the first readers–writers problem, the reader processes share the following data structures:

```
semaphore rw mutex = 1;
semaphore mutex = 1;
int read count = 0;
```

 The semaphores mutex and rw mutex are initialized to 1; read count is initialized to 0.
 The semaphore rw mutex is common to both reader and writer processes.
 The mutex semaphore is used to ensure mutual exclusion when the variable read count is updated.
 The read count variable keeps track of how many processes are currently reading the object.
 The semaphore rw mutex functions as a mutual exclusion semaphore for the writers.

The structure of a writer process.

```
do {
```

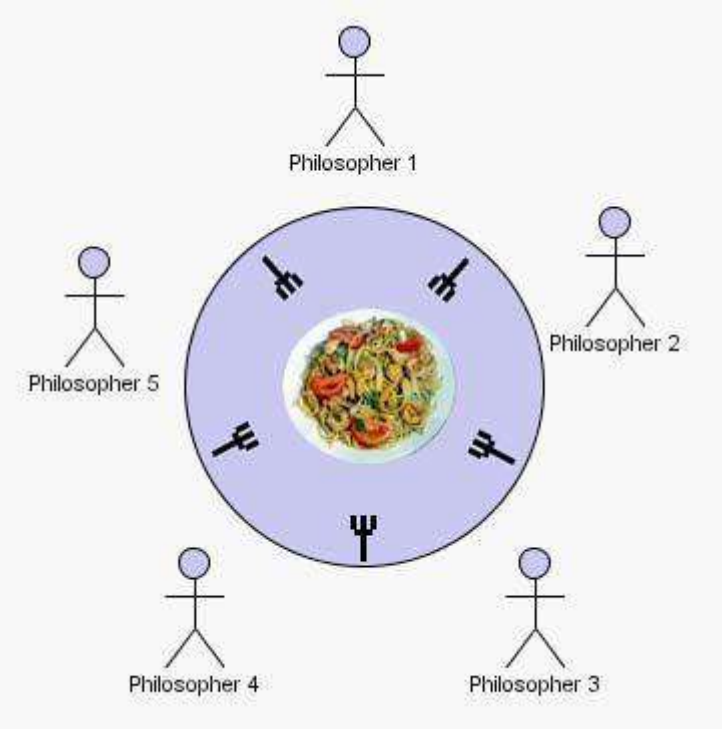
```
        wait(rw mutex);
        . . .
        /* writing is performed */
        . . .
        signal(rw mutex);
        } while (true);
```

The structure of a reader process.

```
        do {
        wait(mutex);
        readcount++;
        if (read count == 1)
        wait(rw mutex);
        signal(mutex);
        . . .
        /* reading is performed */
wait(mutex);
read count--;
if (read count == 0)
signal(rw mutex);
signal(mutex);
} while (true);
```

Dining Philosophers Problem

Consider there are five philosophers sitting around a circular dining table. The dining table has five chopsticks and a bowl of rice in the middle.



At any instant, a philosopher is either eating or thinking. When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right.

When a philosopher wants to think, he keeps down both chopsticks at their original place.

- When a philosopher thinks, he does not interact with his others.
- From time to time, a philosopher gets hungry and tries to pick up the two forks that are closest to him (the forks that are between him and his left and right neighbors).
- A philosopher may pick up only one fork at a time. Obviously, he cannot pick up a fork that is already in the hand of a neighbor.
- When a hungry philosopher has both his forks at the same time, he eats without releasing his forks.
- When he is finished eating, he puts down both of his forks and starts thinking again.

Solution:

From the problem statement, it is clear that a philosopher can think for an indefinite amount of time. But when a philosopher starts eating, he has to stop at some point of time. The philosopher is in an endless cycle of thinking and eating.

An array of five semaphores, **stick[5]**, for each of the five chopsticks.

The code for each philosopher looks like:

```
while(TRUE) {
wait(stick[i]);
wait(stick[(i+1) % 5]);  // mod is used because if i=5, next
                         // chopstick is 1 (dining table is circular)
/* eat */
signal(stick[i]);
signal(stick[(i+1) % 5]);
/* think */
}
```

When a philosopher wants to eat the rice, he will wait for the chopstick at his left and picks up that chopstick. Then he waits for the right chopstick to be available, and then picks it too. After eating, he puts both the chopsticks down.

But if all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever.

The possible solutions for this are:

1) A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.

2) Allow only four philosophers to sit at the table. That way, if all the four philosophers pick up four chopsticks, there will be one chopstick left on the table. So, one philosopher can start eating and eventually, two chopsticks will be available. In this way, deadlocks can be avoided.

MONITORS

Definition: Monitor is a high-level language construct with a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package.

 Processes may call the procedures in a monitor whenever they want to, but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor.

 Monitors have an important property that makes them useful for achieving mutual exclusion: only one process can be active in a monitor at any instant.

Monitor Usage

 A monitor type presents a set of programmer-defined operations that are provided mutual exclusion within the monitor.

 The monitor type also contains the declaration of variables whose values define the state of an instance of that type, along with the bodies of procedures or functions that operate on those variables.

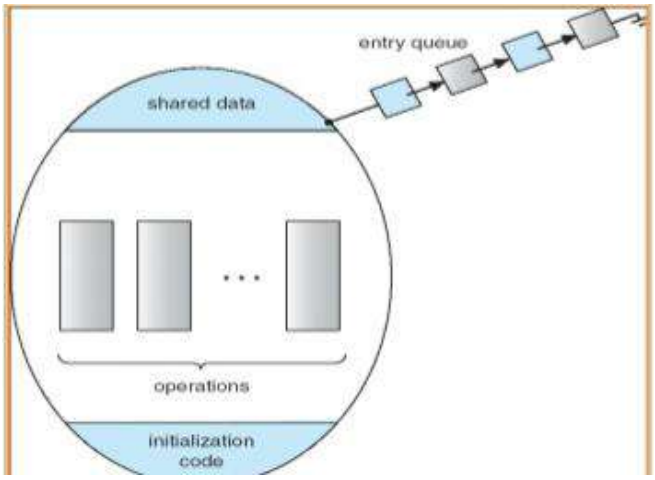
```
monitor monitor name
{
/* shared variable declarations */
function P1 ( . . . ) {
. . .
}
function P2 ( . . . ) {
. . .
}
.
.
function Pn ( . . . ) {
. . .
}
initialization code ( . . . ) {
. . .
}
}
```

 The representation of a monitor type cannot be used directly by the various processes. Thus, a procedure defined within a monitor can access only those variables declared locally within the monitor and its formal parameters.

 Similarly, the local variables of a monitor can be accessed by only the local procedures.

 The monitor construct ensures that only one process at a time can be active within the monitor.

Schematic view of a Monitor



The monitor construct is not sufficiently powerful for modeling some synchronization schemes.
 For this purpose, we need to define additional synchronization mechanisms. These mechanisms are provided by the condition construct condition x, y;

The only operations that can be invoked on a condition variable are wait() and signal(). The operation

x.wait();

means that the process invoking this operation is suspended until another process invokes

x.signal();

The x.signal() operation resumes exactly one suspended process.

A monitor solution to the dining-philosopher problem.

```
monitor DiningPhilosophers
{
enum {THINKING, HUNGRY, EATING} state[5];
condition self[5];
void pickup(int i)
{
state[i] = HUNGRY;
test(i);
if (state[i] != EATING)
self[i].wait();
}
void putdown(int i)
{
state[i] = THINKING;
test((i + 4) % 5);
```

```
test((i + 1) % 5);
}
void test(int i)
{
if ((state[(i + 4) % 5] != EATING) && (state[i] == HUNGRY) && (state[(i + 1) % 5] != EATING))
{
state[i] = EATING;
self[i].signal();
}
}
initialization code()
{
for (int i = 0; i < 5; i++)
state[i] = THINKING;
}
```

CPU SCHEDULING

CPU scheduling is the basis of multi-programmed operating systems.

By switching the CPU among processes, the operating system can make the computer more productive.

Basic Concepts

- The objective of multi-programming is to have some process running at all times, to maximize CPU utilization.
- For a Uni-processor system, there will never be more than one running process.
- Scheduling is a fundamental operating system function.
- The idea of multi-programming is to execute a process until it must wait, typically for the completion of some I/O request.
- The CPU is one of the primary computer resources.
- The CPU scheduling is central to operating system design.

Cpu Scheduler

- When the CPU becomes idle, the operating system must select on of the processes in the ready queue to be executed.
- The selection process is carried out by the short-term scheduler (CPU scheduler)
- The scheduler selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.
- A ready queue may be implemented as a FIFO queue, a priority queue, a tree or simply an unordered link list.
- All the processes in the ready queue are lined up waiting for a chance to run on the CPU.

CPU scheduling decisions may take place when a process.

1. Switches from running to waiting state
2. Switches from running to ready state
3. Switches from waiting to ready

4. Terminates

Scheduling under 1 and 4 is **nonpreemptive.**
All other scheduling is **preemptive.**

Nonpreemptive Scheduling➔A scheduling discipline is non preemptive if, once a process has been given the CPU, the CPU cannot be taken away from that process.

Preemptive Scheduling ➔ A scheduling discipline is preemptive if, once a process has been given the CPU can taken away.

Dispatcher

Dispatcher is a module that gives control of the CPU to the process selected by the short-term scheduler. This function involves the following:

- switching context.
- switching to user mode.
- jumping to the proper location in the user program to restart that program.

Dispatch latency – The time taken for the dispatcher to stop one process and start another running.

Scheduling criteria

1. **CPU utilization** – keep the CPU as busy as possible Throughput – # of processes that complete their execution per time unit .
2. **Turnaround time** – amount of time to execute a particular process
3. **Waiting time** – amount of time a process has been waiting in the ready queue
4. **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)
5. **Throughput** – The number of processes that complete their execution per time unit.

Best Algorithm consider following:
- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

Formulas to calculate Turn-around time & waiting time is:
Waiting time = Finishing Time – (CPU Burst time + Arrival Time)
Turnaround time = Waiting Time + Burst Time

Scheduling Algorithms

A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms.

1. First-Come, First-Served (FCFS) Scheduling
2. Shortest-Job-First (SJF) Scheduling
3. Priority Scheduling
4. Round Robin(RR) Scheduling

First-Come, First-Served (FCFS) Scheduling algorithm.

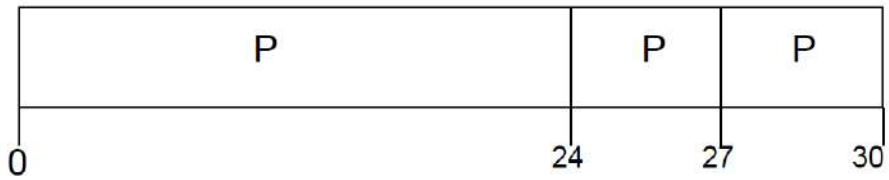
- This is the simplest CPU-scheduling algorithm.
- According to this algorithm, the process that requests the CPU first is allocated the CPU first.
- The implementation of FCFS is easily managed with a FIFO queue.
- When a process enters the ready queue, its PCB is linked onto the tail of the queue.
- When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue.

Example Problem

Consider the following set of processes that arrive at time 0, with the length of the CPU burst time given in milliseconds:

Process	Burst Time(ms)
P1	24
P2	3
P3	3

Suppose that the processes arrive in the order: P1 , P2 , P3 The Gantt Chart:



Waiting time

- Waiting time for P1 = 0; P2 = 24; P3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17 ms.

Turnaround Time = Waiting Time + Burst Time

- Turnaround Time for P1 = (0+24)=24; P2 = (24+3)=27; P3 = (27+3)=30
- Average Turnaround Time = (24+27+30)/3 = 27 ms

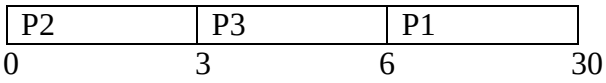
Shortest-Job-First (SJF) Scheduling

- This algorithm associates with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.
- When the CPU is available, it is assigned to the process that has the smallest next CPU burst. It is also called as shortest next CPU burst.

 If two processes have the same length next CPU burst, FCFS scheduling is used to break the tie.

Process	Burst Time
P1	24
P2	3
P3	3

Gantt Chart



Waiting time

For P1=6,P2=0,P3=3

Average Waiting Time=(6+0+3)/3=3 ms.

Turnaround Time = Waiting Time + Burst Time

Turnaround Time for P1=(6+24)=30,P2=(0+3)=3,P3=(3+3)=6

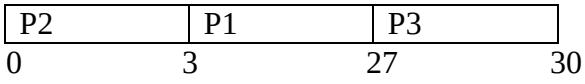
Average Turnaround Time=(30+3+6)/3=13ms

Priority Scheduling

- The SJF algorithm is a special case of the general priority-scheduling algorithm.
- A priority number (integer) is associated with each process and the CPU is allocated to the process with the highest priority.
- Equal-priority processes are scheduled in FCFS order.
- The CPU is allocated to the process with the highest priority (smallest integer º highest priority) .

Process	Burst Time	Priority
P1	24	2
P2	3	1
P3	3	3

Gantt Chart



Waiting time

For P1=3,P2=0,P3=27

Average Waiting Time=(3+0+27)/3=10ms

Turnaround Time = Waiting Time + Burst Time

Turnaround Time for P1=(3+24)=27,P2=(0+3)=3,P3=(27+3)=30

Average Turn Around Time=(27+3+30)/3=20ms.

Round robin scheduling

- Round robin scheduling is designed especially for time-sharing systems.
- It is similar to FCFS, but preemption is added to switch between processes.
- Each process gets a small unit of CPU time called a time quantum or

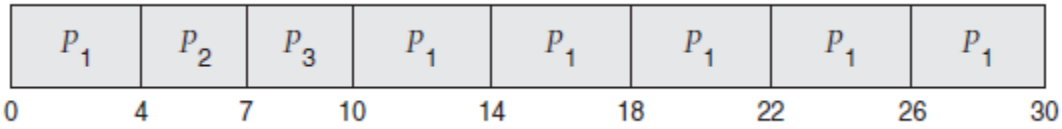
time slice.

- To implement RR scheduling, the ready queue is kept as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum and dispatches the process.

- If the CPU burst time is less than the time quantum, the process itself will release the CPU voluntarily. Otherwise, if the CPU burst of the currently running process is longer than the time quantum a context switch will be executed and the process will be put at the tail of the ready queue.

Process	**Burst Time**
P_1	24
P_2	3
P_3	3

Gantt Chart



Waiting time

Average waiting time = [6+4+7]/3 = 17/3 =5.66

Turnaround Time = Waiting Time + Burst Time

Turnaround Time for P1=(6+24)=30,P2=(4+3)=7,P3=(7+3)=10

Average Turnaround Time=(30+7+10)/3=15.6ms.

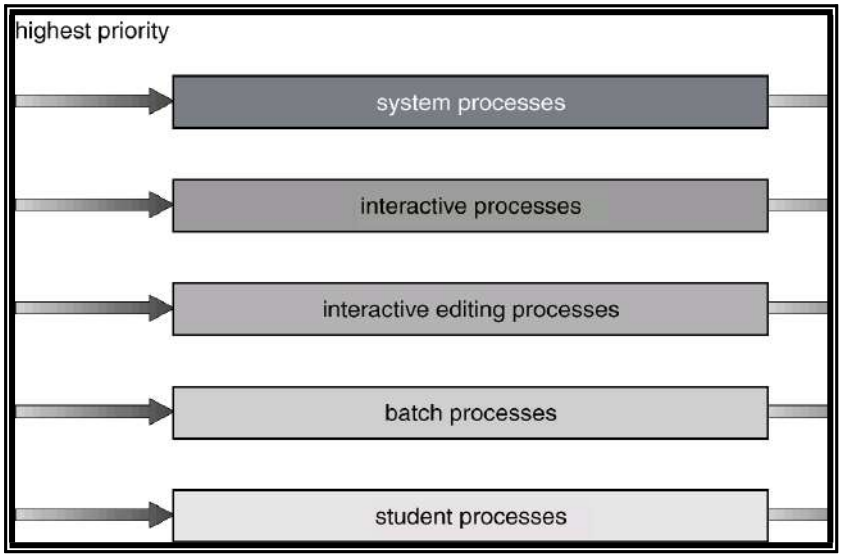
Multilevel Queue Scheduling

- It partitions the ready queue into several separate queues .
- The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type.
- There must be scheduling between the queues, which is commonly implemented as a fixed-priority preemptive scheduling.
- For example the foreground queue may have absolute priority over the background queue.

Example : of a multilevel queue scheduling algorithm with five queues

1. System processes
2. Interactive processes
3. Interactive editing processes
4. Batch processes
5. Student processes

Each queue has absolute priority over lower-priority queue.



Multilevel Feedback Queue Scheduling

- It allows a process to move between queues.
- The idea is to separate processes with different CPU-burst characteristics.
- If a process uses too much CPU time, it will be moved to a lower-priority queue.
- This scheme leaves I/O-bound and interactive processes in the higher- priority queues.
- Similarly, a process that waits too long in a lower priority queue may be moved to a higher-priority queue.
- This form of aging prevents starvation.

Example:

- Consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2 .

- The scheduler first executes all processes in queue 0.
- Only when queue 0 is empty will it execute processes in queue 1.
- Similarly, processes in queue 2 will be executed only if queues 0 and 1 are empty.
- A process that arrives for queue 1 will preempt a process in queue 2.
- A process that arrives for queue 0 will, in turn, preempt a process in queue 1.

- A multilevel feedback queue scheduler is defined by the following parameters:

  1. The number of queues
  2. The scheduling algorithm for each queue
  3. The method used to determine when to upgrade a process to a higher priority queue
  4. The method used to determine when to demote a process to a lower-priority queue
  5. The method used to determine which queue a process will enter when that process needs service

Multiple Processor Scheduling

- If multiple CPUs are available, the scheduling problem is correspondingly more complex.
- If several identical processors are available, then load-sharing can occur.
- It is possible to provide a separate queue for each processor.
- In this case however, one processor could be idle, with an empty queue, while another processor was very busy.
- To prevent this situation, we use a common ready queue.
- All processes go into one queue and are scheduled onto any available processor.
- In such a scheme, one of two scheduling approaches may be used.

  1. **Self Scheduling** - Each processor is self-scheduling. Each processor examines the common ready queue and selects a process to execute. We must ensure that two processors do not choose the same process, and that processes are not lost from the queue.
  2. **Master – Slave Structure** - This avoids the problem by appointing one processor as scheduler for the other processors, thus creating a master-slave structure.

Real-Time Scheduling

- Real-time computing is divided into two types.

  1. Hard real-time systems
  2. Soft real-time systems

Hard real-time systems

- Hard RTS are required to complete a critical task within a guaranteed amount of time.
- Generally, a process is submitted along with a statement of the amount of time in which it needs to complete or perform I/O.
- The scheduler then either admits the process, guaranteeing that the process will complete on time, or rejects the request as impossible. This is known as **resource reservation**.

Soft real-time systems

- Soft real-time computing is less restrictive. It requires that critical processes recieve

priority over less fortunate ones.

- The system must have priority scheduling, and real-time processes must have the highest priority.
- The priority of real-time processes must not degrade over time, even though the priority of non-real-time processes may.
- Dispatch latency must be small. The smaller the latency, the faster a real- time process can start executing.
- The high-priority process would be waiting for a lower-priority one to finish. This situation is known as **priority inversion**.

DEAD LOCK

Definition:
A process request resources, if the resources are not available at that time, the process enters in to a wait state. It may happen that waiting processes will never again change the state, because the resources they have requested are held by other waiting processes. *This situation is called as **dead lock.***

System Model

- A system consists of a finite number of resources to be distributed among a number of competing processes.

- The resources may be partitioned into several types (or classes), each consisting of some number of identical instances.

- CPUcycles, files,and I/O devices (such as printers and DVD drives) are examples of resource types.

A process must request a resource before using it and must release the resource after using it.

Under the normal mode of operation, a process may utilize a resource in only the following sequence:
1.Request. The process requests the resource. If the request cannot be granted immediately then the requesting process must wait until it can acquire the resource.
2. Use. The process can operate on the resource
3. Release. The process releases the resource.

Deadlock Characterizations:-
In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting.

Necessary Conditions for Deadlock:-

A dead lock situation can arise if the following four conditions hold simultaneously in a system.
1) MUTUAL EXCLUSION:- At least one resource must be held in a on-sharable mode. i.e only

one process can hold this resource at a time . other requesting processes should wait till it is released.

2) HOLD & WAIT:- there must exist a process that is holding at least one resource and is waiting to acquire additional resources that are currently being held by other processes.

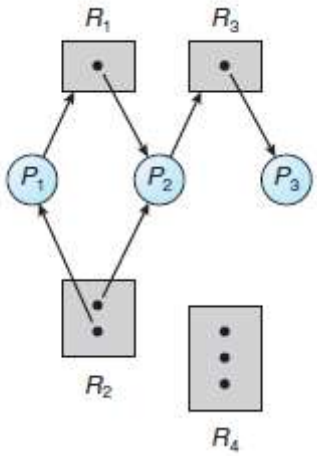
3) NO PREEMPTION:- Resources cannot be preempted, that is a resource can be released voluntarily by the process holding it, after that process has completed its task.

4) CIRCULAR WAIT:- There must exist a set {p0,p1,p1….pn} of waiting processes such that p0 is waiting for a resource that is held by the p1, p1 is waiting for the resource that is held by the p2…. And so on. pn is waiting for a resource that is held by the p0.

Resource-Allocation Graph

A deadlock can be described in terms of a directed graph called system resource-allocation graph.

- A set of vertices V and a set of edges E.
  - V is partitioned into two types:
  - 7 P = {P1, P2, …, Pn}, the set consisting of all the processes in the system.
  - 7 R = {R1, R2, …, Rm}, the set consisting of all resource types in the system.
    - request edge – directed edge Pi → Rj
    - assignment edge – directed edge Rj → P



The resource-allocation graph depicts the following situation.

The sets P, R, and E:

- P = {P1, P2, P3}
- R = {R1, R2, R3, R4}
- E = {P1 → R1, P2 → R3, R1 → P2, R2 → P2, R2 → P1, R3 → P3}

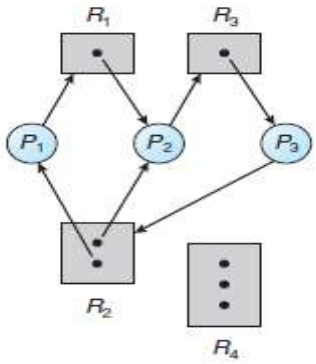
Resource instances:

- One instance of resource type R1
- Two instances of resource type R2
- One instance of resource type R3
- Three instances of resource type R4

Process states:

- Process P1 is holding an instance of resource type R2 and is waiting for an instance of resource type R1.
- Process P2 is holding an instance of R1 and an instance of R2 and is waiting for an instance of R3.
- Process P3 is holding an instance of R3.

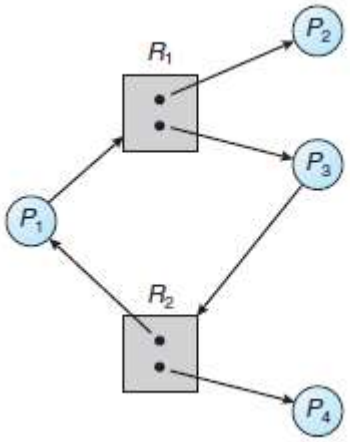
Resource-allocation graph with a deadlock.



- Processes P1, P2, and P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by process P3. Process P3 is waiting for either process P1 or process P2 to release resource R2. In addition, process P1 is waiting for process P2 to release resource R1.
- We also have a cycle: P1 → R1 → P3 → R2 → P1
- If the graph contains no cycles, then no process in the system is deadlocked.

If the graph does contain a cycle, then a deadlock may exist.

Resource-allocation graph with a cycle but no deadlock.



Methods for Handling Deadlocks

We can deal with the deadlock problem in one of three ways:

1. We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state
2. We can allow the system to enter a deadlocked state, detect it, and recover.
3. We can ignore the problem altogether and pretend that deadlocks never occur in the system.

The third solution is the one used by most operating systems, including Linux and Windows.

- **Deadlock prevention** provides a set of methods to ensure that at least one of the necessary conditions cannot hold.
- **Deadlock avoidance** requires that the operating system be given additional information in advance concerning which resources a process will request and use during its lifetime.

DEADLOCK PREVENTION

- For a deadlock to occur, each of the four necessary conditions must hold.
- By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

1. Mutual Exclusion

– not required for sharable resources; must hold for non-sharable resources.
– For example, a printer cannot be simultaneously shared by several processes.
– A process never needs to wait for a sharable resource.

2. Hold and Wait

– must guarantee that whenever a process requests a resource, it does not hold any other resources.

– One protocol requires each process to request and be allocated all its resources before it begins execution,

– Or another protocol allows a process to request resources only when the process has none. So, before it can request any additional resources, it must release all the resources that it is currently allocated.

3. Denying No preemption

– If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.

– Preempted resources are added to the list of resources for which the process is waiting.

– Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

4. Denying Circular wait

- Impose a total ordering of all resource types and allow each process to request for resources in an increasing order of enumeration.
- Let R = {R1,R2,...Rm} be the set of resource types.
- Assign to each resource type a unique integer number.
- If the set of resource types R includes tapedrives, disk drives and printers.

  F(tapedrive)=1,
  F(diskdrive)=5,
  F(Printer)=12.
- Each process can request resources only in an increasing order of enumeration.

DEADLOCK AVOIDANCE

An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested.

Each request requires that in making this decision the system consider

- the resources currently available,
- the resources currently allocated to each process,
- the future requests and releases of each process.

A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist.

The resource-allocation state is defined by the number of available and allocated resources and the maximum demands of the processes.

Safe State

• When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.

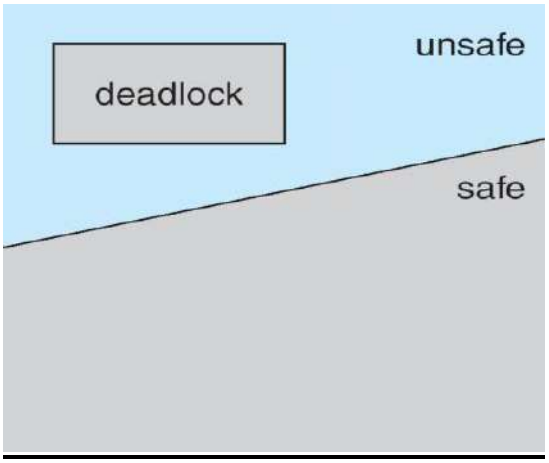
• System is in safe state if there exists a sequence <P1, P2, …, Pn> of ALL the processes is the systems such that for each Pi, the resources that Pi can still request can be satisfied by currently available resources + resources held by all the Pj, with j < i.

• That is:

– If Pi resource needs are not immediately available, then Pi can wait until all Pj have finished.

– When Pj is finished, Pi can obtain needed resources, execute, return allocated resources, and terminate.

– When Pi terminates, Pi +1 can obtain its needed resources, and so on.



Banker's Algorithm

 The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type.

 The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.

Multiple instances.

Each process must a priori claim maximum use.

- When a process requests a resource it may have to wait.
- When a process gets all its resources it must return them in a finite amount of time.
- Let n = number of processes, and m = number of resources types.
    1. **Available:** indicates the number of available resources of each type.
    2. **Max:** Max[i, j]=k then process Pi may request at most k instances of resource type Rj
    3. **Allocation :** Allocation[i. j]=k, then process Pi is currently allocated K instances of resource type Rj
    4. **Need :** if Need[i, j]=k then process Pi may need K more instances of resource type Rj ,Need [i, j]=Max[i, j]-Allocation[i, j]

Need [i,j] = Max[i,j] – Allocation [i,j].

Safety algorithm

1. Initialize work := available and Finish [i]:=false for i=1,2,3 .. n

2.Find an i such that both

a. Finish[i]=false

b. Needi<= Work if no such i exists, goto step 4

3. work :=work+ allocationi; Finish[i]:=true goto step 2

4. If finish[i]=true for all i, then the system is in a safe state

Example:

Given the following state for the Banker's Algorithm.

5 processes P0 through P4

3 resource types A (6 instances), B (9 instances) and C (5 instances).

Snapshot at time T0:

	Max	*Allocation*
	A B C	*A B C*
P_0	6 7 3	1 1 1
P_1	2 2 2	1 1 2
P_2	2 6 3	0 3 0
P_3	2 2 2	2 1 1
P_4	4 6 3	1 1 1

a) Calculate the available vector.

b) Calculate the Need matrix.

c) Is the system in a safe state? If so, show one sequence of processes which allows the system to complete. If not, explain why.

d) Given the request (1, 2, 0) from Process P2. Should this request be granted? Why or why not?

a) Calculate the available vector.

Available
A B C
1 2 0

b) Calculate the Need matrix.

	Need
	A B C
P_0	5 6 2
P_1	1 1 0
P_2	2 3 3
P_3	0 1 1
P_4	3 5 2

c) Is the system in a safe state? If so, show one sequence of processes which allows the system to complete. If not, explain why.

1. Initialize the *Work* and *Finish* vectors.

Work = *Available* = (1, 2, 0)

Finish = (false, false, false, false, false)

2. Find index *i* such that *Finish*[*i*] = false and *Needi* <= *Work*

i	$Work = Work + Allocation_i$	*Finish*
1	(1, 2, 0) + (1, 1, 2) = (2, 3, 2)	(false, true, false, false, false)
3	(2, 3, 2) + (2, 1, 1) = (4, 4, 3)	(false, true, false, true, false)
2	(4, 4, 3) + (0, 3, 0) = (4, 7, 3)	(false, true, true, true, false)
4	(4, 7, 3) + (1, 1, 1) = (5, 8, 4)	(false, true, true, true, true)
0	(5, 8, 4) + (1, 1, 1) = (6, 9, 5)	(true, true, true, true, true)

3. Since *Finish*[*i*] = true for all *i*, hence the system is in a safe state. The sequence of processes which allows the system to complete is P1, P3, P2, P4, P0**.**

d) Given the request (1, 2, 0) from Process P2. Should this request be granted? Why or why not?

1. Check that *Request2* <= *Need2*.

   Since (1, 2, 0) <= (2, 3, 3), hence, this condition is satisfied.

2. Check that *Request2* <= *Available*.

   Since (1, 2, 0) <= (1, 2, 0), hence, this condition is satisfied.

3. Modify the system's state as follows:

   Available = *Available* – *Request2* = (1, 2, 0) – (1, 2, 0) = (0, 0, 0)

   Allocation2 = *Allocation2* + *Request2* = (0, 3, 0) + (1, 2, 0) = (1, 5, 0)

   Need2 = *Need2* – *Request2* = (2, 3, 3) – (1, 2, 0) = (1, 1, 3)

4. Apply the safety algorithm to check if granting this request leaves the system in a safe state.

   1. Initialize the *Work* and *Finish* vectors.

      Work = *Available* = (0, 0, 0)

      Finish = (false, false, false, false, false)

   2. At this point, there does not exist an index *i* such that *Finish*[*i*] = false and *Needi* <= *Work*.

   Since *Finish*[*i*] ≠ true for all *i*, hence the system is not in a safe state.

   Therefore, this request from process *P2* should not be granted.

Resource-Request Algorithm

Let Requesti be the request vector for process Pi . If Requesti [j] == k, then process Pi wants k instances of resource type Rj . When a request for resources is made by process Pi , the following actions are taken:

1. If Requesti ≤Needi , go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.

2. If Requesti ≤ Available, go to step 3. Otherwise, Pi must wait, since the resources are not available.

3. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:

   Available = Available–Requesti ;

   Allocationi = Allocationi + Requesti ;

Needi = Needi –Requesti ;

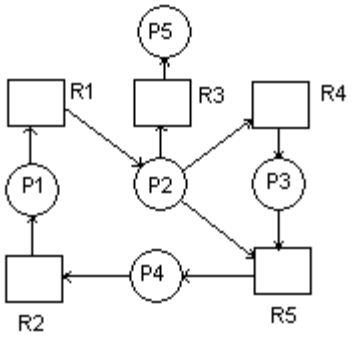
DEADLOCK DETECTION

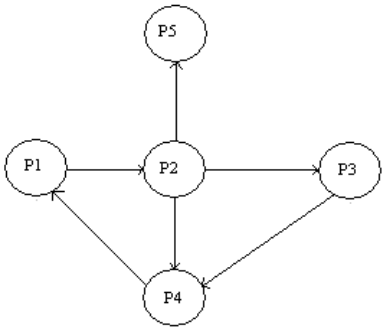
Deadlock Detection

(i) Single instance of each resource type

If all resources have only a single instance, then we can define a deadlock detection algorithm that use a variant of resource-allocation graph called a wait for graph.

Resource Allocation Graph **Wait for Graph**





(ii) Several Instance of a resource type

Available : Number of available resources of each type

Allocation : number of resources of each type currently allocated to each process

Request : Current request of each process

If Request [i,j]=k, then process P_i is requesting K more instances of resource typeR_j.

1. Initialize work := available
   Finish[i]=false, otherwise finish [i]:=true
2. Find an index i such that both
   a. Finish[i]=false
   b. Request$_i$<=work
   if no such i exists go to step4.
3. Work:=work+allocation$_i$
   Finish[i]:=true goto step2
4. If finish[i]=false then process P_i is deadlocked

DEADLOCK RECOVERY.

- There are three basic approaches to recovery from deadlock:
  1. Inform the system operator, and allow him/her to take manual intervention.
  2. Terminate one or more processes involved in the deadlock
  3. Preempt resources.

1. Process Termination

Two basic approaches, both of which recover resources allocated to terminated processes:

➔Terminate all processes involved in the deadlock. This definitely solves the

deadlock, but at the expense of terminating more processes than would be absolutely necessary.

➔Terminate processes one by one until the deadlock is broken. This is more conservative, but requires doing deadlock detection after each step.

➔In the latter case there are many factors that can go into deciding which processes to terminate next:

- Process priorities.
- How long the process has been running, and how close it is to finishing.
- How many and what type of resources is the process holding. (Are they easy to preempt and restore?)
    1. How many more resources does the process need to complete.
    2. How many processes will need to be terminated
    3. Whether the process is interactive or batch.
    4. (Whether or not the process has made non-restorable changes to any resource.)

2. Resource Preemption

➔When preempting resources to relieve deadlock, there are three important issues to be addressed:

1. Selecting a victim - Deciding which resources to preempt from which processes involves many of the same decision criteria outlined above.
2. Rollback - Ideally one would like to roll back a preempted process to a safe state prior to the point at which that resource was originally allocated to the process. Unfortunately it can be difficult or impossible to determine what such a safe state is, and so the only safe rollback is to roll back all the way back to the beginning. (I.e. abort the process and make it start over.)
3. Starvation - How do you guarantee that a process won't starve because its resources are constantly being preempted? One option would be to use a priority system, and increase the priority of a process every time its resources get preempted. Eventually it should get a high enough priority that it won't get preempted any more.

WINDOWS 7 – THREAD AND SMP MANAGEMENT

The native process structures and services provided by the Windows Kernel are relatively simple and general purpose, allowing each OS subsystem to emulate a particular process structure and functionality.

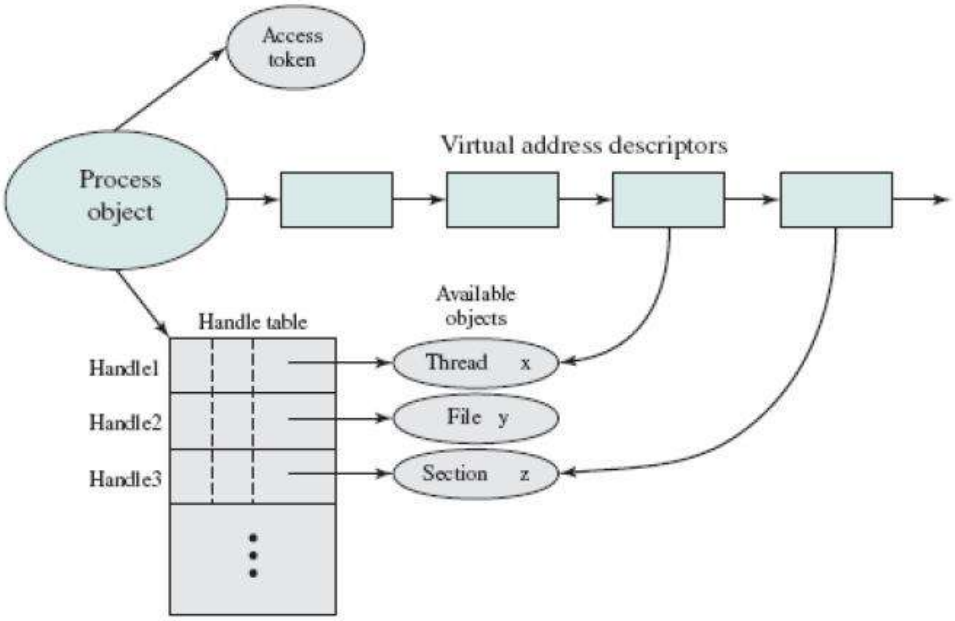
Characteristics of Windows processes:

- Windows processes are implemented as objects.
- A process can be created as new process, or as a copy of an existing process.
- An executable process may contain one or more threads.
- Both process and thread objects have built-in synchronization capabilities.

A Windows Process and Its Resources

- Each process is assigned a security access token, called the primary token of the process. When a user first logs on, Windows creates an access token that includes the security ID for the user.

- Every process that is created by or runs on behalf of this user has a copy of this access token.
- Windows uses the token to validate the user's ability to access secured objects or to perform restricted functions on the system and on secured objects. The access token controls whether the process can change its own attributes.



- Related to the process is a series of blocks that define the virtual address space currently assigned to this process.

- The process cannot directly modify these structures but must rely on the virtual memory manager, which provides a memory allocation service for the process.

- The process includes an object table, with handles to other objects known to this process. The process has access to a file object and to a section object that defines a section of shared memory.

Process and Thread Objects

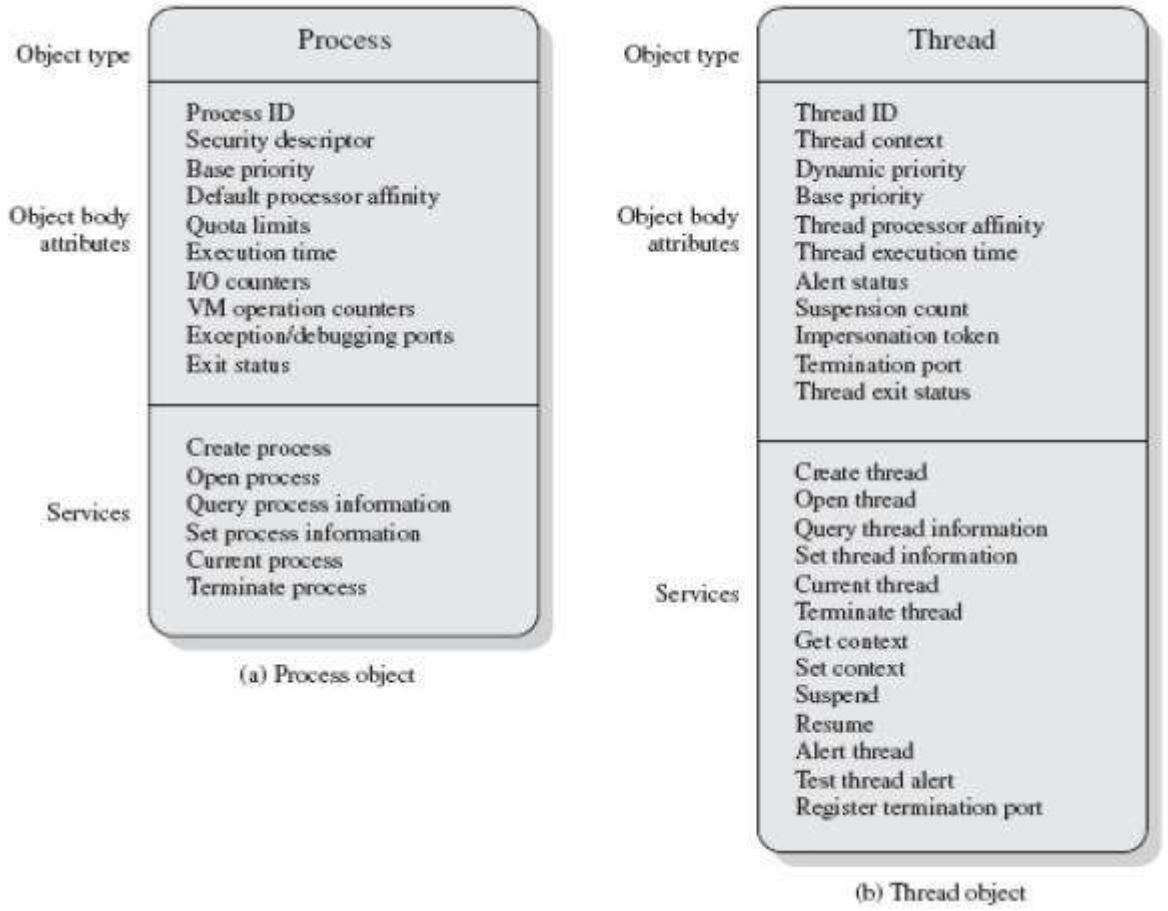
 The object-oriented structure of Windows facilitates the development of a general-purpose process facility.

 Windows makes use of two types of process-related objects: processes and threads.

 A process is an entity corresponding to a user job or application that owns resources, such as memory and open files.

 A thread is a dispatchable unit of work that executes sequentially and is interruptible, so that the processor can turn to another thread.

Windows Process and Thread Objects



(a) Process object

(b) Thread object

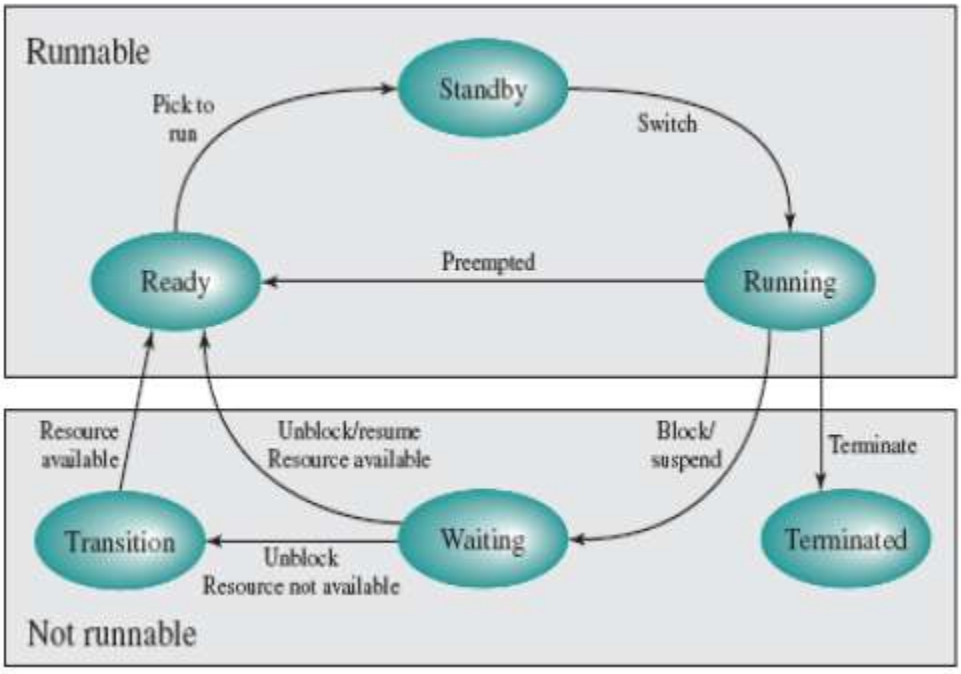
Windows Process Object Attributes

Process ID	A unique value that identifies the process to the operating system.
Security descriptor	Describes who created an object, who can gain access to or use the object, and who is denied access to the object.
Base priority	A baseline execution priority for the process's threads.
Default processor affinity	The default set of processors on which the process's threads can run.
Quota limits	The maximum amount of paged and nonpaged system memory, paging file space, and processor time a user's processes can use.
Execution time	The total amount of time all threads in the process have executed.
I/O counters	Variables that record the number and type of I/O operations that the process's threads have performed.
VM operation counters	Variables that record the number and types of virtual memory operations that the process's threads have performed.
Exception/debugging ports	Interprocess communication channels to which the process manager sends a message when one of the process's threads causes an exception. Normally, these are connected to environment subsystem and debugger processes, respectively.
Exit status	The reason for a process's termination.

Windows Thread Object Attributes

Thread ID	A unique value that identifies a thread when it calls a server.
Thread context	The set of register values and other volatile data that defines the execution state of a thread.
Dynamic priority	The thread's execution priority at any given moment.
Base priority	The lower limit of the thread's dynamic priority.
Thread processor affinity	The set of processors on which the thread can run, which is a subset or all of the processor affinity of the thread's process.
Thread execution time	The cumulative amount of time a thread has executed in user mode and in kernel mode.
Alert status	A flag that indicates whether a waiting thread may execute an asynchronous procedure call.
Suspension count	The number of times the thread's execution has been suspended without being resumed.
Impersonation token	A temporary access token allowing a thread to perform operations on behalf of another process (used by subsystems).
Termination port	An interprocess communication channel to which the process manager sends a message when the thread terminates (used by subsystems).
Thread exit status	The reason for a thread's termination.

Thread States



Problem

1.Consider the following set of processes, with the length of the CPU-burst time given in milliseconds:

Process	Burst Time	Arrival Time	Priority
P1	23	0	2
P2	3	1	1
P3	6	2	4
P4	2	3	3

a. Draw four *Gantt charts* illustrating the execution of these processes using FCFS, SJF(Preemptive), a non-preemptive priority (a smaller priority number implies a higher priority), and RR (quantum = 1) scheduling.

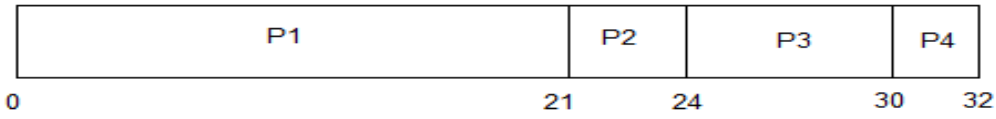
c. What is the *waiting time* of each process for each of the scheduling algorithms in part a?

d. Which of the schedules in part a results in the *minimal average waiting time* (over all processes)?

FCFS SCHEDULING

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

The average waiting time will be = (0 + 21 + 24 + 30)/4 = 18.75 ms



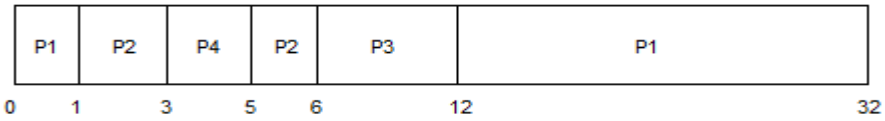
This is the GANTT chart for the above processes

SJF(SHORTEST JOB FIRST)

In Pre-emptive Shortest Job First Scheduling, jobs are put into ready queue as they arrive, but as a process with short burst time arrives, the existing process is pre-empted.

PROCESS	BURST TIME	ARRIVAL TIME
P1	21	0
P2	3	1
P3	6	2
P4	2	3

The GANTT chart for Preemptive Shortest Job First Scheduling will be,



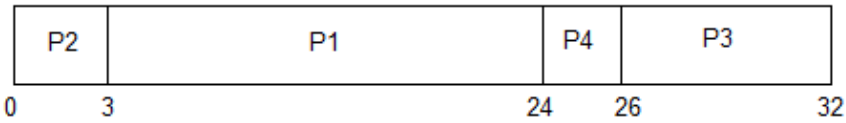
The average waiting time will be, ((5-3) + (6-2) + (12-1))/4 = 4.25 ms

The average waiting time for preemptive shortest job first scheduling is less than both, non-preemptive SJF scheduling and FCFS scheduling.

PRIORITY

PROCESS	BURST TIME	PRIORITY
P1	21	2
P2	3	1
P3	6	4
P4	2	3

The GANTT chart for following processes based on Priority scheduling will be,

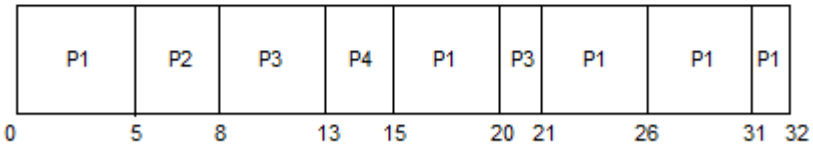


The average waiting time will be, (0 + 3 + 24 + 26)/4 = 13.25 ms

ROUND ROBIN

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

The GANTT chart for round robin scheduling will be,

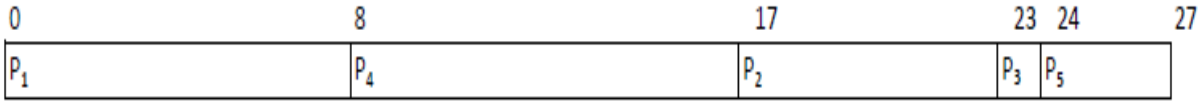


The average waiting time will be, 11 ms.

2.

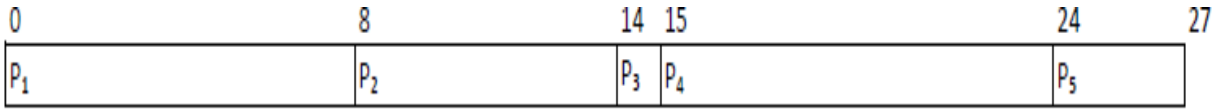
Process	Burst	Priority	Arrival Time
P_1	8	4	0
P_2	6	1	2
P_3	1	2	2
P_4	9	2	1
P_5	3	3	3

First Come First Served



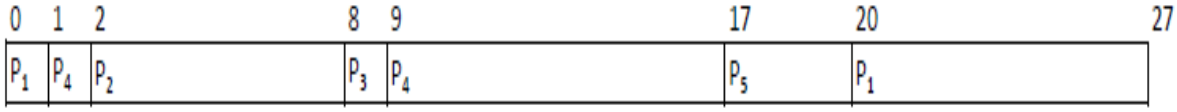
Avg. Wait = 0+8-1+17-2+23-2+24-3 = 0+7+15+21+21=64/5 = 12.8 AVG TAT = 8+17-1+23-2+24-2+27-3 = 8+16+21+22+24=91/5=18.2

SJF



Avg. Wait = 8-2+14-2+15-1+24-3 = 6+12+14+21 = 53/5=10.6ms AVG TAT = 8+14-2+15-2+24-1+27-3 = 8+12+13+23+24=80/5=16ms

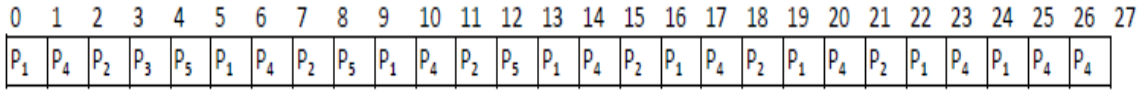
Priority



Avg. Wait Time = 0+20-1+2-2+8-2+9-2+17-3 = 0+19+0+6+7+14 = 46/5=9.2ms AVG TAT = 27+8-2+9-2+16+20-3 = 73/5 = 14.6ms

Round Robin

Round Robin (1ms Quantum)



Wait Time P_1 = 0+4+3+3+2+2+1+1 = 16

Wait Time P2 = 0+4+3+3+2+2+2+1 = 17

Wait Time P3 = 1

Wait Time P4 = 4+4+3+2+3+2+1 = 19

Wait Time P5 = 1+3+3 = 7

Avg Wait Time = 60/5 = 12ms

Avg TAT =25+21+2+26+10 = 84/5 = 16.8

UNIT III STORAGE MANAGEMENT

Main Memory-Contiguous Memory Allocation, Paging, Segmentation, Segmentation with paging, 32 and 64 bit architecture Examples; Virtual Memory- Background, Demand Paging, Page Replacement, Allocation, Thrashing; Allocating Kernel Memory, OS Examples.

1. MEMORY MANAGEMENT: BACKGROUND

Memory management is the functionality of an operating system which handles or manages primary memory and moves processes back and forth between main memory and disk during execution.

Memory management keeps track of each and every memory location, regardless of either it is allocated to some process or it is free. It checks how much memory is to be allocated to processes.

It decides which process will get memory at what time.

It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.

1.1 Basic Hardware

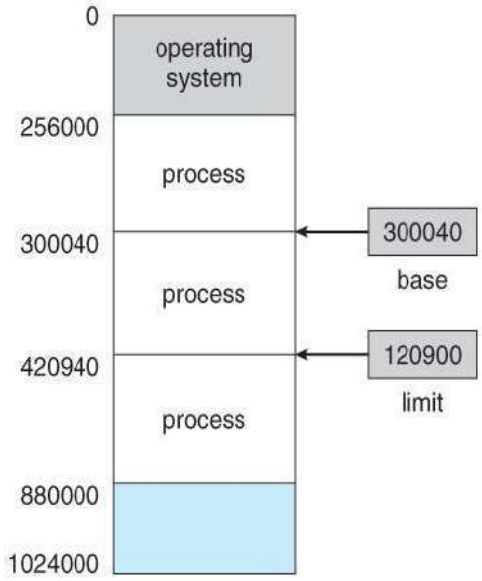
Program must be brought (from disk) into memory and placed within a process for it to be run

- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- Cache sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

➔ We can provide this protection by using two registers, usually a **base** and a **limit**

The base register holds the smallest legal physical memory address; The limit register specifies the size of the range.

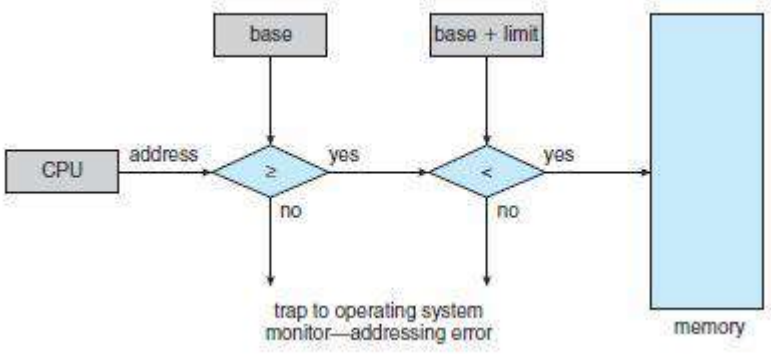
For example, if the base register holds 300040 and limit register is 120900, then the program can legally access all addresses from 300040 through 420940 (inclusive).



Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers.

Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error.

This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.



1.2 Address Binding

Definition

Converting the address used in a program to an actual physical address.

Address binding is the process of mapping the program's logical or virtual addresses to corresponding physical or main memory addresses.

In other words, a given logical address is mapped by the MMU (Memory Management Unit) to a physical address.

User programs typically refer to memory addresses with symbolic names such as "i", "count", and "average Temperature".

These symbolic names must be mapped or bound to physical memory addresses, which typically occurs in several stages:

Three different stages of binding:

1. **Compile time**. The compiler translates symbolic addresses to absolute addresses. If you know at compile time where the process will reside in memory, then absolute code can be generated (Static).

2. **Load time**. The compiler translates symbolic addresses to relative (relocatable) addresses. The loader translates these to absolute addresses. If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code (Static).

3. **Execution time**. If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. The absolute addresses are generated by hardware. Most general-purpose OS use this method (Dynamic).

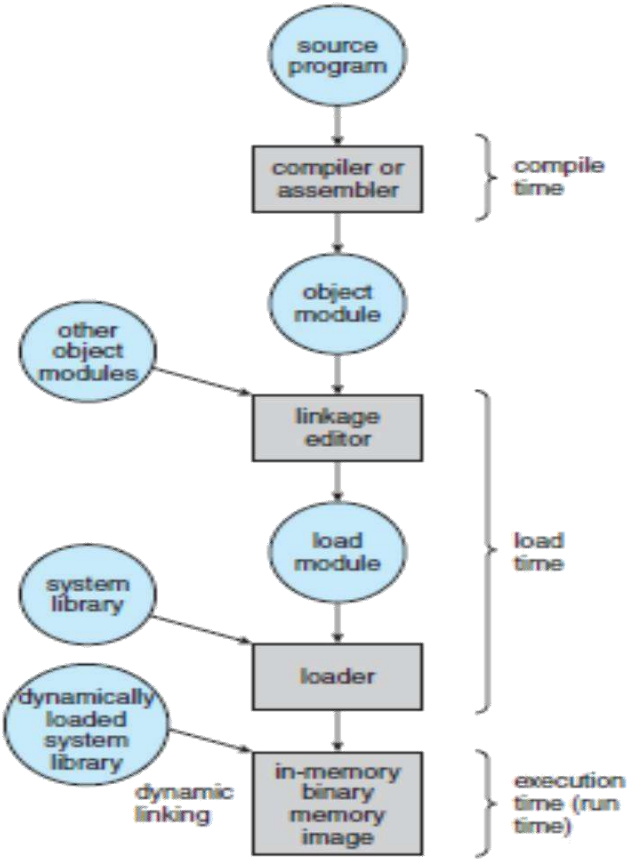


Figure 8.3 Multistep processing of a user program.

1.3 Logical vs. Physical Address Space

Logical address – generated by the CPU; also referred to as **"virtual address"**

Physical address – address seen by the memory unit.

Logical and physical addresses are the **same** in compile-time and load-time address-binding schemes

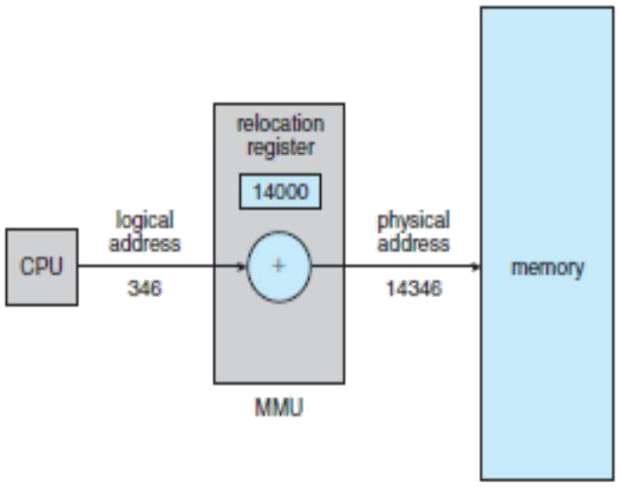
Logical (virtual) and physical addresses **differ** in execution-time address- binding scheme

Memory-Management Unit (MMU)

It is a hardware device that maps virtual / Logical address to physical address.
In this scheme, the relocation register's value is added to Logical address generated by a user process.

The Base register is called a relocation register.

- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346.
- The user program never sees the real physical addresses. The program can create a pointer to location 346, store it in memory, manipulate it, and compare it with other addresses -all as the number 346.
- The user program deals with logical addresses.



1.4 Dynamic Loading

Dynamic loading is a mechanism by which a computer program can, at run time, load a library (or other binary) into memory, retrieve the addresses of functions and variables contained in the library, execute those functions or access those variables, and unload the library from memory.

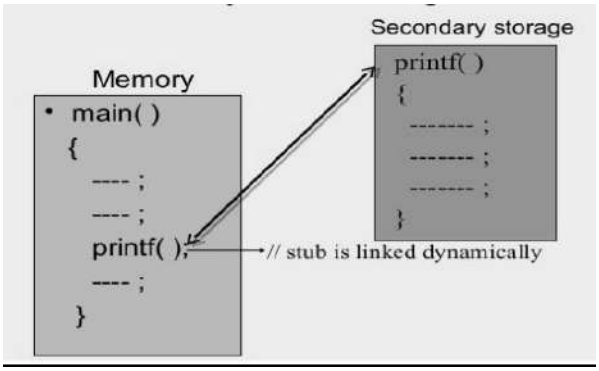
Dynamic loading means loading the library (or any other binary for that matter) into the memory during load or run-time.

Dynamic loading can be imagined to be similar to plugins, that is an exe can actually execute before the dynamic loading happens(The dynamic loading for example can be created using Load Library call in C or C++)

1.5 Dynamic Linking and shared libraries

Dynamic linking refers to the linking that is done during load or run-time and not when the exe is created.
In case of dynamic linking the linker while creating the exe does minimal work. For the dynamic linker to work it actually has to load the libraries too. Hence it's also called linking loader.
Small piece of code, *stub*, used to indicate how to load library routine.
Stub replaces itself with the address of the routine, and executes the routine.
Operating system needed to check if routine is in processes memory address.
Dynamic linking is particularly useful for libraries.

- Shared libraries: Programs linked before the new library was installed will continue using the older library.



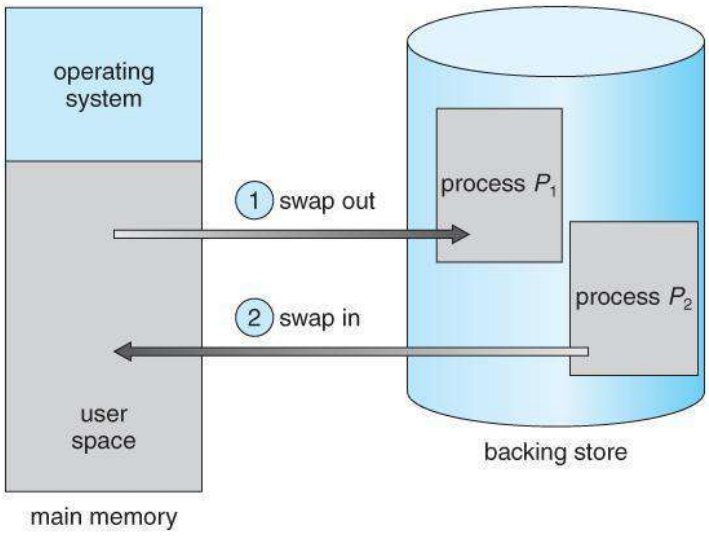
2. SWAPPING

2.1 Basic

- A process can be swapped temporarily out of memory to a backing store (SWAP OUT)and then brought back into memory for continued execution (SWAP IN).
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users & it must provide direct access to these memory images
- **Roll out, roll in –** swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- **Transfer time:** Major part of swap time is transfer time. Total transfer time is directly proportional to the amount of memory swapped.

□ **Example**: Let us assume the user process is of size 1MB & the backing store is a standard hard disk with a transfer rate of 5MBPS.

Transfer time = 1000KB/5000KB per second
= 1/5 sec = 200ms



A process with dynamic memory requirements will need to issue system calls (request memory() and release memory()) to inform the operating system o f its changing memory needs.

2.2 Swapping on Mobile Systems

Swapping is typically not supported on mobile platforms, for several reasons:

Mobile devices typically use flash memory in place of more spacious hard drives for persistent storage, so there is not as much space available.

Flash memory can only be written to a limited number of times before it becomes unreliable.

The bandwidth to flash memory is also lower.

Apple's IOS asks applications to voluntarily free up memory

Read-only data, e.g. code, is simply removed, and reloaded later if needed.

Modified data, e.g. the stack, is never removed.

Apps that fail to free up sufficient memory can be removed by the

OS Android follows a similar strategy.

Prior to terminating a process, Android writes its application state to flash memory for quick restarting.

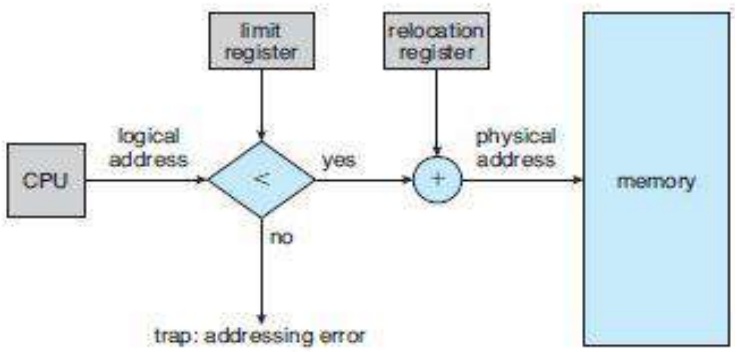
3. CONTIGUOUS MEMORY ALLOCATION

One approach to memory management is to load each process into a contiguous space.

The operating system is allocated space first, usually at either low or high memory locations, and then the remaining available memory is allocated to processes as needed.

3.1 Memory Protection

Protection against user programs accessing areas that they should not, allows programs to be relocated to different memory starting addresses as needed, and allows the memory space devoted to the OS to grow or shrink dynamically as needs change.



3.2 Memory Allocation

In contiguous memory allocation each process is contained in a single contiguous block of memory. Memory is divided into several fixed size partitions. Each partition contains exactly one process.

When a partition is free, a process is selected from the input queue and loaded into it.

There are two methods namely:

- Fixed – P**a**rtition Method
- Variable – Partition Method

- **Fixed – Partition Method:**

Divide memory into fixed size partitions, where each partition has exactly one process. The drawback is Memory space unused within a partition is wasted.(eg. When process size < partition size)

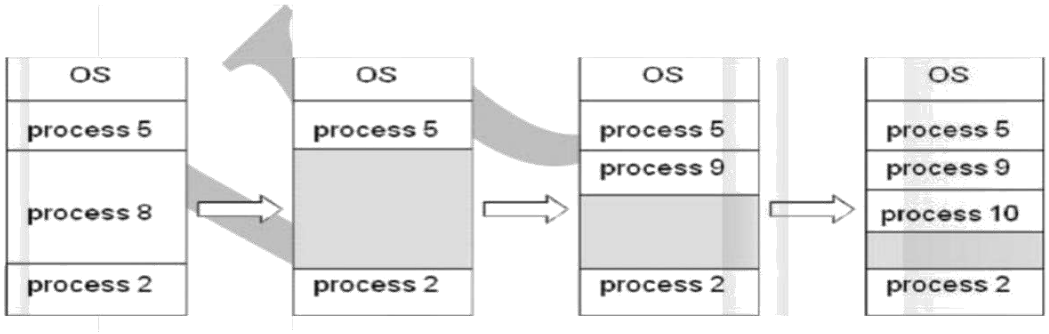
- **Variable-partition method:**

o Divide memory into variable size partitions, depending upon the size of the incoming process.

o When a process terminates, the partition becomes available for another process.

o As processes complete and leave they create holes in the main memory.

o ***Hole*** – block of available memory; holes of various size are scattered throughout memory.



Dynamic Storage- Allocation Problem:

How to satisfy a request of size n‘from a list of free holes?

➔The free blocks of memory are known as holes. The set of holes is searched to determine which hole is best to allocate.

Solution:

- First-fit: Allocate the first hole that is big enough.
- Best-fit: Allocate the smallest hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
- Worst-fit: Allocate the largest hole; must also search entire list. Produces the largest leftover hole.

Example :

Given five memory partitions of 100 KB, 500 KB, 200 KB, 300 KB, and 600 KB (in order), how would each of the first-fit, best-fit, and worst-fit algorithms place processes of 212 KB, 417 KB, 112 KB, and 426 KB (in order)?Which algorithm makes the most efficient use of memory?

a. First-fit:
1. 212K is put in 500K partition
2. 417K is put in 600K partition
3. 112K is put in 288K partition (new partition 288K = 500K − 212K)
4. 426K must wait

b. Best-fit:
1. 212K is put in 300K partition
2. 417K is put in 500K partition
3. 112K is put in 200K partition
4. 426K is put in 600K partition

c. Worst-fit:
1. 212K is put in 600K partition
2. 417K is put in 500K partition
3. 112K is put in 388K partition
4. 426K must wait

In this example, best-fit turns out to be the best.

NOTE: First-fit and best-fit are better than worst-fit in terms of speed and storage utilization

3.3Fragmentation:

Fragmentation is a phenomenon in which storage space is used inefficiently, reducing capacity or performance and often both.

1. **External Fragmentation** – This takes place when enough total memory space exists to satisfy a request, but it is not contiguous i.e, storage is fragmented into a large number of small holes scattered throughout the main memory.

2. Internal Fragmentation – Allocated memory may be slightly larger than requested memory.

Example: hole = 184 bytes Process size = 182 bytes.

We are left with a hole of 2 bytes.

➔ **Solutions**

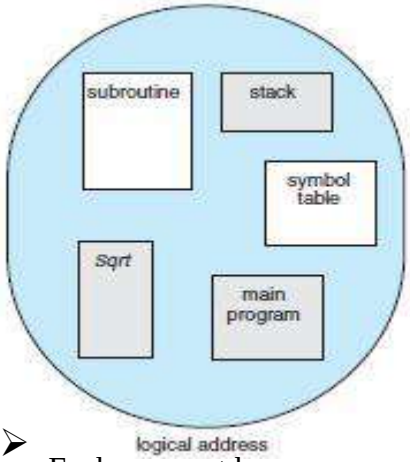
Compaction: Move all processes towards one end of memory, hole towards other end of memory, producing one large hole of available memory. This scheme is expensive as it can be done if relocation is dynamic and done at execution time.

4. SEGMENTATION

4.1 Basic Method

o Memory-management scheme that supports user view of memory

o A program is a collection of segments. A segment is a logical unit such as:Main program, Procedure, Function, Method, Object, Local variables, global variables, Common block, Stack, Symbol table, arrays



- Each segment has a name and a length.
- The addresses specify both the segment name and the offset within the segment.
- The programmer therefore specifies each address by two quantities:

a segment name and an offset.

A logical address consists of a two tuple:

<segment-number, offset>.

4.2Segmentation Hardware

Each entry in the segment table has a segment base and a segment limit.

The segment base contains the starting physical address where the segment resides in memory, and the segment limit specifies the length of the segment

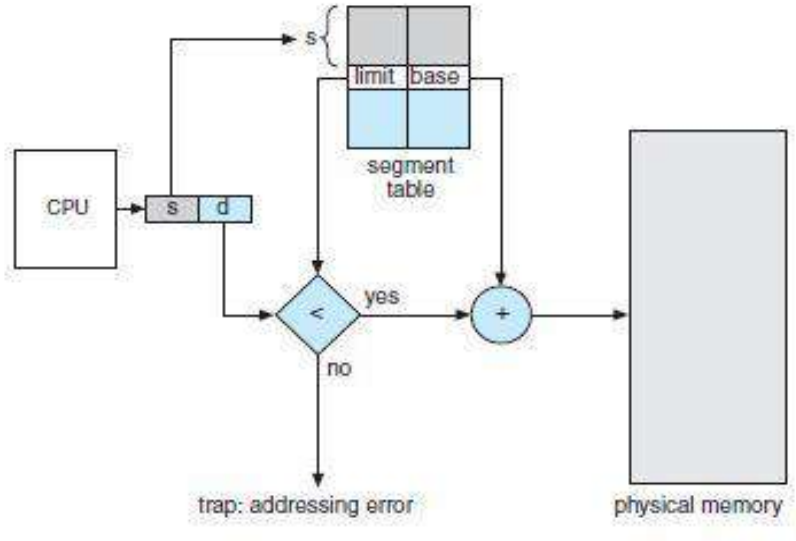
A logical address consists of two parts:

a **segment number** ➔ s, and an **offset** into that segment ➔ d. The **segment number** is used as an index to the segment table.

The **offset** d of the logical address must be between 0 and the segment limit.

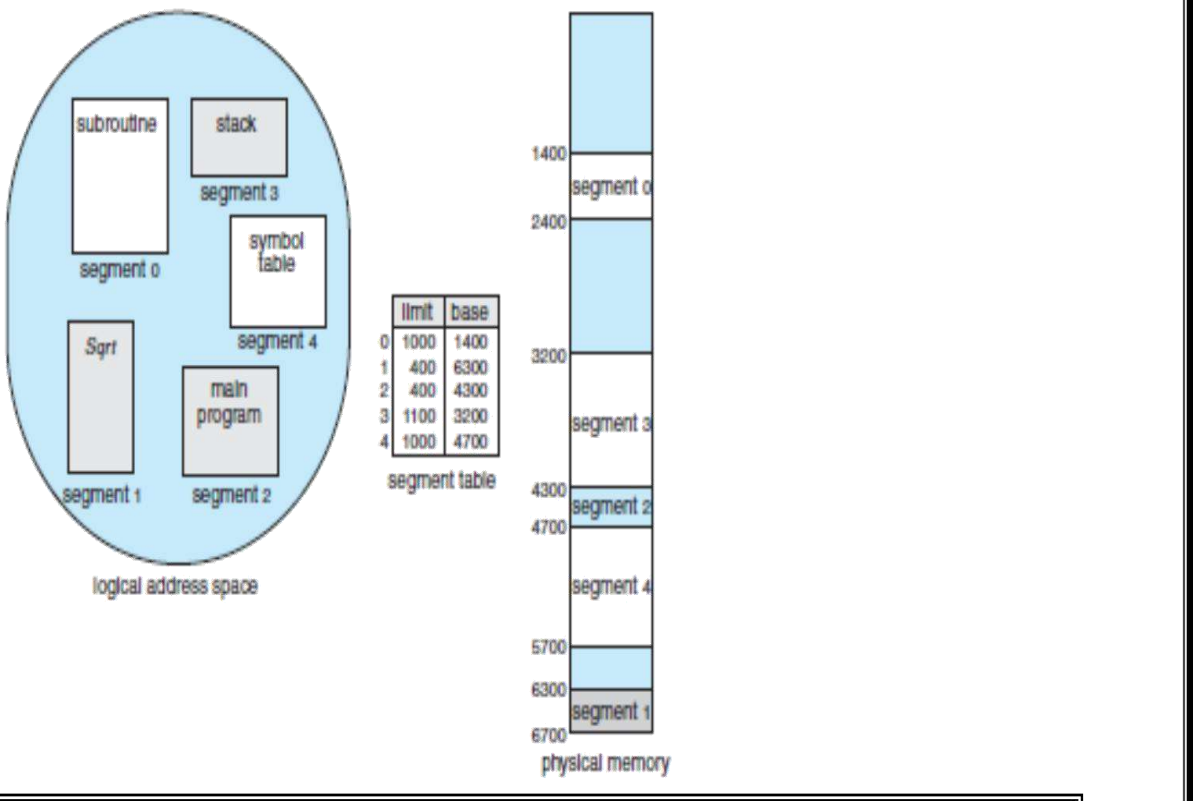
If it is not, we trap to the operating system (logical addressing attempt beyond end of segment).

When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte.



For example,

segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353. A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052. A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1,000 bytes long.



5. PAGING

- It is a memory management scheme that permits the physical address space of a process to be noncontiguous.
- It avoids the considerable problem of fitting the varying size memory chunks on to the backing store.

5.1Basic Method

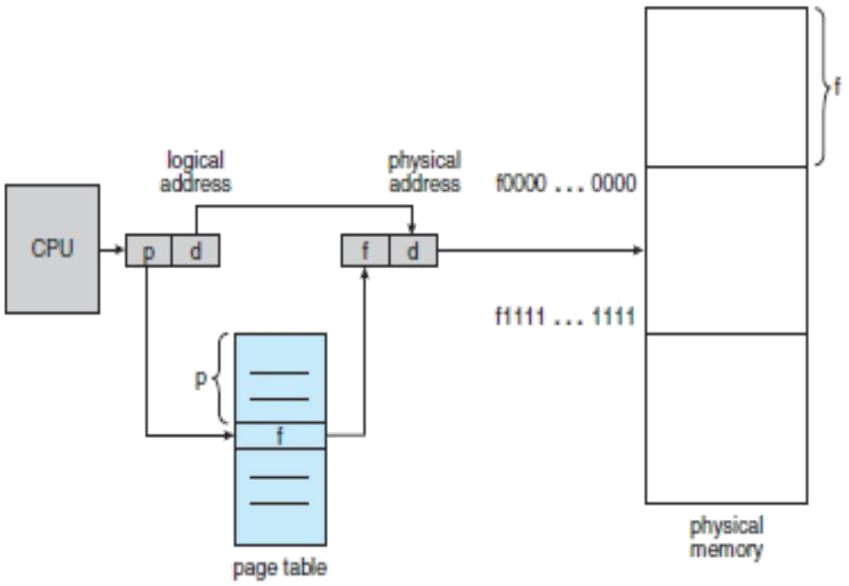
- Divide logical memory into blocks of same size called **"pages"**.
- Divide physical memory into fixed-sized blocks called **"frames"**
- Page size is a power of 2, between 512 bytes and 16MB.

Address Translation Scheme

each page Address generated by CPU (logical address) is divided into:

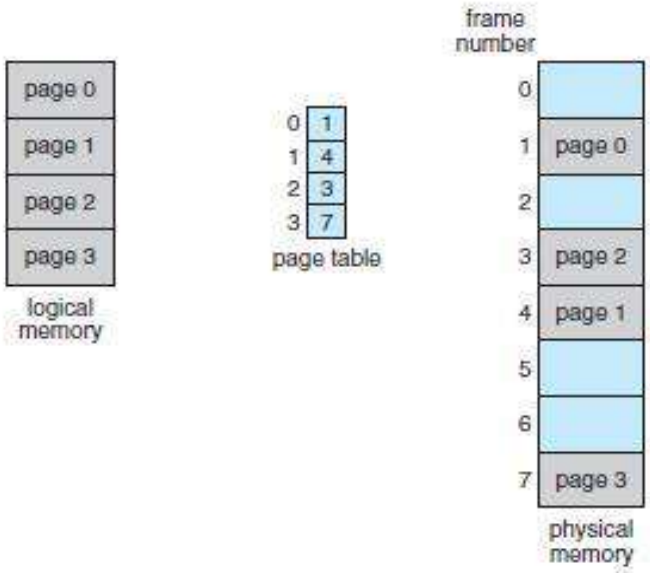
Page number *(p)* – used as an index into a page table which contains base address of in physical memory

Page offset *(d)* – combined with base address to define the physical address
i.e., Physical address = base address + offset



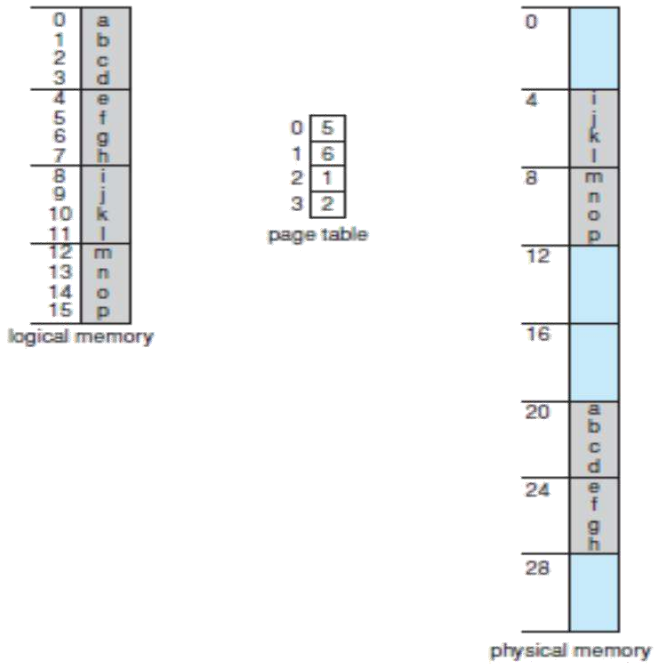
The page number is used as an index into a page table. The page table contains the base address of each page in physical memory.

This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.



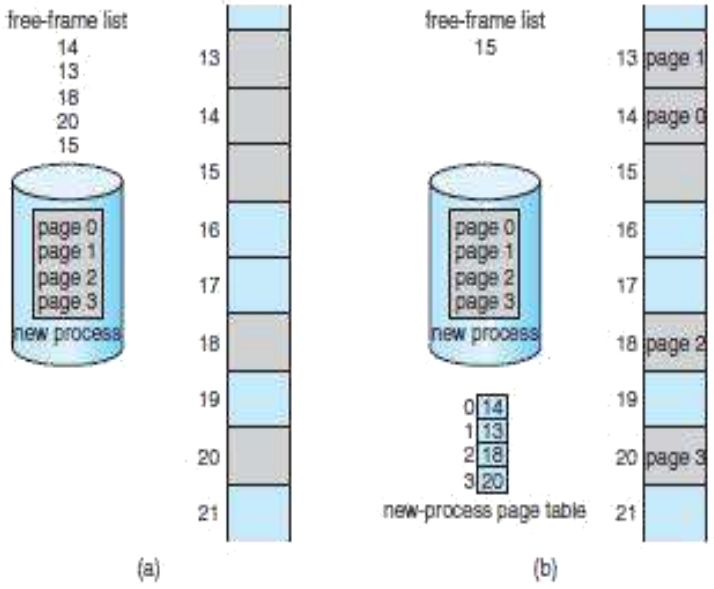
Consider the memory in the logical address, n= 2 and m = 4. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the programmer's view of memory can be mapped into physical memory. Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5.

Thus, logical address 0 maps to physical address 20 [= (5 × 4) + 0]. Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 × 4) + 3]. Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [= (6 × 4) + 0]. Logical address 13 maps to physical address 9.



Since the operating system is managing physical memory, it must be aware of the allocation details of physical memory, which frames are allocated, which frames are available, how many total frames there are, and so on.

This information is generally kept in a data structure called a frame table.
The frame table has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process or processes.



5.2 Hardware Support

- The TLB is associative, high-speed memory.
- Each entry in the TLB consists of two parts:
  - a key (or tag) and a value.
- When the associative memory is presented with an item, the item is compared with all keys simultaneously.
- If the item is found, the corresponding value field is returned.
- The TLB contains only a few of the page-table entries.
- When a logical address is generated by the CPU, its page number is presented to the TLB.
- If the page number is not in the TLB (known as a TLB miss), a memory reference to the page table must be made.
- Depending on the CPU, this may be done automatically in hardware or via an interrupt to the operating system.
- If the page number is found, its frame number is immediately available and is used to access Memory.

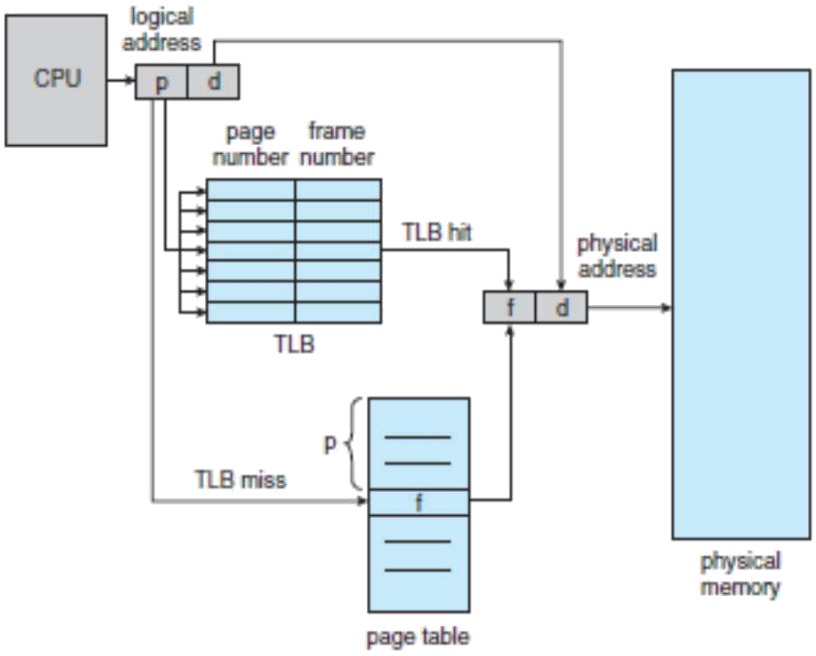
  Hit Ratio - The percentage of times that the page number of interest is found in the TLB is called the hit ratio.

- An 80-percent hit ratio, for example, means that we find the desired page number in the TLB 80 percent of the time. If it takes 100 nanoseconds to access memory, then a mapped-memory access takes 100 nanoseconds when the page number is in the TLB.

- If we fail to find the page number in the TLB then we must first access memory for the page table and frame number (100 nanoseconds) and then access the desired byte in memory (100 nanoseconds), for a total of 200 nanoseconds.

  effective access time = $0.80 \times 100 + 0.20 \times 200$
  = 120 nanoseconds

  For a 99-percent hit ratio, which is much more realistic, we have effective access time = $0.99 \times 100 + 0.01 \times 200 = 101$ nanoseconds

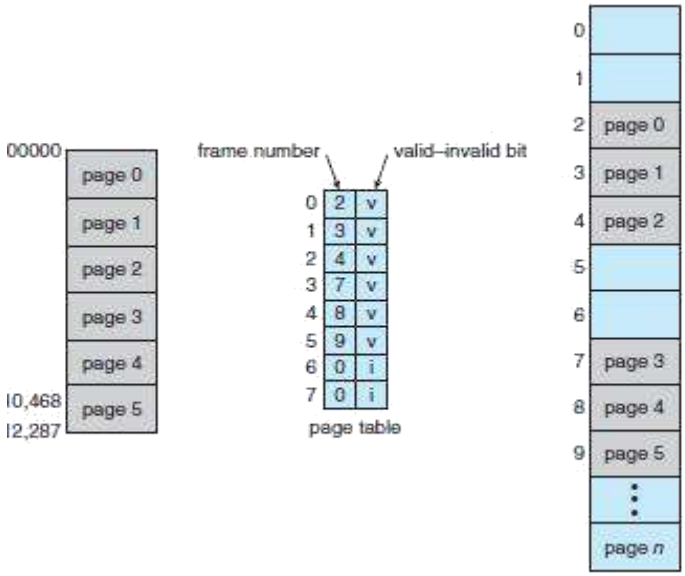


5.3 Protection

Memory protection in a paged environment is accomplished by protection bits associated with each frame.
One additional bit is generally attached to each entry in the page table: a valid–invalid bit.

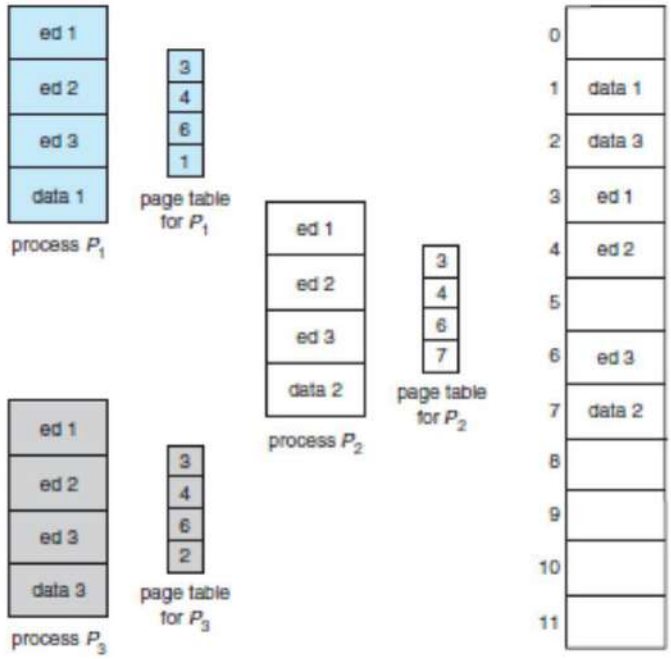
When this bit is set to valid, the associated page is in the process's logical address space and is thus a legal (or valid) page.

When the bit is set to invalid, the page is not in the process's logical address space. Illegal addresses are trapped by use of the valid–invalid bit.



5.4 Shared Pages

An advantage of paging is the possibility of sharing common code.



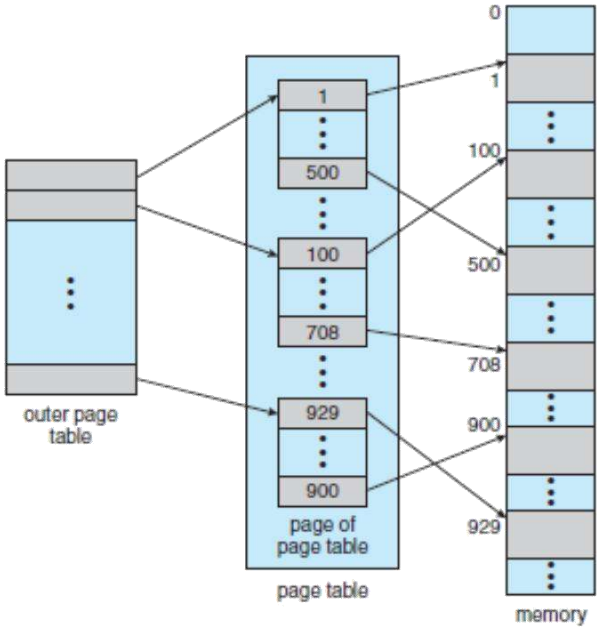
6.STRUCTURE OF PAGE TABLE

The most common techniques for structuring the page table, including hierarchical paging, hashed page tables, and inverted page tables.

1.Hierarchical Paging

The page table itself becomes large for computers with large logical address space (232 to 264).
Example:

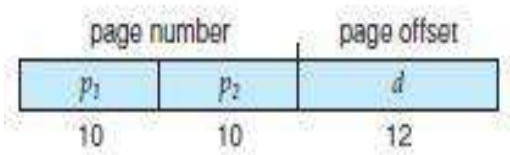
- Consider a system with a 32-bit logical address space. If the page size in such a system is 4 KB(212), then a page table may consist of up to 1 million entries (232/212).
- Assuming that each entry consists of 4 bytes, each process may need up to 4 MB of physical address space for the page table alone.
- The page table should be allocated contiguously in main memory.
- The solution to this problem is to divide the page table into smaller pieces.
- One way of dividing the page table is to use a two-level paging algorithm,



For example, consider again the system with a 32-bit logical address space and a page size of 4 KB. A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12
bits.

Because we page the page table, the page number is further divided into a 10-bit page number and a 10-bit page offset.

Thus, a logical address is as follows:

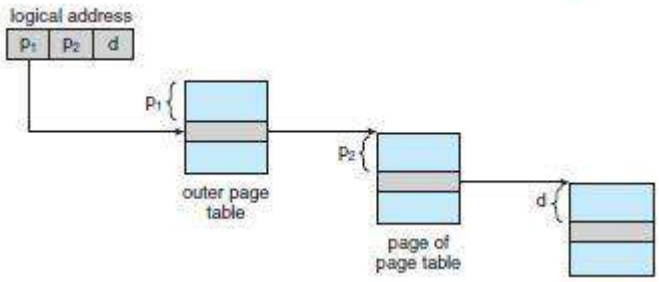


where

p1 - an index into the outer page table

p2 - the displacement within the page of the inner page table.

The address-translation method for this architecture is shown in the figure. Because address translation
works from the outer page table inward, this scheme is also known as a forward-mapped page table.



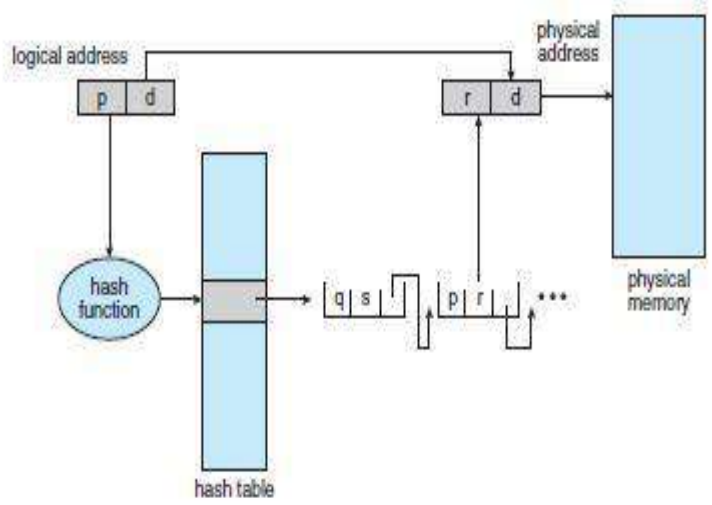
2.Hashed Page Tables

- A common approach for handling address spaces larger than 32 bits is to use a hashed page table, with the hash value being the virtual page number.
- Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions).

- Each element consists of three fields:
  - The virtual page number
  - The value of the mapped page frame
  - A pointer to the next element in the linked list.

Algorithm:

- The virtual page number in the virtual address is hashed into the hash table.
- The virtual page number is compared with field 1 in the first element in the linked list.
- If there is a match, the corresponding page frame (field 2) is used to form the desired physical address.
- If there is no match, subsequent entries in the linked list are searched for a matching virtual page number.



3. Inverted Page Table

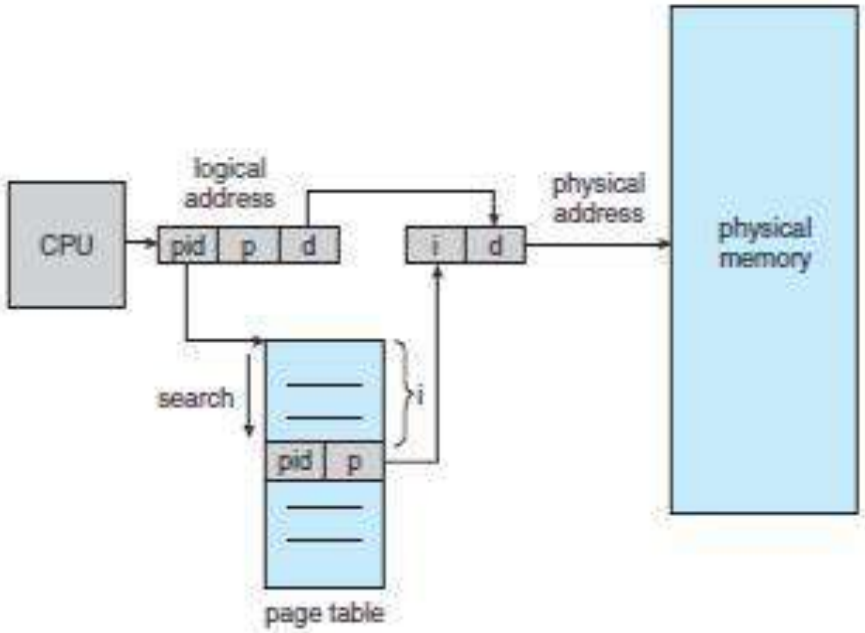
With each process having its own page table, and with each page table consuming considerable amount of memory

We use a lot of memory to keep track of memory.

Inverted page table has one entry for each real page of memory.

Lookup time is increased because it requires a search on the inverted table.

Hash table can be used to reduce this problem.

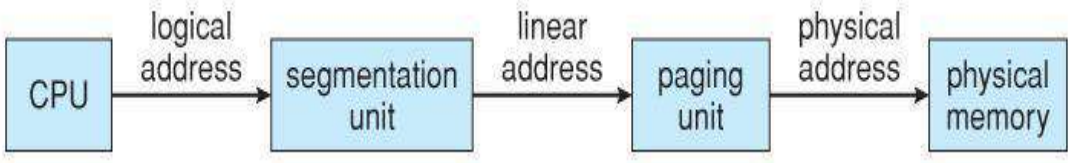


Each virtual address in the system consists of a triple:
<process-id, page-number, offset>.

7.INTEL 32 AND 64-BIT ARCHITECTURES

IA-32 Segmentation

The Pentium CPU provides both pure segmentation and segmentation with paging. In the latter case, the CPU generates a logical address (segment-offset pair), which the segmentation unit converts into a logical linear address, which in turn is mapped to a physical frame by the paging unit



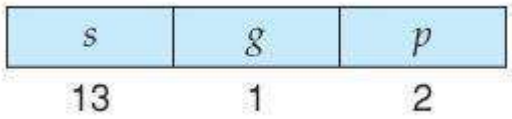
IA-32 Segmentation

The Pentium architecture allows segments to be as large as 4 GB, (24 bits of offset).
Processes can have as many as 16K segments, divided into two 8K groups:

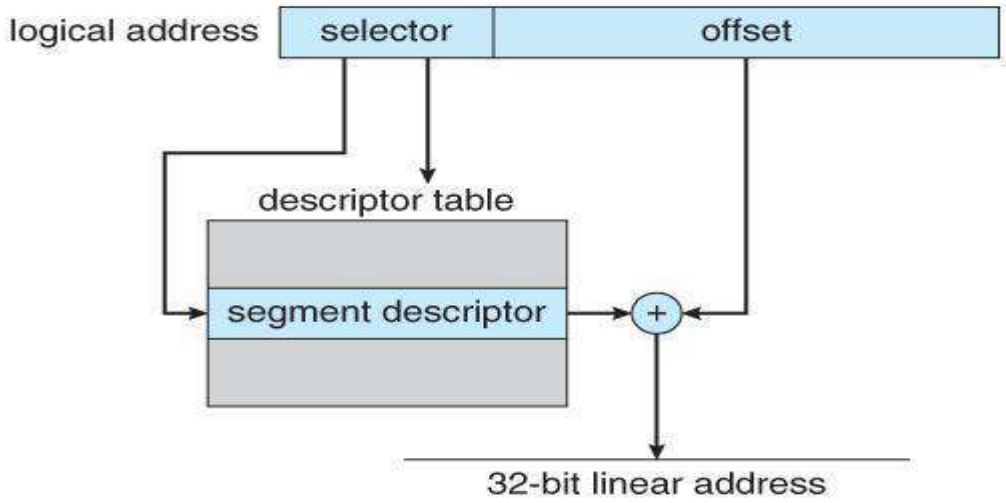
- 8K private to that particular process, stored in the Local Descriptor Table, LDT.
- 8K shared among all processes, stored in the Global Descriptor Table, GDT.

Logical addresses are (selector, offset) pairs, where the selector is made up of 16 bits:

- A 13 bit segment number (up to 8K)
- A 1 bit flag for LDT vs. GDT.
- 2 bits for protection codes.

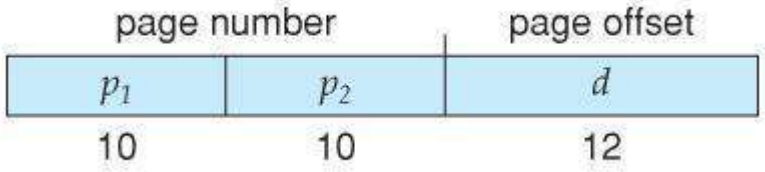


The descriptor tables contain 8-byte descriptions of each segment, including base and limit registers. Logical linear addresses are generated by looking the selector up in the descriptor table and adding the appropriate base address to the offset.



IA-32 Paging

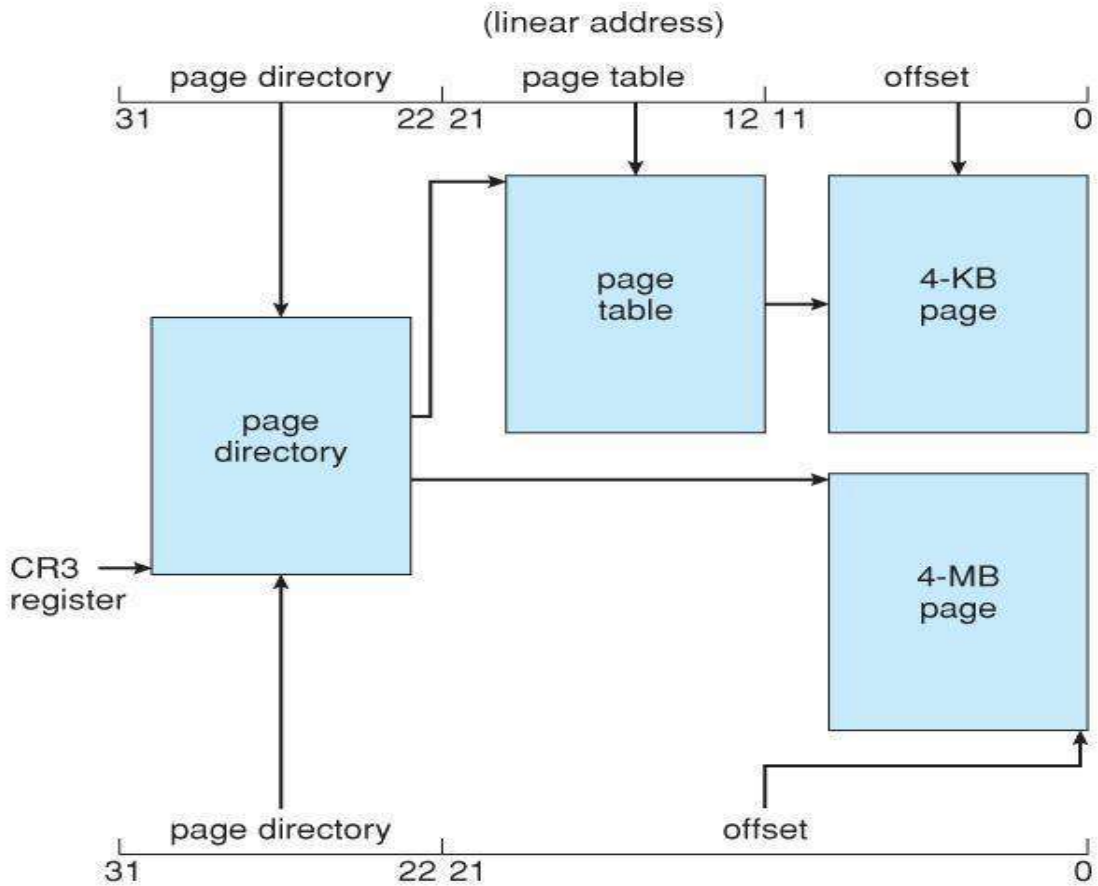
Pentium paging normally uses a two-tier paging scheme, with the first 10 bits being a page number for an outer page table (a.k.a. page directory), and the next 10 bits being a page number within one of the 1024 inner page tables, leaving the remaining 12 bits as an offset into a 4K page.



A special bit in the page directory can indicate that this page is a 4MB page, in which case the remaining 22 bits are all used as offset and the inner tier of page tables is not used.

The CR3 register points to the page directory for the current process.

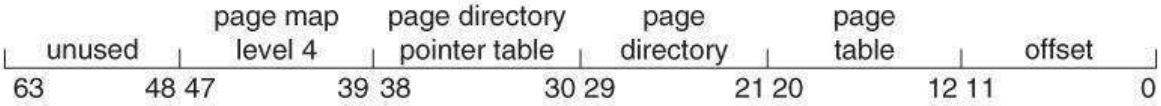
If the inner page table is currently swapped out to disk, then the page directory will have an "invalid bit" set, and the remaining 31 bits provide information on where to find the swapped out page table on the disk.



x86-64

The initial entry of Intel developing 64-bit architectures was the IA-64 (later named Itanium) architecture, but was not widely adopted.

- Meanwhile, AMD —began developing a 64-bit architecture known as x86-64 that was based on extending the existing IA-32 instruction set.
- The x86-64 supported much larger logical and physical address spaces, as well as several other architectural advances.
- Support for a 64-bit address space yields an astonishing 264 bytes of addressable memory— a number greater than 16 quintillion (or 16 exabytes).



8.VIRTUAL MEMORY

- It is a technique that allows the execution of processes that may not be completely in main memory.

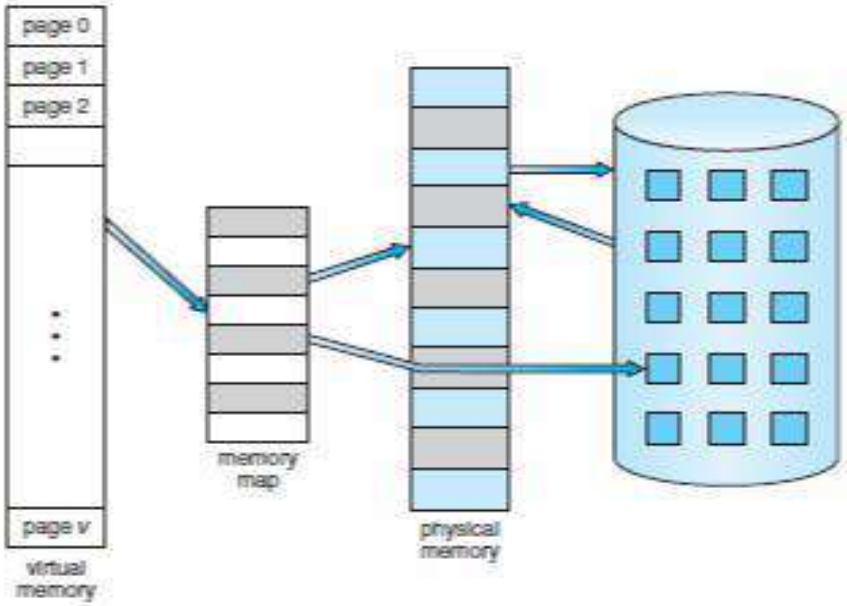
  Virtual memory is the separation of user logical memory from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available.

  – Only part of the program needs to be in memory for execution.

  – Logical address space can therefore be much larger than physical address space.

  – Need to allow pages to be swapped in and out.

- **Advantages:**
  - Allows the program that can be larger than the physical memory.
  - Separation of user logical memory from physical memory
  - Allows processes to easily share files & address space.
  - Allows for more efficient process creation.



- Virtual memory can be implemented using
  - Demand paging
  - Demand segmentation

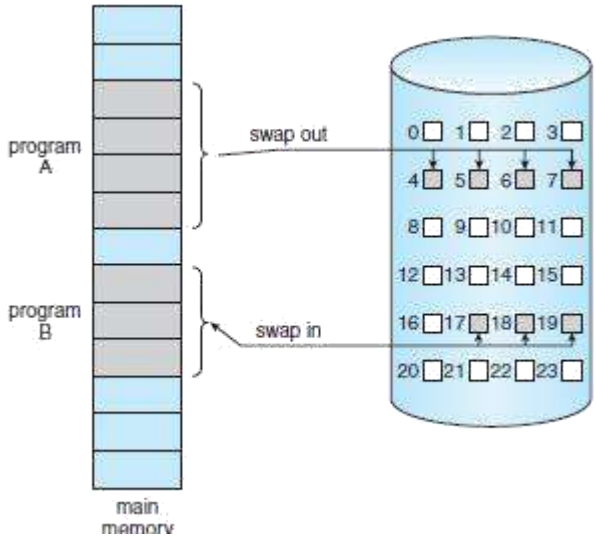
9. DEMAND PAGING

9.1 Concept

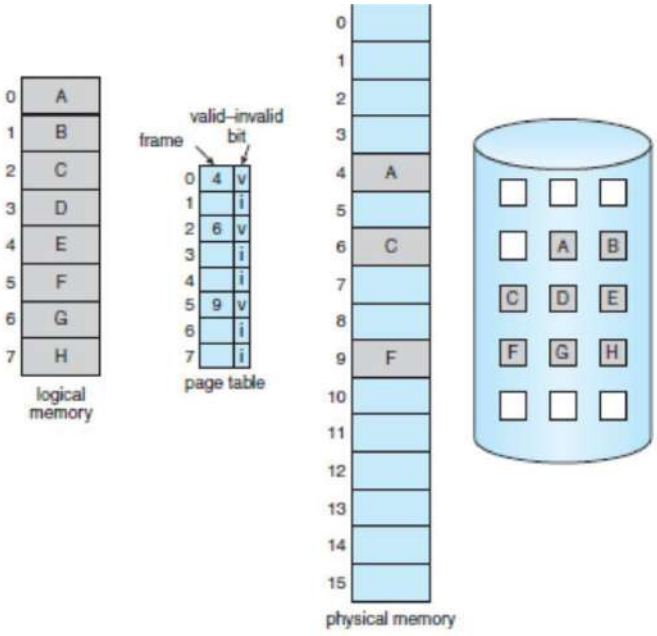
The basic idea behind demand paging is that when a process is swapped in, its pages are not swapped in all at once. Rather they are swapped in only when the process needs them (On demand). This is termed as lazy swapper.

Advantages

- Less I/O needed
- Less memory needed
- Faster response
- More users



Page table when some pages are not in main memory.



The procedure for handling this page fault

1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.

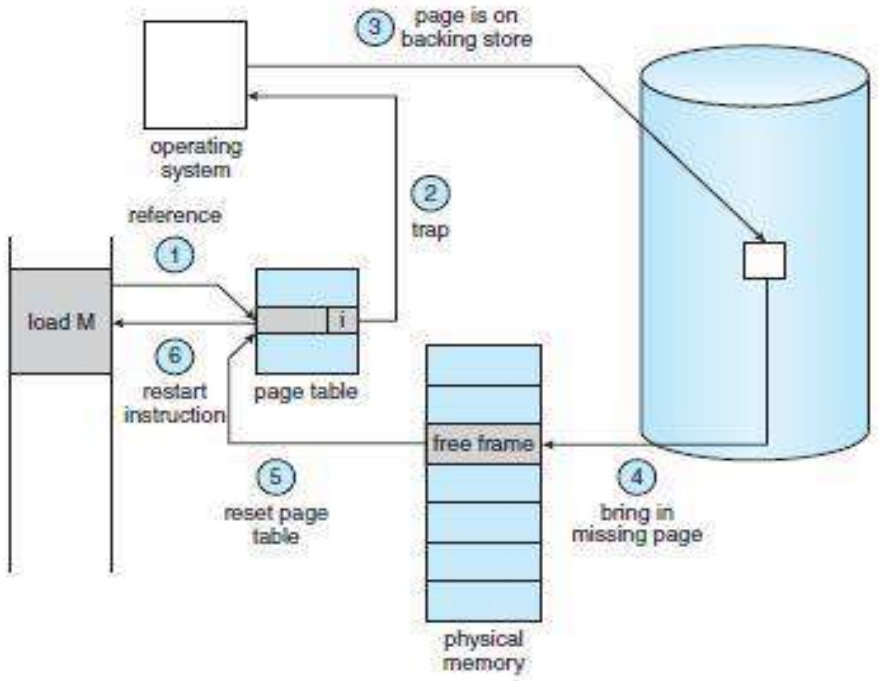
2. If the reference was invalid, we terminate the process. If it was valid but we have not yet brought in that page, we now page it in.

3. We find a free frame (by taking one fromthe free-frame list, for example).

4. We schedule a disk operation to read the desired page into the newly allocated frame.

5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.

6. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.



9.2 Performance of Demand Paging

Effective Access Time (EAT) for a demand-paged memory.
Memory Access Time (ma) for most computers now ranges from 10 to 200 nanoseconds.
If there is no page fault, then EAT = ma.
If there is page fault, then

EAT = (1 – p) x (ma) + p x (page-fault time).
p: the probability of a page fault ($0 \leq p \leq 1$),
we expect p to be close to zero (a few page faults).
If p=0 then no page faults, but if p=1 then every reference is a fault

If a page fault occurs, we must first read the relevant page from disk, and then access the desired word.

Example

Assume an average page-fault service time of 25 milliseconds (10-3), and a Memory Access Time of 100 nanoseconds (10-9). Find the Effective Access Time?

Solution: Effective Access Time (EAT)

= (1 – p) x (ma) + p x (page fault time)
= (1 – p) x 100 + p x 25,000,000
= 100 – 100 x p + 25,000,000 x p
= 100 + 24,999,900 x p.

•Note: The Effective Access Time is directly proportional to the page-fault rate.

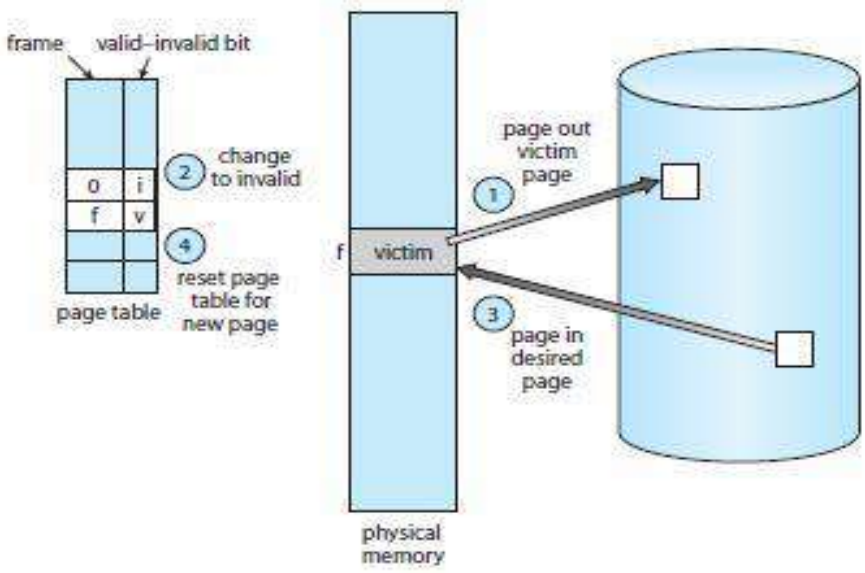
10. PAGE REPLACEMENT

10.1 Page fault

A page fault is a type of interrupt, raised by the hardware when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory.

Need for page replacement

Page replacement is needed to decide which page needed to be replaced when new page comes in.



1. Find the location of the desired page on the disk.
2. Find a free frame:
    a. If there is a free frame, use it.
    b. If there is no free frame, use a page-replacement algorithm to select a victim frame.
    c. Write the victim frame to the disk; change the page and frame tables accordingly.
3. Read the desired page into the newly freed frame; change the page and frame tables.
4. Continue the user process from where the page fault occurred.

10.2 Page replacement algorithms

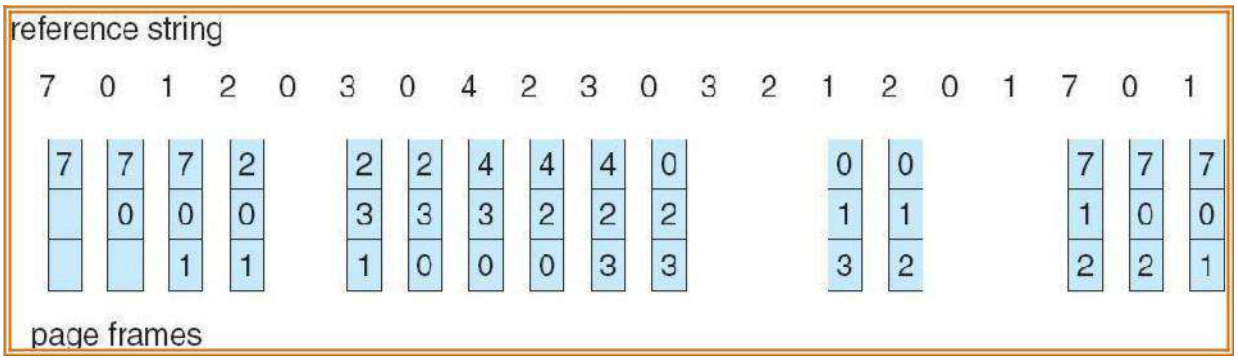
(a) FIFO page replacement algorithm

This is the simplest page replacement algorithm. In this algorithm, operating system keeps track of all pages in the memory in a queue, oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

Example:

Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

No.of available frames = 3 (3 pages can be in memory at a time per process)

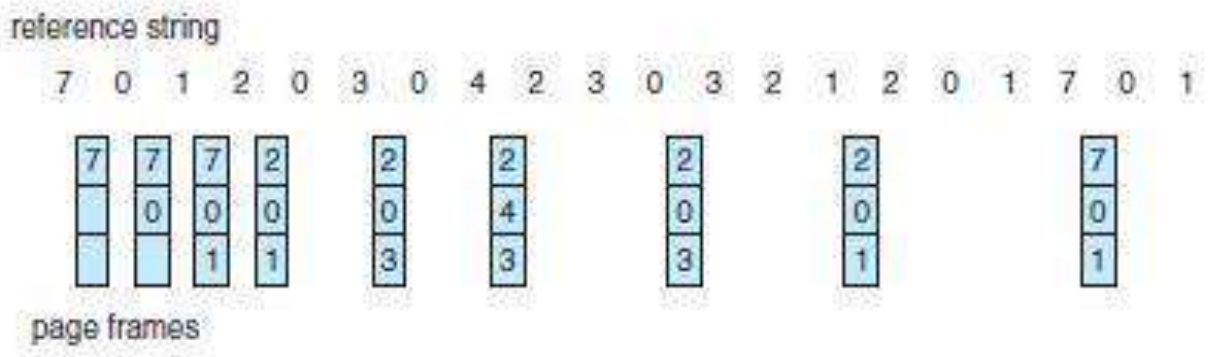


No. of page faults = 15

(b) Optimal page replacement algorithm

In this algorithm, pages are replaced which are not used for the longest duration of time in the future.

Example:



No. of page faults = 9

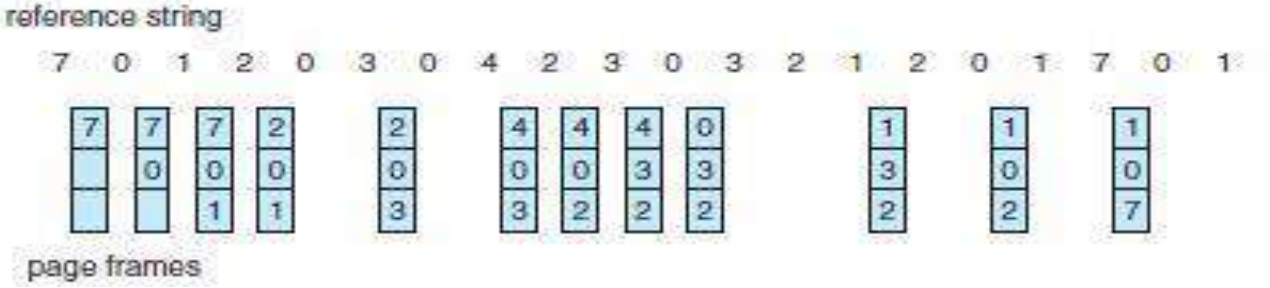
(c) LRU(Least Recently Used) page replacement algorithm

In this algorithm page will be replaced which is least recently used.

Example:

Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

No.of available frames = 3

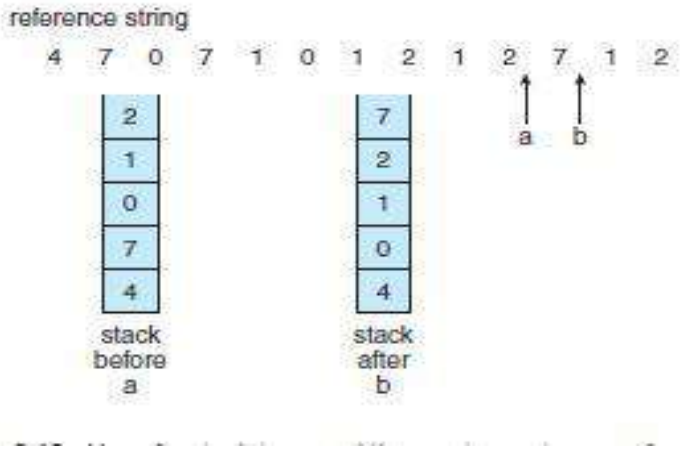


Page Fault =12

Implementation of LRU

1. **Counter**
   - The counter or clock is incremented for every memory reference.
   - Each time a page is referenced , copy the counter into the time- of-use field.
   - When a page needs to be replaced, replace the page with the smallest counter value.
2. **Stack**
   - Keep a stack of page numbers
   - Whenever a page is referenced, remove the page from the stack and put it on top of the stack.
   - When a page needs to be replaced, replace the page that is at the bottom of the stack.(LRU page)

Use of A Stack to Record The Most Recent Page References



(d) LRU Approximation Page Replacement

- Reference bit
  - With each page associate a reference bit, initially set to 0
  - When page is referenced, the bit is set to 1
- When a page needs to be replaced, replace the page whose reference bit is 0
- The order of use is not known , but we know which pages were used and which were not used.

(i) Additional Reference Bits Algorithm

- Keep an 8-bit byte for each page in a table in memory.
- At regular intervals , a timer interrupt transfers control to OS.
- The OS shifts reference bit for each page into higher- order bit shifting the other bits right 1 bit and discarding the lower-order bit.

Example:

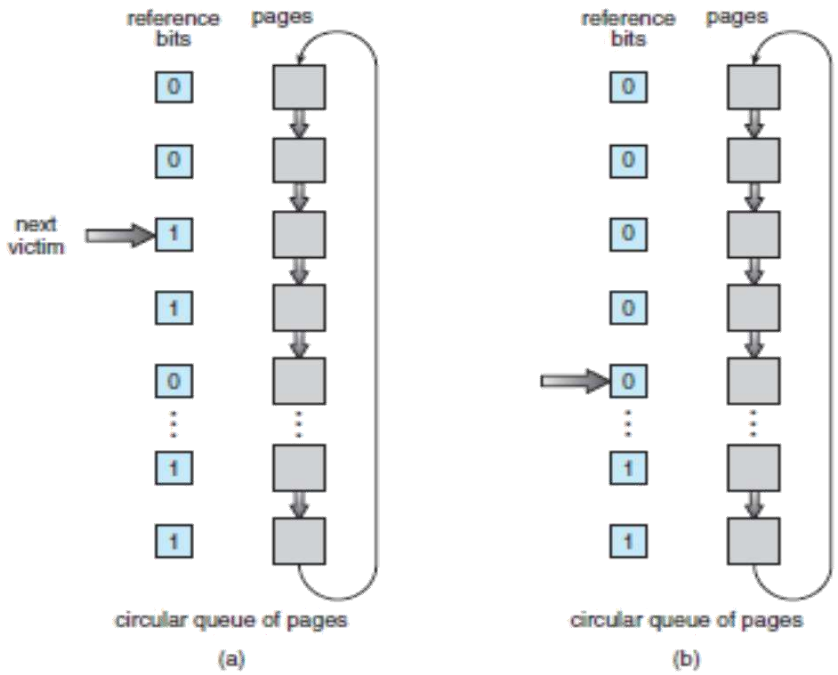
oIf reference bit is 00000000 then the page has not been used for 8 time periods.
oIf reference bit is 11111111 then the page has been used atleast once each time period. oIf the reference bit of page 1 is 11000100 and page 2 is 01110111 then page 2 is the LRU page.

(ii) Second Chance Algorithm

oBasic algorithm is FIFO

oWhen a page has been selected , check its reference bit.

- If 0 proceed to replace the page
- If 1 give the page a second chance and move on to the next FIFO page.
- When a page gets a second chance, its reference bit is cleared and arrival time is reset to current time.
- Hence a second chance page will not be replaced until all other pages are replaced.



(iii) Enhanced Second Chance Algorithm

o Consider both reference bit and modify bit o

There are four possible classes

1. (0,0) – neither recently used nor modified est page to replace
2. (0,1) – not recently used but modifiedpage has to be written out before replacement.
3. (1,0) - recently used but not modified page may be used again
4. (1,1) – recently used and modifiedpage may be used again and page has to be written to disk

(iv) Counting-Based Page Replacement

o Keep a counter of the number of references that have been made to each page

1. **Least Frequently Used (LFU)Algorithm**: replaces page with smallest count
2. **Most Frequently Used (MFU)Algorithm**: replaces page with largest count
   - □ is based on the argument that the page with the smallest count was probably just brought in and has yet to be used

11. ALLOCATION OF FRAMES

11.1 Allocation of Frames

o There are two major allocation schemes

- □ Equal Allocation
- □ Proportional Allocation

Equal allocation

- If there are n processes and m frames then allocate m/n frames to each process.
- **Example:** If there are 5 processes and 100 frames, give each process 20 frames.
- Allocate according to the size of process

Let si be the size of process i.

Let m be the total no. of

frames Then $S = \sum si$

$ai = si / S * m$

where ai is the no.of frames allocated to process i.

11.2 Global vs. Local Replacement

- **Global replacement** – each process selects a replacement frame from the set of all frames; one process can take a frame from another.
- **Local replacement** – each process selects from only its own set of allocated frames.

With proportional allocation, we would split 62 frames between two processes, one of 10 pages and one of 127 pages, by allocating 4 frames and 57 frames

$10/137 \times 62 \approx 4$, and

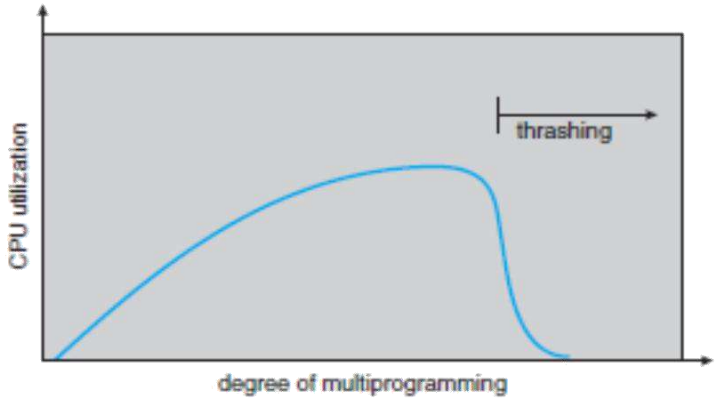
$127/137 \times 62 \approx 57$.

12. THRASHING

Thrashing

- High paging activity is called **thrashing**.
- If a process does not have enough pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system thinks that it needs to increase the degree of multiprogramming
  - another process is added to the system
- When the CPU utilization is low, the OS increases the degree of multiprogramming.
- If global replacement is used then as processes enter the main memory they tend to steal frames belonging to other processes.
- Eventually all processes will not have enough frames and hence the page fault rate becomes very high.

o Thus swapping in and swapping out of pages only takes place.
o This is the cause of thrashing.

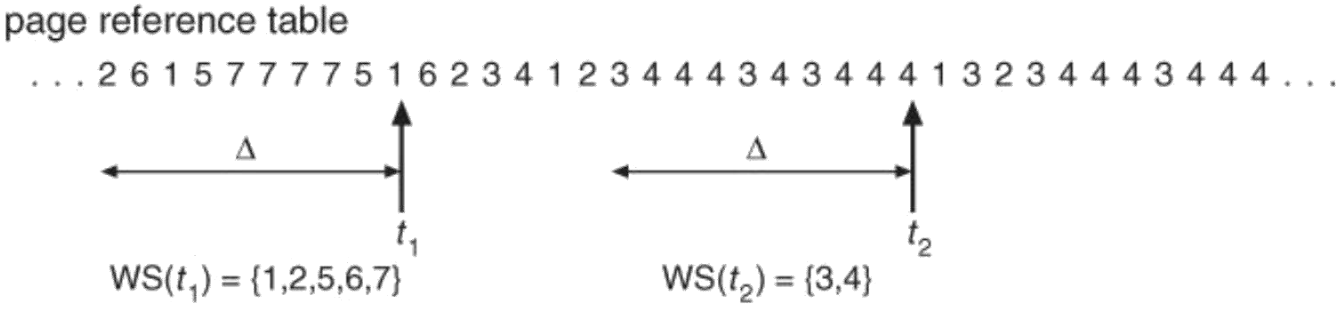


o To **limit thrashing**, we can use a **local replacement** algorithm. o To prevent thrashing, there are two methods namely ,

- Working Set Strategy
- Page Fault Frequency

1. Working-Set Strategy

o It is based on the assumption of the model of locality.
o Locality is defined as the set of pages actively used together.
o Whatever pages are included in the most recent page references are said to be in the processes working set window, and comprise its current working set .



If a page is in active use, it will be in the working set. If it is no longer being used, it will drop from the working set time units after its last reference.

- Thus, the working set is an approximation of the program's locality.
- if Δ = 10 memory references, then the working set at time t1 is {1, 2, 5,6, 7}.
- By time t2, the working set has changed to {3, 4}.
- The accuracy of the working set depends on the selection of .
- If Δ is too small, it will not encompass the entire locality; if is too large, it may overlap several localities.
- In the extreme, if Δ is infinite, the working set is the set of pages touched during the process execution.

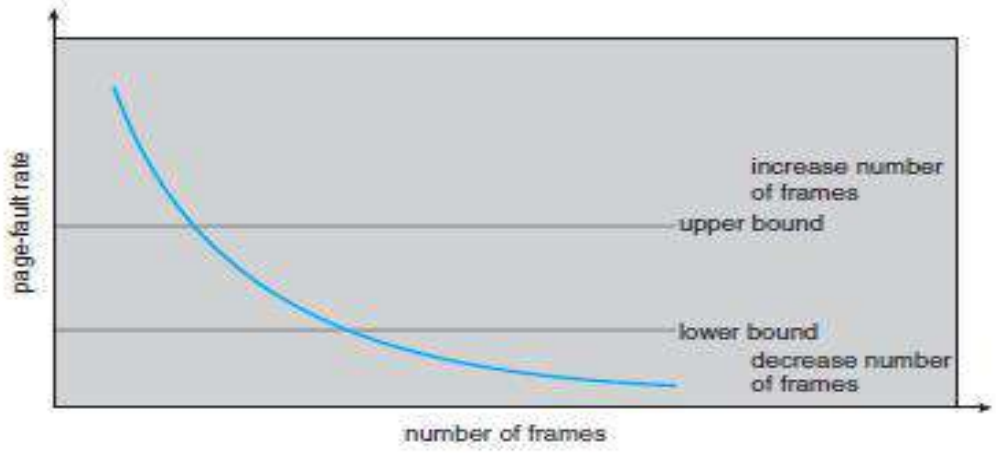
 The most important property of the working set, then, is its size.
 If we compute the working-set size, WSSi , for each process in the system, we can then consider that

$$D = \sum WSS_i,$$

 where D is the total demand for frames. Each process is actively using the pages in its working set.
 Thus, process i needs WSSi frames. If the total demand is greater than the total number of available frames (D> m), thrashing will occur, because some processes will not have enough frames.

2. Page-Fault Frequency Scheme

- Thrashing has a high page-fault rate. Thus, we want to control the page-fault rate.
- When it is too high, we know that the process needs more frames. Conversely, if the page-fault rate is too low, then the process may have too many frames.
- We can establish upper and lower bounds on the desired page-fault rate.
- If the actual page-fault rate exceeds the upper limit, we allocate the process another frame.
- If the page-fault rate falls below the lower limit, we remove a frame from the process.
- Thus, we can directly measure and control the page-fault rate to prevent thrashing.



13. ALLOCATING KERNEL MEMORY

Allocating Kernel Memory

When a process running in user mode requests additional memory, pages are allocated from the list of free page frames maintained by the kernel. This list is typically populated using a page-replacement algorithm such as those discussed in Section 9.4 and most likely contains free pages scattered throughout physical memory, as explained earlier. Remember, too, that if a user process requests a single byte of memory, internal fragmentation will result, as the process will be granted an entire page frame.

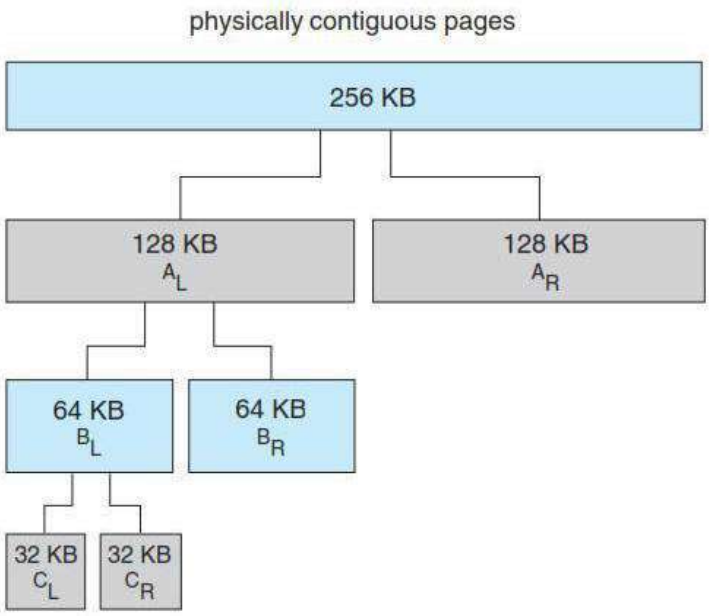
Kernel memory is often allocated from a free-memory pool different from the list used to satisfy ordinary user-mode processes. There are two primary reasons for this:

1. The kernel requests memory for data structures of varying sizes, some of which are less than a page in size. As a result, the kernel must use memory conservatively and attempt to minimize waste due to fragmentation. This is especially important because many operating systems do not subject kernel code or data to the paging system.

2. Pages allocated to user-mode processes do not necessarily have to be in contiguous physical memory. However, certain hardware devices interact directly with physical memory—without the benefit of a virtual memory interface—and consequently may require memory residing in physically contiguous pages.

Buddy System

The buddy system allocates memory from a fixed -size segment consisting of physically contiguous pages. Memory is allocated from this segment using a **power-of-2 allocator**, which satisfies requests in units sized as a power of 2 (4KB, 8KB, 16KB, and so forth). A request in units not appropriately sized is rounded up to the next highest power of 2. For example, a request for 11 KB is satisfied with a 16K segment



Let's consider a simple example. Assume the size of a memory segment is initially 256 KB and the kernel requests 21 KB of memory.

The segment is initially divided into two buddies—which we will call AL and AR—each 128 KB in size. One of these buddies is further divided into two 64-KB buddies— BL and BR.

However, the next-highest power of 2 from 21 KB is 32 KB so either BL or BR is again divided into two 32-KB buddies, CL and CR. One of these buddies is used to satisfy the 21-KB request.

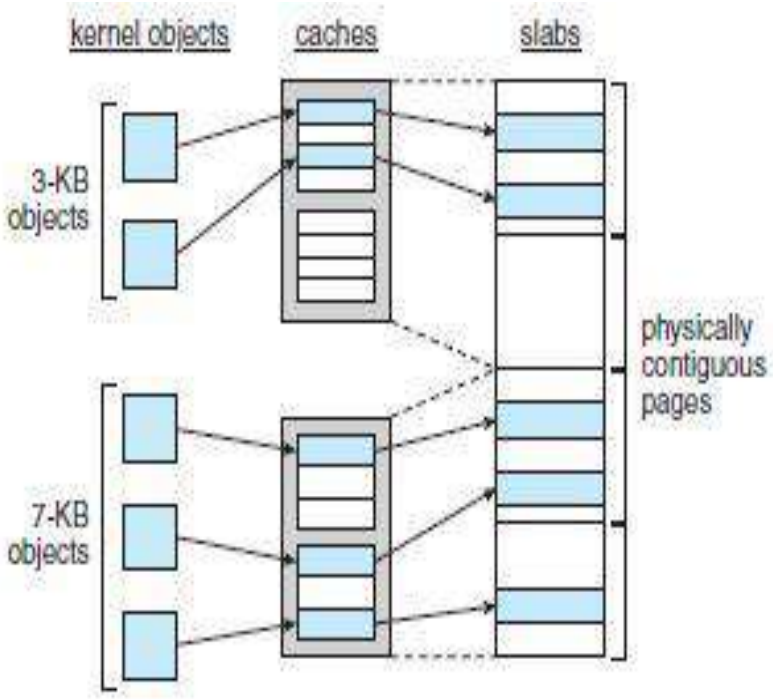
Slab Allocation

A second strategy for allocating kernel memory is known as slab allocation. A slab is made up of one or more physically contiguous pages. A cache consists of one or more slabs.

The slab-allocation algorithm uses caches to store kernel objects.

When a cache is created, a number of objects which are initially marked as free are allocated to the cache. The number of objects in the cache depends on the size of the associated slab.

For example, a 12-KB slab (made up of three contiguous 4-KB pages) could store six 2-KB objects.



In Linux, a slab may be in one of three possible states:

1. Full. All objects in the slab are marked as used.
2. Empty. All objects in the slab are marked as free.
3. Partial. The slab consists of both used and free objects.

The slab allocator first attempts to satisfy the request with a free object in a partial slab.

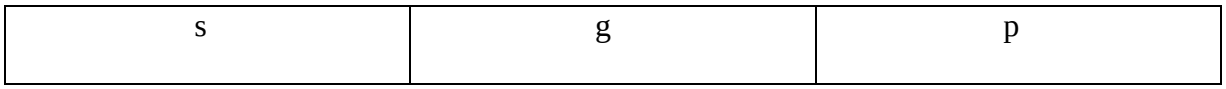
If none exists, a free object is assigned from an empty slab.

If no empty slabs are available, a new slab is allocated from contiguous physical pages and assigned to a cache; memory for the object is allocated from this slab.

14. SEGMENTATION WITH PAGING

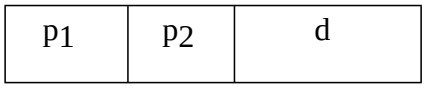
- The IBM OS/ 2.32 bit version is an operating system running on top of the Intel 386 architecture. The 386 uses segmentation with paging for memory management. The maximum number of segments per process is 16 KB, and each segment can be as large as 4 gigabytes.
- The local-address space of a process is divided into two partitions.
  The first partition consists of up to 8 KB segments that are private to that process.
  The second partition consists of up to 8KB segments that are shared among all the processes.
- Information about the first partition is kept in the **local descriptor table (LDT)**, information about the second partition is kept in the **global descriptor table (GDT)**.
- Each entry in the LDT and GDT consist of 8 bytes, with detailed information about a particular segment including the base location and length of the segment.

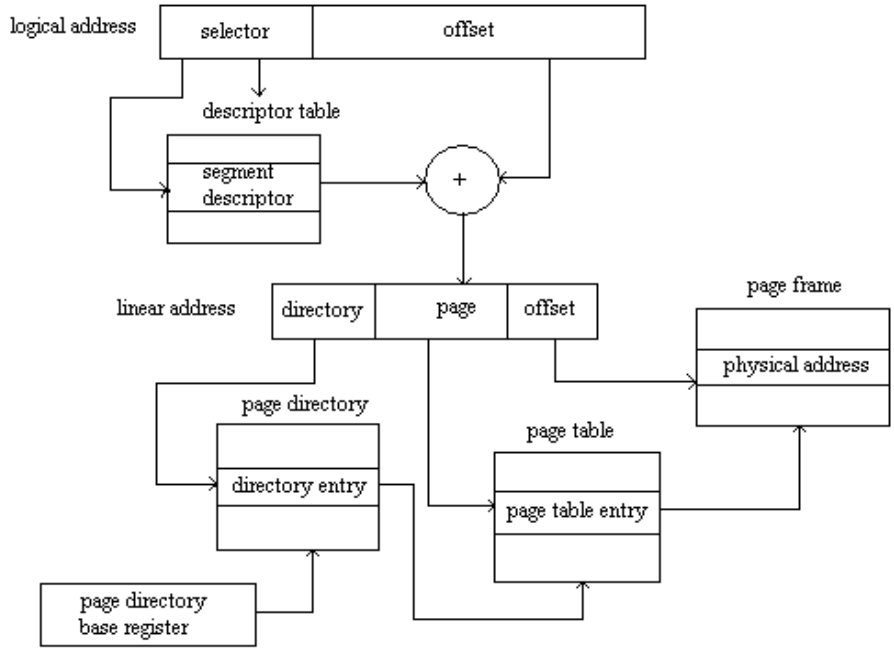
The logical address is a pair (selector, offset) where the selector is a16-bit number:



Where s designates the segment number, g indicates whether the segment is in the GDT or LDT, and p deals with protection. The offset is a 32-bit number specifying the location of the byte within the segment in question.

- The base and limit information about the segment in question are used to generate a linear-address.
- First, the limit is used to check for address validity. If the address is not valid, a memory fault is generated, resulting in a trap to the operating system. If it is valid, then the value of the offset is added to the value of the base, resulting in a 32-bit linear address. This address is then translated into a physical address.
- The linear address is divided into a page number consisting of 20 bits, and a page offset consisting of 12 bits. Since we page the page table, the page number is further divided into a 10-bit page directory pointer and a 10-bit page table pointer. The logical address is as follows.





oTo improve the efficiency of physical memory use. Intel 386 page tables can be swapped to disk. In this case, an invalid bit is used in the page directory entry to indicate whether the table to which the entry is pointing is in memory or on disk.

oIf the table is on disk, the operating system can use the other 31 bits to specify the disk location of the table; the table then can be brought into memory on demand.

UNIT IV FILE SYSTEMS AND I/O SYSTEMS

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

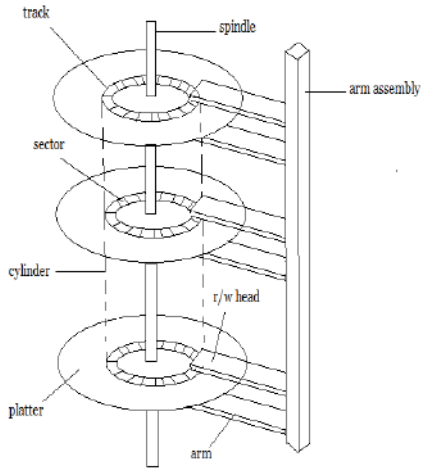
MASS STORAGE STRUCTURE

1. Overview of Mass Storage Structure

Magnetic Disks

- In modern computers, most of the secondary storage is in the form of magnetic disks.
- A magnetic disk contains several platters. Each platter is divided into circular shaped tracks.
- The length of the tracks near the centre is less than the length of the tracks farther from the centre.
- Each track is further divided into sectors.
- Tracks of the same distance from centre form a cylinder.
- A read-write head is used to read data from a sector of the magnetic disk.
- The speed of the disk is measured as two parts:

Transfer rate: This is the rate at which the data moves from disk to the computer.
Random access time: It is the sum of the seek time and rotational latency.
Seek time is the time taken by the arm to move to the required track.
Rotational latency is defined as the time taken by the arm to reach the required sector in the track.



Solid-State Disks

SD is non-volatile memory that is used like a hard drive. SSDs have the same characteristics as traditional hard disks but can be more reliable because they have no

moving parts and faster because they have no seek time or latency. In addition, they consume less power.

SSDs have less capacity than the larger hard disks, and may have shorter life spans. use for SSDs is in storage arrays, where they hold file- system metadata that require high performance. Some SSDs are designed to connect directly to the system bus.

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.

2. Disk Structure

In Disk drives are addressed as large 1-dimensional arrays of logical blocks, where the logical block is the smallest unit of transfer. n The 1-dimensional array of logical blocks is mapped into the sectors of the disk sequentially.

➔In Sector 0 is the first sector of the first track on the outermost cylinder.

➔In Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

3. Disk Scheduling

The operating system is responsible for using hardware efficiently.

For the disk drives, this means having a fast access time & disk bandwidth.

➔Access time has two major components:

➔Seek time is the time for the disk to move the heads to the cylinder containing the desired sector

➔Rotational latency time waiting for the disk to rotate the desired sector to the disk head

➔We like to minimize seek time.

➔ Disk bandwidth is the total number of bytes transferred divided by the total time between the first request for service and the completion of the last transfer.

➔Several algorithms exist to schedule the servicing of disk I/O requests.

We illustrate them with a Request Queue (cylinder range 0-199):

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

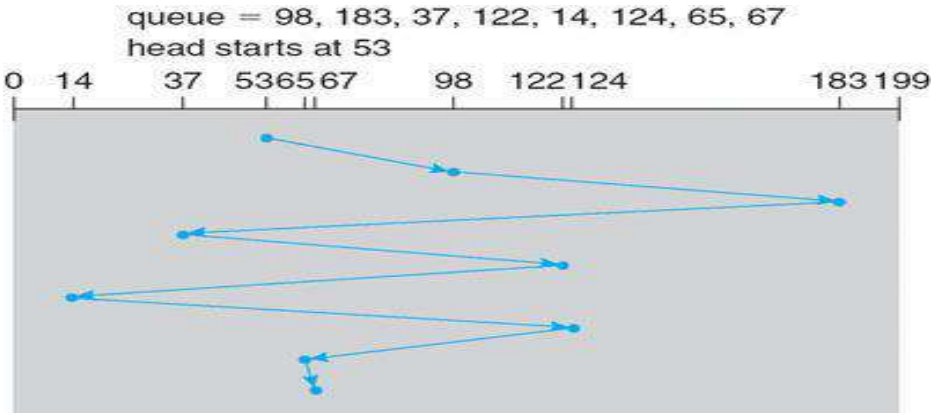
Head pointer: cylinder 53

1. First Come First Serve

This algorithm performs requests in the same order asked by the system. Let's take an example where the queue has the following requests with cylinder numbers as follows:

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

Illustration shows total head movement of 640 cylinders

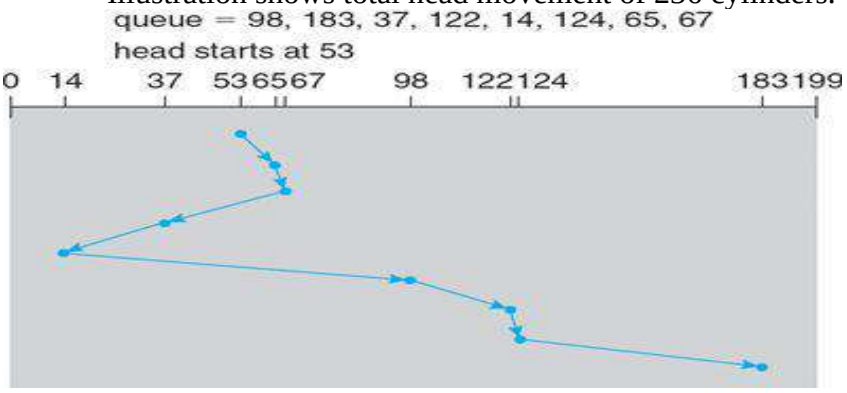


2. SSTF (Shortest Seek Time First)

Selects the request with the minimum seek time from the current head position.

SSTF scheduling is a form of SJF scheduling; may cause starvation of some requests.

Illustration shows total head movement of 236 cylinders.

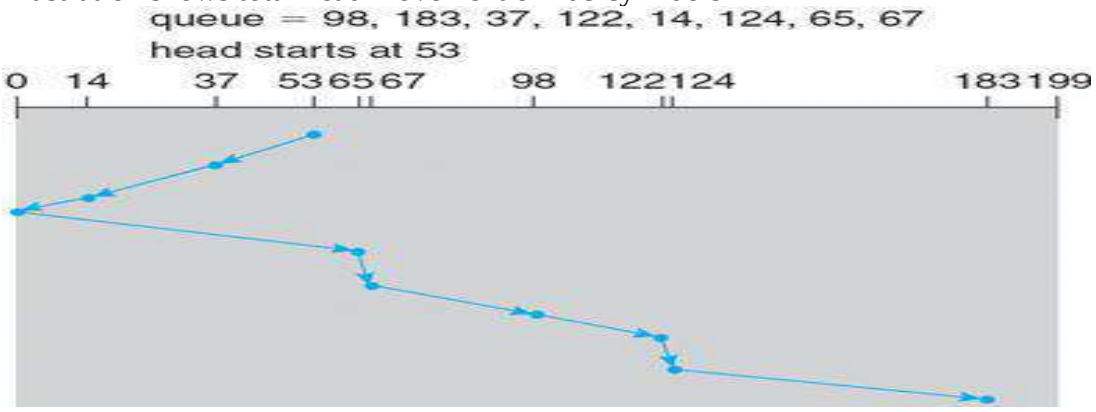


3. **SCAN**

The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.

SCAN algorithm sometimes called the elevator algorithm.

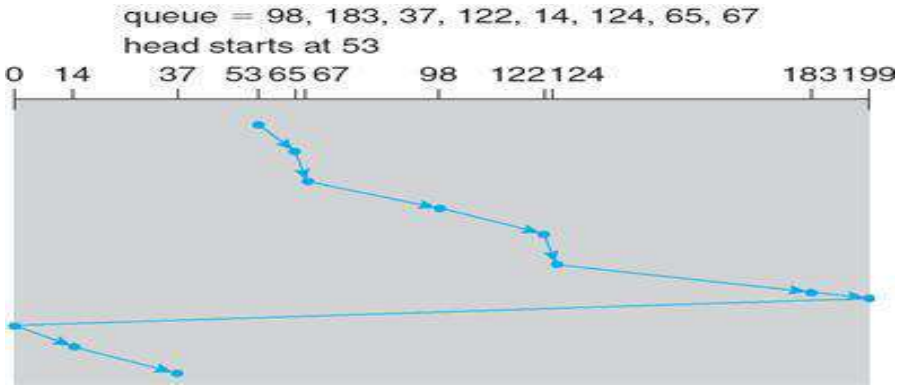
Illustration shows total head movement of 208 cylinders



4. C-SCAN

Provides a more uniform wait time than SCAN. The head moves from one end of the disk to the other, servicing requests as it goes When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip.

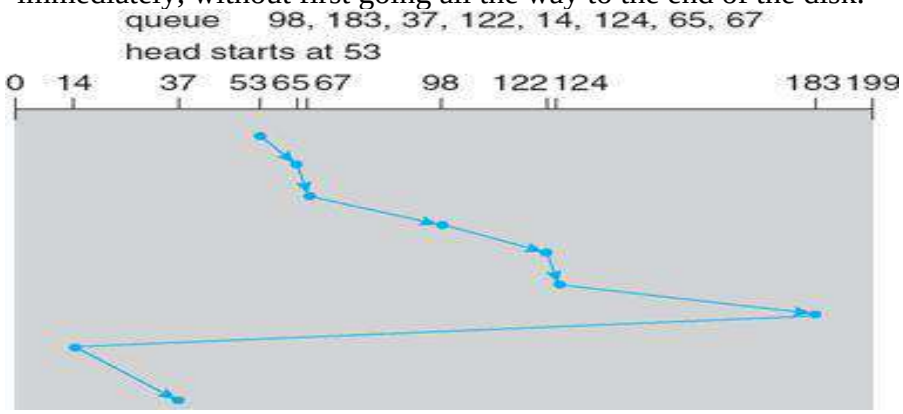
Treats the cylinders as a Circular list that wraps around from the last cylinder to the first one .



5.LOOK

➔ Version of C-SCAN

➔ Arm only goes as far as the **last request** in each direction,then reverses direction immediately, without first going all the way to the end of the disk.



4. Disk Management

The operating system is responsible for disk initialization, booting from disk, and bad-block recovery.

Disk Formatting

A new magnetic disk must be divided into sectors that the disk controller can read and write. This process is called **low-level formatting, or physical formatting**. Low-level formatting fills the disk with a special data structure for each sector. The data structure for a sector typically consists of a header, a data area (usually 512 bytes in size), and a trailer.

The header and trailer contain information used by the disk controller, such as a sector number and an **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.

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.

The bootstrap is stored in read-only memory (ROM). This location is convenient, because ROM needs no initialization and is at a fixed location that the processor can start executing when powered up or reset. And, since ROM is read only, it cannot be infected by a computer virus.

The full bootstrap program is stored in the “**boot blocks**” at a fixed location on the disk. A disk that has a boot partition is called a boot disk or system disk. The work of boot block as follows

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 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.

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:

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

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.

5. Swap-Space Management

➔Swap-space — virtual memory uses disk space as an extension of main memory.

➔Main goal for the design and implementation of swap space is to provide the best throughput for VM system

1.Swap-space use

Swapping –use swap space to hold entire process image

Paging –store pages that have been pushed out of memory

➔Some OS may support multiple swap-space

–Put on separate disks to balance the load

➔Better to overestimate than underestimate

–If out of swap-space, some processes must be aborted or system crashed

2**. Swap-Space Location**

➔Swap-space can be carved out of the normal file system, or in a separate disk partition

➔A large file within the file system: simple but inefficient

–Navigating the directory structure and the disk-allocation data structure takes time and potentially extra disk accesses

–External fragmentation can greatly increase swapping times by forcing multiple seeks during reading or writing of a process image

–Improvement

Caching block location information in main memory

•Contiguous allocation for the swap file

But, the cost of traversing FS data structure still remains

➔In a separate partition: raw partition

–Create a swap space during disk partitioning

–A separate swap-space storage manager is used to allocate and de-allocate blocks

–Use algorithms optimized for speed, rather than storage efficiency

–Internal fragment may increase

Linux supports both approaches

➔Swap-space Management: Example

Solaris 1

–Text-segment pages are brought in from the file system and are thrown away if selected for paged out

More efficient to re-read from FS than write it to the swap space

-Swap space: only used as a backing store for pages of anonymous memory

Stack, heap, and uninitialized data

Solaris 2

–Allocates swap space only when a page is forced out of physical memory

Not when the virtual memory page is first created.

FILE SYSTEM INTERFACE

1. File Concepts

A file is a named collection of related information that is recorded on secondary storage. From user's perspective a , a file is the smallest allotment of that logical secondary storage; unless they are within a file. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary.

In general, a file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user.

A **text file** is a sequence of characters organized into lines (and possibly pages).

An **executable** file is a series of code sections that the loader can bring into memory and execute.

2. File Attributes

The information about all files is kept in the directory structure, a directory entry consists of the file's name and its unique identifier. The identifier in turn locates the other file attributes.

- **Name:** The symbolic file name is the only information kept in human readable form.

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

3.File Operations

The operating system can provide system calls to create, write, read, reposition, delete, and truncate files.

Creating a file - First, space in the file system must be found for the file, Second, an entry for the new file must be made in the directory.

Writing a file - System call specifying both the name of the file and the information to be written to the file.

Reading a file - we use a system call that specifies the name of the file and where (in memory) the next block of the file should be put.

Repositioning within a file - The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value.

Deleting a file - search the directory for the named file. Having found the associated directory entry, we release all file space

Truncating a file - this function allows all attributes to remain unchanged—except for file length—but lets the file be reset to length zero and its file space released.

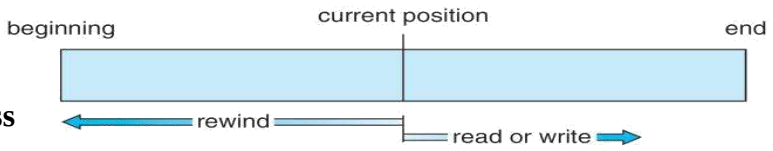
4. **File Types**

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, 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

5. **Access Methods**

1.Sequential Access

- Data is accessed one record right after another is an order.
- Read command cause a pointer to be moved ahead by one.
- Write command allocate space for the record and move the pointer to the new End Of File.
- Such a method is reasonable for tape.



2. **Direct Access**

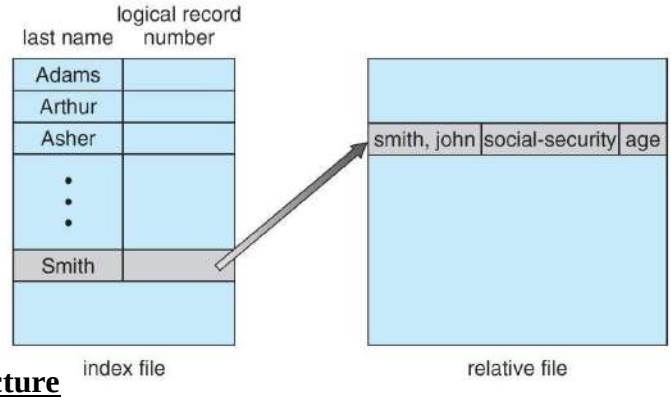
- This method is useful for disks.
- The file is viewed as a numbered sequence of blocks or records.
- There are no restrictions on which blocks are read/written, it can be dobe in any order.
- User now says "read n" rather than "read next".
- "n" is a number relative to the beginning of file, not relative to an absolute physical disk location.

As a simple example, on an **airline – reservation system**, we might store all the information about a particular flight (for example, flight 713) in the block identified by the flight number.

Thus, the number of available seats for flight 713 is stored in block 713 of the reservation file. To store information about a larger set, such as people, we might compute a hash function on the people's names, or search a small in-memory index to determine a block to read and search.

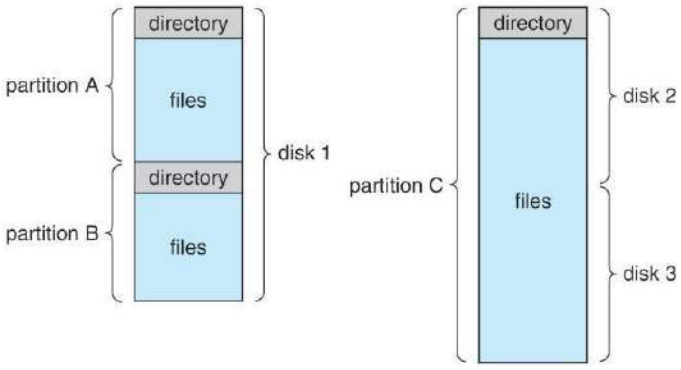
3. Indexed Access

- If a file can be sorted on any of the filed then an index can be assigned to a group of certain records.
- However, A particular record can be accessed by its index.
- The index is nothing but the address of a record in the file.
- In index accessing, searching in a large database became very quick and easy but we need to have some extra space in the memory to store the index value.



6.Directory Structure

- Directory can be defined as the listing of the related files on the disk.
- The directory may store some or the entire file attributes.
- Each partition must have at least one directory in which, all the files of the partition can be listed.
- A directory entry is maintained for each file in the directory which stores all the information related to that file.



Operations that are to be performed on a directory

Search for a file. We need to be able to search a directory structure to find the entry for a particular file. Since files have symbolic names, and similar names

may indicate a relationship among files, we may want to be able to find all files whose names match a particular pattern.

Create a file. New files need to be created and added to the directory.
Delete a file. When a file is no longer needed, we want to be able to remove it from the directory.
List a directory. We need to be able to list the files in a directory and the contents of the directory entry for each file in the list.
Rename a file. Because the name of a file represents its contents to its users, we must be able to change the name when the contents or use of the file changes. Renaming a file may also allow its position within the directory structure to be changed.
Traverse the file system. We may wish to access every directory and every file within a directory structure. For reliability, it is a good idea to save the contents and structure of the entire file system at regular intervals.

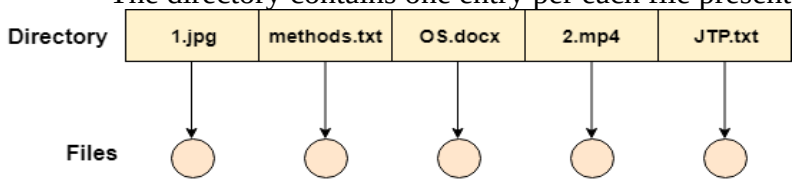
➔Often, we do this by copying all files to magnetic tape. This technique provides a backup copy in case of system failure.
➔In addition, if a file is no longer in use, the file can be copied to tape and the disk space of that file released for reuse by another file.

Logical Structure (or) Level of Directory

- Single-level directory
- Two-level directory
- Tree-Structured directory
- Acyclic Graph directory
- General Graph directory

Single – Level Directory

- 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.



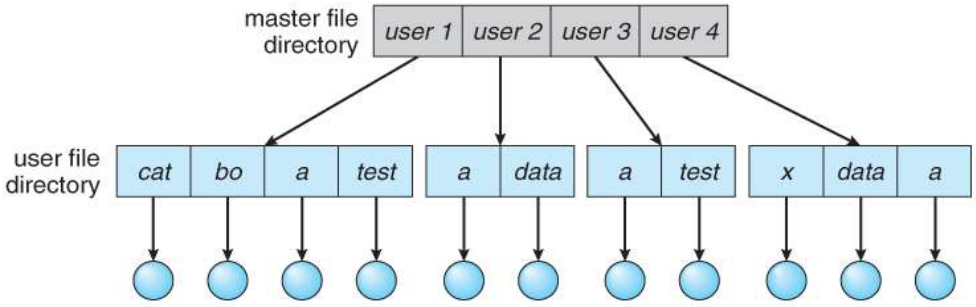
Single Level Directory

Disadvantages

1. We cannot have two files with the same name.
2. The directory may be very big therefore searching for a file may take so much time.
3. Protection cannot be implemented for multiple users.
4. There are no ways to group same kind of files.

Two Level Directory

- In two level directory systems, we can create a separate directory for each user.
- There is one master directory which contains separate directories dedicated to each user. For each user, there is a different directory present at the second level, containing group of user's file.
- The system doesn't let a user to enter in the other user's directory without permission.



Characteristics of two level directory system

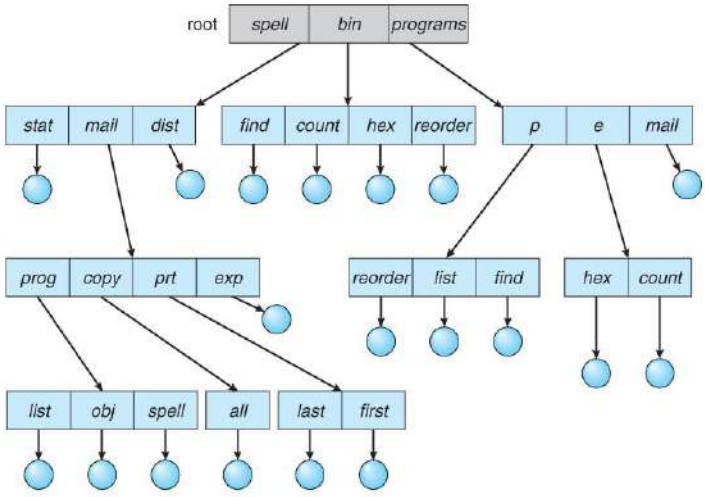
1.Each files has a path name as /User-name/directory-name/

2. Different users can have the same file name.

3. Searching becomes more efficient as only one user's list needs to be traversed.

Tree Structured Directory

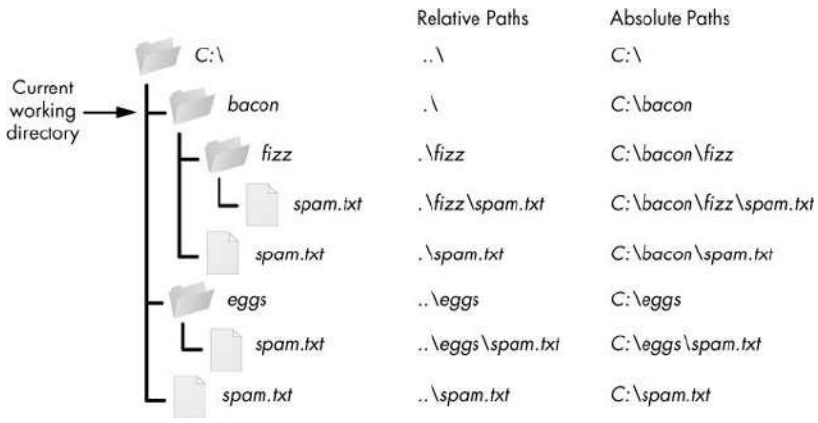
- Tree structured directory system overcomes the drawbacks of two level directory system.
- The similar kind of files can now be grouped in one directory.
- Each user has its own directory and it cannot enter in the other user's directory.
- Searching is more efficient in this directory structure



A file can be accessed by two types of path, either ➔ 1.Relative or 2. Absolute.

1. **Absolute path** is the path of the file with respect to the root directory of the system.

2. **Relative path** is the path with respect to the current working directory of the system

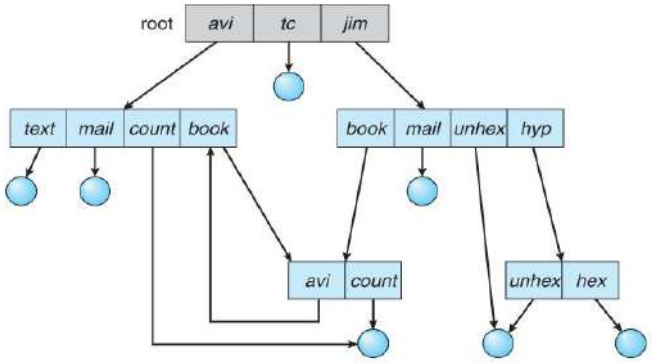


Acyclic-Graph Structured Directories

- When the **same files need to be accessed** in **more than one place** in the directory structure it can be useful to provide an acyclic-graph structure.
- In this system two or more directory entry can point to the same file or sub directory. That file or sub directory is shared between the two directory entries.
  It provides two types of ***links*** for implementing the acyclic-graph structure
  Soft link, the file just gets deleted and we are left with a dangling pointer.
  Hard link, the actual file will be deleted only if all the references to it gets deleted.

General Graph Directory

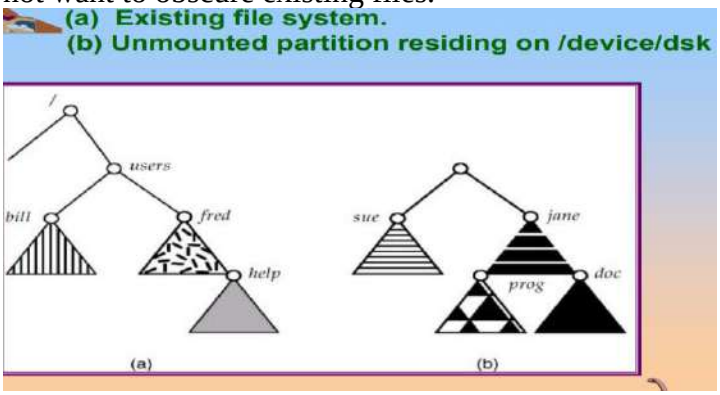
- In general graph directory structure, cycles are allowed within a directory structure where multiple directories can be derived from more than one parent directory
- The main problem with this kind of directory structure is to calculate total size or space that have been taken by the files and directories.

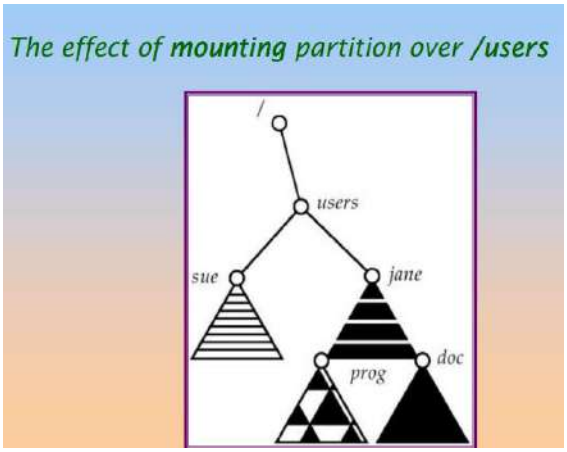


7.File System Mounting

- Before you can access the files on a file system, you need to mount the file system.

- Mounting a file system attaches that file system to a directory (mount point) and makes it available to the system.
- The root (/) file system is always mounted. Any other file system can be connected or disconnected from the root (/) file system.
- When you mount a file system, any files or directories in the underlying mount point directory are unavailable as long as the file system is mounted.
- These files are not permanently affected by the mounting process, and they become available again when the file system is unmounted.
- However, mount directories are typically empty, because you usually do not want to obscure existing files.





8. File Sharing

- File sharing is the accessing or sharing of files by one or more users.
- File sharing is performed on computer networks as an easy and quick way to transmit data.

For example, a user may share an instruction document on his computer that is connected to a corporate network allowing all other employees to access and read that document.

1.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.

2. 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.
- ❖ The Client-Server Model(the system which physically owns the files acts as a *server*, and the system which mounts them is the *client.)*
- ❖ Distributed Information Systems➔ service that runs on a single 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. Remote access systems allow for blocking or delayed response.

3. 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?

1. UNIX Semantics

➔ Writes to an open file are immediately visible to any other user who has the file open.

2. Session Semantics

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.

3. Immutable-Shared-Files Semantics

➔ 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.

9. File Protection

- ❖ Files must be kept safe for reliability (against accidental damage), and protection (against deliberate malicious access.) The former is usually managed with backup copies. This section discusses the latter.
- ❖ One simple protection scheme is to remove all access to a file. However this makes the file unusable, so some sort of controlled access must be arranged.

Types of Access

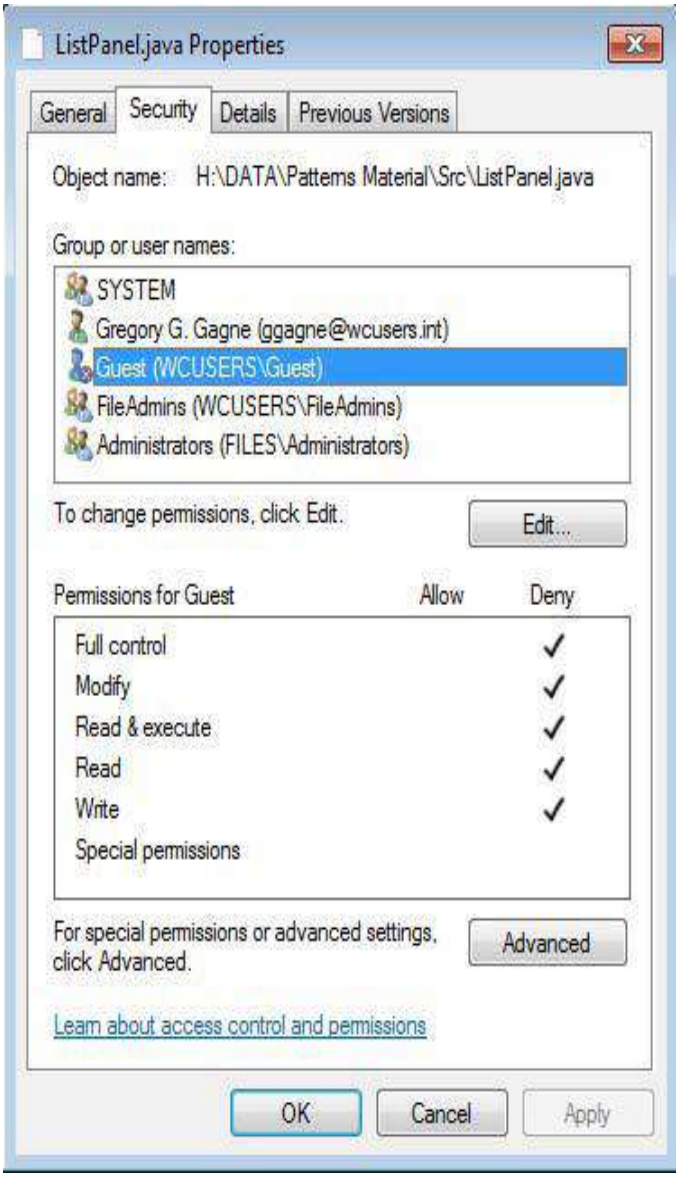
- The following low-level operations are often controlled:

    - 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 IMPLEMENTATION

1. File System Structure

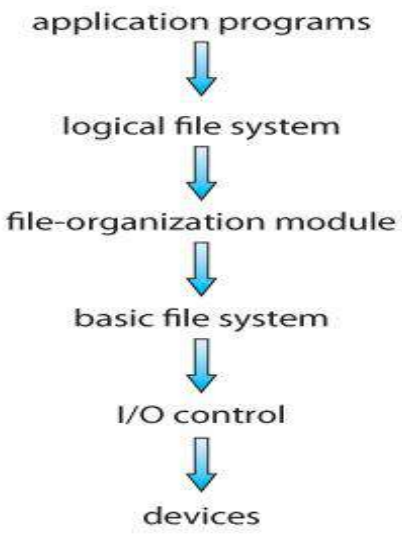
- File System provide efficient access to the disk by allowing data to be stored, located and retrieved in a convenient way.
- A file System must be able to store the file, locate the file and retrieve the file.
- Most of the Operating Systems use layering approach for every task including file systems.
- Every layer of the file system is responsible for some activities.

Logical file system

✂ **Provides** users the view of a contiguous sequence of words, bytes stored somewhere.

✂ Uses a directory structure, symbolic name
✂Provides protection and security
✂OS/user interface

☎E.g., to create a new file the API provides a call that calls the logical file system



The file organization module

✂Knows about files and their logical blocks (say 1,..N)
✂Files are organized in blocks of 32 bytes to 4K bytes
✂Translates logical blocks into physical
✂Knows location of file, file allocation type
✂Includes a free space manages that tracks unallocated blocks

Basic file system

✂Issues commands to the device driver (layer of software that directly controls disk hardware) to read and write physical blocks on the disk,
✂Each physical block identified by a disk address (e.g., drive 2, cylinder 34, track 2, sector 11)

IO control

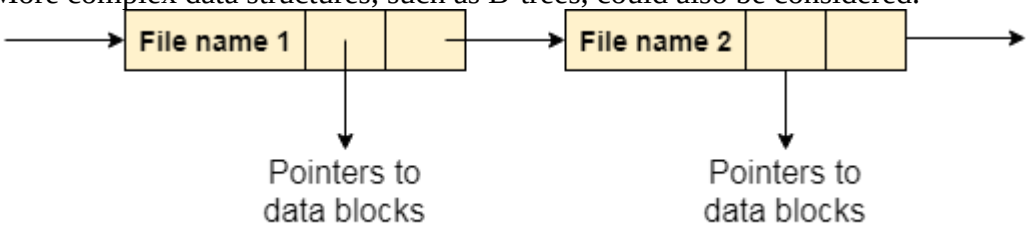
✂The lowest level in the file system
✂Consists of device drivers and interrupt handlers to transfer information between the memory and the disk
✂A device driver translates commands such as "get me block 111" into hardware specific ISA used by hardware controller. This is accomplished by writing specific bits into IO registers

2.Directory Implementation

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

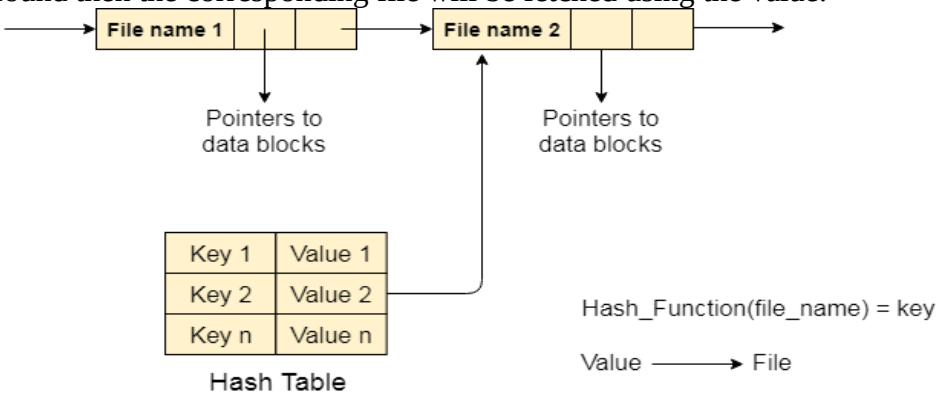
Linear List

- ❖ 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.
- ❖ A key-value pair for each file in the directory gets generated and stored in the hash table.
- ❖ The key can be determined by applying the hash function on the file name while the key points to the corresponding file stored in the directory.
- ❖ **Searching➔** Only hash table entries are checked using the key and if an entry found then the corresponding file will be fetched using the value.

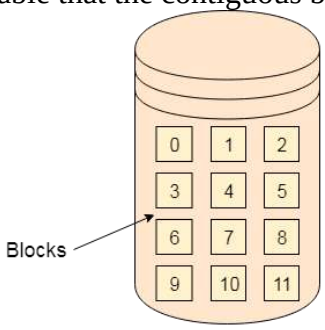


3. Allocation Methods

- There are various methods which can be used to allocate disk space to the files. Selection of an appropriate allocation method will significantly affect the performance and efficiency of the system.
- Allocation method provides a way in which the disk will be utilized and the files will be accessed.

Contiguous Allocation

- If the blocks are allocated to the file in such a way that all the logical blocks of the file get the contiguous physical block in the hard disk then such allocation scheme is known as contiguous allocation.
- In the image shown below, there are three files in the directory.
- The starting block and the length of each file are mentioned in the table. We can check in the table that the contiguous blocks are assigned to each file as per its need.



Hard Disk

File Name	Start	Length	Allocated Blocks
abc.text	0	3	0,1,2
video.mp4	4	2	4,5
jtp.docx	9	3	9,10,11

Directory

- All these algorithms suffer from the problem of external fragmentation.
- As files are allocated and deleted, the free disk space is broken into little pieces. External fragmentation exists whenever free space is broken into chunks.
- It becomes a problem when the largest contiguous chunk is insufficient for a request; storage is fragmented into a number of holes, none of which is large enough to store the data.
- This scheme effectively **compacts** all free space into one contiguous space, solving the fragmentation problem.

Advantages

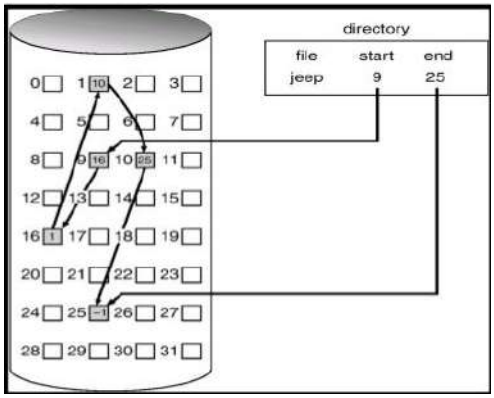
- It is simple to implement.
- We will get Excellent read performance.
- Supports Random Access into files.

Disadvantages

- The disk will become fragmented.
- It may be difficult to have a file grow.

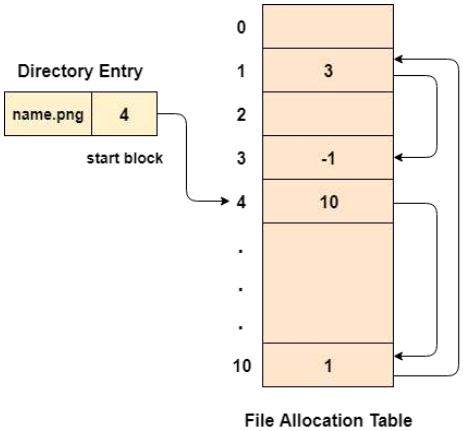
Linked List Allocation

- Linked List allocation solves all problems of contiguous allocation.
- In linked list allocation, each file is considered as the linked list of disk blocks.
- However, the disks blocks allocated to a particular file need not to be contiguous on the disk.
- Each disk block allocated to a file contains a pointer which points to the next disk block allocated to the same file.
- For example, a file of five blocks might start at block 9 and continue at block 16, then block 1, then block 10, and finally block 25 (See Figure). Each block contains a pointer to the next block. These pointers are not made available to the user. Thus, if each block is 512 bytes in size, and a disk address (the pointer) requires 4 bytes, then the user sees blocks of 508 bytes.



File Allocation Table

- The main disadvantage of linked list allocation is that the Random access to a particular block is not provided. In order to access a block, we need to access all its previous blocks.
- File Allocation Table overcomes this drawback of linked list allocation. In this scheme, a file allocation table is maintained, which gathers all the disk block links. The table has one entry for each disk block and is indexed by block number.
- File allocation table needs to be cached in order to reduce the number of head seeks. Now the head doesn't need to traverse all the disk blocks in order to access one successive block.



File Allocation Table

Advantages

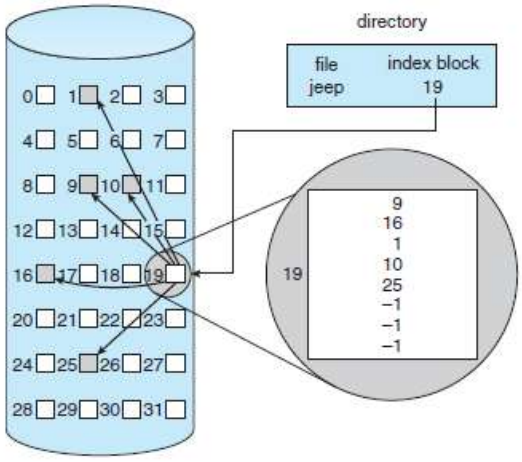
- There is no external fragmentation with linked allocation.
- Any free block can be utilized in order to satisfy the file block requests.
- File can continue to grow as long as the free blocks are available.
- Directory entry will only contain the starting block address.

Disadvantages

- Random Access is not provided.
- Pointers require some space in the disk blocks.
- Any of the pointers in the linked list must not be broken otherwise the file will get corrupted.
- Need to traverse each block.

Indexed Allocation

- Indexed allocation solves this problem by bringing all the pointers together into one location: **the index block.**
- Each file has its own index block, which is an array of disk-block addresses.
- The ith entry in the index block points to the ith block of the file. The directory contains the address of the index block.
- To find and read the *ith* block, we use the pointer in the *ith* index-block entry. This scheme is similar to the paging scheme.
- When the file is created, all pointers in the index block are set to null.
- When the *ith* block is first written, a block is obtained from the free-space manager, and its address is put in the *i*th index-block entry.
- Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk can satisfy a request for more space.



Advantages

1. Supports direct access
2. A bad data block causes the lost of only that block.

Disadvantages

1. A bad index block could cause the lost of entire file.

2. Size of a file depends upon the number of pointers, a index block can hold.
3. Having an index block for a small file is totally wastage.
4. More pointer overhead

4.Free Space Management

Since disk space is limited, we need to reuse the space from deleted files for new files, if possible. To keep track of free disk space, the system maintains a free-space list. The free-space list records all free disk blocks – those not allocated to some file or directory.

To create a file, we search the free-space list for the required amount of space, and allocate that space to the new file. This space is then removed from the free-space list. When a file is deleted, its disk space is added to the free-space list.

1. Bit Vector

The free-space list is implemented as a bit map or bit vector. Each block is represented by 1 bit.

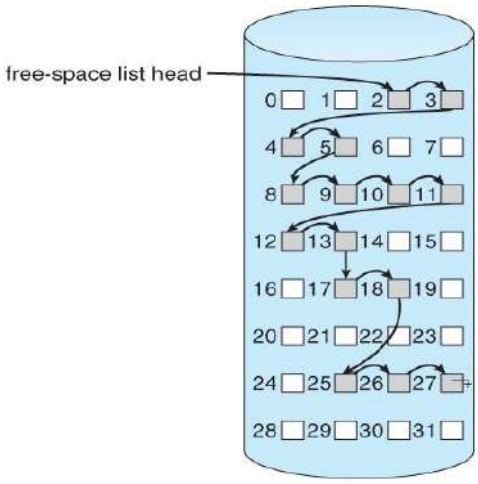
If the block is free, the bit is 1; if the block is allocated, the bit is 0. For example, Consider a disk where block 2,3,4,5,8,9,10,11,12,13,17,18,25,26 and 27 are free, and the rest of the block are allocated. The free space bit map would be 00111100111111000110000001110000 0 …

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

2. Linked List

Another approach to free-space management is to link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory. This first block contains a pointer to the next free disk block, and so on.

In our example, we would keep a pointer to block 2, as the first free block. Block 2 would contain a pointer to block 3, which would point to block 4, which would point to block 5, which would point to block 8, and so on. However, this scheme is not efficient; to traverse the list, we must read each block, which requires substantial I/O time. The FAT method incorporates free-block accounting data structure. No separate method is needed.

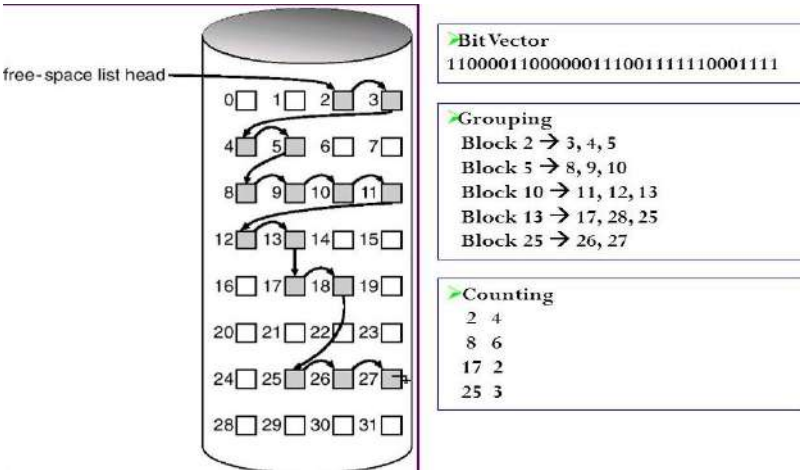


3. Grouping

A modification of the free-list approach is to store the addresses of n free blocks in the first free block. The first n-1 of these blocks are actually free. The last block contains the addresses of another n free blocks, and so on. The importance of this implementation is that the addresses of a large number of free blocks can be found quickly.

4. Counting

We can keep the address of the first free block and the number n of free contiguous blocks that follow the first block. Each entry in the free-space list then consists of a disk address and a count. Although each entry requires more space than would a simple disk address, the overall list will be shorter, as long as the count is generally greater than1.



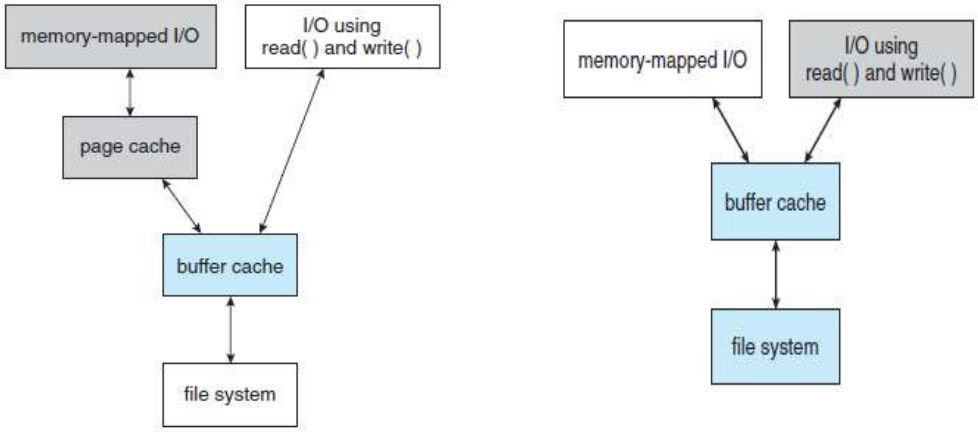
5. Efficiency and Performance

Efficiency

- The efficient use of disk space depends heavily on the disk-allocation and directory algorithms in use.
- Let's reconsider the clustering scheme, which improves file-seek and file-transfer performance at the cost of internal fragmentation. To reduce this fragmentation, BSD UNIX varies the cluster size as a file grows. Large clusters are used where they can be filled, and small clusters are used for small files and the last cluster of a file. This
- The types of data normally kept in a file's directory (or inode) entry also require consideration. Commonly, a "last write date" is recorded to supply information to the user and to determine whether the file needs to be backed up. Some systems also keep a "last access date," so that a user can determine when the file was last read.
- The result of keeping this information is that, whenever the file is read, a field in the directory structure must be written to. That means the block must be read into memory, a section changed, and the block written back out to disk, because operations on disks occur only in block (or cluster) chunks. So any time a file is opened for reading, its directory entry must be read and written as well.
- Generally, every data item associated with a file needs to be considered for its effect on efficiency and performance.

Performance

- Some systems maintain a separate section of main memory for a **buffer cache**, where blocks are kept under the assumption that they will be used again shortly. Other systems cache file data using a **page cache**.
- The **page cache** uses virtual memory techniques to cache file data as pages rather than as file-system-oriented blocks.
- Caching file data using virtual addresses is far more efficient than caching through physical disk blocks, as accesses interface with virtual memory rather than the file system.
- Several systems—including Solaris, Linux, and Windows —use page caching to cache both process pages and file data. This is known as **unified virtual memory**.



- The two alternatives for opening and accessing a file. One approach is to use memory mapping the second is to use the standard system calls **read()** and **write()**.
- Here, the read() and write() system calls go through the buffer cache.
- The memory-mapping call, however, requires using two caches—the **page cache** and the **buffer cache.**
- A memory mapping proceeds by reading in disk blocks from the file system and storing them in the buffer cache. Because the virtual memory system does not interface with the buffer cache, the contents of the file in the buffer cache must be copied into the page cache. This situation, known as **double caching**, requires caching file-system data twice.

6. 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 (fsck in UNIX, chkdsk or scandisk in Windows) 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.
  - Link counts for an inode not matching the number of references to that inode in the directory structure.
  - Two or more identical file names in the same directory.
  - Illegally linked directories, e.g. cyclical relationships where those are not allowed, or files/directories that are not accessible from the root of the directory tree.
  - Consistency checkers will often collect questionable disk blocks into new files with names such as chk00001.dat. These files may contain valuable information that would otherwise be lost, but in most cases they can be safely deleted, (returning those disk blocks to the free list.)

UNIX caches directory information for reads, but any changes that affect space allocation or metadata changes are written synchronously, before any of the corresponding data blocks are written to.

Log-Structured File Systems

- Log-based transaction-oriented (a.k.a. journaling) 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.

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.
- A typical backup schedule may then be as follows:

Day 1. Copy to a backup medium all files from the disk. This is called a full backup.

Day 2. Copy to another medium all files changed since day 1. This is an incremental backup.

Day 3. Copy to another medium all files changed since day 2.

Day N. Copy to another medium all files changed since day N−1. Then go back to day 1.

I/O SYSTEMS

1. I/O Hardware

The role of the operating system in computer I/O is to manage and control I/O operations and I/O devices. A device communicates with a computer system by sending signals over a cable or even through the air.

Port: The device communicates with the machine via a connection point (or port), for example, a serial port.

Bus: If one or more devices use a common set of wires, the connection is called a bus.

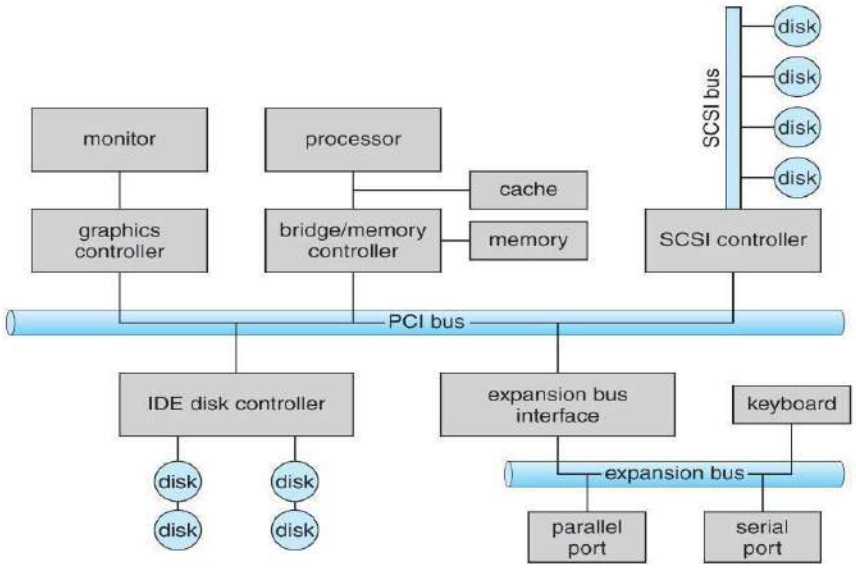
Daisy chain: Device _A 'has a cable that plugs into device _B ', and device _B 'has a cable that plugs into device _C ', and device _C 'plugs into a port on the computer, this arrangement is called a daisy chain. A daisy chain usually operates as a bus.

PC bus structure

A PCI bus that connects the processor-memory subsystem to the fast devices, and an expansion bus that connects relatively slow devices such as the keyboard and serial and parallel ports. In the upper- right portion of the figure, four disks are connected together on a SCSI bus plugged into a SCSI controller.

A **controller or host adapter** is a collection of electronics that can operate a port, a bus, or a device. A serial-port controller is a simple device controller. It is a single chip in the computer that controls the signals on the wires of a serial port. By contrast, a SCSI bus controller is not simple.

Because the SCSI protocol is complex, the SCSI bus controller is often implemented as a separate circuit board. It typically contains a processor, microcode, and some private memory. Some devices have their own built- in controllers.



How can the processor give commands and data to a controller to accomplish an I/O transfer?

- Direct I/O instructions
- Memory-mapped I/O

Direct I/O instructions

Use special I/O instructions that specify the transfer of a byte or word to an I/O port address. The I/O instruction triggers bus lines to select the proper device and to move bits into or out of a device register

Memory-mapped I/O

The device-control registers are mapped into the address space of the processor. The CPU executes I/O requests using the standard data-transfer instructions to read and write the device- control registers.

Status register	Read by the host to indicate states such as whether the current command has completed, whether a byte is available to be read from the data-in register, and whether there has been a device error.
Control register	Written by the host to start a command or to change the mode of a device.
data-in register	Read by the host to get input
data-out register	Written by the host to send output

- An I/O port typically consists of four registers: status, control, data-in, and data-out registers.

1. Polling

Interaction between the host and a controller

- The controller sets the busy bit when it is busy working, and clears the busy bit when it is ready to accept the next command.
- The host sets the command ready bit when a command is available for the controller to execute.

Coordination between the host & the controller is done by handshaking as follows:

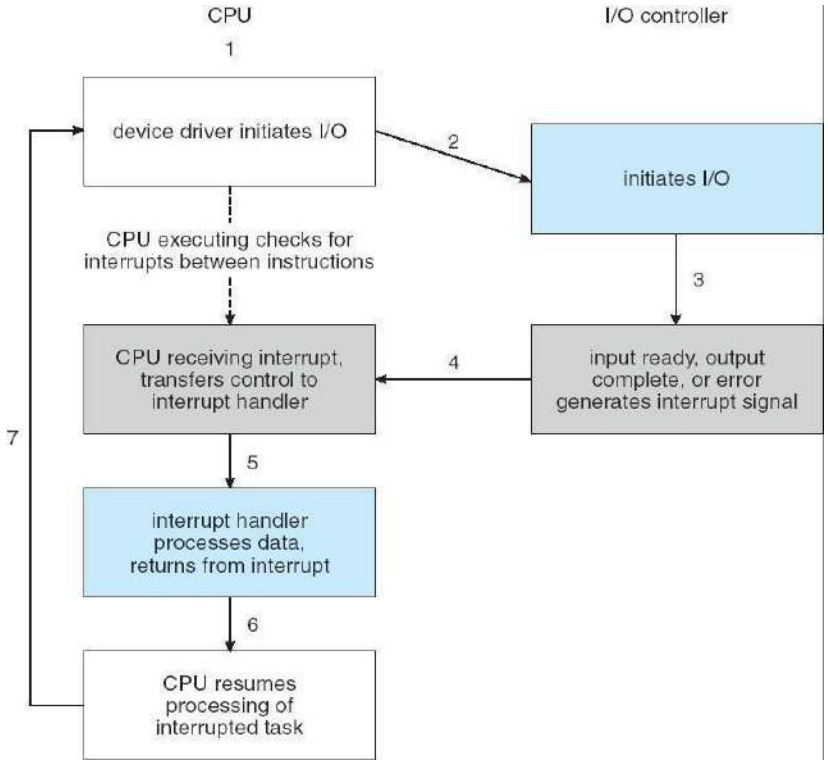
1. The host repeatedly reads the busy bit until that bit becomes clear.
2. The host sets the write bit in the command register and writes a byte into the data-out register.
3. The host sets the command-ready bit.
4. When the controller notices that the command-ready bit is set, it sets the busy bit.
5. The controller reads the command register and sees the write command. It reads the data-out register to get the byte, and does the I/O to the device.
6. The controller clears the command-ready bit, clears the error bit in the status register to indicate that the device I/O succeeded, and clears the busy bit to indicate that it is finished.
7. In step 1, the host is ―**busy-waiting or polling**‖: It is in a loop, reading the status register over and over until the busy bit becomes clear.

2. Interrupts

The CPU hardware has a wire called the ―interrupt-request line‖.
The basic interrupt mechanism works as follows;

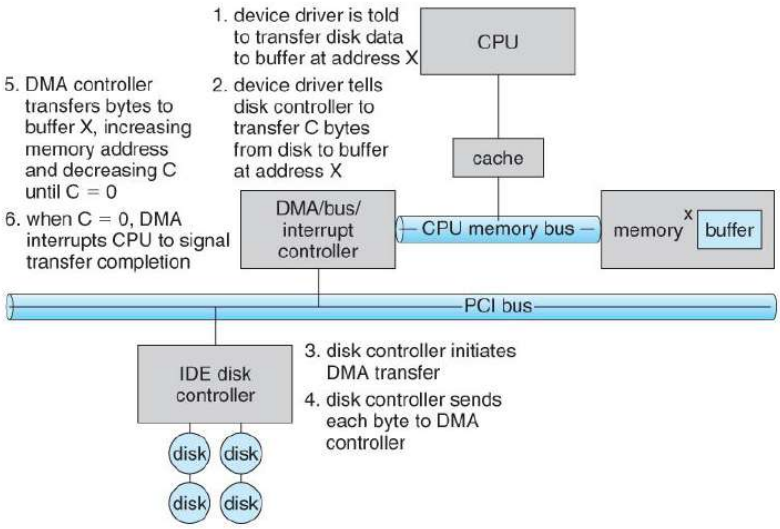
1. Device controller raises an interrupt by asserting a signal on the interrupt request line.
2. The CPU catches the interrupt and dispatches to the interrupt handler and
3. The handler clears the interrupt by servicing the device.

- **Nonmaskable interrupt**: which is reserved for events such as unrecoverable memory errors?
- **Maskable interrupt**: Used by device controllers to request service



3. Direct Memory Access (DMA)

In general it is tough for the CPU to do the large transfers between the memory buffer & disk; because it is already equipped with some other tasks ,then this will create overhead. So a special-purpose processor called a direct memory-access **(DMA)** controller is used.



2. Application I/O Interface

I/O system calls encapsulate device behaviours in generic classes. Device-driver layer hides differences among I/O controllers from kernel

Devices vary on many dimensions, as illustrated in

• **Character-stream or block**. A character-stream device transfers bytes one by one, whereas a block device transfers a block of bytes as a unit.

• **Sequential or random access**. A sequential device transfers data in a fixed order determined by the device, whereas the user of a random-access device can instruct the device to seek to any of the available data storage locations.

• **Synchronous or asynchronous**. A synchronous device performs data transfers with predictable response times, in coordination with other aspects of the system. An asynchronous device exhibits irregular or unpredictable response times not coordinated with other computer events.

• **Sharable or dedicated**. A sharable device can be used concurrently by several processes or threads; a dedicated device cannot.

Speed of operation. Device speeds range from a few bytes per second to a few gigabytes per second.

• **Read–write, read only, or write only**. Some devices perform both input and output, but others support only one data transfer direction.

1. Block and Character Devices

Block-device: The block-device interface captures all the aspects necessary for accessing disk drives and other block-oriented devices. The device should understand the commands such as read () & write (), and if it is a random access device, it has a seek() command to specify which block to transfer next.

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

2. Network Devices

Because the performance and addressing characteristics of network I/O differ

significantly from those of disk I/O, most operating systems provide a network I/O interface that is different from the read0
-write() -seek() interface used for disks.

- Windows NT provides one interface to the network interface card, and a second interface to the network protocols.

- In UNIX, we find half-duplex pipes, full-duplex FIFOs, full-duplex STREAMS, message queues and sockets.

3. Clocks and Timers

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

- Give the current time
- Give the elapsed time
- Set a timer to trigger operation X at time T

Programmable interval timer: The hardware to measure elapsed time and to trigger operations is called a programmable interval timer. It can be set to wait a certain amount of time and then to generate an interrupt. To generate periodic interrupts, it can be set to do this operation once or to repeat.

Uses of Programmable interval timer:

Scheduler	To generate an interrupt that will pre-empt a process at the end of its time slice.
Disk I/O subsystem	To invoke the flushing of dirty cache buffers to disk periodically
Network subsystem	To cancel operations those are proceeding too slowly because of network congestion or failures.

When the timer interrupts, the kernel signals the requester, and reloads the timer with the next earliest time.

Counter: The hardware clock is constructed from a high frequency counter.

In some computers, the value of this counter can be read from a device register, in which the counter can be considered to be a high-resolution clock.

4. Blocking and Non-blocking I/O (or) synchronous & asynchronous: Blocking I/O: When an application issues a blocking system call;

- The execution of the application is suspended.
- The application is moved from the operating system's run queue to a wait queue.
- After the system call completes, the application is moved back to the run queue, where it is eligible to resume execution, at which time it will receive the values returned by the system call.

Non-blocking, I/O: Some user-level processes need non-blocking

I/O***Examples:***

User interface that receives keyboard and mouse input while processing and displaying data on the screen.

Video application that reads frames from a file on disk while simultaneously decompressing and displaying the output on the display.

3.Kernel I/O Subsystem

Kernels provide many services related to I/O.

- One way that the I/O subsystem improves the efficiency of the computer is by scheduling I/O operations.
- Another way is by using storage space in main memory or on disk, via techniques called buffering, caching, and spooling.

1. I/O Scheduling:

To determine a good order in which to execute the set of I/O requests. Uses:

- It can improve overall system performance,
- It can share device access fairly among processes, and
- It can reduce the average waiting time for 1/0 to complete.

Implementation: OS developers implement scheduling by maintaining a ―queue of requests for each device.

- When an application issues a blocking I/O system call,
- The request is placed on the queue for that device.
- The I/O scheduler rearranges the order of the queue to improve the overall system efficiency and the average response time experienced by applications.

2. Buffering:

Buffer: A memory area that stores data while they are transferred between two devices or between a device and an application.

Reasons for buffering:

- To cope with a speed mismatch between the producer and consumer of a data stream.
- To adapt between devices that have different data-transfer sizes.
- To support copy semantics for application I/O.

Copy semantics Suppose that an application has a buffer of data that it wishes to write to disk. It calls the write () system call, providing a pointer to the buffer and an integer specifying the number of bytes to write.

3. Caching

A cache is a region of fast memory that holds copies of data. Access to the cached copy is more efficient than access to the original

Cache vs buffer: A buffer may hold the only existing copy of a data item, whereas a cache just holds a copy on faster storage of an item that resides elsewhere.

When the kernel receives a file I/O request,

1. The kernel first accesses the buffer cache to see whether that region of the file is already available in main memory.

2. If so, a physical disk I/O can be avoided or deferred. Also, disk writes are accumulated in the buffer cache for several seconds, so that large transfers are gathered to allow efficient write schedules.

4. Spooling and Device Reservation:

Spool: A buffer that holds output for a device, such as a printer, that cannot accept interleaved data streams.

A printer can serve only one job at a time, several applications may wish to print their output concurrently, without having their output mixed together

The OS provides a control interface that enables users and system administrators ;

- To display the queue,
- To remove unwanted jobs before those jobs print,
- To suspend printing while the printer is serviced, and so on.

Device reservation - provides exclusive access to a device

- System calls for allocation and de-allocation
- Watch out for deadlock

5. Error Handling

An operating system that uses protected memory can guard against many kinds of hardware and application errors. OS can recover from disk read, device unavailable, transient write failures Most return an error number or code when I/O request fails System error logs hold problem reports

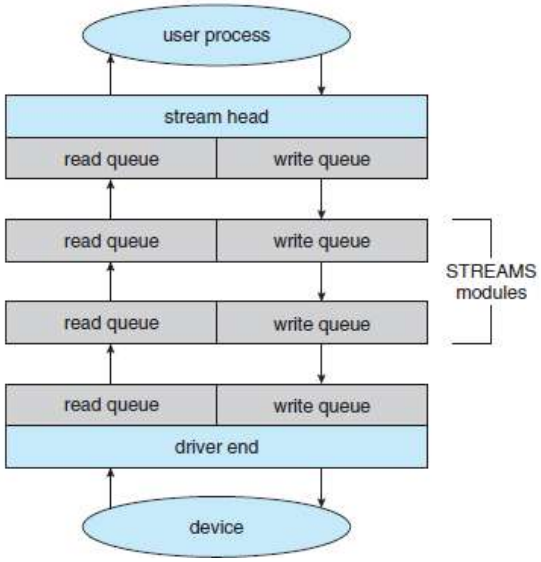
STREAMS

Stream is a full-duplex communication channel between a user-level process and a device in Unix System V and beyond A STREAM consists of:

- STREAM head interfaces with the user process
- Driver end interfaces with the device
- Zero or more STREAM modules between them.

Each module contains a read queue and a write queue. Message passing is used to communicate between queues. Modules provide the functionality of STREAMS processing and they are pushed onto a stream using the ioct () system call.

Flow control: Because messages are exchanged between queues in adjacent modules, a queue in one module may overflow an adjacent queue. To prevent this from occurring, a queue may support flow control.



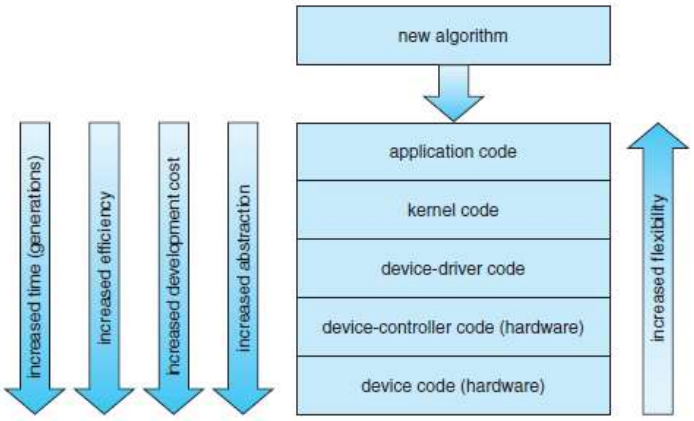
PERFORMANCE

I/O a major factor in system performance:

- Heavy demands on CPU to execute device driver, kernel I/O code. So context switches occur due to interrupts.
- Interrupt handling is a relatively expensive task: Each interrupt causes the system to perform a state change, to execute the interrupt handler & then to restore state
- Network traffic especially stressful.
- Systems use separate ―front-end processors" for terminal I/O, to reduce the interrupt burden on the main CPU.

We can employ several principles to improve the efficiency of I/O:

- Reduce the number of context switches.
- Reduce the number of times that data must be copied in memory while passing between device and application.
- Reduce the frequency of interrupts by using large transfers, smart controllers & polling.
- Increase concurrency by using DMA-knowledgeable controllers or channels to offload simple data copying from the CPU.
- Move processing primitives into hardware, to allow their operation in device controllers concurrent with the CPU and bus operation.
- Balance CPU, memory subsystem, bus, and I/O performance, because an overload in any one area will cause idleness in others.



Device functionality progression.

a) **<u>An application-level implementation</u>**: Implement experimental I/O algorithms at the application level, because application code is flexible, and application bugs are unlikely to cause system crashes.

It can be inefficient;

- Because of the overhead of context switches and
- Because the application cannot take advantage of internal kernel data structures and kernel functionality

b) **<u>In-kernel implementation</u>**: Re-implement application-level algorithm in the kernel. This can improve the performance, but the development effort is more challenging, because an operating-system kernel is a large, complex software system. Moreover, an in-kernel implementation must be thoroughly debugged to avoid data corruption and system crashes.

c) **<u>A hardware implementation</u>**: The highest performance may be obtained by a specialized implementation in hardware, either in the device or in the controller.

- ❖ Difficult and expense of making further improvements or of fixing bugs, (-) Increased development time
- ❖ Decreased flexibility.

UNIT V CASE STUDY

Linux System - Design Principles, Kernel Modules, Process Management, Scheduling, Memory Management, Input-Output Management, File System, Inter-process Communication; Mobile OS - iOS and Android - Architecture and SDK Framework, Media Layer, Services Layer, Core OS Layer, File System.

1. LINUX SYSTEM

1.1 Linux History

- Its development began in 1991, when a Finnish university student, Linus Torvalds, began developing a small but self-contained kernel for the 80386 processor, the first true 32-bit processor in Intel's range of PC-compatible CPUs.
- Early in its development, the Linux source code was made available free— both at no cost and with minimal distributional restrictions—on the Internet.
- The **Linux kernel** is an original piece of software developed from scratch by the Linux community.
- The **Linux system**, includes a multitude of components, some written from scratch, others borrowed from other development projects, and still others created in collaboration with other teams.
- A **Linux distribution** includes all the standard components of the Linux system, plus a set of administrative tools to simplify the initial installation and subsequent upgrading of Linux and to manage installation and removal of other packages on the system.

1.2The Linux Kernel

- The first Linux kernel released to the public was version 0.01, dated May 14, 1991. It had no networking, ran only on 80386-compatible Intel processors and PC hardware, and had extremely limited device-driver support.
- The next milestone, **Linux 1.0**, was released on March 14, 1994.
- This release culminated three years of rapid development of the Linux kernel. Perhaps the single biggest new feature was networking: 1.0 included support for UNIX's standard TCP/IP networking protocols such as socket interface for networking programming.
- In **March 1995**, the **1.2 kernel** was released. This release did not offer nearly the same improvement in functionality as the 1.0 release, but it did support a much wider variety of hardware, including the new PCI hardware bus architecture.
- In **June 1996** as Linux **version 2.0**was released. This release was given a major version-number increment because of two major new capabilities: support for multiple architectures, including a 64-bit native Alpha port, and symmetric multiprocessing (SMP) support
- Improvements continued with the release of **Linux 2.2** in **1999**. A port to UltraSPARC systems was added. Networking was enhanced with more

flexible firewalling, improved routing and traffic management, and support for TCP large window and selective acknowledgement.

- Linux kernel version 3.0 was released in July 2011.

2. DESIGN PRINCIPLES

- Linux runs on a wide variety of platforms, it was originally developed exclusively on PC architecture.
- Linux can run happily on a multiprocessor machine with many gigabytes of main memory and many terabytes of disk space, but it is still capable of operating usefully in under 16 MB of RAM.

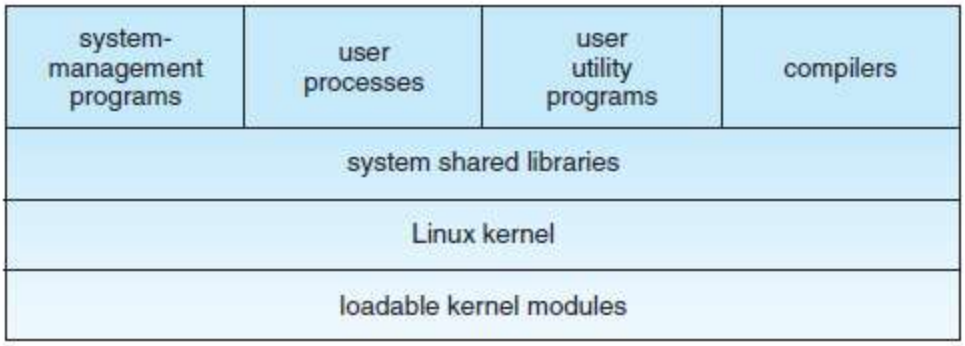
➔Components of a Linux System

The Linux system is composed of three main bodies of code

1. **Kernel**. The kernel is responsible for maintaining all the important abstractions of the operating system, including such things as virtual memory and processes.

2. **System libraries**. The system libraries define a standard set of functions through which applications can interact with the kernel. These functions implement much of the operating-system functionality that does not need the full privileges of kernel code. The most important system library is the C library, known as libc. In addition to providing the standard C library, libc implements the user mode side of the Linux system call interface, as well as other critical system-level interfaces.

3. **System utilities**. The system utilities are programs that perform individual, specialized management tasks. Some system utilities are invoked just once to initialize and configure some aspect of the system. Others —known as daemons in UNIX terminology—run permanently, handling such tasks as responding to incoming network connections, accepting logon requests from terminals, and updating log files.



- All the kernel code executes in the processor's privileged mode with full access to all the physical resources of the computer.
- Linux refers to this **privileged mode** as kernel mode.
- Under Linux, no user code is built into the kernel.
- Any operating-system-support code that does not need to run in **kernel mode** is placed into the system libraries and runs in user **mode**.
- Unlike **kernel mode**, **user mode** has access only to a controlled subset of the system's resources.

3. KERNEL MODULES

- The Linux kernel has the ability to load and unload arbitrary sections of kernel code on demand.
- These loadable kernel modules run in privileged kernel mode and as a consequence have full access to all the hardware capabilities of the machine on which they run.
- Kernel modules are convenient for several reasons.
  1. **Linux's source code is free**, so anybody wanting to write kernel code is able to compile a modified kernel and to reboot into that new functionality.
  2. However, **recompiling, relinking, and reloading** the entire kernel is a cumbersome cycle to undertake when you are developing a new driver.
  3. If you use kernel modules, you **do not have to make a new kernel** to test a new driver—the driver can be compiled on its own and loaded into the already running kernel.
- Kernel modules allow a Linux system to be set up with a standard minimal kernel, without any extra device drivers built in.
- Any device drivers that the user needs can be either loaded explicitly by the system at startup or loaded automatically by the system on demand and unloaded when not in use.
- For example, a mouse driver can be loaded when a USB mouse is plugged into the system and unloaded when the mouse is unplugged.

1. **The module-management** system allows modules to be loaded into memory and to communicate with the rest of the kernel.
2. **The module loader and unloader**, which are user-mode utilities, work with the module-management system to load a module into memory.
3. **The driver-registration** system allows modules to tell the rest of the kernel that a new driver has become available.
4. **A conflict-resolution mechanism** allows different device drivers to reserve hardware resources and to protect those resources from accidental use by another driver.

Module Management

- Loading a module requires more than just loading its binary contents into kernel memory.
- Linux maintains an internal symbol table in the kernel.
- The loading of the module is performed in **two stages**.

  ➔First, **the module loader utility** asks the kernel to reserve a continuous area of virtual kernel memory for the module. The kernel returns the address of the memory allocated, and the loader utility can use this address to relocate the module's machine code to the correct loading address.

  ➔A **second system call then passes** the module, plus any symbol table that the new module wants to export, to the kernel.

Driver Registration

- provides a set of routines to allow drivers to be added to or removed.

- A module may register many types of functionality
- For example, a device driver might want to register two separate mechanisms for accessing the device. Registration tables include, among others, the following items:

  Device drivers. These drivers include character devices (such as printers, terminals, and mice), block devices (including all disk drives), and network interface devices.

  File systems. The file system may be anything that implements Linux's virtual file system calling routines. It might implement a format for storing files on a disk, but it might equally well be a network file system, such as NFS, or a virtual file system whose contents are generated on demand, such as Linux's /proc file system.

  Network protocols. A module may implement an entire networking protocol, such as TCP or simply a new set of packet-filtering rules for a network firewall.

  Binary format. This format specifies a way of recognizing, loading, and executing a new type of executable file.

Conflict Resolution

Linux provides a central conflict-resolution mechanism to help arbitrate access to certain hardware resources. Its aims are as follows:

➔ To prevent modules from clashing over access to hardware resources

➔ To prevent autoprobes—device-driver probes that auto-detect device configuration—from interfering with existing device drivers

➔ To resolve conflicts among multiple drivers trying to access the same hardware—as, for example, when both the parallel printer driver and the parallel line IP (PLIP) network driver try to talk to the parallel port

4. PROCESS MANAGEMENT

A process is the basic context in which all user-requested activity is serviced within the operating system.

➔The fork() and exec() Process Model

- The basic principle of UNIX process management is to separate into two steps two operations that are usually combined into one: The **creation of a new process** and the **running of a new program**.
- A new process is created by the fork() system call, and a new program is run after a call to exec().
- These are two distinctly separate functions.
- We can create a new process with fork() without running a new program—the new subprocess simply continues to execute exactly the same program, at exactly the same point, that the first (parent) process was running.

1.Process Identity

➔ A process identity consists mainly of the following items:

Process ID (PID). Each process has a unique identifier.

Credentials. Each process must have an associated user ID and one or more group IDs that determine the rights of a process to access system resources and files.

Personality: Personalities are primarily used by emulation libraries to request the system calls be compatible with certain varieties of UNIX.

Namespace: Each process is associated with a specific view of the file system hierarchy, called its namespace. Most processes share a common namespace and thus operate on a shared file-system hierarchy.

➔ Process Environment

A process's environment is inherited from its parent and is composed of **two null-terminated vectors**: **the argument vector** and the **environment vector**.

The argument vector simply lists the command-line arguments used to invoke the running program; it conventionally starts with the name of the program itself.

The environment vector is a list of "NAME=VALUE" pairs that associates named environment variables with arbitrary textual values. The environment is not held in kernel memory but is stored in the process's own user-mode address space as the first datum at the top of the process's stack.

➔ Process Context

- Process context is the state of the running program at any one time; it changes constantly. Process context includes the following parts:

  Scheduling context: The most important part of the process context is its scheduling context—the information that the scheduler needs to suspend and restart the process. This information includes saved copies of all the process's registers.

  Accounting: The kernel maintains accounting information about the resources currently being consumed by each process and the total resources consumed by the process in its entire lifetime so far.

  File table. The file table is an array of pointers to kernel file structures representing open files.

  File-system context : Whereas the file table lists the existing open files, the file-system context applies to requests to open new files. The file-system context includes the process's root directory, current working directory, and namespace.

  Signal-handler table: The signal-handler table defines the action to take in response to a specific signal.

  Virtual memory context : The virtual memory context describes the full contents of a process's private address space.

2. Processes and Threads

- Linux provides the fork() system call, which duplicates a process without loading a new executable image. Linux also provides the ability to create threads via the clone() system call.
- The clone() system call behaves identically to fork(), except that it accepts as arguments a set of flags that dictate what resources are shared between the parent and child.
- The flags include

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

5. SCHEDULING

- Scheduling is the job of allocating CPU time to different tasks within an operating system.
- Linux, like all UNIX systems, supports preemptive multitasking.
- In such a system, the process scheduler decides which process runs and when.

➔Process Scheduling

Linux has two separate process-scheduling algorithms.

1. One is a time-sharing algorithm for fair, preemptive scheduling among multiple processes.
2. The other is designed for real-time tasks, where absolute priorities are more important than fairness.

Completely Fair Scheduler (CFS).

- In CFS each core of the CPU has its own run queue.
- Each task has a so called nice value and weight assigned to it. The nice value represents how "kind" the specific task is to other tasks.
- In other words, a task with a high nice value has a lower priority and is thus less likely to take more of the CPUs bandwidth than a task with a low nice value.
- CFS introduced a new scheduling algorithm called **fair scheduling** that eliminates **time slices** in the traditional sense. **Instead of time slices**, all processes are **allotted a proportion of the processor's time**. CFS calculates how long a process should run as a function of the total number of runnable processes.
- To calculate the actual length of time a process runs, CFS relies on a configurable variable called **target latency**, which is the interval of time during which every runnable task should run at least once.

➔Real-Time Scheduling

- Linux implements the two real-time scheduling classes: first-come, first served (FCFS) and round-robin.
- In both cases, each process has a priority in addition to its scheduling class.
- The scheduler always runs the process with the highest priority. Among processes of equal priority, it runs the process that has been waiting longest.
- The only difference between FCFS and round-robin scheduling is that FCFS processes continue to run until they either exit or block, whereas a round-robin process will be preempted after a while and will be moved to the end of the scheduling queue, so round-robin processes of equal priority will automatically time-share among themselves.
- Linux's real-time scheduling is **soft— and hard—real time**.

  ➔ A hard real time system guarantees that critical tasks complete on time, whereas in soft real time system, a critical real time task gets priority over other tasks and retains that priority until it completes.

➔ Kernel Synchronization

- The way the kernel schedules its own operations is fundamentally different from the way it schedules processes.
- A request for kernel-mode execution can occur in two ways.
- A running program may request an operating-system service, either explicitly via a system call or implicitly—for example, when a page fault occurs.
- Alternatively, a device controller may deliver a hardware interrupt that causes the CPU to start executing a kernel-defined handler for that interrupt.
- The problem for the kernel is that all these tasks may try to access the same internal data structures.
- If one kernel task is in the middle of accessing some data structure when an interrupt service routine executes, then that service routine cannot access or modify the same data without risking data corruption.
- The Linux kernel provides spinlocks and semaphores (as well as reader–writer versions of these two locks) for locking in the kernel.
- Linux uses an interesting approach to disable and enable kernel preemption. It provides two simple kernel interfaces—**preempt disable()** and **preempt enable().**

single processor	multiple processors
Disable kernel preemption.	Acquire spin lock.
Enable kernel preemption.	Release spin lock.

- The counter is incremented when a lock is acquired and decremented when a lock is released.
- Linux implements this architecture by separating interrupt service routines into two sections: **the top half** and the **bottom half**.

  The **top half** is the standard interrupt service routine that runs with recursive interrupts disabled.

Interrupts of the same number (or line) are disabled, but other interrupts may run.
The **bottom half** of a service routine is run, with all interrupts enabled, by a miniature scheduler that ensures that bottom halves never interrupt themselves.

Interrupt protection levels.

➔ Symmetric Multiprocessing

Linux kernel to support symmetric multiprocessor (SMP) hardware, allowing separate processes to execute in parallel on separate processors. The original implementation of SMP imposed the restriction that only one processor at a time could be executing kernel code.

6.MEMORY MANAGEMENT

Memory management under Linux has two components. The first deals with allocating and freeing physical memory—pages, groups of pages, and small blocks of RAM. The second handles virtual memory, which is memory-mapped into the address space of running processes.

➔Management of Physical Memory

- Due to specific hardware constraints, Linux separates physical memory into four different zones, or regions:

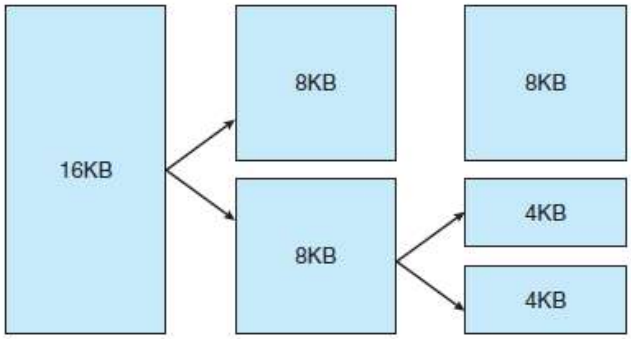
  • ZONE DMA
  • ZONE DMA32
  • ZONE NORMAL
  • ZONE HIGHMEM

- ZONE_DMA. This zone contains pages that can undergo DMA.
- ZONE_DMA32. Like ZOME_DMA, this zone contains pages that can undergo DMA. Unlike ZONE_DMA, these pages are accessible only by 32-bit devices. On some architectures, this zone is a larger subset of memory.
- ZONE_NORMAL. This zone contains normal, regularly mapped, pages.

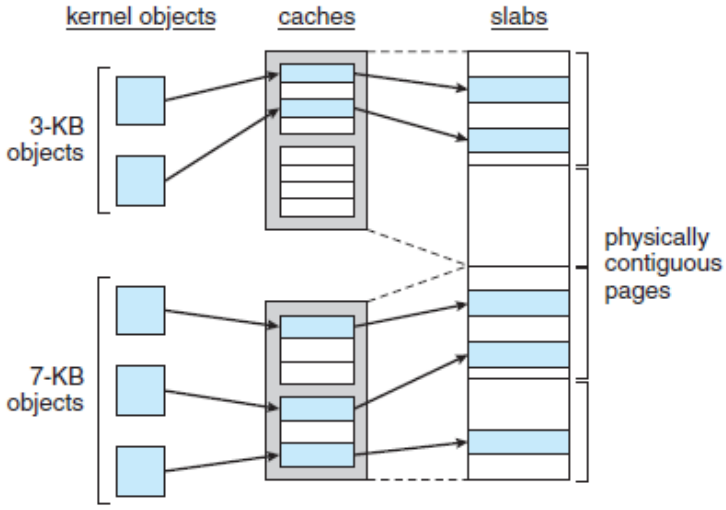
- ZONE_HIGHMEM. This zone contains "high memory", which are pages not permanently mapped into the kernel's address space. The relationship of zones and physical addresses on the Intel x86-32 architecture is shown Below

zone	physical memory
ZONE_DMA	< 16 MB
ZONE_NORMAL	16 .. 896 MB
ZONE_HIGHMEM	> 896 MB

- The primary physical-memory manager in the Linux kernel is the **page allocator**.
- Each zone has its own allocator, which is responsible for allocating and freeing all physical pages for the zone and is capable of allocating ranges of physically contiguous pages on request.
- The allocator uses a **buddy system** to keep track of available physical pages.
- Each allocatable memory region has an adjacent partner (or buddy). Whenever two allocated partner regions are freed up, they are combined to form a larger region—a buddy heap.



- Another strategy adopted by Linux for allocating kernel memory is known as slab allocation. A slab is used for allocating memory for kernel data structures and is made up of one or more physically contiguous pages. A cache consists of one or more slabs.



In Linux, a slab may be in one of three possible states:

1. Full. All objects in the slab are marked as used.
2. Empty. All objects in the slab are marked as free.
3. Partial. The slab consists of both used and free objects.

➔Virtual Memory

- The Linux virtual memory system is responsible for maintaining the address space accessible to each process.
- It creates pages of virtual memory on demand and manages loading those pages from disk and swapping them back out to disk as required.
- Under Linux, the virtual memory manager maintains two separate views of a process's address space: as a set of **separate regions** and as a **set of pages**.

1. Virtual Memory Regions

- Linux implements several types of virtual memory regions.
- One property that characterizes virtual memory is the backing store for the region, which describes where the pages for the region come from.
- Most memory regions are backed either by a file or by nothing.
- A region backed by nothing is the simplest type of virtual memory region.
- Such a region represents demand-zero memory: when a process tries to read a page in such a region, it is simply given back a page of memory filled with zeros.
- A virtual memory region is also defined by its reaction to writes. The mapping of a region into the process's address space can be either private or shared.

2. Lifetime of a Virtual Address Space

- The kernel creates a new virtual address space in two situations:
- when a process runs a new program with the exec() system call and when a new process is created by the fork() system call.

3. Swapping and Paging

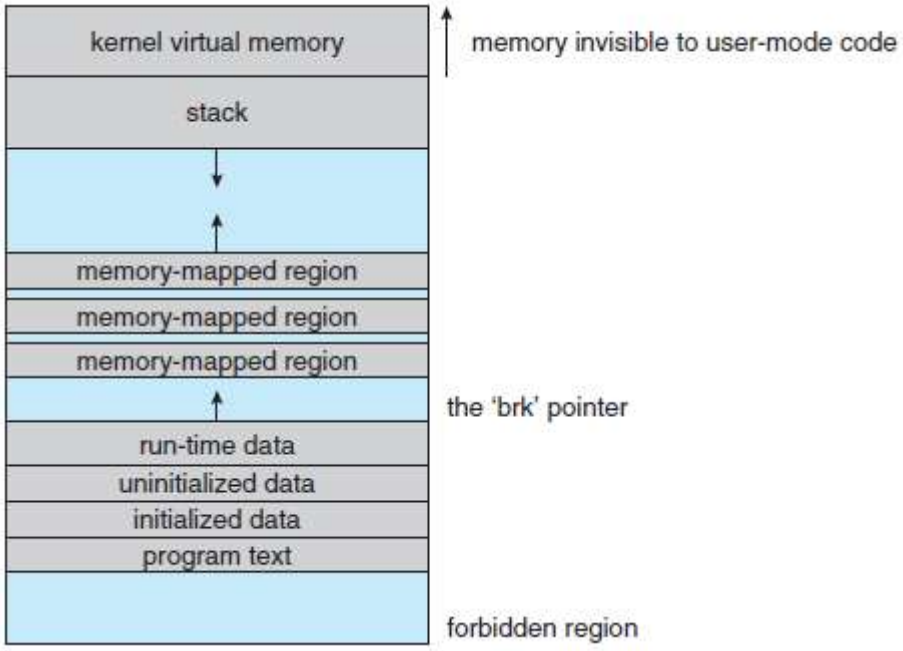
- An important task for a virtual memory system is to relocate pages of memory from physical memory out to disk when that memory is needed.
- The paging system can be divided into two sections.
- First, the policy algorithm decides which pages to write out to disk and when to write them.
- Second, the paging mechanism carries out the transfer and pages data back into physical memory when they are needed again.

4. Kernel Virtual Memory

- kernel virtual memory area contains two regions.
- The first is a static area that contains page-table references to every available physical page of memory in the system, so that a simple translation from physical to virtual addresses occurs when kernel code is run.
- The remainder of the kernel's reserved section of address space is not reserved for any specific purpose.

➔Execution and Loading of User Programs

- The Linux kernel's execution of user programs is triggered by a call to the exec() system call.
- This exec() call commands the kernel to run a new program within the current process, completely overwriting the current execution context with the initial context of the new program.
- The first job of this system service is to verify that the calling process has permission rights to the file being executed.
- Newer Linux systems use the more modern ELF format, now supported by most current UNIX implementations.



Memory layout for ELF programs.

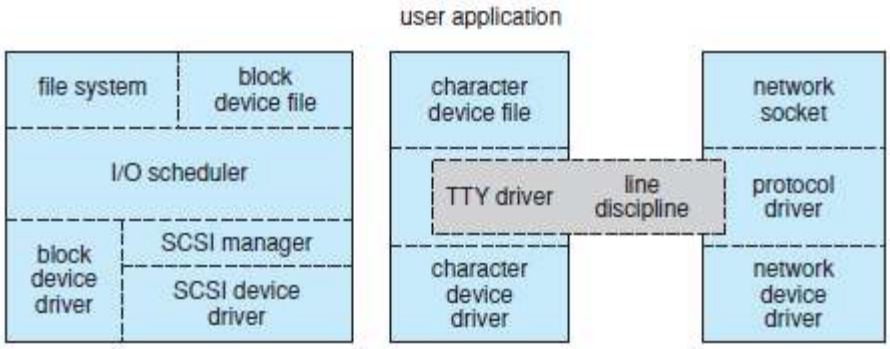
7. INPUT AND OUTPUT MANAGEMENT

- Linux splits all devices into three classes: block devices, character devices, and network devices.

➔Block Devices

- Block devices provide the main interface to all disk devices in a system.
- Performance is particularly important for disks, and the block-device system must provide functionality to ensure that disk access is as fast as possible.
- This functionality is achieved through the scheduling of I/O operations.

- In the context of block devices, a block represents the unit with which the kernel performs I/O.
- When a block is read into memory, it is stored in a buffer.
- The request manager is the layer of software that manages the reading and writing of buffer contents to and from a block-device driver.
- A separate list of requests is kept for each block-device driver.
- These requests have been scheduled according to a unidirectional-elevator (C-SCAN) algorithm that exploits the order in which requests are inserted in and removed from the lists.
- When a request is accepted for processing by a block-device driver, it is not removed from the list.
- It is removed only after the I/O is complete, at which point the driver continues with the next request in the list, even if new requests have been inserted in the list before the active request.



Device-driver block structure.

➔**Character Devices**

- A character-device driver can be almost any device driver that does not offer random access to fixed blocks of data.
- Any character-device drivers registered to the Linux kernel must also register a set of functions that implement the file I/O operations that the driver can handle.
- The kernel performs almost no preprocessing of a file read or write request to a character device. It simply passes the request to the device in question and lets the device deal with the request.
- A line discipline is an interpreter for the information from the terminal device.
- The most common line discipline is the **tty discipline**, which glues the terminal's data stream onto the standard input and output streams of a user's running processes, allowing those processes to communicate directly with the user's terminal

➔**Network devices**

- Network devices are dealt with differently from block and character devices.
- Users cannot directly transfer data to network devices. Instead, they must communicate indirectly by opening a connection to the kernel's networking subsystem.

8. INTERPROCESS COMMUNICATION

Linux provides a rich environment for processes to communicate with each other.

➔Synchronization and Signals

- The standard Linux mechanism for informing a process that an event has occurred is the signal.
- Signals can be sent from any process to any other process, with restrictions on signals sent to processes owned by another user.
- The kernel also generates signals internally. For example, it can send a signal to a server process when data arrive on a network channel, to a parent process when a child terminates, or to a waiting process when a timer expires.
- Internally, the Linux kernel does not use signals to communicate with processes running in kernel mode. If a kernel-mode process is expecting an event to occur, it will not use signals to receive notification of that event.
- Rather, communication about incoming asynchronous events within the kernel takes place through the use of scheduling states and **wait queue structures**
- Whenever a process wants to wait for some event to complete, it places itself on a wait queue associated with that event and tells the scheduler that it is no longer eligible for execution.
- Once the event has completed, every process on the wait queue will be awoken.

➔Passing of Data among Processes

- Linux offers several mechanisms for passing data among processes.
- The standard UNIX pipe mechanism allows a child process to inherit a communication channel from its parent; data written to one end of the pipe can be read at the other.
- Under Linux, pipes appear as just another type of inode to virtual file system software, and each pipe has a pair of wait queues to synchronize the reader and writer.
- Another process communications method, shared memory, offers an extremely fast way to communicate large or small amounts of data.
- Any data written by one process to a shared memory region can be read immediately by any other process that has mapped that region into its address space.

9. FILE SYSTEMS

- The Linux kernel handles all types of files by hiding the implementation details of any single file type behind a layer of software, the virtual file system (VFS).

➔The Virtual File System

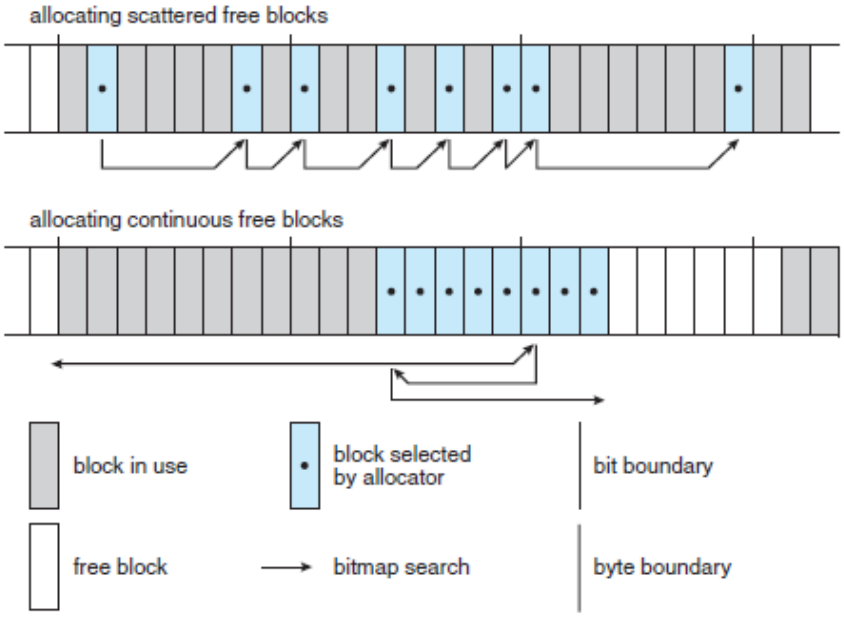
- The Linux VFS is designed around object-oriented principles. It has two components: a set of definitions that specify what file-system objects are allowed to look like and a layer of software to manipulate the objects.
- The VFS defines four main object types:
  - ➔ An inode object represents an individual file.
  - ➔ A file object represents an open file.
  - ➔ A superblock object represents an entire file system.
  - ➔ A dentry object represents an individual directory entry.
- For each of these four object types, the VFS defines a set of operations.

- ❖ Every object of one of these types contains a pointer to a function table.
- ❖ The function table lists the addresses of the actual functions that implement the defined operations for that object.
- ❖ For example, an abbreviated API for some of the file object's operations includes:

  int open(. . .) — Open a file.
  ssize t read(. . .) — Read from a file.
  ssize t write(. . .) — Write to a file.
  int mmap(. . .) — Memory-map a file.

➔The Linux ext3 File System

- ❖ The standard on-disk file system used by Linux is called ext3, for historical reasons.
- ❖ Linux was originally programmed with a Minix-compatible file system, to ease exchanging data with the Minix development system, but that file system was severely restricted by 14-character file-name limits and a maximum file-system size of 64 MB.
- ❖ The ext3 allocation policy works as follows: As in FFS, an ext3 file system is partitioned into multiple segments. In ext3, these are called block groups.
- ❖ FFS uses the similar concept of cylinder groups, where each group corresponds to a single cylinder of a physical disk.
- ❖ When allocating a file, ext3 must first select the block group for that file.
- ❖ For data blocks, it attempts to allocate the file to the block group to which the file's inode has been allocated. For inode allocations, it selects the lock group in which the file's parent directory resides for nondirectory files.



ext3 block-allocation policies.

➔Journaling

- The ext3 file system supports a popular feature called journaling, whereby modifications to the file system are written sequentially to a journal.
- A set of operations that performs a specific task is a transaction.
- Once a transaction is written to the journal, it is considered to be committed. Meanwhile, the journal entries relating to the transaction are replayed across the actual file system structures.
- As the changes are made, a pointer is updated to indicate which actions have completed and which are still incomplete. When an entire committed transaction is completed, it is removed from the journal.
- If the system crashes, some transactions may remain in the journal.
- Those transactions were never completed to the file system even though they were committed by the operating system, so they must be completed once the system recovers.
- The transactions can be executed from the pointer until the work is complete, and the file-system structures remain consistent.

➔The Linux Process File System

- The Linux process file system, known as the /proc file system, is an example of a file system whose contents are not actually stored anywhere but are computed on demand according to user file I/O requests.
- The /proc file system contains a illusionary file system.
- It does not exist on a disk. Instead, the kernel creates it in memory. It is used to provide information about the system (originally about processes, hence the name).
- Some of the more important files and directories are explained below. The /proc file system is described in more detail in the proc manual page.
- The /proc file system must implement **two things**: a **directory structure** and the **file contents within**.
- To allow efficient access to these variables from within applications, the /proc/sys subtree is made available through a special system call, sysctl(), that reads and writes the same variables in binary, rather than in text, without the overhead of the file system. sysctl() is not an extra facility; it simply reads the /proc dynamic entry tree to identify the variables to which the application is referring.

MOBILE OPERATING SYSTEMS

- A mobile operating system (OS) is software that allows smartphones, tablet PCs and other devices to run applications and programs.
- A mobile OS typically starts up when a device powers on, presenting a screen with icons or tiles that present information and provide application access. Mobile operating systems also manage cellular and wireless network connectivity, as well as phone access.
- Examples of mobile device operating systems include **Apple iOS**, **Google Android**, Research in Motion's **BlackBerry OS**, Nokia's Symbian, Hewlett-Packard's webOS (formerly Palm OS) and Microsoft's **Windows Phone OS**. Some, such as Microsoft's Windows 8, function as both a traditional desktop OS and a mobile operating system.

- ❖ Most mobile operating systems are tied to specific hardware, with little flexibility. Users can jailbreak or root some devices, however, which allows them to install another mobile OS or unlock restricted applications.

10.ANDRIOD VS IOS

IOS

It is Apple's mobile operating system used to run the popular iPhone, iPad, and iPod Touch devices. Formerly known as the iPhone OS, the name was changed with the introduction of the iPad. It interprets the commands of software applications ("apps") and it gives those apps access to features of the device, such as the multi-touch screen or the storage.

Features of IOS

- System Fonts
- Folders
- Notification center
- Accessibility
- Multitasking
- Switching Applications(application does not execute any code and may be removed from memory at any time)
- Task Completion (helps to ask extra time for completion of task)
- Background audio (helps to run in background)
- Voice over IP (in case phone call is not in progress)
- Background location (notified when location changes)
- Push notifications

ANDROID

Android is a software package and Linux based operating system for mobile devices such as tablet computers and smartphones. It is developed by Google in 2007 and later the OHA (Open Handset Alliance). Because Google developed Android, it comes with a lot of Google app services installed right out of the box. Gmail, Google Calendar, Google Maps, and Google Now are all pre-installed on most Android phones

Android OS has many features, among which are the following:

- Enhanced interface with the array of icons on the menu. Android adapts to high quality 2D and 3D graphics, with multi-touch support.
- Android supports multitasking, i.e. many applications will run at the same time, like in a computer. This is not possible with simple mobile phones and many other smartphones.
- All new means of connectivity are support: GSM, 3G, 4G, Wi-Fi, Bluetooth, GPS etc.
- Android supports many languages, including those with right-to-left text.
- Multimedia messaging system (MMS) is supported.

- Java runs great on Android. Applications for Android are developed in Java, but instead of a Java Runtime Environment, Android uses the Dalvik Executer, which is lighter on resources.
- Android supports most voice and video media formats, including streaming media.
- Additional hardware like sensors, gaming devices, other touchscreens can be integrated in Android.
- Voice and Video over IP. VoIP has many benefits, and Android manages cameras and has embedded support for seamless use of VoIP for free and cheap calls.
- On versions 2.2 and up, tethering is possible, which is the ability to use the Android device as a mobile WiFi hot spot.

Comparison chart

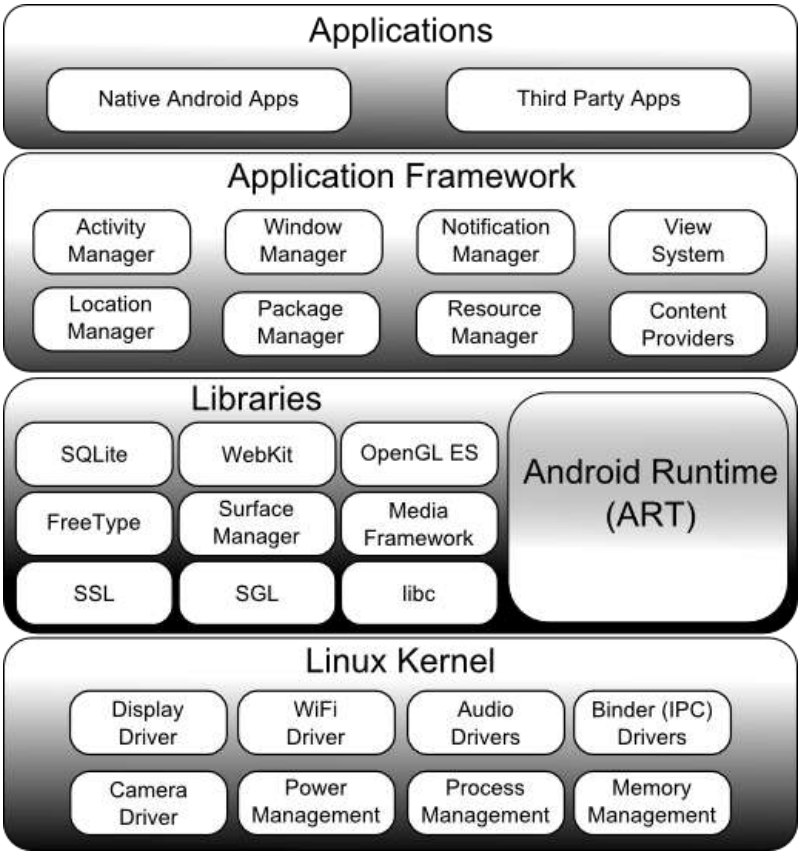
	Andriod	**iOS**
Source model	Open source	Closed, with open source components.
OS family	Linux	OS X, UNIX
Initial release	September 23, 2008	July 29, 2007
Customizability	A lot. Can change almost anything.	Limited unless jailbroken
Developer	Google, Open Handset Alliance	Apple Inc.
Widgets	Yes	No, except in NotificationCenter
Available language(s)	100+ Languages	34 Languages
File transfer	Easier than iOS. Using USB port and Android File Transfer desktop app. Photos can be transferred via USB without apps.	More difficult. Media files can be transferred using iTunes desktop app. Photos can be transferred out via USB without apps.
Available on	Many phones and tablets. Major manufacturers are Samsung, Motorola, LG, HTC and Sony.. Nexus and Pixel line of devices is pure Android, others bundle manufacturer software.	iPod Touch, iPhone, iPad, Apple TV (2nd and 3rd generation)
Calls and messaging	Google Hangouts. 3rd party apps like Facebook Messenger, WhatsApp, Google Duo and Skype all work on	iMessage, FaceTime (with other Apple devices only). 3rd party apps like Google Hangouts, Facebook

	Android and iOS both.	Messenger, WhatsApp, Google Duo and Skype all work on Android and iOS both.
Internet browsing	Google Chrome (or Android Browser on older versions; other browsers are available)	Mobile Safari (Other browsers are available)
App store , Affordability and interface	Google Play – 1,000,000+ apps. Other app stores like Amazon and Getjar also distribute Android apps. (unconfirmed ".APKs")	Apple app store – 1,000,000+ apps
Video chat	Google Duo and other 3rd party apps	FaceTime (Apple devices only) and other 3rd party apps
Voice commands	Google Now, Google Assistant	Siri
Working state	Current	Current
Maps	Google Maps	Apple Maps (Google Maps also available via a separate app download)
Latest stable release and Updates	Android 8.0.0, Oreo (Aug 21, 2017)	11 (Sep 19, 2017)
Alternative app stores and side loading	Several alternative app stores other than the official Google Play Store. (e.g. Aptoide, Galaxy Apps)	Apple blocks 3rd party app stores. The phone needs to be jailbroken if you want to download apps from other stores.
Battery life and management	Many Android phone manufacturers equip their devices with large batteries with a longer life.	Apple batteries are generally not as big as the largest Android batteries. However, Apple is able to squeeze decent battery life via hardware/software optimizations.
Open source	Kernel, UI, and some standard apps	The iOS kernel is not open source but is based on the open-source Darwin OS.
File manager	Yes. (Stock Android File Manager included on devices running Android	Not available

	7.1.1)	
Photos & Videos backup	Apps available for automatic backup of photos and videos. Google Photos allows unlimited backup of photos. OneDrive, Amazon Photos and Dropbox are other alternatives.	Up to 5 GB of photos and videos can be automatically back up with iCloud. All other vendors like Google, Amazon, Dropbox, Flickr and Microsoft have auto-backup apps for both iOS and Android.
Security	Android software patches are available soonest to Nexus device users. Manufacturers tend to lag behind in pushing out these updates. So at any given time a vast majority of Android devices are not running updated fully patched software.	Most people will never encounter a problem with malware because they don't go outside the Play Store for apps. Apple's software updates support older iOS devices also.
Rooting, bootloaders, and jailbreaking	Access and complete control over your device is available and you can unlock the bootloader.	Complete control over your device is not available.
Cloud services	Native integration with Google cloud storage. 15GB free, $2/mo for 100GB, 1TB for $10. Apps available for Amazon Photos, OneDrive and Dropbox.	Native integration with iCloud. 5GB free, 50GB for $1/mo, 200GB for $3/mo, 1TB for $10/mo. Apps available for Google Drive and Google Photos, Amazon Photos, OneDrive and Dropbox.
Interface	Touch Screen	Touch Screen
Supported versions	Android 5.0 & later (Android 4.4 is also supported but with patches)	iOS 8 & later
First version	Android 1.0, Alpha	iOS 1.0

11. IOS AND ANDROID ARCHITECTURE AND SDK FRAMEWORK

1. Android Architecture



Android System Architecture

The Android software stack generally consists of a Linux kernel and a collection of C/C++ libraries that is exposed through an application framework that provides services, and management of the applications and run time.

Linux Kernel

Android was created on the open source kernel of Linux. One main reason for choosing this kernel was that it provided proven core features on which to develop the Android operating system. The features of Linux kernel are:

1. Security:

The Linux kernel handles the security between the application and the system.

2. Memory Management:

It efficiently handles the memory management thereby providing the freedom to develop our apps.

3.Process Management:

It manages the process well, allocates resources to processes whenever they need them.

4. Network Stack:

It effectively handles the network communication.

5. Driver Model:

It ensures that the application works. Hardware manufacturers can build their drivers into the Linux build.

Libraries:

Running on the top of the kernel, the Android framework was developed with various features. It consists of various C/C++ core libraries with numerous of open source tools. Some of these are:

1. **The Android runtime:**
   The Android runtime consist of core libraries of Java and ART(the Android RunTime). Older versions of Android (4.x and earlier) had Dalvik runtime.
2. **Open GL(graphics library):**
   This cross-language, cross-platform application program interface (API) is used to produce 2D and 3D computer graphics.
3. **WebKit:**
   This open source web browser engine provides all the functionality to display web content and to simplify page loading.
4. **Media frameworks:**
   These libraries allow you to play and record audio and video.
5. **Secure Socket Layer (SSL):**
   These libraries are there for Internet security.

Android Runtime:

It is the third section of the architecture. It provides one of the key components which is called Dalvik Virtual Machine. It acts like Java Virtual Machine which is designed specially for Android. Android uses it's own custom VM designed to ensure that multiple instances run efficiently on a single device.

The Delvik VM uses the device's underlying Linux kernel to handle low-level functionality,including security,threading and memory management.

Application Framework

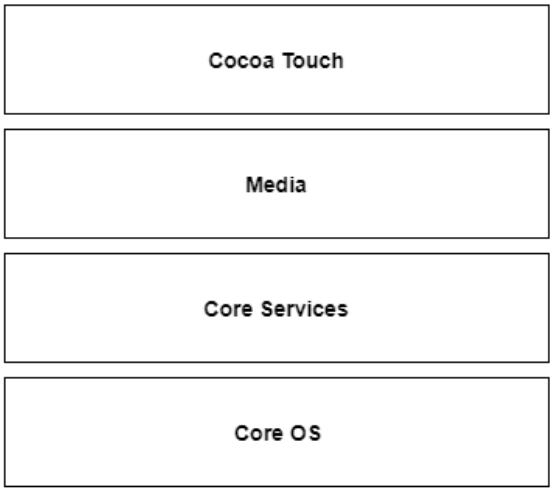
The Android team has built on a known set proven libraries, built in the background, and all of it these is exposed through Android interfaces. These interfaces warp up all the various libraries and make them useful for the Developer. They don't have to build any of the functionality provided by the android. Some of these interfaces include:

1. **Activity Manager:**
   It manages the activity lifecycle and the activity stack.
2. **Telephony Manager:**
   It provides access to telephony services as related subscriber information, such as phone numbers.
3. **View System:**
   It builds the user interface by handling the views and layouts.
4. **Location manager:**
   It finds the device's geographic location.

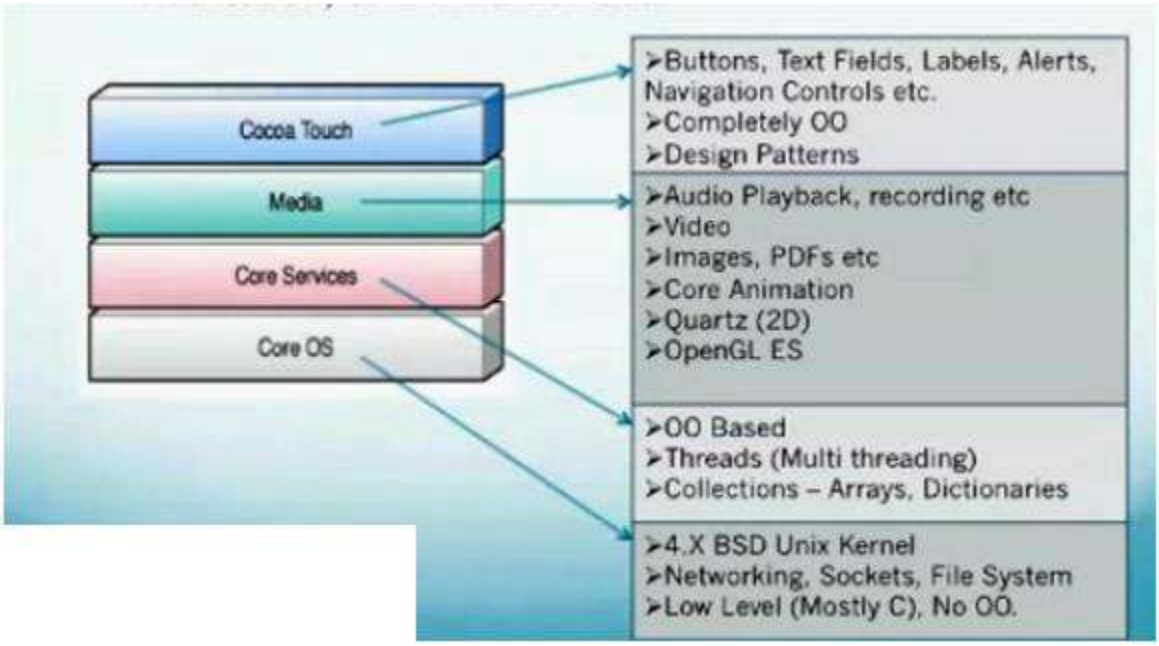
Applications:

Android applications can be found at the topmost layer. At application layer we write our application to be installed on this layer only. Examples of applications are Games, Messages, Contacts etc.

2. iOS Architecture



Apple iOS Architecture



Cocoa Touch Layer

The Cocoa Touch layer sits at the top of the iOS stack and contains the frameworks that are most commonly used by iPhone application developers. Cocoa Touch is primarily written in Objective-C, is based on the standard Mac OS X Cocoa API (as found on Apple

desktop and laptop computers) and has been extended and modified to meet the needs of the iPhone.

- **Primarily Objective-C**
- **Based off the Mac OS X Cocoa API**
- **Frameworks**
  - UIKit - UI Elements, lifecycle management, touch, gestures
  - Address Book UI – Contacts, adding, editing
  - Event Kit UI – Calendar events
  - Game Kit Framework – P2P networking, Game Center
  - iAd – Apple's advertising platform
  - Map Kit – Google maps
  - Message UI – Email and SMS

The iOS Media Layer

The role of the Media layer is to provide iOS with audio, video, animation and graphics capabilities. As with the other layers comprising the iOS stack, the Media layer comprises a number of frameworks that may be utilized when developing iPhone apps.

Core OS Layer:

All the iOS technologies are build on the low level features provided by the Core OS layer. These technologies include Core Bluetooth Framework, External Accessory Framework, Accelerate Framework, Security Services Framework, Local Authorisation Framework etc.

iOS Core Services

The iOS Core Services layer provides much of the foundation on which the previously referenced layers are built

********** **End of Andriod and iOS Architecture Framework**********

12. THE iOS MEDIA LAYER

- The role of the Media layer is to provide iOS with audio, video, animation and graphics capabilities.
- As with the other layers comprising the iOS stack, the Media layer comprises a number of frameworks that may be utilized when developing iPhone apps.

Core Video Framework (CoreVideo.framework)

A new framework introduced with iOS 4 to provide buffering support for the Core Media framework. This may be utilized by application developers it is typically not necessary to use this framework.

Core Text Framework (CoreText.framework)

- The iOS Core Text framework is a C-based API designed to ease the handling of advanced text layout and font rendering requirements.

Image I/O Framework (ImageIO.framework)

- The Image IO framework, the purpose of which is to facilitate the importing and exporting of image data and image metadata, was introduced in iOS 4.
- The framework supports a wide range of image formats including PNG, JPEG, TIFF and GIF.

Assets Library Framework (AssetsLibrary.framework)

- The Assets Library provides a mechanism for locating and retrieving video and photo files located on the iPhone device.
- In addition to accessing existing images and videos, this framework also allows new photos and videos to be saved to the standard device photo album.

Core Graphics Framework (CoreGraphics.framework)

- The iOS Core Graphics Framework (otherwise known as the Quartz 2D API) provides a lightweight two dimensional rendering engine.
- Features of this framework include PDF document creation and presentation, vector based drawing, transparent layers, path based drawing, anti-aliased rendering, color manipulation and management, image rendering and gradients.
- Those familiar with the Quartz 2D API running on MacOS X will be pleased to learn that the implementation of this API is the same on iOS.

Quartz Core Framework (QuartzCore.framework)

- The purpose of the Quartz Core framework is to provide animation capabilities on the iPhone.
- It provides the foundation for the majority of the visual effects and animation used by the UIKit framework and provides an Objective-C based programming interface for creation of specialized animation within iPhone apps.

OpenGL ES framework (OpenGLES.framework)

- For many years the industry standard for high performance 2D and 3D graphics drawing has been OpenGL.
- Originally developed by the now defunct Silicon Graphics, Inc (SGI) during the 1990s in the form of GL, the open version of this technology (OpenGL) is now under the care of a non-profit consortium comprising a number of major companies including Apple, Inc., Intel, Motorola and ARM Holdings.

- OpenGL for Embedded Systems (ES) is a lightweight version of the full OpenGL specification designed specifically for smaller devices such as the iPhone,iOS 3 or later supports both OpenGL ES 1.1 and 2.0 on certain iPhone models (such as the iPhone 3GS and iPhone 4). Earlier versions of iOS and older device models support only OpenGL ES version 1.1.

iOS Audio Support

- iOS is capable of supporting audio in AAC, Apple Lossless (ALAC), A-law, IMA/ADPCM, Linear PCM, µ-law, DVI/Intel IMA ADPCM, Microsoft GSM 6.10 and AES3-2003 formats through the support provided by the following frameworks.

AV Foundation framework (AVFoundation.framework)

- An Objective-C based framework designed to allow the playback, recording and management of audio content.

Core Audio Frameworks (CoreAudio.framework, AudioToolbox.framework and AudioUnit.framework)

- The frameworks that comprise Core Audio for iOS define supported audio types, playback and recording of audio files and streams and also provide access to the device's built-in audio processing units.

Open Audio Library (OpenAL)

- OpenAL is a cross platform technology used to provide high-quality, 3D audio effects (also referred to as positional audio).
- Positional audio can be used in a variety of applications though is typically using to provide sound effects in games.

Media Player framework (MediaPlayer.framework)

- The iOS Media Player framework is able to play video in .mov, .mp4, .m4v, and .3gp formats at a variety of compression standards, resolutions and frame rates.

Core Midi Framework (CoreMIDI.framework)

- Introduced in iOS 4, the Core MIDI framework provides an API for applications to interact with MIDI compliant devices such as synthesizers and keyboards via the iPhone's dock connector.

*****************************End of iOS Media Layer*****************************

13. THE iOS CORE SERVICES LAYER

- The iOS Core Services layer provides much of the foundation on which the previously referenced layers are built and consists of the following frameworks.

Address Book framework (AddressBook.framework)

- The Address Book framework provides programmatic access to the iPhone Address Book contact database allowing applications to retrieve and modify contact entries.

CFNetwork Framework (CFNetwork.framework)

- The CFNetwork framework provides a C-based interface to the TCP/IP networking protocol stack and low level access to BSD sockets.
- This enables application code to be written that works with HTTP, FTP and Domain Name servers and to establish secure and encrypted connections using Secure Sockets Layer (SSL) or Transport Layer Security (TLS).

Core Data Framework (CoreData.framework)

- This framework is provided to ease the creation of data modeling and storage in Model-View-Controller (MVC) based applications.
- Use of the Core Data framework significantly reduces the amount of code that needs to be written to perform common tasks when working with structured data in an application.

Core Foundation Framework (CoreFoundation.framework)

- The Core Foundation is a C-based Framework that provides basic functionality such as data types, string manipulation, raw block data management, URL manipulation, threads and run loops, date and times, basic XML manipulation and port and socket communication. Additional XML capabilities beyond those included with this framework are provided via the libXML2 library.
- Though this is a C-based interface, most of the capabilities of the Core Foundation framework are also available with Objective-C wrappers via the Foundation Framework.

Core Media Framework (CoreMedia.framework)

- The Core Media framework is the lower level foundation upon which the AV Foundation layer is built.
- While most audio and video tasks can, and indeed should, be performed using the higher level AV Foundation framework, access is also provided for situations where lower level control is required by the iOS application developer.

Core Telephony Framework (CoreTelephony.framework)

- The iOS Core Telephony framework is provided to allow applications to interrogate the device for information about the current cell phone service provider and to receive notification of telephony related events.

EventKit Framework (EventKit.framework)

- An API designed to provide applications with access to the calendar and alarms on the device.

Foundation Framework (Foundation.framework)

- The Foundation framework is the standard Objective-C framework that will be familiar to those that have programmed in Objective-C on other platforms (most likely Mac OS X).

- Essentially, this consists of Objective-C wrappers around much of the C-based Core Foundation Framework.

Core Location Framework (CoreLocation.framework)

- The Core Location framework allows you to obtain the current geographical location of the device (latitude and longitude) and compass readings from with your own applications.
- The method used by the device to provide coordinates will depend on the data available at the time the information is requested and the hardware support provided by the particular iPhone model on which the app is running (GPS and compass are only featured on recent models).
- This will either be based on GPS readings, Wi-Fi network data or cell tower triangulation (or some combination of the three).

Mobile Core Services Framework (MobileCoreServices.framework)

- The iOS Mobile Core Services framework provides the foundation for Apple's Uniform Type Identifiers (UTI) mechanism, a system for specifying and identifying data types.
- A vast range of predefined identifiers have been defined by Apple including such diverse data types as text, RTF, HTML, JavaScript, PowerPoint .ppt files, PhotoShop images and MP3 files.

Store Kit Framework (StoreKit.framework)

- The purpose of the Store Kit framework is to facilitate commerce transactions between your application and the
- Apple App Store. Prior to version 3.0 of iOS, it was only possible to charge a customer for an app at the point that they purchased it from the App Store. iOS 3.0 introduced the concept of the "in app purchase" whereby the user can be given the option make additional payments from within the application.
- This might, for example, involve implementing a subscription model for an application, purchasing additional functionality or even buying a faster car for you to drive in a racing game.

SQLite library

- Allows for a lightweight, SQL based database to be created and manipulated from within your iPhone application.

System Configuration Framework (SystemConfiguration.framework)

- The System Configuration framework allows applications to access the network configuration settings of the device to establish information about the "reachability" of the device (for example whether Wi-Fi or cell connectivity is active and whether and how traffic can be routed to a server).

Quick Look Framework (QuickLook.framework)

- One of the many new additions included in iOS 4, the Quick Look framework provides a useful mechanism for displaying previews of the contents of files types loaded onto the

device (typically via an internet or network connection) for which the application does not already provide support.

- File format types supported by this framework include iWork, Microsoft Office document, Rich Text Format, Adobe PDF, Image files, public.text files and comma separated (CSV).

********************************End of Core Services Layer************************

14.The iOS Core OS Layer

- The Core OS Layer occupies the bottom position of the iOS stack and, as such, sits directly on top of the device hardware.
- The layer provides a variety of services including low level networking, access to external accessories and the usual fundamental operating system services such as memory management, file system handling and threads.

Accelerate Framework (Accelerate.framework)

- Introduced with iOS 4, the Accelerate Framework provides a hardware optimized C-based API for performing complex and large number math, vector, digital signal processing (DSP) and image processing tasks and calculations.

External Accessory framework (ExternalAccessory.framework)

- Provides the ability to interrogate and communicate with external accessories connected physically to the iPhone via the 30-pin dock connector or wirelessly via Bluetooth.

Security Framework (Security.framework)

- The iOS Security framework provides all the security interfaces you would expect to find on a device that can connect to external networks including certificates, public and private keys, trust policies, keychains, encryption, digests and Hash-based Message Authentication Code (HMAC).

System (LibSystem)

- As we have previously mentioned, the iOS is built upon a UNIX-like foundation.
- The System component of the Core OS Layer provides much the same functionality as any other UNIX like operating system. This layer includes the operating system kernel (based on the Mach kernel developed by Carnegie Mellon University) and device drivers.
- The kernel is the foundation on which the entire iOS is built and provides the low level interface to the underlying hardware.
- Amongst other things the kernel is responsible for memory allocation, process lifecycle management, input/output, inter-process communication, thread management, low level networking, file system access and thread management.
- As an app developer your access to the System interfaces is restricted for security and stability reasons. Those interfaces that are available to you are contained in a C-based library called LibSystem.

**********************End od Core OS Layer****************************

15. FILE SYSTEM BASICS

- A *file system* handles the persistent storage of data files, apps, and the files associated with the operating system itself. Therefore, the file system is one of the fundamental resources used by all processes.
- APFS is the default file system in macOS, iOS, watchOS, and tvOS. APFS replaces HFS+ as the default file system for iOS 10.3 and later, and macOS High Sierra and later macOS additionally supports a variety of other formats, as described in Supported File Systems.
- The file system uses directories to create a hierarchical organization
- Before you begin writing code that interacts with the file system, you should first understand a little about the organization of file system and the rules that apply to your code.
- Aside from the basic tenet that you cannot write files to directories for which you do not have appropriate security privileges, apps are also expected to be good citizens and put files in appropriate places.
- Precisely where you put files depends on the platform, but the overarching goal is to make sure that the user's files remain easily discoverable and that the files your code uses internally are kept out of the user's way.

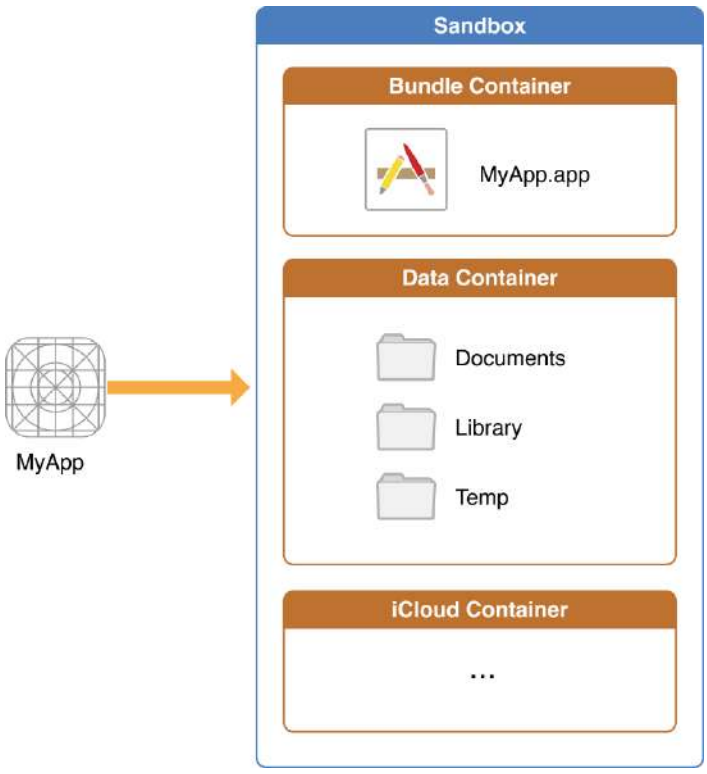
About the iOS File System

- The iOS file system is geared toward apps running on their own. To keep the system simple, users of iOS devices do not have direct access to the file system and apps are expected to follow this convention.

iOS Standard Directories: Where Files Reside ?

- For security purposes, an iOS app's interactions with the file system are limited to the directories inside the app's sandbox directory.
- During installation of a new app, the installer creates a number of container directories for the app inside the sandbox directory.
- Each container directory has a specific role.
- The bundle container directory holds the app's bundle, whereas the data container directory holds data for both the app and the user.
- The data container directory is further divided into a number of subdirectories that the app can use to sort and organize its data.

- The app may also request access to additional container directories—for example, the iCloud container—at runtime.
- These container directories constitute the app's primary view of the file system. The Figure shows a representation of the sandbox directory for an app.
- An iOS app operating within its own sandbox directory



- An app is generally prohibited from accessing or creating files outside its container directories.
- One exception to this rule is when an app uses public system interfaces to access things such as the user's contacts or music.
- In those cases, the system frameworks use helper apps to handle any file-related operations needed to read from or modify the appropriate data stores.
- Table lists some of the more important subdirectories inside the sandbox directory and describes their intended usage.
- This table also describes any additional access restrictions for each subdirectory and points out whether the directory's contents are backed up by iTunes and iCloud.

Table Commonly used directories of an iOS app

Directory	Description

AppName`.app`	This is the app's bundle. This directory contains the app and all of its resources. You cannot write to this directory. To prevent tampering, the bundle directory is signed at installation time. Writing to this directory changes the signature and prevents your app from launching. You can, however, gain read-only access to any resources stored in the apps bundle. For more information, see the *Resource Programming Guide* The contents of this directory are not backed up by iTunes or iCloud. However, iTunes does perform an initial sync of any apps purchased from the App Store.
`Documents/`	Use this directory to store user-generated content. The contents of this directory can be made available to the user through file sharing; therefore, his directory should only contain files that you may wish to expose to the user. The contents of this directory are backed up by iTunes and iCloud.
`Documents/Inbox`	Use this directory to access files that your app was asked to open by outside entities. Specifically, the Mail program places email attachments associated with your app in this directory. Document interaction controllers may also place files in it. Your app can read and delete files in this directory but cannot create new files or write to existing files. If the user tries to edit a file in this directory, your app must silently move it out of the directory before making any changes. The contents of this directory are backed up by iTunes and iCloud.
`Library/`	This is the top-level directory for any files that are not user data files. You typically put files in one of several standard subdirectories. iOS apps commonly use the `Application Support` and `Caches` subdirectories; however, you can create custom subdirectories. Use the `Library` subdirectories for any files you don't want exposed to the user. Your app should not use these directories for user data files. The contents of the `Library` directory (with the exception of the `Caches` subdirectory) are backed up by iTunes and iCloud. For additional information about the Library directory and its commonly used subdirectories, see The Library Directory Stores App-Specific Files.
`tmp/`	Use this directory to write temporary files that do not need to persist between launches of your app. Your app should remove files from this directory when they are no longer needed; however, the system may purge this directory when your app is not running.

	The contents of this directory are not backed up by iTunes or iCloud.

An iOS app may create additional directories in the `Documents`, `Library`, and `tmp` directories. You might do this to better organize the files in those locations.

Where You Should Put Your App's Files

- To prevent the syncing and backup processes on iOS devices from taking a long time, be selective about where you place files. Apps that store large files can slow down the process of backing up to iTunes or iCloud.
- These apps can also consume a large amount of a user's available storage, which may encourage the user to delete the app or disable backup of that app's data to iCloud. With this in mind, you should store app data according to the following guidelines:
- **Put user data in `Documents/`.** User data generally includes any files you might want to expose to the user—anything you might want the user to create, import, delete or edit. For a drawing app, user data includes any graphic files the user might create. For a text editor, it includes the text files. Video and audio apps may even include files that the user has downloaded to watch or listen to later.
- **Put app-created support files in the `Library/Application support/`** directory. In general, this directory includes files that the app uses to run but that should remain hidden from the user. This directory can also include data files, configuration files, templates and modified versions of resources loaded from the app bundle.
- **Remember that files in `Documents/` and `Application Support/`** are backed up by default. You can exclude files from the backup by calling using the **`NSURLIsExcludedFromBackupKey` key**. Any file that can be re-created or downloaded must be excluded from the backup. This is particularly important for large media files. If your application downloads video or audio files, make sure they are not included in the backup.
- **Put temporary data in the `tmp/`** directory. Temporary data comprises any data that you do not need to persist for an extended period of time. Remember to delete those files when you are done with them so that they do not continue to consume space on the user's device. The system will periodically purge these files when your app is not running; therefore, you cannot rely on these files persisting after your app terminates.
- **Put data cache files in the Library/Caches/** directory. Cache data can be used for any data that needs to persist longer than temporary data, but not as long as a support file. Generally speaking, the application does not require cache data to operate properly, but it can use cache data to improve performance.
- Examples of cache data include (but are not limited to) database cache files and transient, downloadable content. Note that the system may delete the **Caches/** directory to free up disk space, so your app must be able to re-create or download these files as needed.

PART A

1. **What are the components of a Linux System?**
   Linux system composed of three main modules . They are:
   i) Kernel
   ii) System libraries
   iii) System utilities
2. **What are the main supports for the Linux modules?**
   The Module support under Linux has three components. They are:
   i) Module management
   ii) Driver registration
   iii) Conflict resolution mechanism.
3. **Define shell.**
   A shell is a program that provides the traditional, text-only user interface for Linux and other Unix-like operating systems. Its primary function is to read commands that are typed into the console.
4. **What is meant by kernel in Linux system?**
   Kernel is responsible for maintaining all the important abstractions of the operating system including such things as virtual memory and processes.
5. **What is meant by system Libraries?**
   System libraries define a standard set of functions through which applications can interact with the kernel and that implement much of the operating-system functionality that doesn't need the full privileges of kernel code.
6. **What is meant by system Utilities?**
   System Utilities are system programs that perform individual, specialized management tasks. Some of the system utilities may be invoked just to initialize and configure some aspect of the system and others may run permanently, handling such tasks as responding to incoming network connections, accepting logon requests from terminals or updating log files.
7. **What do you meant by process?**
   Process is the basic context within in which all user-requested activity is serviced with in the OS.
8. **What is meant by Process-ID**
   A PID is an acronym for process identification number on a Linux or Unix-like operating system. A PID is automatically assigned to each process when it is created. A process is nothing but running instance of a program and each process has a unique PID on a Unix-like system.
9. **What is meant by personality?**
   Process personality are primarily used by emulation libraries to request that system call be compatible with certain version of UNIX
10. **What is meant by buffer cache?**
    It is the kernel's main cache for blocked-oriented devices such as disk drivers and is the main mechanism through which I/O to these devices is performed.
11. **What is the disadvantage of static linking?**
    The main disadvantage of static linking is that every program generated must contain copies of exactly the same common system library functions.

12. What is the function of module management?

The module management allows modules to be loaded into memory and to talk to the rest of the kernel.

13. What is the function of driver registration?

Driver registration allows modules to tell the rest of the kernel that a new driver has become available.

14. What is the function of conflict resolution mechanism?

This mechanism allows different device drivers to reserve hardware resources and to protect those resources from accidental use by another driver.

15. What is meant by device drivers?

Device drivers include i) character devices such as printers, terminals ii) Block devices (including all disk drives) and network interface devices.

16. What is a character device?

A device driver which does not offer random access to fixed blocks of data. A character device driver must register a set of functions which implement the driver's various file I/O operations.

17. What is Mobile OS?

A mobile operating system (mobile OS) is an OS built exclusively for a mobile device, such as a smartphone, personal digital assistant (PDA), tablet or other embedded mobile OS.

18. What is iOS?

iOS is a mobile operating system created and developed by Apple Inc. exclusively for its hardware. It is the operating system that presently powers many of the company's mobile devices, including the iPhone, iPad, and iPod Touch. It is the second most popular mobile operating system globally after Android.

19. List the services available in iOS.

i) Cocoa Touch
ii) Media layer
iii) Service layer
iv) Core OS layer

20. List the features of iOS.

i) System fonts
ii) Folders
iii) Notification center
iv) Accessibility
v) Multitasking
vi) Switching Applications
vii) Task completion
viii) Background audio
ix) Voice over IP
x) Background Location
xi) Push notification

21. List the advantages of iOS

- Best gaming experience.
- A vast number of applications.

- Suits for business and gaming.
- Excellent UI and fluid responsive.
- The latest version has two notification menus.
- Excellent security.
- Multitasking.
- Jailbreaking for customization.
- Wearables are getting launched.
- Feel is awesome.
- Excellent for media entertainment.
- Multi-language support.
- Apple Pay Support.
- Quick settings in the notification bar.

22. List the disadvantages of iOS

- It has a review process, when developers want to publish an app they need to send it to Apple for review that takes around 7 days and it takes even more in some cases.
- Applications are very large when compared to other mobile platforms
- Using iOS are costly Apps and no widget support
- Battery performance is very poor on 3G
- Repair costs are very piracy
- Not flexible only supports iOS devices

23. List the advantages of Android

- Android Is More Customizable Can change almost anything.
- In Android, any new publication can be done easily and without any review process
- Use a Different Messaging App for SMS
- Android Offers an Open Platform
- Easy access to the Android App Market
  - Cost Effective
- Upcoming versions have a support to save RAW images
- Built in Beta Testing and staged rollout

24. List the disadvantages of Android

- Need internet connection
- Advertising
- Wasteful memory
- Many application contain viruses

25. How are iOS and Android similar? How are they different?

Similarities: Both are based on existing kernel. Both have architecture that uses software stacks. Both provide framework for developers

Difference: iOS is closed-source and Android is open source. iOS applications are developed in objective C, Android in java. Android uses a virtual machine, and iOS executes code natively.

26. Describe some challenges of designing OS for mobile device compared with designing OS for traditional PC's

- Less storage capacity means the OS must be manage memory
- Less processing power plus fewer processors mean the operating system carefully apportion
- Processors to applications

Part- B

1. Draw a neat sketch of overview of iOS architecture and explain in detail. (13)
2. Discuss process management and scheduling in LINUX. (13)
3. Illustrate some existing SDK architecture implementation frameworks.(13)
4. Describe about the network structure of LINUX system.(13)
5. Explain in detail the design principles, kernel modules in LINUX system. (7+6)
6. Demonstrate the functions of the kernel, service and command layers ofOS.(13)
7. Generalize the importance of memory management in Operating system.(13)
8. Explain in detail about file system management done in LINUX.(13)
9. Summarize Inter Process Communication with suitable example.(13)
10. Analyze:
    a. mobile OS (5) ii) desktop OS(4) iii) multi-user OS (4)
11. Compare and contrast Andriod OS and IOS. (13)
12. Explain in detail about Linux architecture. (13)
13. Compare the functions of media layer, service layer and core OS layer.
14. Explain the basic concepts of the Linux system
15. 2. Explain about kernel modules
16. 3. Explain in detail about the process management in Linux
17. 4. Explain in detail about the scheduling in Linux
18. 5. Explain the iOS architecture and various layers available in iOS
19. Discuss about various services in the media layer
20. Discuss about various services in the iOS core OS layer
21. Discuss about various services in the iOS service OS layer