Syllubus: Computer System and Operating System Overview: Overview of computer operating systems, operating systems functions(operations,services), protection and security, distributed systems, special purpose systems, operating systems structures and systems calls, operating systems generation.

Computer System Overview:

An operating system (OS) exploits the hardware resources of one or more processors to provide a set of services to system users. The OS also manages secondary memory and I/O input /output) devices on behalf of its users. Accordingly, it is important to have some understanding of the underlying computer system hardware before we begin our examination of operating systems.

Basic Elements: 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.

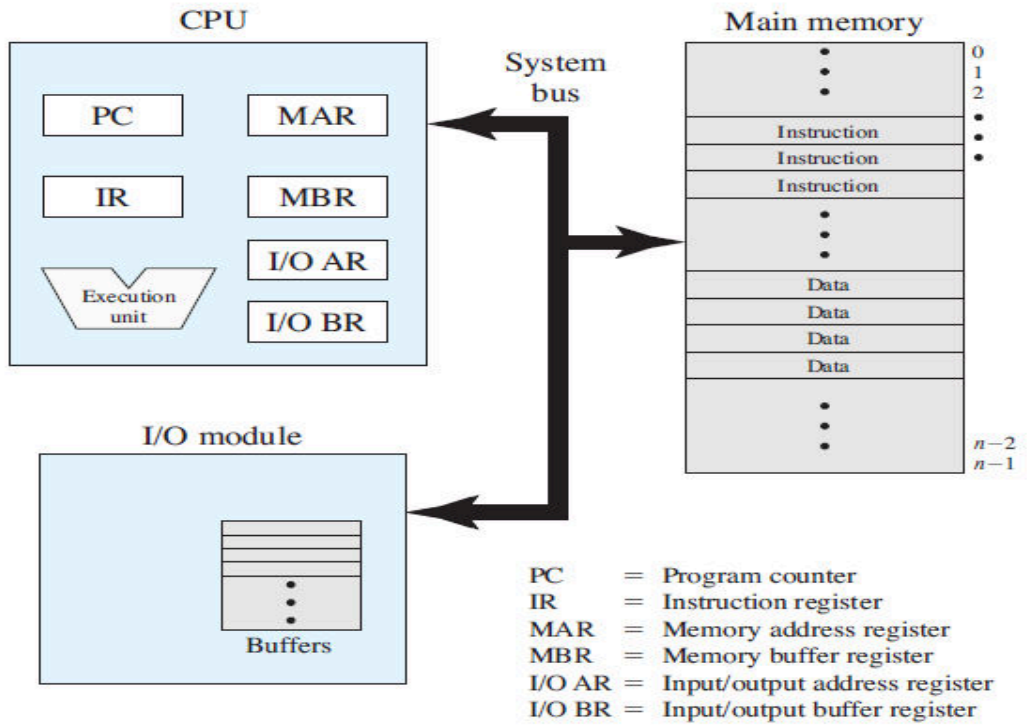


Figure 1.1 Computer Components: Top-Level View

Processor Registers:

A processor includes a set of registers that provide memory that is faster and smaller than main memory. Processor registers are of two types:

- **User-Visible Registers :**A user-visible register may be referenced by means of the machine language that the processor executes and is generally available to all programs, including application programs as well as system programs. Enable the machine or assembly language programmer to minimize main memory references by optimizing register use. Types of registers that are typically available are data, address, and condition code registers.

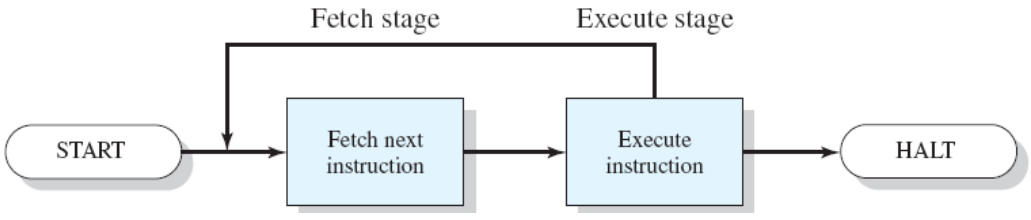
    - **Data registers :** can be assigned to a variety of functions by the programmer. In some cases, they are general purpose in nature and can be used with any machine instruction that performs operations on data. Often, however, there are restrictions. For example, there may be dedicated registers for floating-point operations and others for integer operations.
    - **Address registers :** contain main memory addresses of data and instructions, or they contain a portion of the address that is used in the calculation of the complete or effective address. These registers may themselves be general purpose, or may be devoted to a particular way, or mode, of addressing memory.

- **Control and Status Registers :**A variety of processor registers are employed to control the operation of the processor. On most processors, most of these are not visible to the user. Some of them may be accessible by machine instructions executed in what is referred to as a control or kernel mode. The following are essential to instruction execution:

    - **Program counter (PC):** Contains the address of the next instruction to be fetched.
    - **Instruction register (IR):** Contains the instruction most recently fetched.
    - **Condition codes :** (also referred to as *flags*) are bits typically set by the processor hardware as the result of operations. For example, an arithmetic operation may produce a positive, negative, zero, or overflow result. In addition to the result itself being stored in a register or memory, a condition code is also set following the execution of the arithmetic instruction.The condition code may subsequently be tested as part of a conditional branch operation. Condition code bits are collected into one or more registers.

Instruction Execution: 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. Instruction execution may involve several operations and depends on the nature of the instruction.

The processing required for a single instruction is called an *instruction cycle*. Using a simplified two-step description, the instruction cycle is depicted in the following figure. The two steps are referred to as the *fetch stage* and the *execute 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.



Basic Instruction Cycle

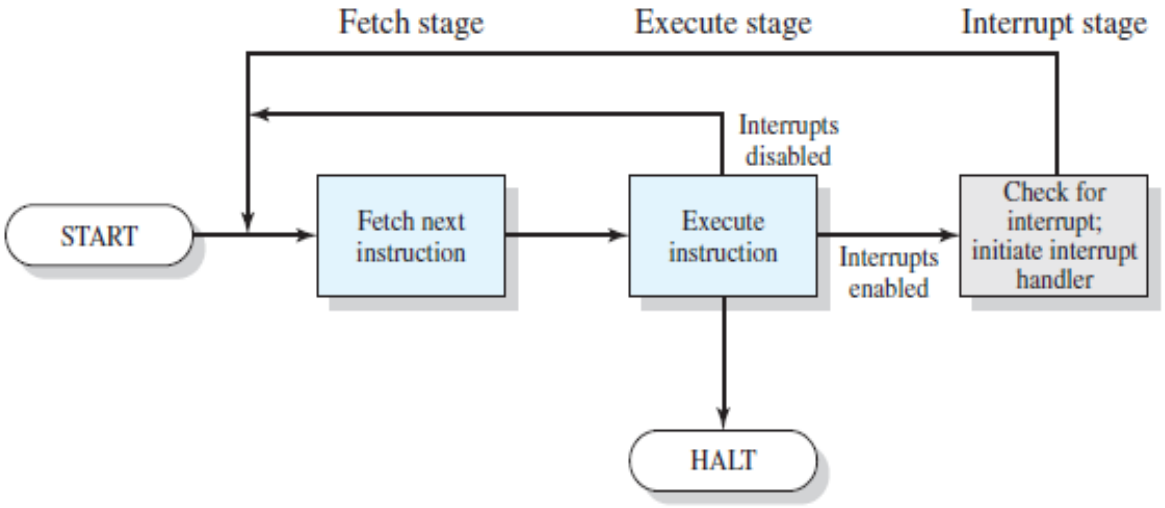
At the beginning of each instruction cycle, the processor fetches an instruction from memory. Typically, the program counter (PC) holds the address of the next instruction to be fetched. Unless instructed otherwise, the processor always increments the PC after each instruction fetch so that it will fetch 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.

Interrupts: Virtually all computers provide a mechanism by which other modules (I/O, memory)may interrupt the normal sequencing of the processor. The following table lists the most common classes of interrupts. Interrupts are provided primarily as a way to improve processor utilization.

Classes of Interrupts

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.
Timer	Generated by a timer within the processor. This allows the operating system to perform certain functions on a regular basis.
I/O	Generated by an I/O controller, to signal normal completion of an operation or to signal a variety of error conditions.
Hardware failure	Generated by a failure, such as power failure or memory parity error.

Interrupts and the Instruction Cycle: With interrupts, the processor can be engaged in executing other instructions while an I/O operation is in progress. For the user program, an interrupt suspends the normal sequence of execution. When the interrupt processing is completed, execution resumes Thus, the user program does not have to contain any special code to accommodate interrupts; the processor and the OS are responsible for suspending the user program and then resuming it at the same point. To accommodate interrupts, an *interrupt stage* is added to the instruction cycle, In the interrupt stage, the processor checks to see if any interrupts have occurred, indicated by the presence of an interrupt signal. If no interrupts are pending, the processor proceeds to the fetch stage and fetches the next instruction of the current program. If an interrupt is pending, the processor suspends execution of the current program and executes an *interrupt-handler* routine. The interrupt-handler routine is generally part of the OS. Typically, this routine determines the nature of the interrupt and performs whatever actions are needed. The following figure describes instruction cycle with interrupts.



Basic Instruction Cycle with Interrupts

Interrupt Processing: An interrupt triggers a number of events, both in the processor hardware and in software. The following figure shows a typical sequence. When an I/O device completes an I/O operation, the following sequence of hardware events occurs:

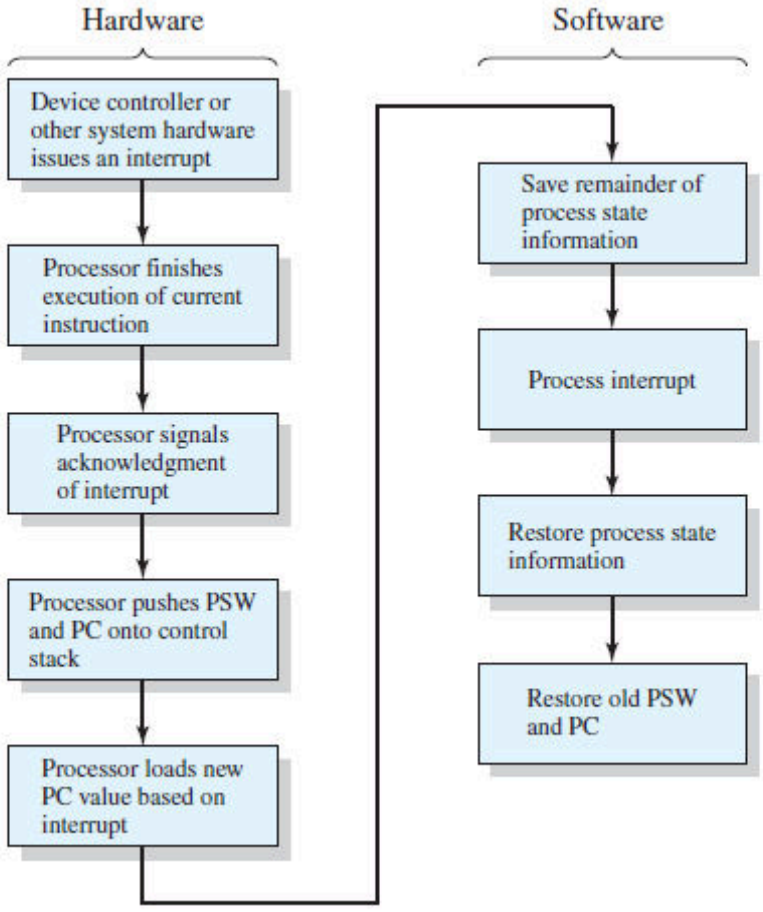
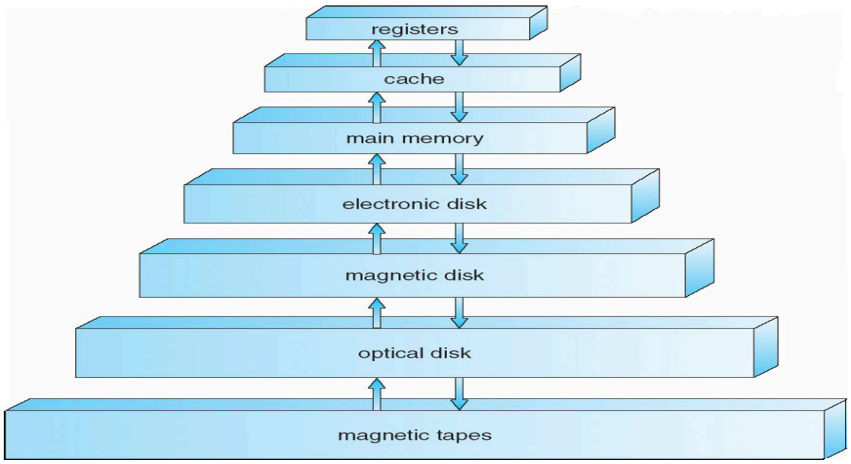


Figure 1.5: Interrupt Processing

1. The device issues an interrupt signal to the processor.

2. The processor finishes execution of the current instruction before responding to the interrupt, as indicated in Figure.
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. To begin, it saves information needed to resume the current program at the point of interrupt. The minimum information required is the program status word (PSW) and the location of the next instruction to be executed, which is contained in the program counter.These can be pushed onto a control stack.
5. The processor then loads the program counter with the entry location of the interrupt-Handling routine that will respond to this interrupt. Depending on the computer architecture and OS design, there may be a single program, one for each type of interrupt, or one for each device and each type of interrupt. If there is more than one interrupt-handling routine, the processor must determine which one to invoke. This information may have been included in the original interrupt signal, or the processor may have to issue a request to the device that issued the interrupt to get a response that contains the needed information.
6. At this point, the program counter and PSW relating to the interrupted program have been saved on the control stack. However, there is other information that is considered part of the state of the executing program. In particular, the contents of the processor registers need to be saved, because these registers may be used by the interrupt handler. So all of these values, plus any other state information, need to be saved. Typically, the interrupt handler will begin by saving the contents of all registers on the stack.
7. The interrupt handler may now proceed to process the interrupt.This includes an examination of status information relating to the I/O operation or other event that caused an interrupt. It may also involve sending additional commands or acknowledgments to the I/O device.
8. When interrupt processing is complete, 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.As a result, the next instruction to be executed will be from the previously interrupted program.

Memory Hierarchy: The Memory Unit is an essential component in any digital computer, because it is needed for storing programs and data. The Memory hierarchy system consists of all storage devices employed in a computer system from the slow but high capacity secondary memory to a relative faster main memory. A typical hierarchy is illustrated in **Figure.**



Memory Hierarchy

As one goes down the hierarchy, the following occur:

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

Thus, smaller, more expensive, faster memories are supplemented by larger, cheaper, slower memories. The key to the success of this organization decreasing frequency of access at lower levels.

Registers: Registers are fastest of all memories, a register is a group of flip-flops with each flip-flop capable of storing one bit of information. An n-bit register has a group of n flip-flops and is capable of storing any binary information of n bits.

Cache Memory: Cache is a special very high speed memory. It is the intermediate Memory between CPU and Main Memory. The Cache is used for storing segments of programs currently being executed by the CPU and temporarily data, frequently needed in the present calculations.

Main Memory: The Memory unit that communicates directly with CPU is called **"Main Memory"**. The main memory is a relatively large and fast memory used to store programs and data during the processor execution. The principle technology used for the main memory is based on semiconductor integrated circuits(RAM chips). It is a volatile memory, it means the stored information remains valid as long as power is applied to the unit.

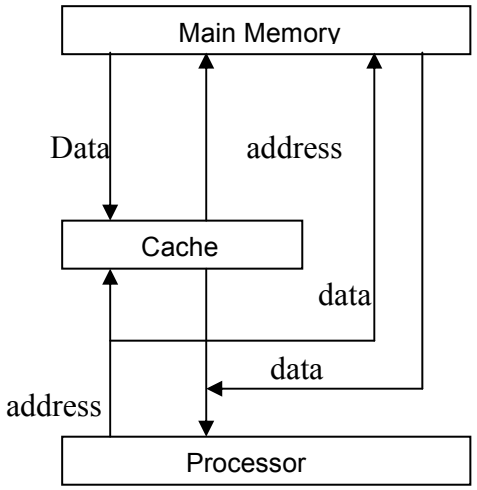
Magnetic Disk: Devices that provide backup storage are called **"Auxiliary Memory" or "Secondary Memory".** The most common auxiliary memory used in computer systems are magnetic disks. They are used for storing system programs, large data files, and other backup information.

Magnetic Tapes: Magnetic Tapes are slow devices compare to magnetic disks, generally tapes are used for backup storage.

Optical Disks: In recent years, Optical disks(CD-ROMS) are become available. They have much higher recording density than magnetic disks. A CD is prepared by using a high power infrared laser to burn 0.8 micron diameter holes in a coated glass master disk. The burned area is called pits and the unburned areas between the "pits" are called "lands".

Cache memory: The Clock speed of the CPU is much faster than the Main Memory, so the CPU requires a fast memory, such a fast small memory is reffered to as a "Cache Memory". The Cache Memory is the intermediate memory between CPU and Main Memory. So it is placed between the CPU and Main Memory.

`



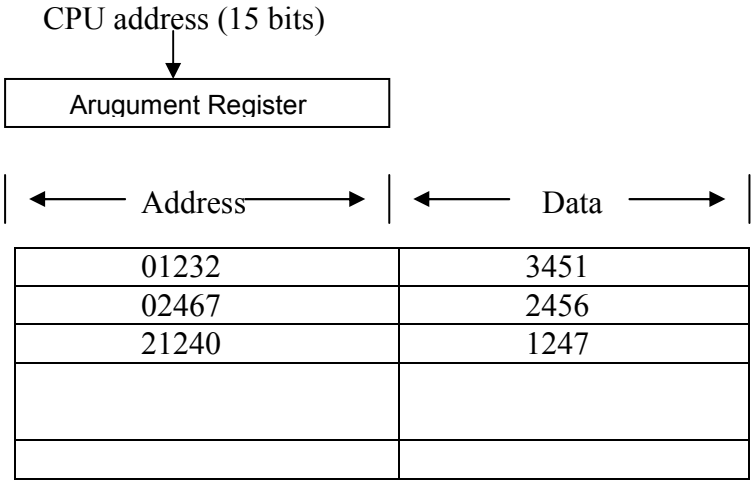
The Basic Operation of the Cache is as follows. When the CPU needs to access memory, the cache is examined. If the word is found in the cache, it is read from cache memory. If the word is not found in the Cache, the CPU read the word from main memory, and the same word copied into cache memory from main memory.

Mapping: The Basic characteristic of cache memory is its fast access time. So very little (or) no time must be wasted when searching for words in the cache. The transformation of data from main memory to cache memory is reffered to as a "Mapping" process. There are 3types of mapping procedures are there for cache memory. These are

1. Associative Mapping
2. Direct Mapping
3. Set-Associative Mapping

1. **Associative Mapping:** The Associative Memory stores both the Address and Content (data) of the memory word. This permits any location in cache to store any word from main memory. The Diagram shows three words presently stored in the cache.

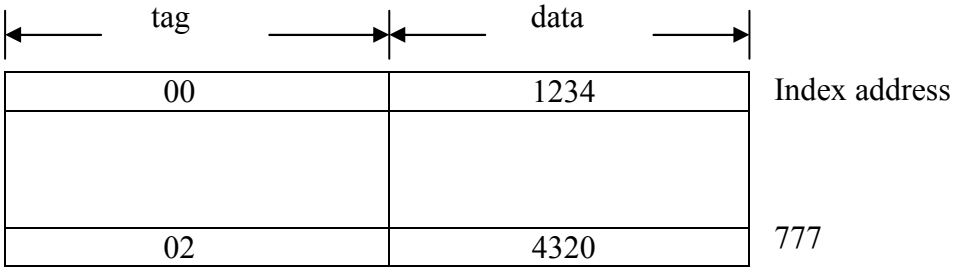
   The CPU address of 15 bits is placed in the argument register and the associative memory is searched for a matching address. If the address is found, the corresponding 12 bit data is read and sent to the CPU. If no match occurs, the main memory is accessed for the word. The address data pair is then transferred to the associative cache memory. If the cache is full, an address data pair must be displaced to make room for a pair that is needed and not presently in the cache.



Address	Data
01232	3451
02467	2456
21240	1247

Associative Mapping Cache (all numbers in octal)

2. **Direct Mapping:** In this Mapping procedure, the CPU address of 15 bits is divided into two fields. One is index field (9 bits) and second is tag field (6 bits). The number of bits in the index field is equal to the number of address bits required to access the cache memory.



tag	data	
00	1234	Index address
02	4320	777

Direct Mapping Cache

When the CPU generate a request, the index field is used for the address to access the cache. The tag field of the CPU address is compared with the tag. If the two tags are match, there is a "hit" otherwise a "miss". If it is a miss then the required word is read from main memory.

3. **Set-Associative Mapping:** A Third type of cache organization called "Set Associative Mapping", in this each word of cache can store two or more words of memory under the same index address. Each data word is stored together with its tag and the number of tag data items in one word of cache is said to form a set.

Consider the below figure for better understanding

Tag	Data		Tag	Data
01	2345		02	1234
01	4567		02	3450

Set-Associative Mapping

I/O communication Techniques: Three techniques are possible for I/O operations:

- **Programmed I/O**
- **Interrupt-driven I/O**
- **Direct memory access (DMA)**
- **Programmed I/O:** When the processor is executing a program and encounters an instruction relating to I/O, it executes that instruction by issuing a command to the appropriate I/O module. 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.
- **Interrupt-Driven I/O:** An alternative 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.

  At the end of each instruction cycle, the processor checks for interrupts. When the interrupt from the I/O module occurs, the processor saves the context of the program it is currently executing and begins to execute an interrupt-handling program that processes the interrupt. After processing interrupt, processor resume backs to normal execution.
- **Direct Memory Access:** Interrupt-driven I/O, though more efficient than simple programmed I/O, still requires the active intervention of the processor to transfer data between memory and an I/O module, and any data transfer must traverse a path through the

processor. When large volumes of data are to be moved, a more efficient technique is required: direct memory access (DMA).

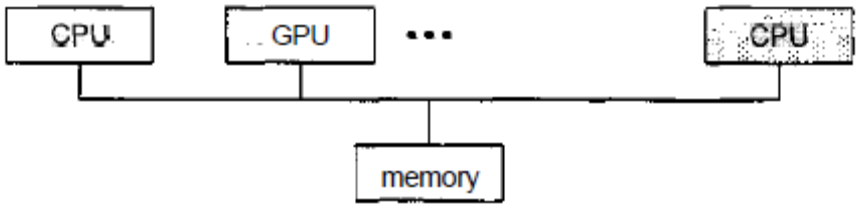
Computer-System Architecture: A computer system may be organized in a number of different ways, according to the number of general-purpose processors used.

- **Single-Processor Systems:** Most systems vise a single processor. On a single-processor system, there is one main CPU capable of executing a general-purpose instruction set, including instructions from user processes. Almost all systems have other special-purpose processors as such as disk, keyboard, and graphics controllers; or, on mainframes. All of these special-purpose processors run a limited instruction set and do not run user processes.
- **Multiprocessor Systems(parallel systems** or **tightly coupled systems):** Multiprocessor system have two or more processors in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices. Multiprocessor systems have three main advantages:

  - ✓ **Increased throughput.** By increasing the number of processors, more work will be done in less time.
  - ✓ **Economy of scale.** Multiprocessor systems can cost less than equivalent multiple single-processor systems, because they can share peripherals, mass storage, and power supplies.
  - ✓ **Increased reliability.** If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, only slow it down. If we have ten processors and one fails, then each of the remaining nine processors can pick up a share of the work of the failed processor. Thus, the entire system runs only 10 percent slower, rather than failing altogether.

  There are two different types of multi processor systems: **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.

  The most common systems use **symmetric multiprocessing (SMP),** in which each processor performs all tasks within the operating system. SMP means that all processors are peers; no master-slave relationship exists between processors.



Symmetric multiprocessing architecture.

- **Clustered Systems:** Another type of multiple-CPU system is the **clustered system.** Clustered computers share storage and are closely linked via a **local-area network (LAN)** or a faster interconnect such as InfiniBand. Clustered systems differ from multiprocessor systems, however, in that they are composed of two or more individual systems coupled together. Clustering is usually used to provide **high-availability** service; that is, service will continue even if one or more systems in the cluster fail. A layer of cluster software runs on the cluster nodes. Each node can monitor one or more of the others (over the LAN). If the monitored

machine fails, the monitoring machine can take ownership of its storage and restart the applications that were running on the failed machine.

Operating-System Structure: A program that acts as an intermediary between a user of a computer and the computer hardware

Operating system goals:

- Execute user programs and make solving user problems easier
- Make the computer system convenient to use
- Use the computer hardware in an efficient manner

One of the most important aspects of operating systems is the ability to multiprogram. A single user cannot, keep either the CPU or the I/O devices busy at all times. **Multiprogramming** increases CPU utilization.

The operating system keeps several jobs in memory simultaneously. This set of jobs can be a subset of the jobs kept in the job pool—which contains all jobs that enter the system—since the number of jobs that can be kept simultaneously in memory is usually smaller than the number of jobs that can be kept in the job pool. The operating system picks and begins to execute one of the jobs in memory. Eventually, the job may have to wait for some task, such as an I/O operation, to complete. In a system, the operating system simply switches to, and executes, another job.

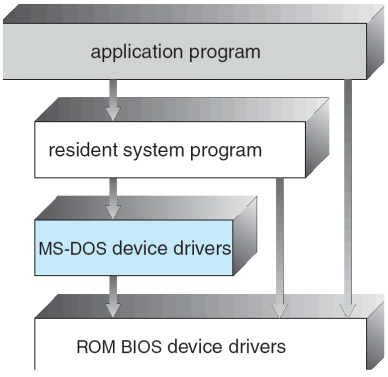
When *that* job needs to wait, the CPU is switched to *another* job, and so on. Eventually, the first job finishes waiting and gets the CPU back. As long as at least one job needs to execute, the CPU is never idle.

Multiprogrammed systems do not provide for user interaction with the computer system. **Time sharing** (or **multitasking)** is a logical extension of multiprogramming. In time-sharing systems, the CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running. Time sharing requires an **interactive** (or **hands-on) computer system,** which provides direct communication between the user and the system. The user gives instructions to the operating system or to a program directly, using a input device such as a keyboard or a mouse, and waits for immediate results on an output device. Accordingly, the **response time** should be short—typically less than one second.

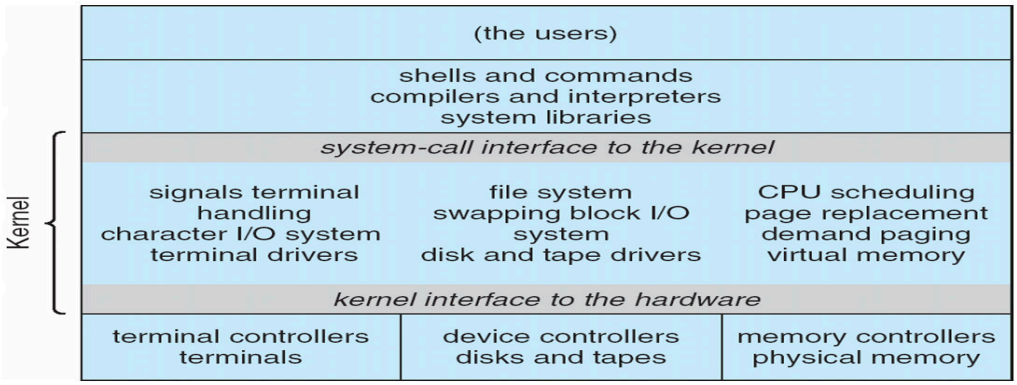
A time-shared operating system allows many users to share the computer simultaneously. Since each action or command in a time-shared system tends to be short, only a little CPU time is needed for each user. As the system switches rapidly from one user to the next, each user is given the impression that the entire computer system is dedicated to his use, even though it is being shared among many users.

Types of Operating System Structure: The following are different types of operating system structures:

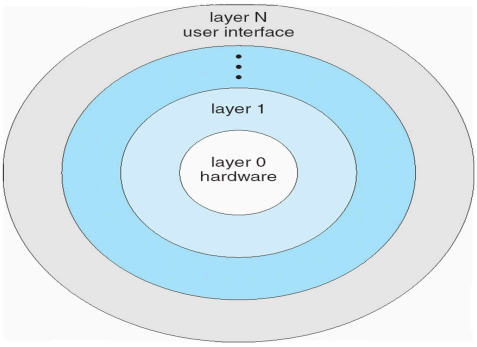
- **Simple Structure:** Many commercial systems do not have well-defined structures. Frequently, such operating systems started as small, simple, and limited systems and then grew beyond their original scope. MS-DOS is an example of such a system. It was written to provide the most functionality in the least space. In MS-DOS, the interfaces and levels of functionality are not well separated. For instance, 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 crashes when user programs fail.



Another example of limited structuring is the original UNIX operating system. UNIX is another system that initially was limited by hardware functionality. 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.



- **Layered Approach:** A system can be made modular in many ways. One method is the **layered approach,** in which the operating system is broken up into a number of layers (levels). The bottom layer (layer 0) is the hardware; the highest (layer N) 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.



The major difficulty with the layered approach involves appropriately defining the various layers.

A final problem with layered implementations is that they tend to be less efficient than other types.

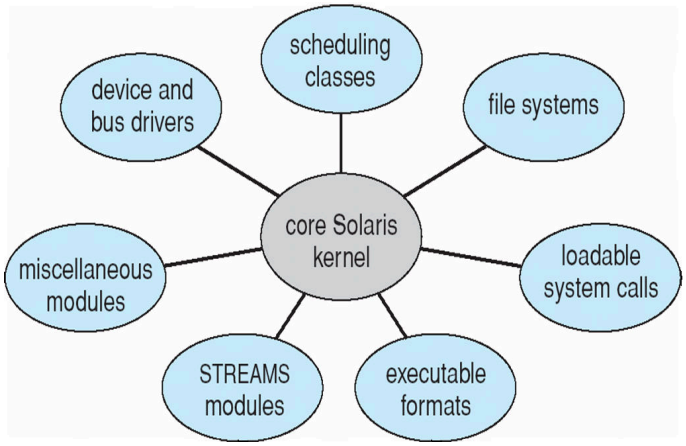
- **Microkernels:** This method structures the operating system by removing all nonessential components from the kernel and implementing them as system and user-level programs. The result is a smaller kernel.

The main function of the microkernel is to provide a communication facility between the client program and the various services that are also running in user space. The client program and service never interact directly. Rather, they communicate indirectly by exchanging messages
with the microkernel.

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. When the kernel does have to be modified, the changes tend to be fewer, because the microkernel is a smaller kernel.

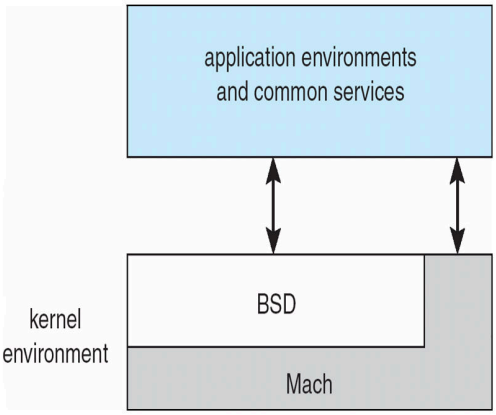
The microkernel also provides more security and reliability, since most services are running as user—rather than kernel—processes. If a service fails, the rest of the operating system remains untouched. Examples are Tru64 UNIX, QNX.

- **Modular Kernels:** the best 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 dynamically links in additional services either during boot time or during run time. Examples are Solaris, Mac OS X, Linux. For example, the Solaris operating system structure is shonw below is is organized around a core kernel with seven types of loadable kernel modules:



This structure is more flexible than a layered system in that any module can call any other module.

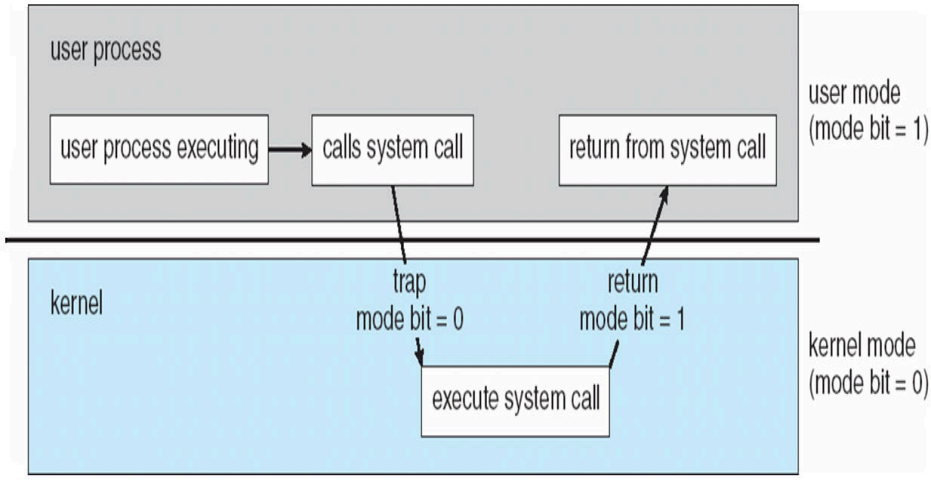
The Apple Macintosh Mac OS X operating system uses a hybrid structure. Mac OS X (also known as *Danvin)* structures the operating system using a layered technique where one layer consists of the Mach microkernel. The structure of Mac OS X appears as shown below.



The top layers include application environments and a set of services providing a graphical interface to applications. Below these layers is the kernel environment, which consists primarily of the Mach microkernel and the BSD kernel. Mach provides memory management; support for remote procedure calls (RPCs) and interprocess communication (IPC) facilities, including message passing; and thread scheduling. The BSD component provides a BSD command line interface, support for networking and file systems.

Operating System Operations: The following are different types of operations performed by each and every operating system.

- **Dual-Mode Operation:** Every operating system executes applications in two different modes: **user mode** and **kernel mode** (also called **supervisor mode, system mode, or privileged mode).** A bit, called the **mode bit,** is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1). When the computer system is executing on behalf of a user application, the system is in user mode. However, when a user application requests a service from the operating system (via a system call), it must transition from user to kernel mode to fulfill the request, which is shown in the following diagram.



At system boot time, the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode. Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the state of the mode bit to 0). Thus, whenever the operating system gains control of the computer, it is in kernel mode. The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program.

The hardware allows privileged instructions to be executed 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 it as illegal and traps it to the operating system.

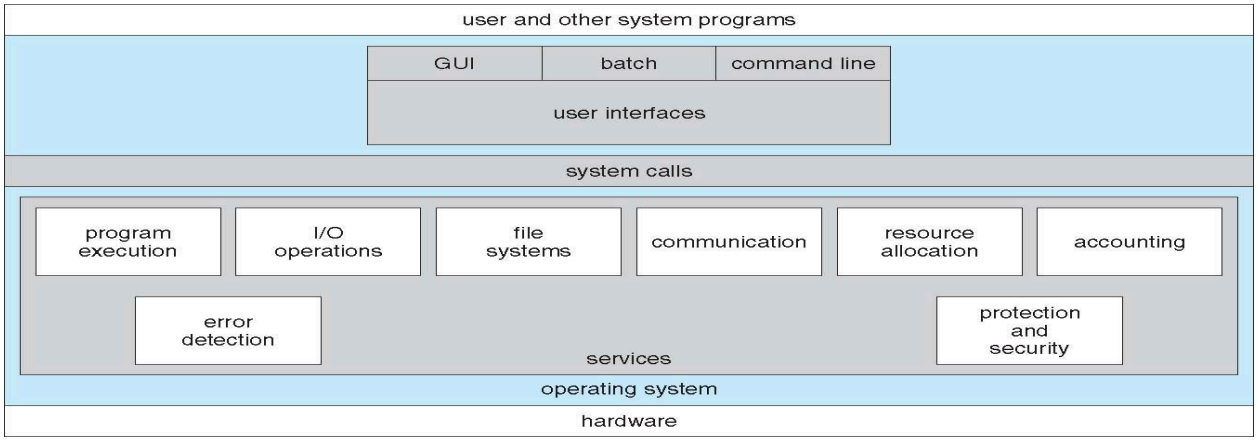
- **Timer:** A timer can be set to interrupt the computer after a specified period. The period may be fixed (for example, 1/60 second) or variable (for example, from 1 millisecond to 1 second). Times prevents a user program from getting stuck in an infinite loop or not calling system services and never returning control to the operating system.

Operating System Services: The following are different types of services provided by Operating System.

- **User interface:** Almost all operating systems have a **user interface** (UI). This interface can take several forms. One is a **command-line interface (CLI),** which uses text commands and a method for entering them. Another is a **batch interface,** in which commands and directives to control those commands are entered into files, and those files are executed. Most commonly a **graphical user interface (GUI)** is used. Here, the interface is a window system with a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text.
- **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).
- **I/O operations:** A running program may require I/O device. For specific devices, special functions may be desired (such as recording to a CD or DVD drive). For efficiency and

protection, users usually cannot control I/O devices directly. Therefore, the operating system must provide a means to do I/O.

- **File-system manipulation:** Operating System is used to control operations on files like creating, opening, reading, and writing.
- **Communications:** There are many cases in which one process needs to exchange information with another process. Such communication may occur between processes that are executing on the same computer or between processes that are executing on different computer systems tied together by a computer network. Communications may be implemented via *shared memory* or through *message passing,* in which packets of information are moved between processes by the operating system.
- **Error detection.** The operating system needs to be constantly aware of possible errors. Errors may occur in the CPU and memor, in I/O devices, and in the user program. For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing.
- **Resource allocation.** When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them.
- **Accounting:** Accounting keep track of which users use how much and what kinds of computer resources. This record keeping may be used for accounting (so that users can be billed) or simply for accumulating usage statistics.
- **Protection and security.** The owners of information stored in a multiuser or networked computer system may want to control use of that information. When several separate processes execute concurrently, it should not be possible for one process to interfere with the others or with the operating system itself. Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders is also important. Such security starts with requiring each user to authenticate himself or herself to the system, usually by means of a password, to gain access to system resources.



A View of Operating System Services

Distributed Systems: A distributed system is a collection of physically separate, possibly heterogeneous computer systems that are networked to provide the users with access to the various resources that the system maintains. Access to a shared resource increases computation speed, functionality, data availability, and reliability. A **network,** in the simplest terms, is a communication path between two or more systems. Distributed systems depend on networking for their functionality. Networks are characterized based on the distances between their nodes. A **local-area**

network (LAN) connects computers within a room, a floor, or a building. A **wide-area network (WAN)** usually links buildings, cities, or countries.

Distributed operating systems depend on networking for their operation. Distributed OS runs on and controls the resources of multiple machines. It provides resource sharing across the boundaries of a single computer system.

Advantages:

- Resource sharing
- High availability
- High throughput
- High performance
- Incremental growth

Special purpose systems:

- **Real time embedded systems:** Real Time embedded systems are specially designed for serving autonomous computers like satellites, robotics, hydroelectric dams etc…A real time system uses priority scheduling algorithm to meet the responsive requirement of a real time application.

  Memory management in real time system is comparatively less demanding than in other types of multiprogramming systems.
- **Multimedia systems:** Most operating systems are designed to handle conventional data such as text files, programs, word-processing documents, and spreadsheets. Multimedia data consist of audio and video files as well as conventional files.

  Multimedia describes a wide range of applications that are in popular use today. These include audio files such as MP3 DVD movies, video conferencing, and short video clips of movie previews or news stories downloaded over the Internet.
- **Handheld systems: Handheld systems** include personal digital assistants (PDAs), such as Palm and Pocket-PCs, and cellular telephones, many of which use special-purpose embedded operating systems. Developers of handheld systems and applications face many challenges, most of which are due to the limited size of such devices. For example, a PDA is typically about 5 inches in height and 3 inches in width, and it weighs less than one-half pound. Because of their size, most handheld devices have a small amount of memory, slow processors, and small display screens.

  A second issue of concern to developers of handheld devices is the speed of the processor used in the devices. Processors for most handheld devices run at a fraction of the speed of a processor in a PC. Most handheld devices use smaller, slower processors that consume less power.

Protection and Security:

Protection – any mechanism for controlling access of processes or users to resources defined by the OS.
Security – Defense of the system against internal and external attacks.

Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. A system can have adequate protection but still be rone to failure and allow inappropriate access.

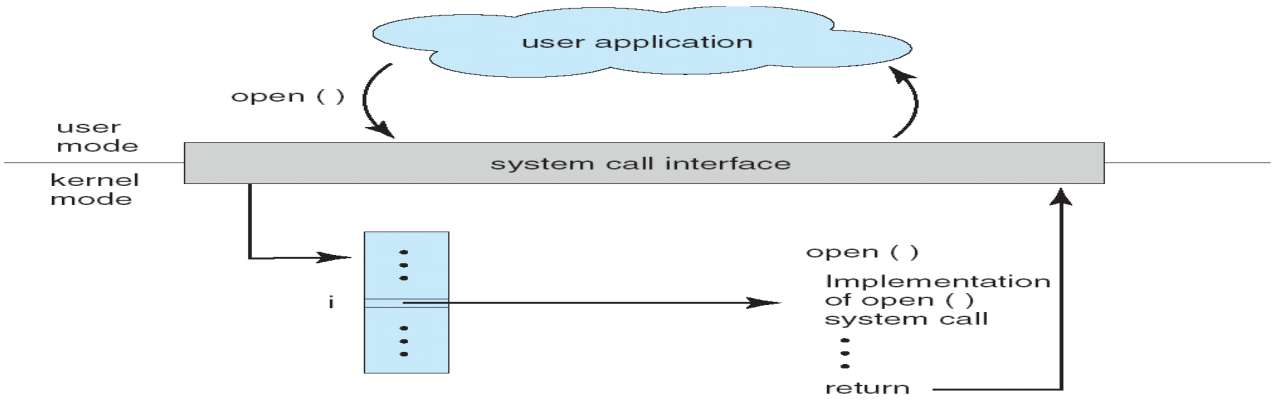
For protection and security, systems generally first distinguish among users, to determine who can do what. Most operating systems maintain a list of user names and associated **user identifiers (user IDs).** User ID then associated with all files processes of that user to determine

access control. Group identifier allows set of users to be defined and controls managed, then also associated with each process, file.

Previliged escalation allows user ot change to effective ID with more rights. The user may need access to a device that is restricted. Various operating systems provide different methods to allow privilege escalation.

System Calls: A System call or Kernel call is a software interrupt request in UNIX like operating systems made via active process for services provided by kernel. **System calls** provide an interface to the services made available by an operating
System.

The run-time support system for most programming languages provides a **system-call interface** that serves as the link to system calls made available by the operating system. The system-call interface intercepts function calls in the API and invokes the necessary system call within the operating system. Typically, a number is associated with each system call, and the system-call interface maintains a table indexed according to these numbers. The system call interface then invokes the intended system call in the operating system kernel and returns the status of the system call and any return values. The following figure represents handling of open() system call.



System calls can be grouped roughly into six major categories:

- **Process control: fork(), wait(), exit()**
- **File manipulation: open(), read(), write(), close()**
- **Device manipulation: iocntl(), read(), write()**
- **Information maintenance: alarm(), sleep()**
- **Communications: pipe(), shmget(), mmap()**
- **Protection and Security: chmod(), umask(), chown()**

Operating System Generation:

- Operating systems are designed to run on any of a class of machines; the system must be configured for each specific computer site
- SYSGEN program obtains information concerning the specific configuration of the hardware system *Booting* – starting a computer by loading the kernel
- *Bootstrap program* – code stored in ROM that is able to locate the kernel, load it into memory, and start its execution

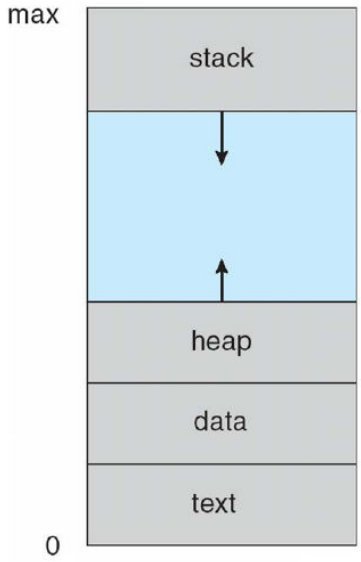
System Boot

- Operating system must be made available to hardware so hardware can start it

- Small piece of code – **bootstrap loader**, locates the kernel, loads it into memory, and starts it
- Sometimes two-step process where **boot block** at fixed location loads bootstrap loader
- When power initialized on system, execution starts at a fixed memory location

Syllubus: Process Management – Process concept- process scheduling, operations, Inter process communication. Multi Thread programming models. Process scheduling criteria and algorithms, and their evaluation.

Process Concept: A process is an instance of a program that is running in a computer. 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. The structure of a process in memory is shown in the following figure.

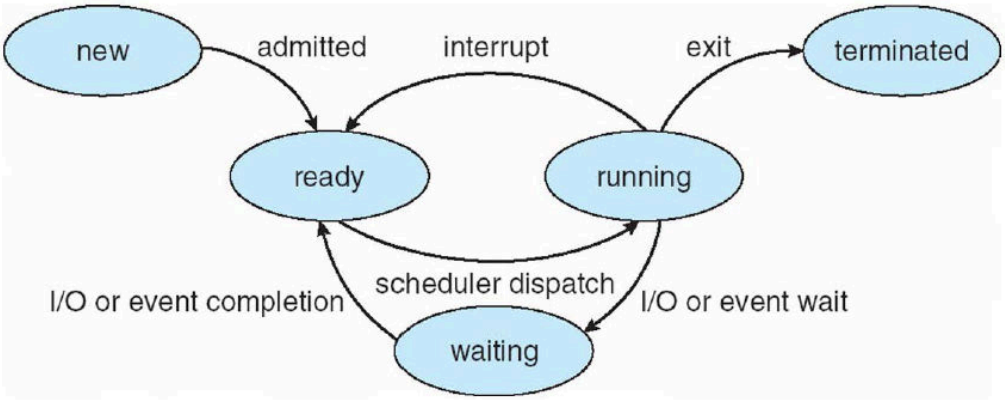


A program is a passive entity, such as a file containing a list of instructions stored on disk (often called an **executable file),** whereas a process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory. Two common techniques for loading executable files are double-clicking an icon representing the executable file and entering the name of the executable file on the command line (as in prog. exe or a. out.)

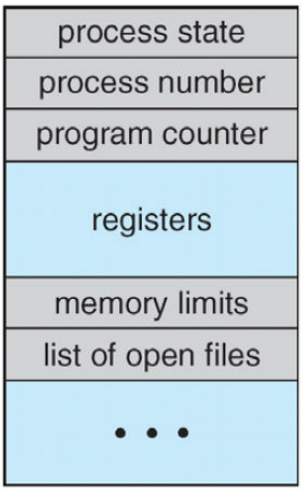
Process States (or) Life Cycle of a Process: During execution of a process changes its state. The state of a process is defined in part by the current activity of that process. Each process may be in one of the following states:

- **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.

The state diagram corresponding to these states is as shown below.



Process Control Block: Each process is represented in the operating system by a **process control block (PCB)**—also called a task control block. A PCB is shown in the following figure.



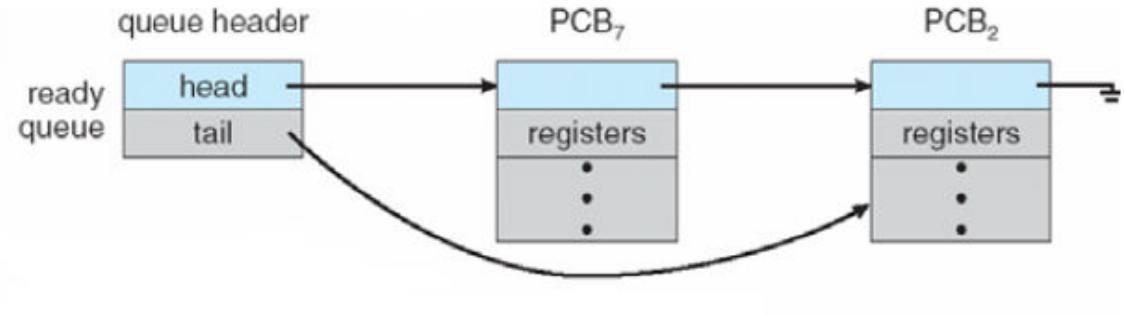
It contains many pieces of information associated with a specific process, including these:

- **Process state.** The state may be new, ready, running, waiting, halted, and so on.
- **Program counter:** The counter indicates the address of the next instruction to be executed for this process.
- CPU **registers:** This block indicates type of registers used by the process. The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information.
- **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 mimbers, job or process numbers, and so on.
- **I/O status information:** This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

Process Scheduling: The **process scheduler** selects an available process (possibly from a set of several available processes) for program execution on the CPU. For this processor maintains the following three queues:

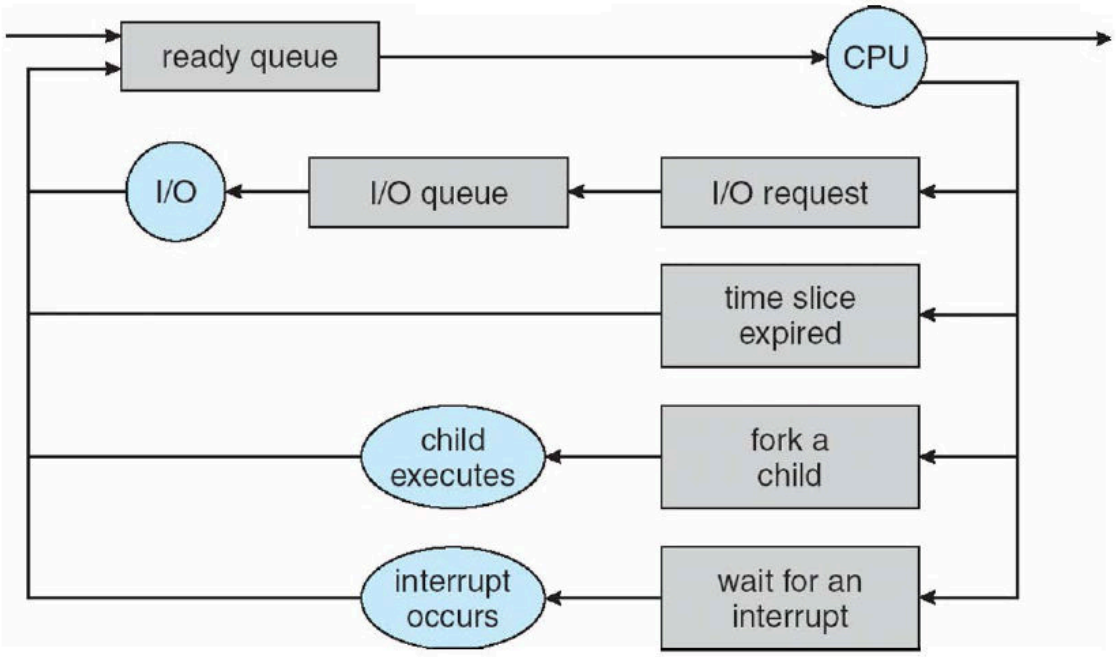
- **Job Queue** - which consists of all processes in the system.
- **Ready Queue**:The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue.** This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.
- **Device Queue**: The list of processes waiting for a particular I/O device is called a device **queue.**

Each queue is maintained in the form of linked list as shown below.



A common representation for a discussion of process scheduling is a **queueing diagram** which is shown below. Two types of queues are present: the ready queue and a set of device queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system. A new process is initially put in the ready queue. It waits there tmtil it is selected for execution, or is **dispatched.** Once the process is allocated the CPU and is executing, one of several events 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 the subprocess's termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.



Schedulers: A process migrates among the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some fashion.

The selection process is carried out by the appropriate scheduler. There are three different types of schedules:

The **long-term scheduler,** or **job scheduler,** selects processes from this pool and loads them into memory for execution. The long-term scheduler executes much less frequently.

The long-term scheduler controls the **degree of multiprogramming** (the number of processes in memory). If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system. Thus, the long-term scheduler may need to be invoked only when a process leaves the system.

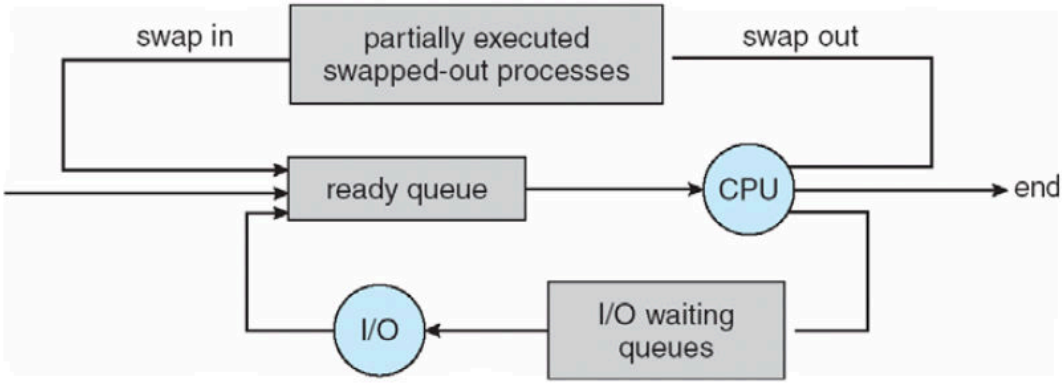
In general, most processes can be described as either I/O bound or CPU bound. A process which spends more amount of time with I/O devices is called as I/O bound process. A process which spends more amount of time with CPU is called as CPU bound process.

If all processes are I/O bound, the ready queue will almost always be empty. If all processes are CPU bound, the I/O waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced. The system with the best performance will have a combination of CPU-bound and I/O-bound processes.

The long-term scheduler is also used to control selection of good **Process Mix(Combination of I/O and CPU bound Process)**.

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.

The key idea behind a **medium-term scheduler** is that sometimes it can be advantageous to remove processes from memory and thus reduce the degree of multiprogramming. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called swapping. The process is swapped out, and is later swapped in, by the medium-term scheduler. Swapping may be necessary to improve the process mix. The following diagram represents swapping process.



Context Switch: Switching the CPU to another process requires performing a stata save of the current process and a state restore of a different process. This task is known as a **context switch.** When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.

Operations on Processes: The processes in most systems can execute concurrently, and they may be created and deleted dynamically. The following are different types of operations that can be performed on processes.

Process Creation: A process may create several new processes using fork() system call. The creating process is called a **parent** process, and the new processes are called the **children** of that process. Each of these new processes may in turn create other processes, forming a **tree** of processes.

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

- The parent continues to execute concurrently with its children.
- 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:

- The child process is a duplicate of the parent process (it has the same program and data as the parent).
- The child process has a new program loaded into it.

The following C program illustrates creation of a new process using fork() system call.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
        pid-t pid;      /* fork a child process */
        pid = fork();
        if (pid < 0)
        {/* error occurred */
                fprintf(stderr, "Fork Failed");
                exit (-1) ;
        }
        else if (pid == 0}
        {/* child process */
        execlpf"/bin/Is","Is",NULL);
        }
        else
        {       /* parent process. parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
        exit (0) ;
        }
}
```

A new process is created by the **fork()** system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process. Both processes (the parent and the child) continue execution at the instruction after the f ork(), with one difference: The return code for the forkO is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

Typically, the execO system call is used after a fork() system call by one of the two processes to replace the process's memory space with a new program. The exec () system call loads a binary file into memory and starts its execution.

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 a status value (typically an integer) to its parent process (via the wait() system call). All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system.

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

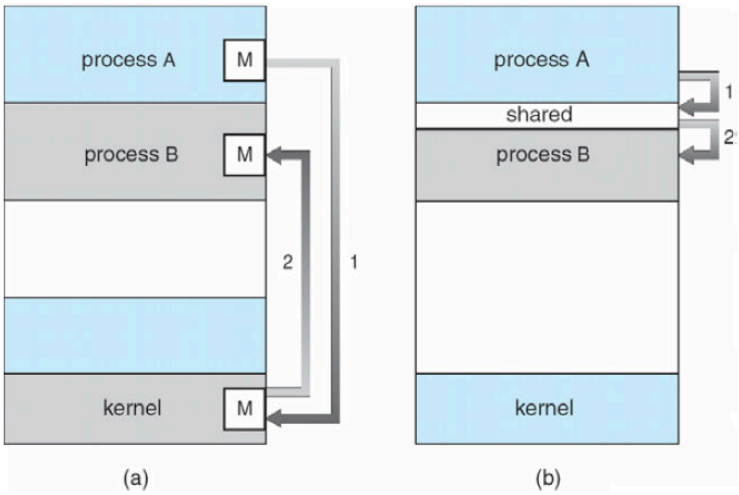
- The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

Interprocess Communication: 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, sharing provides an environment to allow concurrent access to such information.
- **Computation speedup:** To execute a particular task faster, break it into subtasks, each of which will be executing in parallel with the others.
- **Modularity:** To construct the system in a modular fashion, divide 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, printing, and compiling in parallel.

Cooperating processes require an **interprocess communication (IPC)** mechanism that will allow them to exchange data and information. There are two fundamental models of interprocess communication: **(1) shared memory** and (2) **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 messagepassing model, communication takes place by means of messages exchanged between the cooperating processes. The two communications models are contrasted in the following figure.



Communications models, (a) Message passing, (b) Shared memory.

Shared-Memory Systems: Interprocess communication using shared memory requires communicating processes to establish a region of shared memory. Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment. Other processes that wish to communicate using this shared-memory segment must attach it to their address space.

To illustrate the concept of cooperating processes, let's consider the producer-consumer problem, which is a common paradigm for cooperating processes. A **producer** process produces information that is consumed by a **consumer** process.

One solution to the producer-consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, must have available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the

consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

Two types of buffers can be used. The **unbounded buffer** places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items. The **bounded buffer** assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

The following variables reside in a region of memory shared by the producer and consumer processes:

```
#define BUFFER_SIZE 10
typedef struct
{
…..
}item;
item buffer [BUFFER_SIZE] ;
int in = 0 ;
int out = 0 ;
```

The shared buffer is implemented as a circular array with two logical pointers: in and out. The variable in points to the next free position in the buffer; out points to the first full position in the buffer. The buffer is empty when in == out; the buffer is full when ((in + 1) % BUFFER_SIZE) == out.

The code for the producer and consumer processes is shown below.

The Producer process

```
item nextProduced;
while (true)
{
        while (((in + 1) % BUFFER-SIZE) == out)
          ; /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
}
```

The Consumer process

```
item nextConsumed;
while (true)
{
        while (in == out)
        ; //do nothing
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFEFLSIZE;
}
```

The producer process has a local variable nextProduced in which the new item to be produced is stored. The consumer process has a local variable nextConsumed in which the item to be consumed is stored.

Message Passing Systems: Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. A message-passing facility provides at least two operations: send(message) and receive(message). Messages sent by a process can be of either fixed or variable size.

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. This link can be implemented in a variety of ways.

- **Direct or indirect communication**
- **Synchronous or asynchronous communication**

- **Automatic or explicit buffering**
- **Direct or indirect communication**: Under **Symmetry direct communication**, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the send.0 and receive() primitives are defined as:
  - ❖ send(P, message)—Send a message to process P.
  - ❖ receive (Q, message)—Receive a message from process Q.

  A communication link in this scheme has the following properties:
  - ➔ A link is established automatically between every pair of processes that want to communicate.
  - ➔ A link is associated with exactly two processes.
  - ➔ Between each pair of processes, there exists exactly one link.

  In **asymmetry direct communication** only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the send() and receive () primitives are defined as follows:
  - ❖ send(P, message)—Send a message to process P.
  - ❖ receive(id, message)—-Receive a message from any process; the variable id is set to the name of the process with which communication has taken place.

  With **indirect communication**, the messages are sent to and received from mailboxes, or ports. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification. In this scheme, a process can communicate with some other process via a number of different mailboxes. Two processes can communicate only if the processes have a shared mailbox, however. The sendC) 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.

  In this scheme, a communication link has the following properties:
  - ➔ A link is established between a pair of processes only if both members of the pair have a shared mailbox.
  - ➔ A link may be associated with more than two processes.
  - ➔ Between each pair of communicating processes, there may be a number of different links, with each link corresponding to one mailbox.

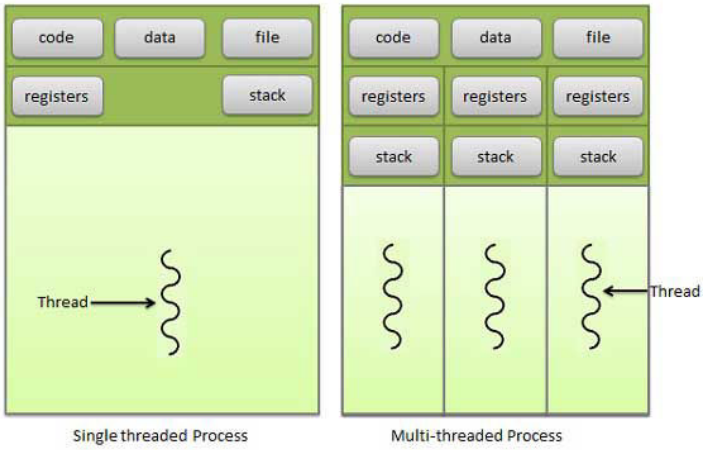
  A mailbox may be owned either by a process or by the operating system. If the mailbox is owned by a process there can be no confusion about who should receive a message sent to this mailbox(). When a process that owns a mailbox terminates, the mailbox disappears.

  In contrast, a mailbox that is owned by the operating system has an existence of its own. The operating system then must provide a mechanism that allows a process to do the following:
  - ❖ Create a new mailbox.
  - ❖ Send and receive messages through the mailbox.
  - ❖ Delete a mailbox.
- **Synchronous or asynchronous communication:** Communication between processes takes place through calls to sendO and receive () primitives. There are different design options for implementing each primitive. Message passing may be either **blocking** or **nonblocking**— also known as **synchronous** and **asynchronous.**
  - ❖ **Blocking send.** The sending process is blocked until the message is received by the receiving process or by the mailbox.
  - ❖ **Nonblocking send.** The sending process sends the message and resumes operation.
  - ❖ **Blocking receive.** The receiver blocks until a message is available.
  - ❖ **Nonblocking receive.** The receiver retrieves either a valid message or a null.

- **Buffering:** Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:
  - ❖ **Zero capacity:** The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
  - ❖ **Bounded capacity:** The queue has finite length n; thus, at most n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue, and the sender can continue execution without waiting. If the link is full, the sender must block until space is available in the queue.
  - ❖ **Unbounded capacity:** The queues length is infinite; thus, any number of messages can wait in it. The sender never blocks.

    The zero-capacity case is sometimes referred to as a message system with no buffering; the other cases are referred to as systems with automatic buffering.

Thread: A thread is a flow of execution through the process code, with its own program counter, system registers and stack. A thread is also called a light weight process. Threads provide a way to improve application performance through parallelism. Folowing figure shows the working of the single and multithreaded processes.

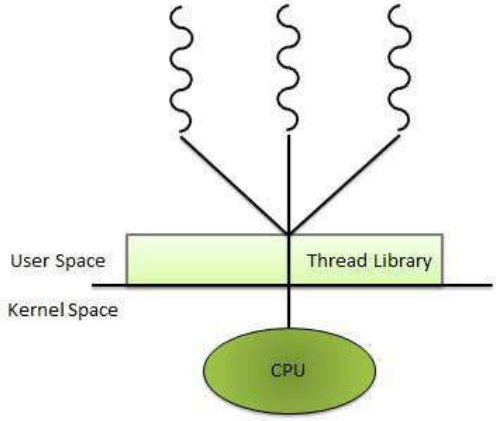


Difference between Process and Thread:

Process	Thread
Process is heavy weight.	Thread is light weight taking lesser resources than a process.
Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
If one process is blocked then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, second thread in the same task can run.
In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Types of Threads: Threads are implemented in following two ways:

- **User Level Threads:** User level threads are managed by a user level library. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. User level threads are typically fast. Creating threads, switching between threads and synchronizing threads only needs a procedure call.



Advantages:

- Thread switching does not require Kernel mode privileges.
- User level thread can run on any operating system.
- Scheduling can be application specific in the user level thread.
- User level threads are fast to create and manage.

Disadvantages:

- In a typical operating system, most system calls are blocking.
- Multithreaded application cannot take advantage of multiprocessing.

- **Kernel Level Threads:** In this case, thread management done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. They are slower than user level threads due to the management overhead.

  Advantages:

  - Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
  - If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
  - Kernel routines themselves can multithreaded.

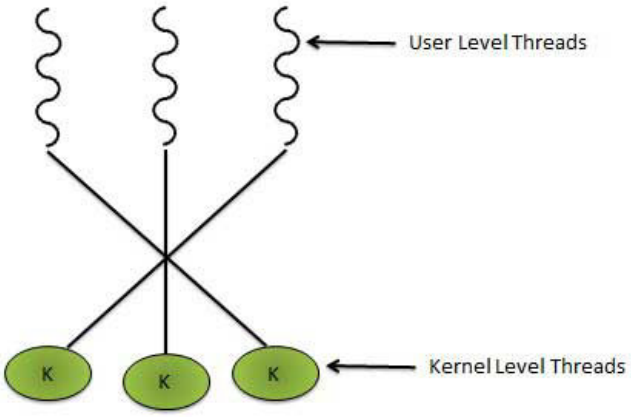
  Disadvantages:

  - Kernel threads are generally slower to create and manage than the user threads.
  - Transfer of control from one thread to another within same process requires a mode switch to the Kernel.

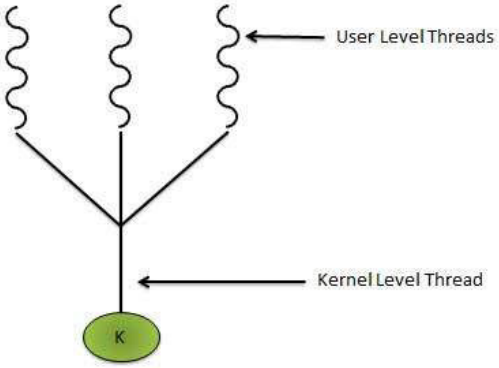
Multi Thread programming Models: Some operating system provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process. Multithreading models are three types

- Many to Many Model
- Many to One Model
- One to One Model

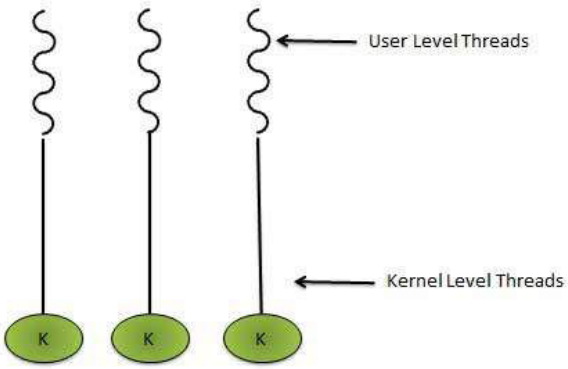
Many to Many Model: In this model, many user level threads multiplexes to the Kernel thread of smaller or equal numbers. The number of Kernel threads may be specific to either a particular application or a particular machine. Following diagram shows the many to many model. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallels on a multiprocessor.



Many to One Model: 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 blocked. 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.



One to One Model: 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 support multiple thread to execute in parallel on microprocessors. Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, windows NT and windows 2000 use one to one relationship model.



Process Scheduling Criteria: Many criteria have been suggested for comparing CPU scheduling algorithms. The criteria include the following:

- **CPU Utilization:** In general, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).
- **Throughput:** One measure of work is the number of processes that are completed per time unit, called throughput. For long processes, this rate may be one process per hour; for short transactions, it may be 10 processes per second.
- **Turnaround time:** The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
- **Waiting time:** Waiting time is the sum of the periods spent waiting in the ready queue.
- **Response time:** Response time, is the time it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device.

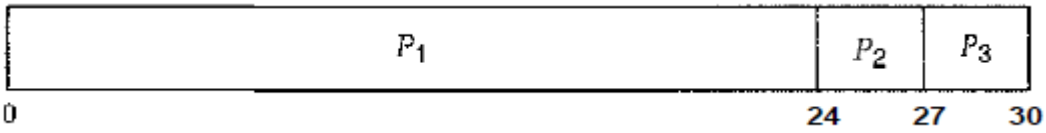
Process Scheduling Algorithms: CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. There are many different CPU scheduling algorithms as shown below.

1. **First Come First Serve (FCFS) Scheduling:** In this scheduling, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy 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.

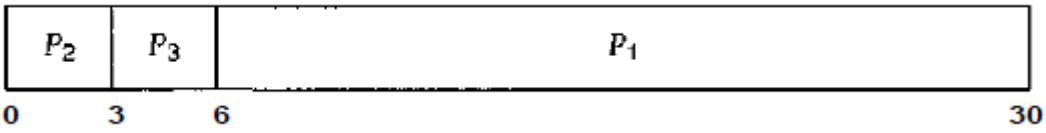
   Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

If the processes arrive in the order Pi, *Po, P3,* and are served in FCFS order, the result will be as shown in the following **Gantt chart:**



The waiting time is 0 milliseconds for process Pi, 24 milliseconds for process *Pn,* and 27 milliseconds for process *Pj.* Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds. If the processes arrive in the order *P2,* P3, P1, the results will be as shown in the following Gantt chart:



The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds. This reduction is substantial. Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the process's CPU burst times vary greatly.

The FCFS scheduling algorithm is nonpreemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O.

Advantages: Suitable for batch system. It is simple to understand and code

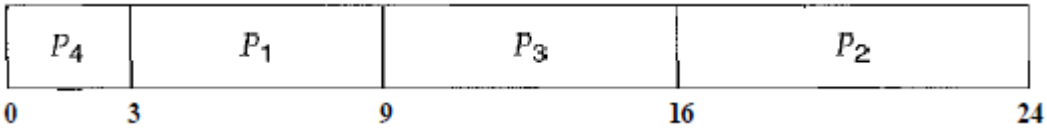
Disadvantages:

- Waiting time can be large if short requests wait behind the long ones.
- It is not suitable for time sharing systems where it is important that each user should get the CPU for an equal amount of time interval.
- A proper mix of jobs is needed to achieve good results from FCFS scheduling.

2. **Shortest-Job-First (SJF) Scheduling(***shortest-next-CPU-burst algorithm***):** A different approach to CPU scheduling is the **shortest-job-first (SJF) scheduling algorithm.** This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie. Consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

Using SJF scheduling, the following Gantt chart occurred.

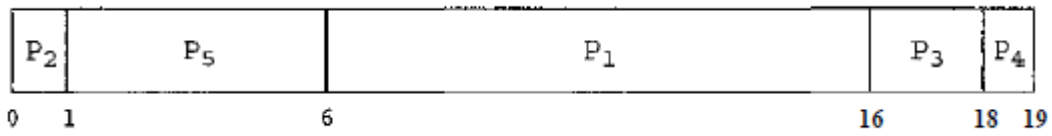


The waiting time is 3 milliseconds for process *P1,* 16 milliseconds for process *P2,* 9 milliseconds for process *P3,* and 0 milliseconds for process P4. Thus, the average waiting time is (3 + 16 + 9 + 0)/4 - *7* milliseconds.

3. **Priority Scheduling:** The SJF algorithm is a special case of the general **priority scheduling algorithm.** A priority 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. An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority. consider the following set of processes, assumed to have arrived at time 0, in the order Pi, P2, • • -, *P5,* with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Using priority scheduling, the following Gantt chart will be occurred. The average waiting time is 8.2 milliseconds.



Priority scheduling can be either preemptive or nonpreemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

A major problem with priority scheduling algorithms is **indefinite blocking,** or **starvation.** A process that is ready to run but waiting for the CPU can be considered blocked. A solution to the problem of indefinite blockage of low-priority processes is **aging.** Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time. For example, if priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes.

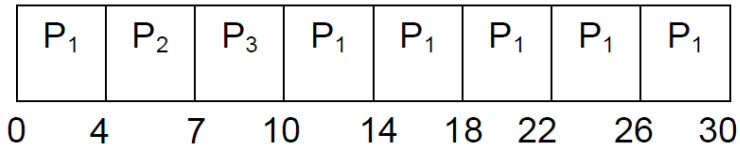
4. **Round Robin(RR) Scheduling:** The **round-robin (RR) scheduling algorithm** is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added to switch between processes. A small unit of time, called a **time quantum** or time slice, is defined. A time quantum is generally from 10 to 100 milliseconds. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

   To implement RR scheduling, keep the ready queue 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.

   The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

   If we use a time quantum of 4 milliseconds, then process Pi gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process *P2.* Since process Pi does not need 4 milliseconds, it quits before its time quantum expires. The CPU is then given to the next process, process P3. Once each process has received 1 time quantum, the CPU is returned to process Pi for an additional time quantum. The resulting RR schedule is as shonw below.

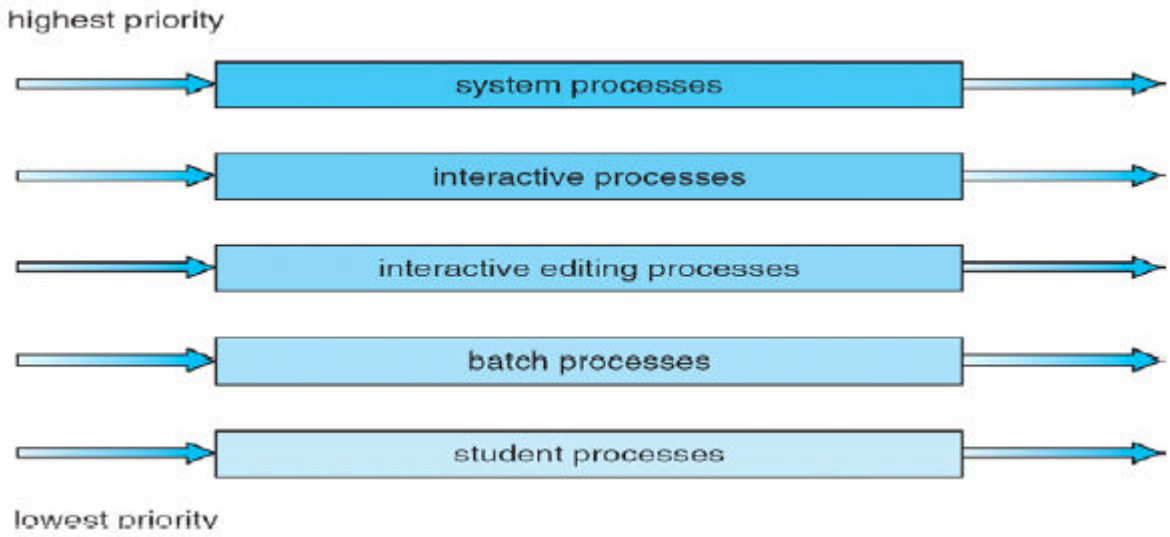


   The average waiting time is 17/3 = 5.66 milliseconds. In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process). If a process's CPU burst exceeds 1 time quantum, that process is *preempted* and is put back in the ready queue. The RR scheduling algorithm is thus preemptive.

5. **Multilevel Queue Scheduling:** A **multilevel queue scheduling algorithm** partitions the ready queue into several separate queues as shown below. 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. Each queue has its own scheduling algorithm. In

addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. For example, the foreground queue may have absolute priority over the background queue. Let's look at an example of a multilevel queue scheduling algorithm with five queues, listed below in order of priority:

- System processes
- Interactive processes
- Interactive editing processes
- Batch processes
- Student processes

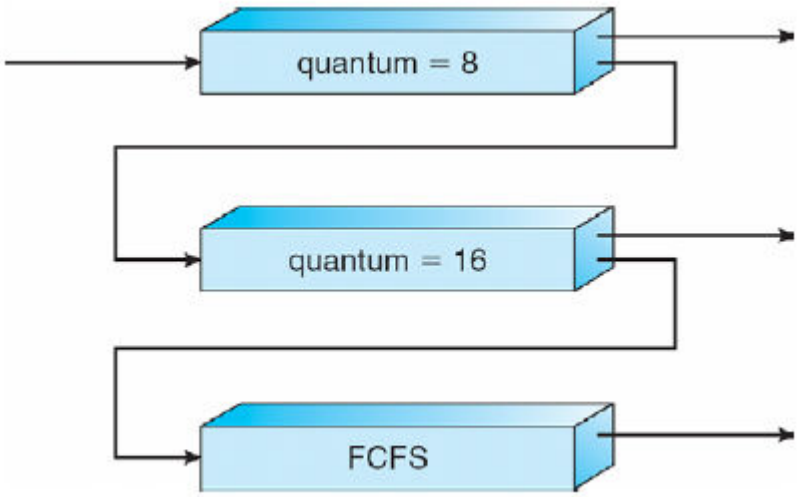


Each queue has absolute priority over lower-priority queues. No process in the batch queue, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.

Another possibility is to time-slice among the queues. Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes. For instance, in the foreground-background queue example, the foreground queue can be given 80 percent of the CPU time for RR scheduling among its processes, whereas the background queue receives 20 percent of the CPU to give to its processes on an FCFS basis.

6. **Multilevel Feedback-Queue Scheduling:** The **multilevel feedback-queue scheduling algorithm,** allows a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower-priority queue. In addition, 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.

   For example, consider a multilevel feedback-queue scheduler with three queues, numbered from 0 to 2 as shonw below. 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 only be executed if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2. A process in queue 1 will in turn be preempted by a process arriving for queue 0.



A process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2. Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty.

This scheduling algorithm gives highest priority to any process with a CPU burst of 8 milliseconds or less. Such a process will quickly get the CPU, finish its CPU burst, and go off to its next I/O burst. Processes that need more than 8 but less than 24 milliseconds are also served quickly, although with lower priority than shorter processes. Long processes automatically sink to queue 2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.

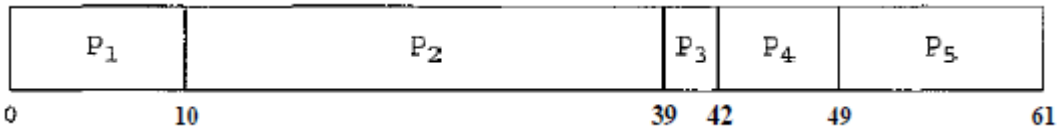
Process Scheduling Algorithms Evalution:

- **Deterministic Modeling:** One major class of evaluation methods is **analytic evaluation.** Analytic evaluation uses the given algorithm and the system workload to produce a formula or number that evaluates the performance of the algorithm for that workload.

  One type of analytic evaluation is **deterministic modeling.** This method takes a particular predetermined workload and defines the performance of each algorithm for that workload. For example, assume that we have the workload shown below. All five processes arrive at time 0, in the order given, with the length of the CPU burst given in milliseconds:

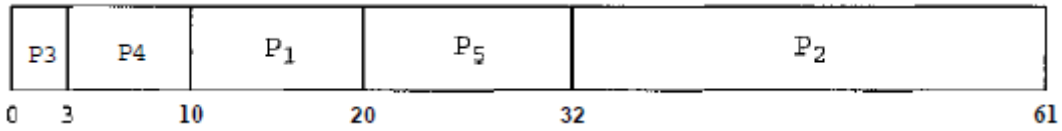
Process	Burst Time
P_1	10
P_2	29
P_3	3
Pi	7
P_5	12

Consider the FCFS, SJF, and RR (quantum = 10 milliseconds) scheduling algorithms for this set of processes. For the FCFS algorithm, Gantt chart looks like shown below.



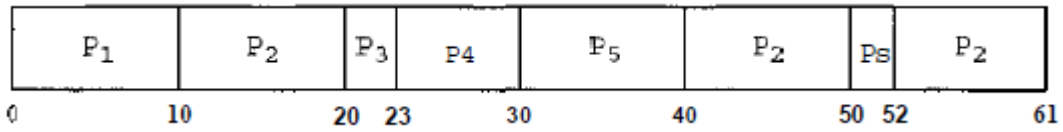
The waiting time is 0 milliseconds for process P1, 10 milliseconds for process P2, 39 milliseconds for process P3, 42 milliseconds for process P4, and 49 milliseconds for process P5. Thus, the average waiting time is (0 + 10 + 39 + 42 + *49)/5* = 28 milliseconds.

With SJF scheduling,



The waiting time is 10 milliseconds for process *P1,* 32 milliseconds for process *P2,* 0 milliseconds for process P3, 3 milliseconds for process P4, and 20 milliseconds for process P5. Thus, the average waiting time is (10 + 32 + 0 + 3 + 20)/5 = 13 milliseconds.

With the RR algorithm,



The waiting time is 0 milliseconds for process Pi, 32 milliseconds for process *P2,* 20 milliseconds for process P3, 23 milliseconds for process P4, and 40 milliseconds for process P5. Thus, the average waiting time is (0 + 32 + 20 + 23 + 40)/5 = 23 milliseconds.

From the above cases(FCFS, SJF, RR), the average waiting time obtained with the SJF policy is less than half that obtained with FCFS scheduling; the RR algorithm gives us an intermediate value.

- **Queueing models:** The computer system is described as a network of servers. Each server has a queue of waiting processes. The CPU is a server with its ready queue, as the I/O system with its device queues. By knowing arrival rates and service rates, we can compute utilization, average queue length, average wait time, and so on. This area of study is called **queueing-network analysis.**

  Let *n* be the average queue length, let W be the average waiting time in the queue, and let X be the average arrival rate for new processes in the queue. If the system is in a steady state, then the number of processes leaving the queue must be equal to the number of processes that arrive. Thus,

$$n = \lambda \times W.$$

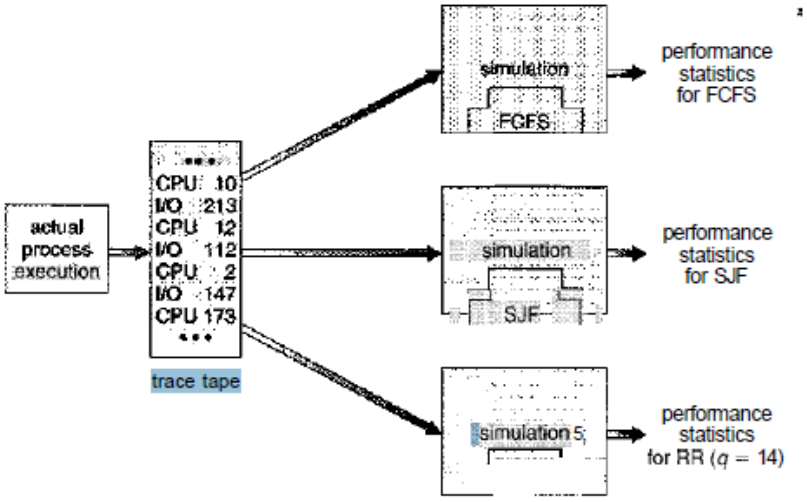
  This equation, known as **Little's formula.**

- **Simulation:** To get a more accurate evaluation of scheduling algorithms, use **simulations.** The simulator has a variable representing a clock; as this variable's value is increased, the simulator modifies the system state to reflect the activities of the devices, the processes, and the scheduler. As the simulation executes, statistics that indicate algorithm performance are gathered and printed.

  The data to drive the simulation can be generated in several ways. The most common method uses a random-number generator, which is programmed to generate processes, CPU burst times, arrivals, departures, and so on, according to probability distributions.

  The distributions can be defined mathematically or empirically. If a distribution is to be defined empirically, measurements of the actual system under study are taken. The results define the distribution of events in the real system; this distribution can then be used to drive the simulation.

  Simulations can be expensive, often requiring hours of computer time. A more detailed simulation provides more accurate results, but it also requires more computer time. In addition, trace tapes can require large amounts of storage space. Finally, the design, coding, and debugging of the simulator can be a major task.



Evaluation of CPU schedulers by simulation.

FREQUENTLY ASKED QUESTIONS

UNIT -1

1. Define Operating System? List the objectives of Operating System?
2. With a neat diagram, explain the layered structure of UNIX Operating System?
3. Explain how multiprogramming increases the utilization of CPU?
4. Explain the dual mode operation of an Operating System?
5. Mention the objectives and functions of realtime embedded systems?
6. With a neat sketch, describe the Operating system Services?
7. What is a System Call? Explain the various types of System Calls?
8. What is Computer System Architecture? Explain number of different ways in which it can be organized?
9. Explain the process of invoking System call with an example. Also write various system calls?

UNIT – 2

1. What is a Process? Explain about various fields of Process Cintrol Block?
2. With a neat diagram, explain various atate of a process?
3. What is scheduler? Explain various types of schedulers and their roles with help of process state diagram?
4. Describe the differences among short term, medium term, and long term scheduling?
5. Explain the following operations on processes: Process Creation and Process Termination
6. What are the advantages of Inter Process Communication? How communicationtakes place in a Shared memory environment? Explain.
7. Write and explain various issues involved in message passing systems?
8. Define a Thread? Give the benefits of multithreading. Differentiate process and Thread?
9. Explain about different types of multithreading models?
10. Define thread. What are the differences between user level and kernel level thread?
11. What are the criteria for evaluating the CPU scheduling algorithms? Why do we need it.
12. Explain Round Robin Scheduling algorithm with an example?
13. Distinguish between preemptive and non preemptive scheduling. Explain each type with an example?
14. What are the parameters that can be used to evaluate algorithms? Also explain different algorithmic evaluation methods with advantages and disadvantages?

Syllubus:Concurrency: Process synchronization, the critical- section problem, Peterson's Solution, synchronization Hardware, semaphores, classic problems of synchronization, monitors, and Synchronization examples

The Critical-Section Problem: Consider a system consisting of n processes (P1, P2, P3…..Pn). Each process has a segment of code, called a **critical section,** 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 critical section, no other process is to be allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time. Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section.** The critical section may be followed by an **exit section.** The remaining code is the **remainder section.** The general structure of a typical process P_i is shonw below.

```
do {
    entry section
        critical section
    exitsection
        remainder section
} while (TRUE);
```

General structure of a typical process P_i.

A solution to the critical-section problem must satisfy the following three requirements:

- **Mutual exclusion:** If process P; is executing in its critical section, then no other processes can be executing in their critical sections.
- **Progress:** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can enter its critical section next.
- **Bounded waiting:** There exists a bound, or limit, 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.

Peterson's Solution: A classic software-based solution to the critical-section problem known as **Peterson's solution.** Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. Peterson's solution requires two data items to be shared between the two processes:

int turn;
boolean flag [2];

The variable turn indicates whose turn it is to enter its critical section. That is, if turn == i, then process P; is allowed to execute in its critical section. The flag array is used to indicate if a process *is ready* to enter its critical section. For example, if f lag[i] is true, this value indicates that P; is ready to enter its critical section.

To enter the critical section, process P, first sets flag[i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section.

If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time. Only one of these assignments will last; the other will occur but will be overwritten immediately. The updated value of turn decides which of the two processes is allowed to enter its critical section first.

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

The structure of process P_i in Peterson's solution.

To prove property 1(Mutual Exclusion), note that each Pi enters its critical section only if either flag[j] == false or turn == i. Also note that, if both processes can be executing in their critical sections at the same time, then flag [0] == flag [1] == true. These two observations imply that P0 and P1 could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both.

To prove properties 2 and 3, note that a process Pj, can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag [j] == true and turn == j; this loop is the only one possible. If Pj is not ready to enter the critical section, then flag [j] == false, and Pi can enter its critical section. If *Pj* has set flag [j] to true and is also executing in its while statement, then either turn == i or turn == j . If turn == i, then Pi will enter the critical section. If turn == j, then *Pj* will enter the critical section. However, once *Pj* exits its critical section, it will reset f lag[j] to false, allowing Pi to enter its critical section. If *Pj* resets flag [j] to true, it must also set turn to i.

Synchronization Hardware: In this, critical section is secured by using locks. Race conditions are prevented by requiring that critical regions be protected by locks. That is, a process must acquire a lock before entering a critical section; it releases the lock when it exits the critical section as shown below.

```
do {
    acquire lock

        critical section

    releaselock

        remainder section

} while (TRUE);
```

Solution to the critical-section problem using locks.

The TestAndSet() instruction can be used to solve Critical Section problem. The important characteristic here is that this instruction is executed atomically. Thus, if two TestAndSet() instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order. If the machine supports the TestAndSet () instruction, then we

can implement mutual exclusion by declaring a Boolean variable lock, initialized to false. The structure of process Pi is shown below.

```
boolean TestAndSet(boolean *target) {
   boolean rv = *target;
   *target = TRUE;
   return rv;
}
```

The definition of the TestAndSet() instruction.

```
do {
   while (TestAndSetLock(&lock))
      ; // do nothing

      // critical section

   lock = FALSE;

      // remainder section
}while (TRUE);
```

Mutual-exclusion implementation with TestAndSet ().

The SwapO instruction, in contrast to the TestAndSet0 instruction, operates on the contents of two words; it is defined as shown below. Like the TestAndSet 0 instruction, it is executed atomically. If the machine supports the Swap() instruction, then mutual exclusion can be provided as follows. A global Boolean variable lock is declared and is initialized to false. In addition, each process has a local Boolean variable key. The structure of process *Pi,* is shown below.

```
void Swap(boolean *a, boolean *b) {
   boolean temp = *a;
   *a = *b;
   *b = temp;
}
```

The definition of the Swap () instruction.

```
do {
   key = TRUE;
   while (key == TRUE)
      Swap(&lock, &key);

      // critical section

   lock = FALSE;

      // remainder section
}while (TRUE);
```

Mutual-exclusion implementation with the Swap() instruction.

All these algorithms satisfy the mutual-exclusion requirement, they do not satisfy the bounded-waiting requirement. The following is another algorithm using the TestAndSet() instruction that satisfies all the critical-section requirements. The common data structures are

boolean waiting[n];
boolean lock;

These data structures are initialized to false. To prove that the mutualexclusion requirement is met, we note that process Pi can enter its critical section only if either waiting[i] == false or key== false. The value of key can become false only if the TestAndSet() is executed. The first process to execute the TestAndSet () will find key == false; all others must wait. The variable waiting[i] can become false only if another process leaves its critical section; only one waiting [i] is set to false, maintaining the mutual-exclusion requirement.

```
do {
   waiting [i] = TRUE;
   key = TRUE;
   while (waiting[i] && key)
      key = TestAndSet(&lock);
   waiting [i] = FALSE;

      // critical section

   j = (i + 1) % n;
   while ((j != i) &&   !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = FALSE;
   else
      waiting[j] = FALSE;

      // remainder section
}while (TRUE);
```

Bounded-waiting mutual exclusion with TestAndSet().

Semaphores: A semaphore S is an integer variable which can be incremented and decremented. The only two atomic operations that can be performed on semaphores are wait () and signal ().The wait() operation was originally termed P; signal () was originally called V. The definition of wait () is as follows:

```
wait(S)
{
while S <= 0
; // no-op
S--;
}
```

The definition of signal () is as follows:

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

All the modifications to the integer value of the semaphore in the wait () and signal() operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

There are two types of semaphores: binary semaphore and counting semaphore. 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. On some systems, binary semaphores are known as **mutex locks,** as they are locks that provide *mutualexclusion*.

Binary semaphores are used to deal with the critical-section problem for multiple processes. The *n* processes share a semaphore, mutex, initialized to 1. Each process Pi is organized as shown below.

```
do {
   waiting(mutex);
      // critical section
   signal(mutex);
      // remainder section
}while (TRUE);
```

Mutual-exclusion implementation with semaphores.

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.

Semaphores are also used 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 P*2* share a common semaphore **synch**, initialized to 0, and by inserting the statements

S1;

signal(synch);

in process *P1,* and the statements

wait(synch);

S2;

in process P2. 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.

Implementation:

The main disadvantage of the semaphore definition 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 entry code. 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. This type of semaphore is also called a **spinlock** because the process "spins" while waiting for the lock.

When a process executes the wait () operation and finds that the semaphore value is not positive, it must wait. However, 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 following C cod eimplements this.

```
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 now be defined as

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

The signal () semaphore operation can now 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.**

For example, consider a system consisting of two processes, P0 and P1, each accessing two semaphores, S and Q, set to the value 1: 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.

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
.	.
.	.
.	.
signal(S);	signal(Q);
signal(Q);	signal(S);

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 add and remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

Classic Problems of Synchronization:

The Bounded-Buffer Problem: In this problem 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 f u l l semaphores count the number of empty and

full buffers. The semaphore empty is initialized to the value n; the semaphore f u l l is initialized to the value 0. The code for the producer and consumer processes is shown below.

```
do {
    . . .
  // produce an item in nextp
    . . .
  wait(empty);
  wait(mutex);
    . . .
  // add nextp to buffer
    . . .
  signal(mutex);
  signal(full);
 }while (TRUE) ,-
```

The structure of the producer process.

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

The structure of the consumer process.

The Readers-Writers Problem: In general, A database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database. If two readers access the shared data simultaneously, no problem will occur. However, if a writer and some other thread (either a reader or a writer) access the database simultaneously, problems may ensue. This synchronization problem is referred to as the *readers-writers problem.*

The readers-writers problem has several variations, all involving priorities. The simplest one, referred to as the *first* readers-writers problem, requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers. The *second* readers-writers problem requires that, once a writer is ready, that writer performs its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

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

semaphore mutex, wrt;

int readcount;

The semaphores mutex and wrt are initialized to 1; readcount is initialized to 0. The semaphore wrt is common to both reader and writer processes. The mutex semaphore is used to ensure mutual exclusion when the variable readcount is updated. The readcount variable keeps track of how many processes are currently reading the object. The semaphore wrt functions as a mutual-exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the

critical section. It is not used by readers who enter or exit while other readers are in their critical sections. The structure of reader and writer processes are shown below.

```
do {
   wait(wrt);
      . . .
   // writing is performed
      . . .
   signal(wrt);
}while (TRUE);
```

The structure of a writer process.

```
do {
   wait(mutex);
   readcount++;
   if (readcount == 1)
      wait(wrt);
   signal(mutex);
      . . .
   // reading is performed
      . . .
   wait(mutex);
   readcount--;
   if (readcount == 0)
      signal(wrt);
   signal(mutex);
}while (TRUE);
```

The structure of a reader process.

Note that, if a writer is in the critical section and *n* readers are waiting, then one reader is queued on wrt, and n — 1 readers are queued on mutex. Also observe that, when a writer executes signal (wrt), we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

The Dining-Philosophers Problem: Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks as shown in the diagram. When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.

One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait () operation on that semaphore; she releases her chopsticks by executing the signal() operation on the appropriate semaphores. Thus, the shared data are

semaphore chopstick[5];

where all the elements of chopstick are initialized to 1. The structure of philosopher i is shown below.

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   // eat
      . . .
   signal(chopstick[i]);

   signal(chopstick[(i+1) % 5]);

   // think

}while (TRUE);
```

The structure of philosopher *i*.

Monitors: Using Sempaphores incorrectly can result in timing errors that are difficult to detect. For example,

- Suppose that a process interchanges the order in which the wait(j and signal () operations on the semaphore mutex are executed, resulting in the following execution:

  signal(mutex);
  //critical section
  wait(mutex);

  In this situation, several processes may be executing in their critical sections simultaneously, violating the rmitual-exclusion requirement.
- Suppose that a process replaces signal (mutex) with wait (mutex). That is, it executes

  wait(mutex);
  //critical section
  wait(mutex);

  In this case, a deadlock will occur.
- Suppose that a process omits the wait (mutex), or the signal (mutex), or both. In this case, either mutual exclusion is violated or a deadlock will occur.

These examples illustrate that various types of errors can be generated easily when programmers use semaphores incorrectly to solve the critical-section problem. To deal with such errors, ne fundamental high-level synchronization construct—the monitor type is used.

A Monitor is a type, or abstract data 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, along with the bodies of procedures or functions. The syntax of a monitor is shown below.

```
monitor monitor name
{
    // shared variable declarations
    procedure P1 ( . . . )
    {
    …….
    }
```

```
    procedure P 2 ( . . . )
    {
    ……
    }
    ……
    ……
    ……
    procedure P n ( . . . )
    {
    …….
    }
    Initializationcode( . . . )
    {
    ……
    }
}
```

The monitor construct ensures that only one process at a time can be active within the monitor, i.e.mutual exclusion is achieved. To overcome synchronization problems, a condition construct is used as shown below.

condition x, y;

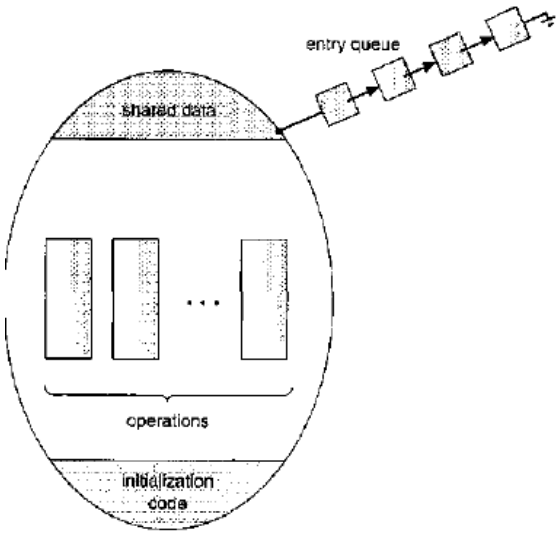
The only operations that can be invoked on a condition variable are wait () and signal(). The operation

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. If no process is suspended, then the signal () operation has no effect; that is, the state of x is the same as if the operation had never been executed.



Schematic view of a monitor.

For example, when the x. **s**ignal () operation is invoked by a process P, there is a suspended process Q associated with condition x. Clearly, if the suspended process Q is allowed to resume its execution, the signaling process P must wait. Otherwise, both P and Q would be active simultaneously within the monitor. Note, however, that both processes can conceptually continue with their execution. Two possibilities exist:

1. Signal and wait. *P* either waits until Q leaves the monitor or waits for another condition.

2. Signal and continue. *Q* either waits until *P* leaves the monitor or waits for another condition.

Dining-Philosophers Solution Using Monitors: This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available. To do this use the following data structure.

enum {thinking, hungry, eating} state[5] ;

Here, Philosopher *i* can set the variable s t a t e [i] = eating only if her two neighbors are not eating: (s t a te [(i+4) °/» 5] != eating) and (s t a te [(i+1) % 5] != eating).

condition self [5];

Where philosopher i can delay herself when she is hungry but is unable to obtain the chopsticks she needs.

The distribution of the chopsticks is controlled by the monitor dp as shown below.

```
dp.pickup(i);
.....
//eat
.....
dp.putdown(i);
```

```
monitor dp
{
   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 til = 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;
    }
}
```

A monitor solution to the dining-philosopher problem.

Implementing a Monitor Using Semaphores: For each monitor, a semaphore mut ex (initialized to 1) is provided. A process must execute wait (mutex) before entering the monitor and must execute signal (mutex) after leaving the monitor.

Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore, next, is introduced, initialized to 0, on which the signaling processes may suspend themselves. An integer variable next-count is also provided to count the number of processes suspended on next.

```
wait(mutex);
    ...
  body of F
    ...
if (next_count > 0)
  signal(next);
else
  signal(mutex);
```

For each condition x, a semaphore x_sem and an integer variable x_count used both initialized to 0. The operation x. wait () can now be implemented as

```
x_count++;
if (next_count > 0)
  signal(next);
else
  signal(mutex);
wait(x_sem);
x_count--;
```

The operation x. signal () can be implemented as

```
if (x_count > 0) {
  next_count++;
  signal(x_sem);
  wait(next);
  next_count--;
}
```

Resuming Processes Within a Monitor: If several processes are suspended on condition x, and an x. signal () operation is executed by some process, then how do we determine which of the suspended processes should be resumed next? One simple solution is to use an FCFS ordering, so that the process waiting the longest is resumed first. In many cases, such a simple scheduling scheme is not adequate. For this purpose, the **conditional-wait** construct can be used; it has the form

x.wait(c);

where c is an integer expression that is evaluated when the wait () operation is executed. The value of c, which is called a **priority number,** is then stored with the name of the process that is suspended. When x. signal () is executed, the process with the smallest associated priority number is resumed next. For example, consider the ResourceAllocator monitor shown below, which controls the allocation of a single resource among competing processes. Each process, when requesting an allocation of this resource, specifies the maximum time it plans to use the resource. The monitor allocates the resource to the process that has the shortest timeallocation request. A process that needs to access the resource in question must observe the following sequence:

R.acquire(t);
//access the resource;
R. release() ;
where R is an instance of type ResourceAllocator.

```
monitor ResourceAllocator
{
   boolean busy;
   condition x;

   void acquire(int time) {
      if (busy)
         x.wait(time);
      busy = TRUE;
   }

   void release() {
      busy = FALSE;
      x.signal();
   }

   initialization_code() {
      busy = FALSE;
   }
}
```

A monitor to allocate a single resource.

Synchronization Examples:

1. **Synchronization in Solaris:** To control access to critical sections, Solaris provides adaptive mutexes, condition variables, semaphores, reader-writer locks, and turnstiles.

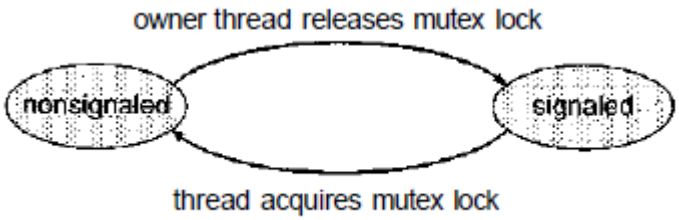
   An adaptive mutex protects access to every critical data item. On a multiprocessor system, an adaptive mutex starts as a standard semaphore implemented as a spinlock. If the data are locked and therefore already in use, the adaptive mutex does one of two things. If the lock is held by a thread that is currently running on another CPU, the thread spins while waiting for the lock to become available, because the thread holding the lock is likely to finish soon. If the thread holding the lock is not currently in run state, the thread blocks, going to sleep until it is awakened by the release of the lock.

   Reader-writer locks are used to protect data that are accessed frequently but are usually accessed in a read-only manner. In these circumstances, reader-writer locks are more efficient than semaphores, because multiple threads can read data concurrently, whereas semaphores always serialize access to the data. Reader-writer locks are relatively expensive to implement, so again they are used on only long sections of code.

   A **turnstile** is a queue structure containing threads blocked on a lock. For example, if one thread currently owns the lock for a synchronized object, all other threads trying to acquire the lock will block and enter the turnstile for that lock. When the lock is released, the kernel selects a thread from the turnstile as the next owner of the lock.

2. **Synchronization in Windows XP:** For thread synchronization Windows XP provides **dispatcher objects.** Using a dispatcher object, threads synchronize according to several different mechanisms, including mutexes, semaphores, events, and timers. The system protects shared data by requiring a thread to gain ownership of a mutex to access the data and to release ownership when it is finished. **Events** are similar to condition variables; that is, they may notify a waiting thread when a desired condition occurs. Finally, timers are used to notify one (or more than one) thread that a specified amount of time has expired.

   Dispatcher objects may be in either a signaled state or a nonsignaled state. A **signaled state** indicates that an object is available and a thread will not block when acquiring the object. A **nonsignaled state** indicates that an object is not available and a thread will block when attempting to acquire the object.



3. **Synchronization in Linux:** Linux was a nonpreemptive kernel, meaning that a process running in kernel mode could not be preempted—even if a higher-priority process became available to run. Now, however, the Linux kernel is fully preemptive, so a task can be preempted when it is running in the kernel.

   The Linux kernel provides spinlocks and semaphores (as well as reader - writer versions of these two locks) for locking in the kernel. On SMP machines, the fundamental locking mechanism is a spinlock, and the kernel is designed so that the spinlock is held only for short durations. On single-processor machines, spinlocks are inappropriate for use and are replaced by enabling and disabling kernel preemption. That is, on single-processor machines, rather than holding a spinlock, the kernel disables kernel preemption; and rather than releasing the spinlock, it enables kernel preemption.
4. **Synchronization in Pthreads:** The Pthreads API provides mutex locks, condition variables, and read-write locks for thread synchronization. Mutex locks represent the fundamentalsynchronization technique used with Pthreads. A mutex lock is used to protect critical sections of code—that is, a thread acquires the lock before entering a critical section and releases it upon exiting the critical section.

FREQUENTLY ASKED QUESTIONS

1. What is a Critical-section Problem? Give the conditions that a solution to the critical section problem must satisfy?
2. What is a Semaphore? Also give the operations for accessing semaphores?
3. What is a semaphore? List the types of semaphores and show that if the wait() and signal() semaphore operations are not executed atomically, then mutual exclusion may be violated?
4. Give the Peterson's solution to the Critical-Section Problem?
5. What is Dining Philosophers Problem? Discuss the solution to Dining Philosophers problem using Monitos?
6. Discuss the Bounded-Buffer Problem?
7. Briefly explain the Readers-Writers Problem?
8. State the Critical Section Problem. Illustrate the software based solution to the Critical Section Problem?
9. How does the signal() operation associated with monitors differ from the corresponding operation defined for semaphores?
10. Define monitor? Explain how it overcomes the drawback of semaphores?
11. What is Synchronization? Explain how semaphores can be used to deal with n-process crictical section problem?
12. Discuss mutual exclusion implementation with test() and set() instruction?

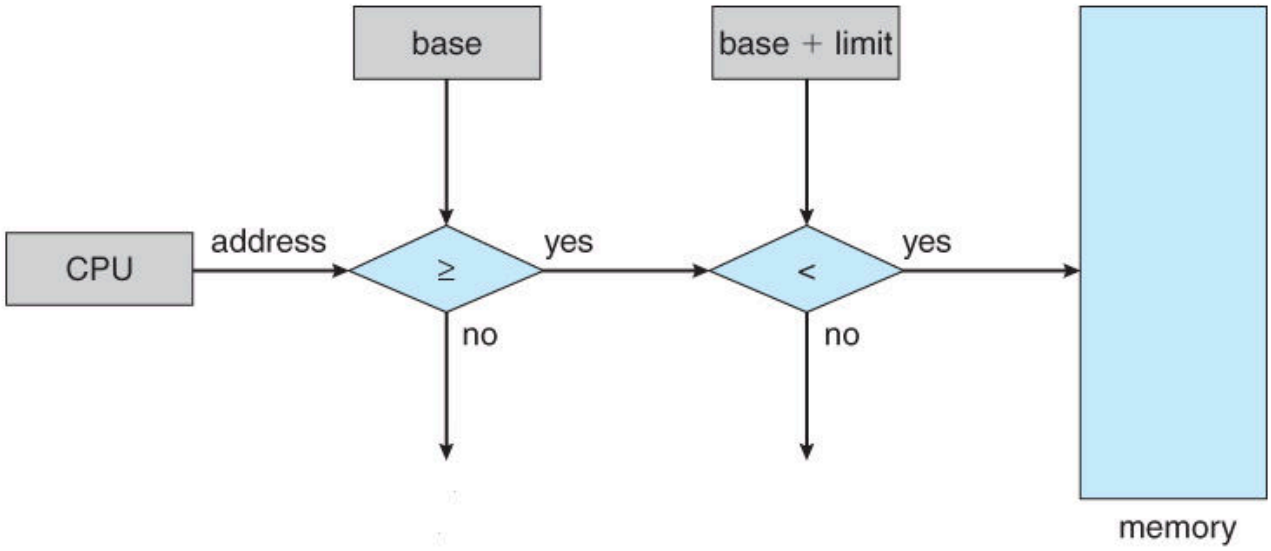
Syllubus: **Memory Management:** Swapping, contiguous memory allocation, paging, structure of the page table, segmentation
Virtual Memory Management: virtual memory, demand paging, page-Replacement, algorithms, Allocation of Frames, Thrashing

Memory Management :

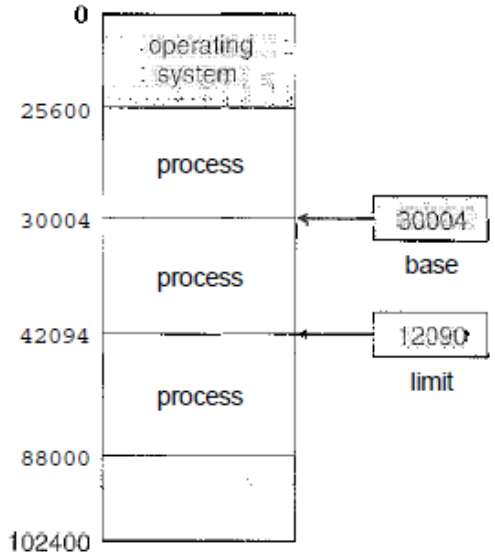
In a uniprogramming system, main memory is divided into two parts: one part for the operating system (resident monitor, kernel) and one part for the program currently being executed. In a multiprogramming system, the "user" part of memory must be further subdivided to accommodate multiple processes. The task of subdivision is carried out dynamically by the operating system and is known as **memory management**.

One impartant function of memory management is protection. This can be done by using Base and Limit registers. The base register holds the smallest legal physical memory address; the **limit register** specifies the size of the range.

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 which is shown in the below diagram.



Hardware address protection with base and limit registers.



A base and a limit register define a logical address space.

Address Binding: Address Binding is a process of mapping Logical address or virtual address to its corresponding physical address in main memory. The binding of instructions and data to memory addresses can be done at any step along the way:

Compile Time: If you know at compile time where the process will reside in memory, then **absolute code** can be generated. For example, if you know that a user process will reside starting at location *R,* then the generated compiler code will start at that location and extend up from there.

Load **time:** If it is not known at compile time where the process will reside in memory, then the compiler must generate **relocatable code. In** this case, final binding is delayed until load time.

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.

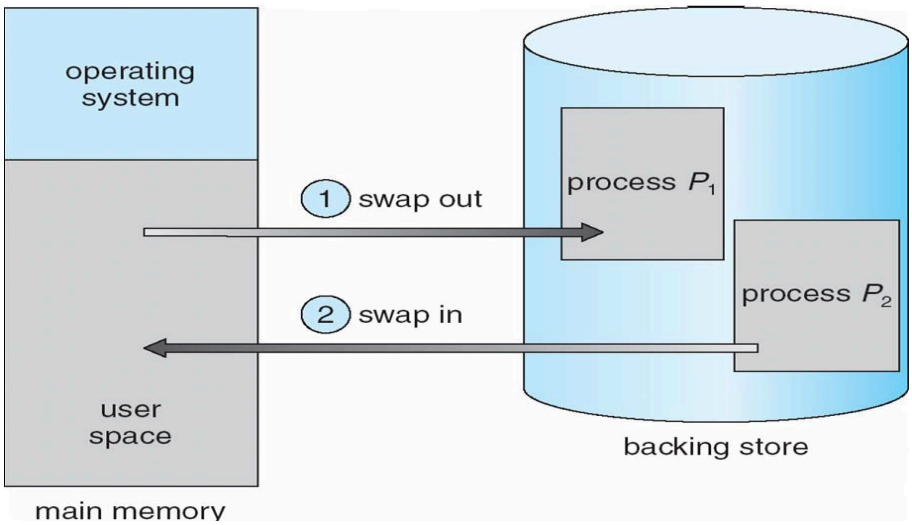
Logical Versus Physical Address Space: An address generated by the CPU is commonly referred to as a **logical address,** whereas an address seen by the memory unit—that is, the one loaded into the **memory-address register** of the memory—is commonly referred to as a **physical address.**

The compile-time and load-time address-binding methods generate identical logical and physical addresses. However, the execution-time addressbinding scheme results in differing logical and physical addresses. The set of all logical addresses generated by a program is a **logical** address **space;** the set of all physical addresses corresponding to these logical addresses is a **physical** address **space.**

The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit (MMU).**

Swapping: A process must be in memory to be executed. A process, however, can be swapped temporarily out of memory to a **backing store** and then brought back into memory for continued execution. For example, assume a multiprogramming environment with a round-robin CPU-scheduling algorithm. When a quantum expires, the memory manager will start to swap out the process that just finished and to swap another process into the memory space that has been freed. In the meantime, the CPU scheduler will allocate a time slice to some other process in memory.

A variant of this swapping policy is used for priority-based scheduling algorithms. If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process and then load and execute the higher-priority process. When the higher-priority process finishes, the lower-priority process can be swapped back in and continued. This variant of swapping is sometimes called **roll out, roll in.**



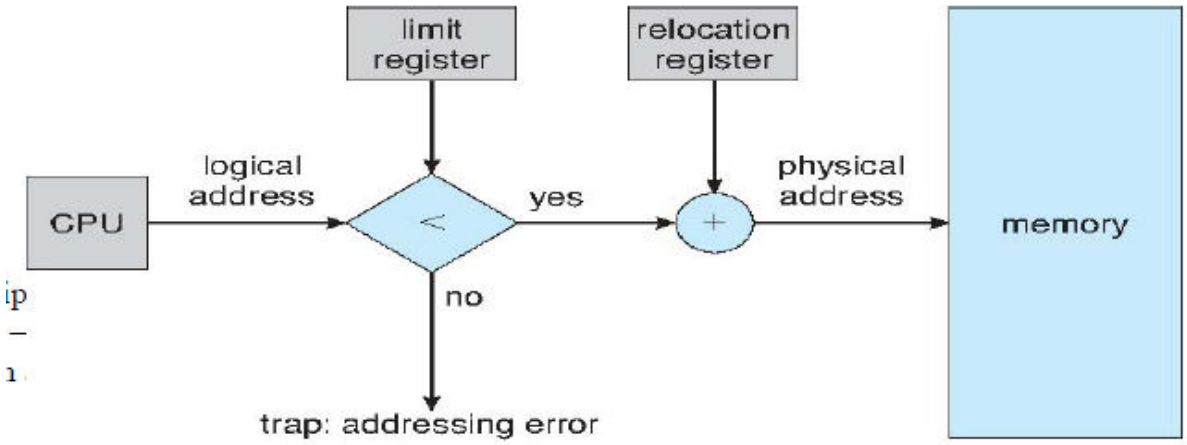
Normally, a process that is swapped out will be swapped back into the same memory space it occupied previously. This depends on the method of address binding. If binding is done at assembly or load time, then the process cannot be easily moved to a different location. If execution-time binding is being used, however, then a process can be swapped into a different memory space.

Swapping requires a backing store. The backing store is commonly a fast disk. It must be large enough to accommodate copies of all memory images for all users. The system maintains a ready queue consisting of all processes whose memory images are on the backing store or in memory and are ready to run. Whenever the CPU scheduler decides to execute a process, it calls the dispatcher. The dispatcher checks to see whether the next process in the queue is in memory. If it is not, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process. It then reloads registers and transfers control to the selected process.

The context-switch time in such a swapping system is fairly high. If we want to swap a process, we must be sure that it is completely idle.

Contiguous Memory Allocation: The memory is usually divided into two partitions: one for the resident operating system and one for the user processes. The operating system may be in either low memory or high memory depends on location of interrupt vector.

Memory Mapping and Protection: Mapping is the process of converting Logical address into Physical address. This can be done by using Base or relocation register and limit registers. Base register contains the value of the smallest physical address; the limit register contains the range of logical addresses. With relocation and limit registers, each logical address must be less than the limit register; the VIMU maps the logical address *dynamically* by adding the value in the relocation register. This mapped address is sent to memory.



Hardware support for relocation and limit registers.

When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Because every address generated by the CPU is checked against these registers, we can protect both the operating system and the other users programs and data from being modified by this running process.

Memory Allocation: One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions. In this multiplepartition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. This method was originally used by the IBM OS/360 operating system. The following are two different partitioning methods:

1. **Fixed partitioning**
2. **Dynamic sized partitioning (or) Variable sized Partitioning**

1. **Fixed partitioning:** In this, Single contiguous memory location is a simple memory management scheme that requires no special hardware features. For allocating memory, it is divided into number of fixed sized partitions. Each partition contains exactly one process. If the partition is free, process is selected from the input queue and is loaded into the free partition of memory. When the process terminates, the memory partition becomes available for another

process. Batch operating systems uses this partition scheme. The operating system maintains the record of allocating and deallocating memory to processes.

When the process arrives in the systemand needs memory, operating system search large enough space for this process. If available, use it. Otherwise, wait.

The following diagrams shows the examples of two alternatives for fixed partitioning: one possibility is to make use of equal size partitions.

8MB
8MB
8MB
8MB
8MB
8MB
8MB
8MB

Equal Size Partitions

8MB
6MB
5MB
3MB
4MB
4MB
6MB
12MB
16MB

Unequal size partitions

Any process whose size is lessthan or equal to the partition size can be loaded into any available partition.

If all memory partitions are full and no process is in the ready or running state, the operating system can swap a process out of any of the partitions and load in another process.

Therer are two difficulties in with the use of equal size fixed partitions:

→ A program may be too big to fit into a partition.

→ Main memory utilization is extremely inefficient.

Advantages and disadvantages of Fixed Partition size:

Advantages:

→ Simple to implement

→ Less overhead

Disadvantages:

→ Memory is not efficiently used

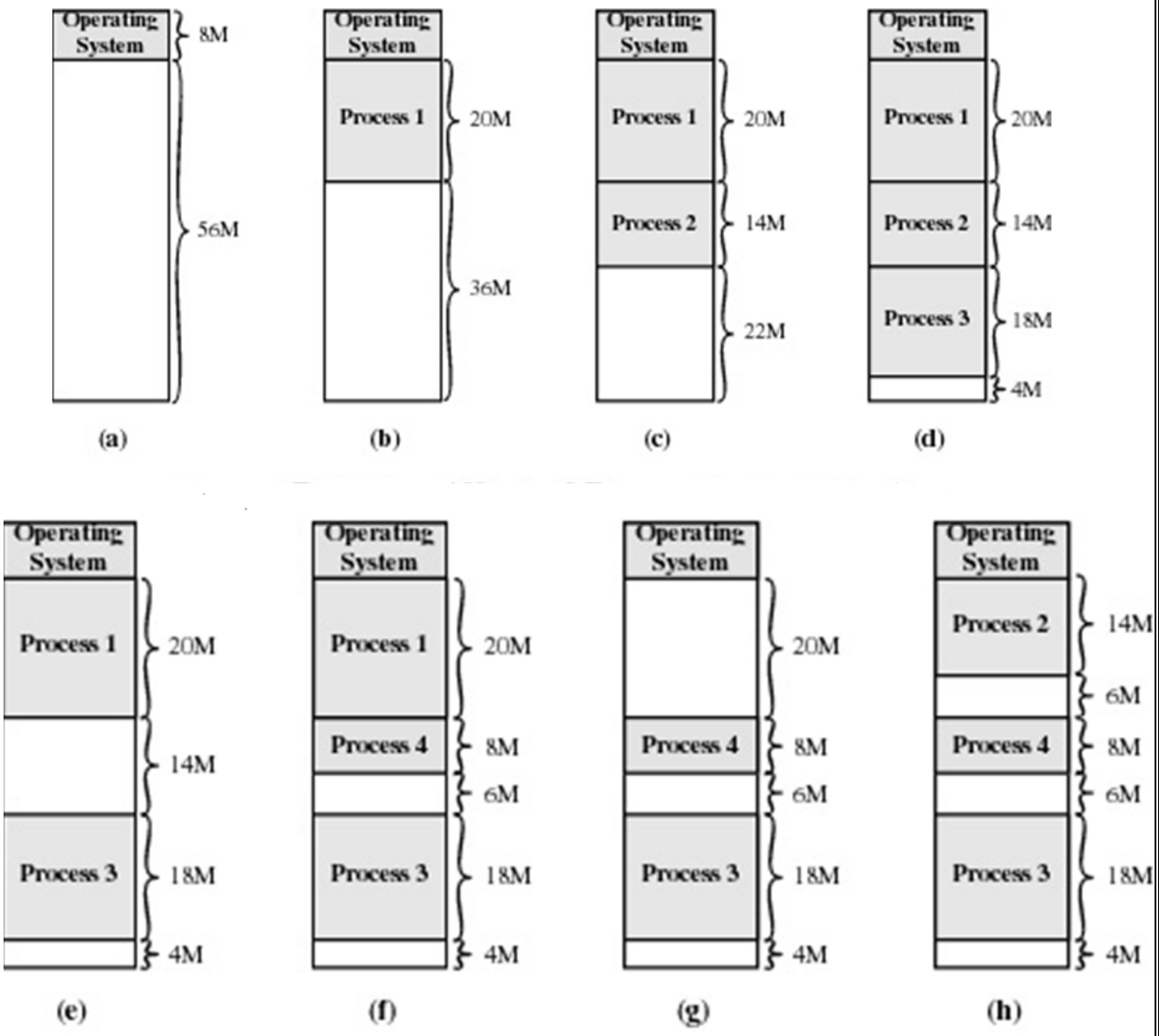
→ Poor utilization of Processors

→ User's process is limited to the size of available mamin memory

2. **Dynamic sized partitioning:** In this technique, Memory Prtitions are of variable length. When process is brought into memory, it is allocated exactly as much memory as it requires and no more. Initially, main memory is empty, except for the operating system as shown below.

The first three processes are loaded in, starting where the operating system ends and occupying just enough space for each process. This leaves a "hole" at end of memory that is too small for a fourth process.

The operating system swaps out process2 , which leaves sufficient room to load a new process, process 4. Since, process 4 is smaller than process 2,a nother small hole is created. At certain point of time, none of the processes in mamn memory is ready, but process 2 is the ready suspend state, is available. Because there is insufficient space in memory for process 2, the operating system swaps out process 1 and swaps in process 2 for exectution.



The Effect of Dynamic Partitioning

The **first-fit, best-fit,** and **worst-fit** strategies are the ones most commonly used to select a free hole from the set of available holes.

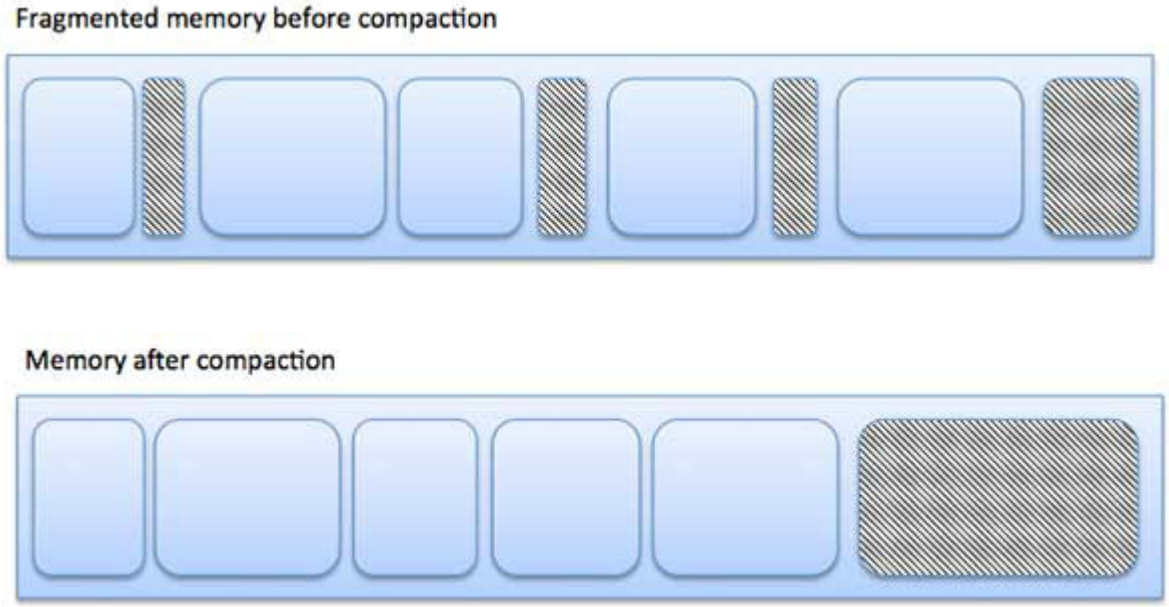
- First fit: Allocate the *first* hole that is big enough. Searching can start either at the beginning of the set of holes or where the previous first-fit search ended.
- Best fit: Allocate the *smallest* hole that is big enough. Search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
- Worst fit: Allocate the *largest* hole. Again, search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Fragmentation: As processes are loaded and removed from memory, the free memory space is broken into little pieces. It happens after sometimes that processes cannot be allocated to memory blocks considering their small size and memory blocks remains unused. This problem is known as Fragmentation. Fragmentation is of two types –

- **Internal Fragmentation:** Unused memory which is internal to a prtition is called as Internal Fragmentation. The internal fragmentation can be reduced by effectively assigning the smallest partition but large enough for the process.

- **External Fragmentation:** External fragmentation exists when there is enough total memory space to satisfy a request, but the available spaces are not contiguous. One solution to the problem of external fragmentation is **Compaction**. **Compaction** is the process of shuffling the memory contents so as to place all free memory together in one large block. Compaction is not always possible, however. If relocation is static and is done at assembly or load time, compaction cannot be done; compaction is possible *only* if relocation is dynamic and is done at execution time.

  The following diagram shows how fragmentation can cause waste of memory and a compaction technique can be used to create more free memory out of fragmented memory –



Another possible solution to the external-fragmentation problem is to permit the logical address space of the processes to be noncontiguous, thus allowing a process to be allocated physical memory wherever the latter is available. Two complementary techniques achieve this solution: paging and Segmentation.

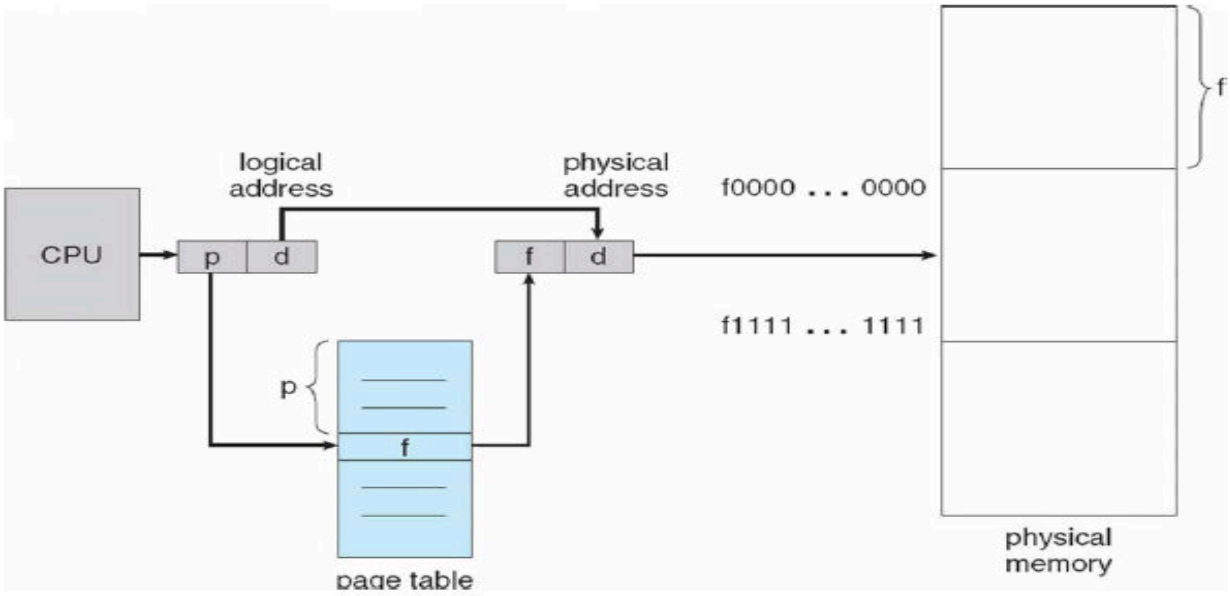
Differences between Internal and External Fragmentation:

S.No	Internal Fragmentation	External Fragmentation
1	Memory allocated to a process may be slightly larger than the requested memory.	External fragmentation exists when there is enough total memory space to satisfy a request, but the available spaces are not contiguous.
2	First fit and best fit memory allocation does not suffer from internal fragmentation.	First fit and best fit memory allocation suffers from internal fragmentation.
3	Systems with fixed sized allocation units, such as the single sized partition scheme and paging suffer from internal fragmentation.	Systems with variable sized allocation units, such as the multiple partition scheme and segmentation suffer from internal fragmentation.

Paging: Paging is a memory-management scheme that permits the physical address. Space of a process to be noncontiguous. Paging avoids external fragmentation and use of Compaction.

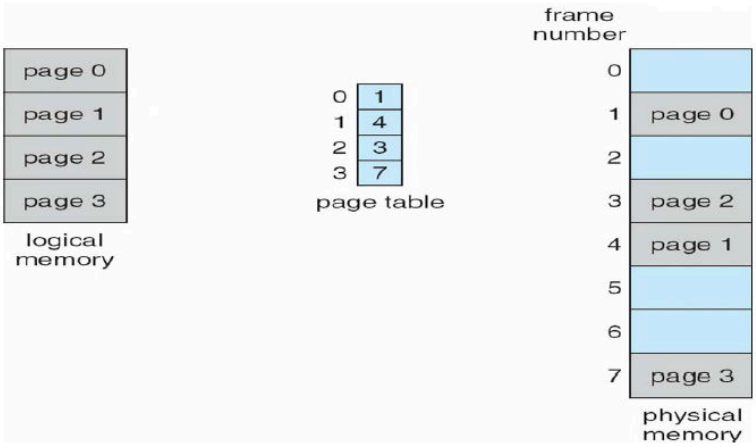
Basic Method: The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called **frames** and breaking logical memory into blocks of the same size called **pages.** When a process is to be executed, its pages are loaded into any available memory frames from the backing store. The backing store is divided into fixed-sized blocks that are of the same size as the memory frames.

The hardware support for paging is illustrated in the following figure.



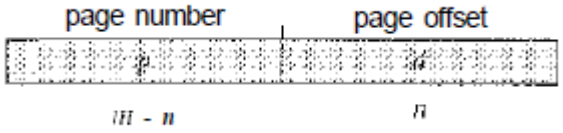
Paging hardware.

Every address generated by the CPU is divided into two parts: a **page number (p)** and a **page offset (d).** 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. The paging model of memory is shown in below figure.



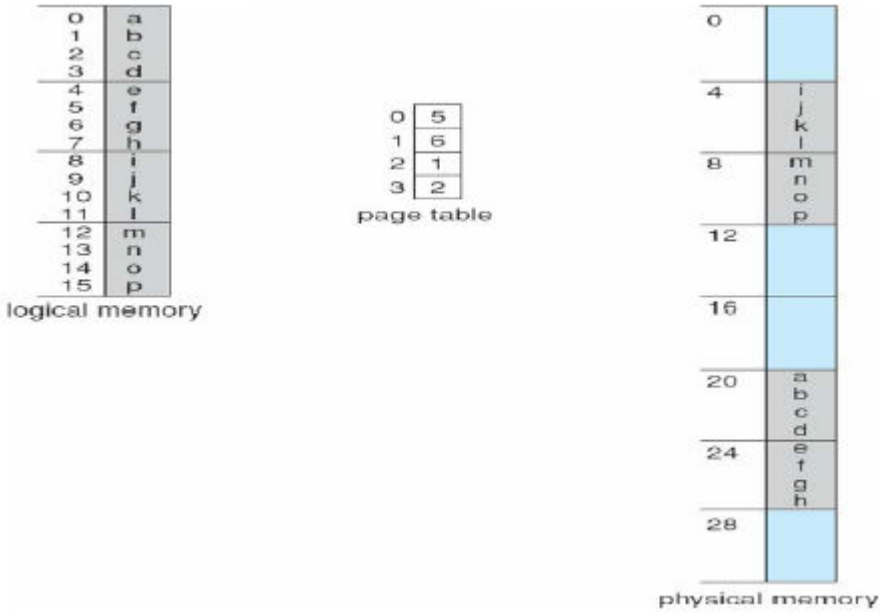
Paging model of logical and physical memory.

Paging model of logical and physical memory. The page size (like the frame size) is defined by the hardware. The size of a page is typically a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy. If the size of logical address space is 2'"* and a page size is 2" addressing units (bytes or words), then the high-order $m - n$ bits of a logical address designate the page number, and the n low-order bits designate the page offset. Thus, the logical address is as follows:



where *p* is an index into the page table and *d* is the displacement within the page.

For example, consider the memory in the following figure. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages)..



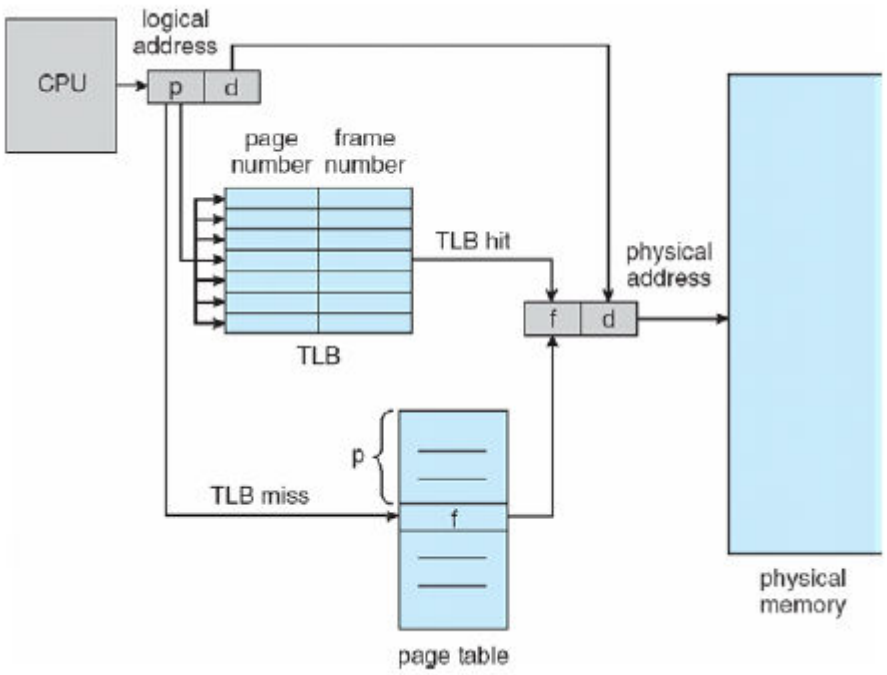
Paging example for a 32-byte memory with 4-byte pages.

Paging example for a 32-byte memory with 4-byte pages. 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 x 4) + 0). Logical address 3 (page 0, offset 3) maps to physical address 23 {- (5x4) + 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 (= (6x4) + 0). Logical address 13 maps to physical address 9.

Hardware Support: As size of the page table increases, memory access time increases. Such that performance of a computer decreases. To overcome this problem, use a special, small, fastlookup hardware cache, called a translation look-aside buffer (TLB). 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 is used with page tables in the following way. 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 found(TLB Hit), its frame number is immediately available and is used to access memory.

If the page number is not in the TLB (known as a TLB miss), a memory reference to the page table must be made. When the frame number is obtained, use it to access memory. In addition, add the page number and frame number to the TLB, so that they will be found quickly on the next reference. If the TLB is already full of entries, the operating system must select one for replacement.

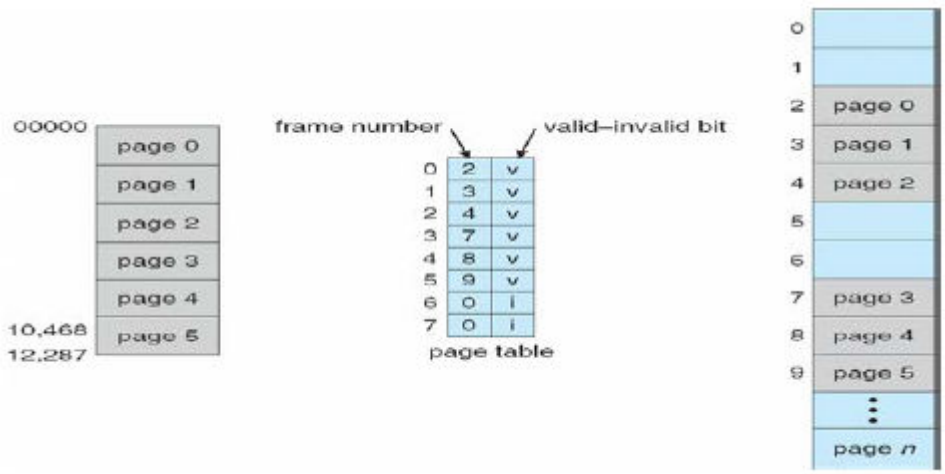


Paging hardware with TLB.

The percentage of times that a particular page number is found in the TLB is called the **hit ratio.** An 80-percent hit ratio means that we find the desired page number in the TLB 80 percent of the time. If it takes 20 nanoseconds to search the TLB and 100 nanoseconds to access memory, then a mapped-memory access takes 120 nanoseconds when the page number is in the TLB. If we fail to find the page number in the TLB (20 nanoseconds), 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 220 nanoseconds. To find the **effective memory-access time,** we weight each case by its probability:

effective access time = 0.80 x 120 + 0.20 x 220
= 140 nanoseconds.

Protection: Memory protection in a paged environment is accomplished by protection bits associated with each frame. Normally, these bits are kept in the page table. 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. The operating system
sets this bit for each page to allow or disallow access to the page.

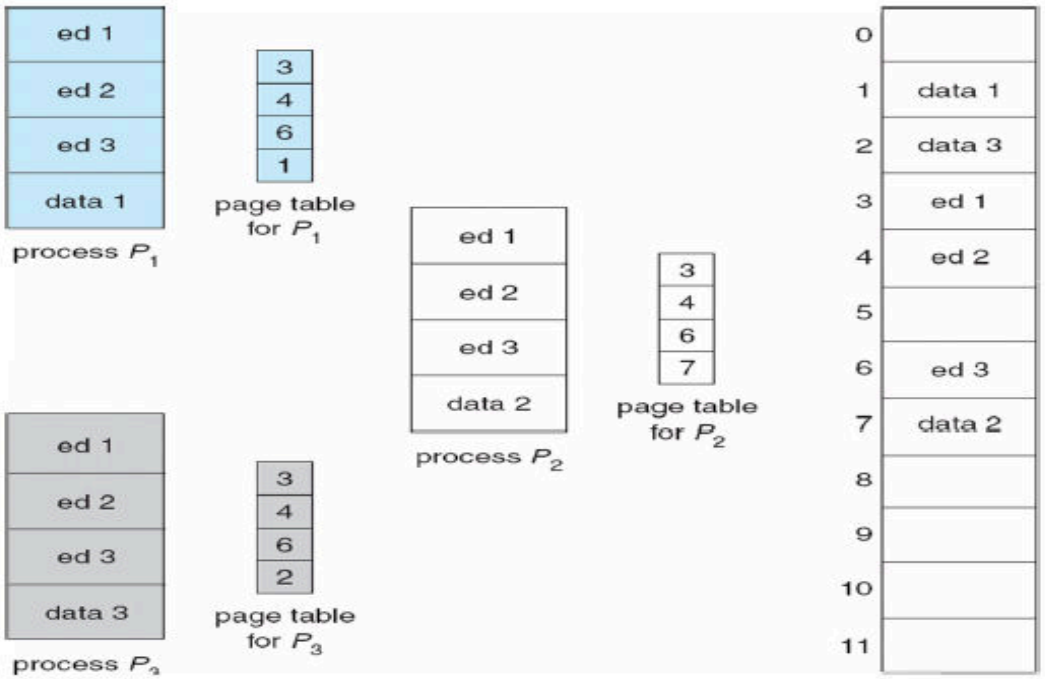


Valid (v) or invalid (i) bit in a page table.

Here, Addresses in pages 0,1, 2,3, 4, and 5 are mapped normally through the page table. Any attempt to generate an address in pages 6 or 7, however, will find that the valid-invalid bit is set to invalid, and the computer will trap to the operating system.

Shared Pages: An advantage of paging is the possibility of *sharing* common code. This consideration is particularly important in a time-sharing environment. Consider a system that supports 40 users, each of whom executes a text editor. If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to support the 40 users.

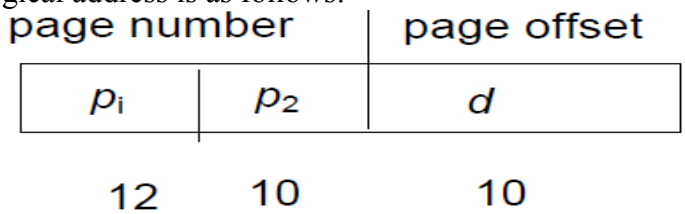
In this case, Only one copy of the editor need be kept in physical memory. Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames. Thus, to support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user. The total space required is now 2,150 KB instead of 8,000 KB which is shown in the figure below.



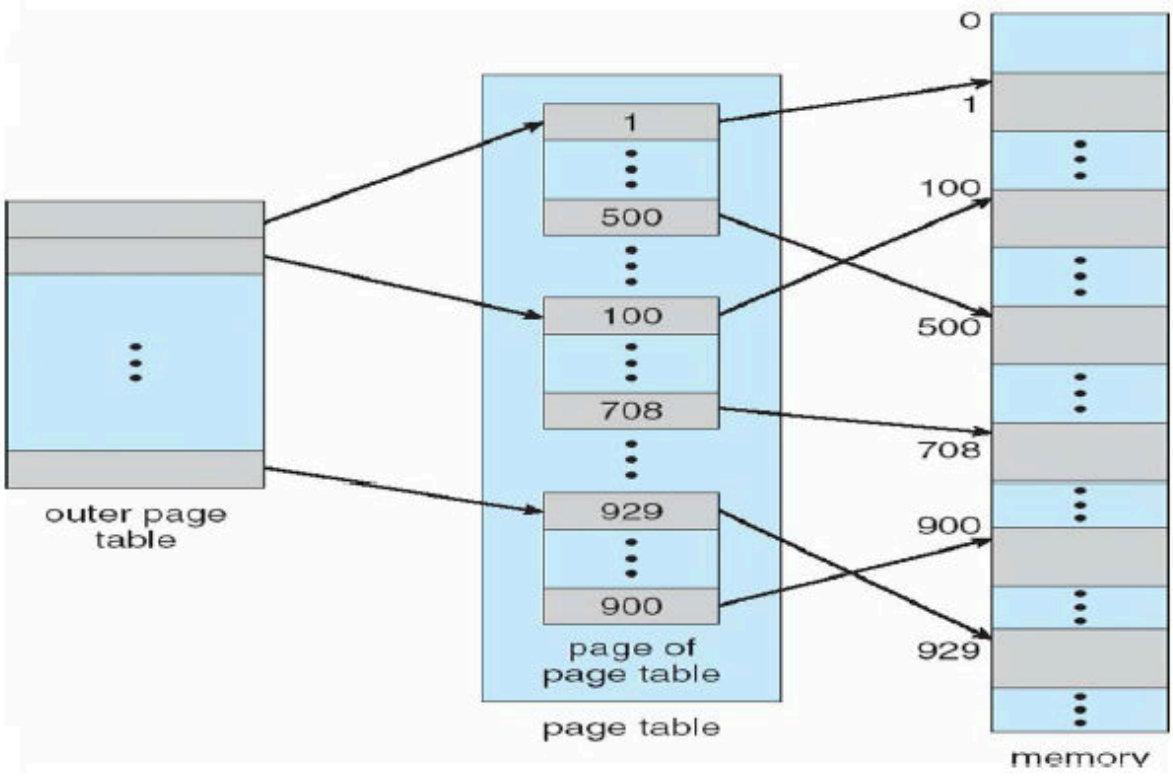
Sharing of code in a paging environment.

Structure of the Page Table: Some of the most common techniques for structuring the page table are : Hierarchical Paging, Hashed page tables, and Inverted page tables.

- **Hierarchical Paging:** In this, page table will be divided into smaller pieces. Example for Hierarchical Paging is two-level paging algorithm, in which the page table itself is also paged. Consider a 32 bit machine with 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:



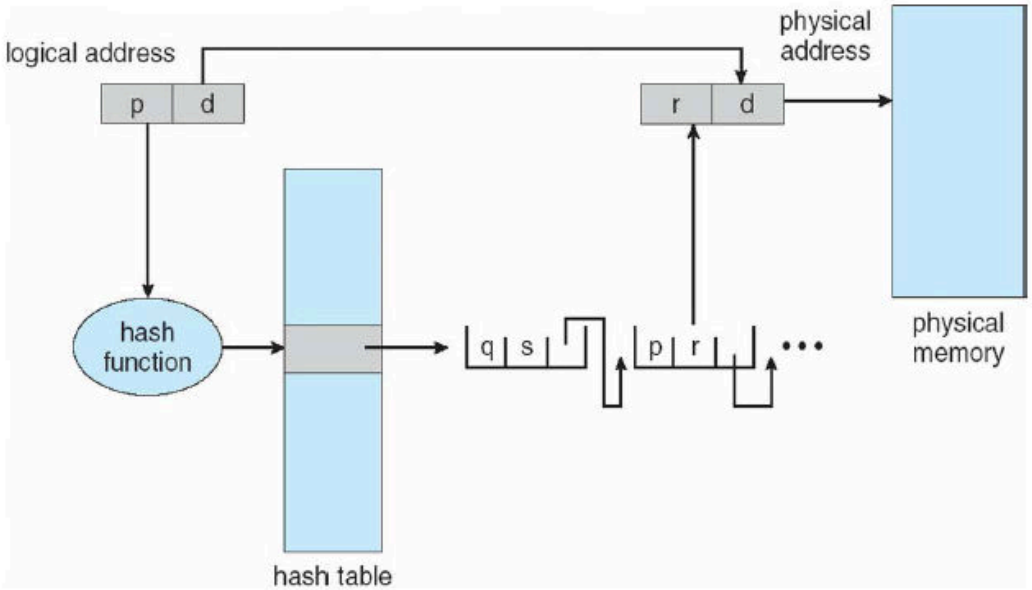
where $p\backslash$ is an index into the outer page table and *p2* is the displacement within the page of the outer page table.



A two-level page-table scheme

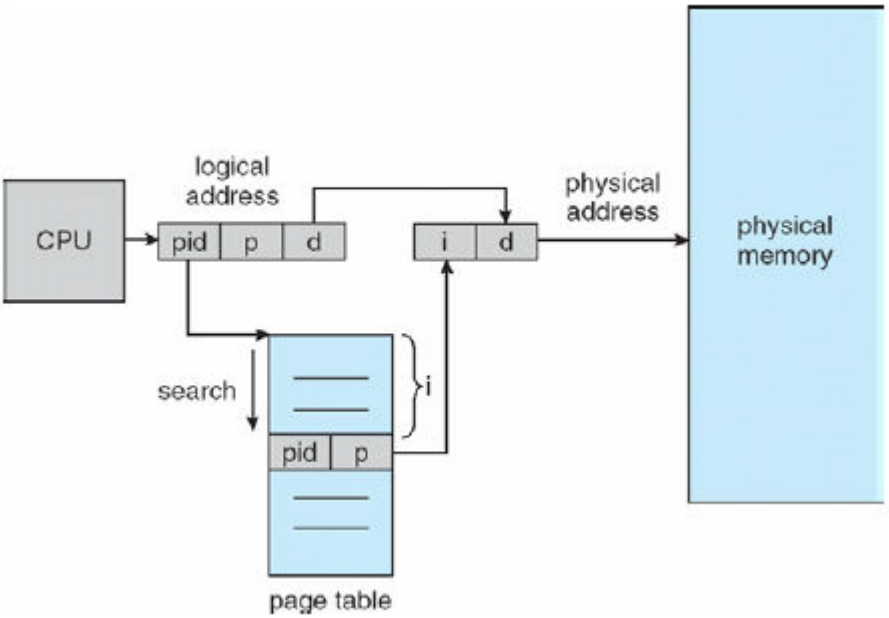
- **Hashed page tables:** A common approach for handling address spaces larger than 32 bits is to use a **hashed** page table, with the hash value being the virtual page number. Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions). Each element consists of three fields: (1) the virtual page number, (2) the value of the mapped page frame, and (3) a pointer to the next element in the linked list.

  The algorithm works as follows: 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. This scheme is shown in the following figure.



Hashed page table.

- **Inverted page tables:** An inverted page table has one entry for each real page (or frame) of memory. Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page. Thus, only one page table is in the system, and it has only one entry for each page of physical memory. The following diagram shows the operation of an inverted page table.



Inverted page table.

Each virtual address in the system consists of a triple

<process-id, page-number, offset>.

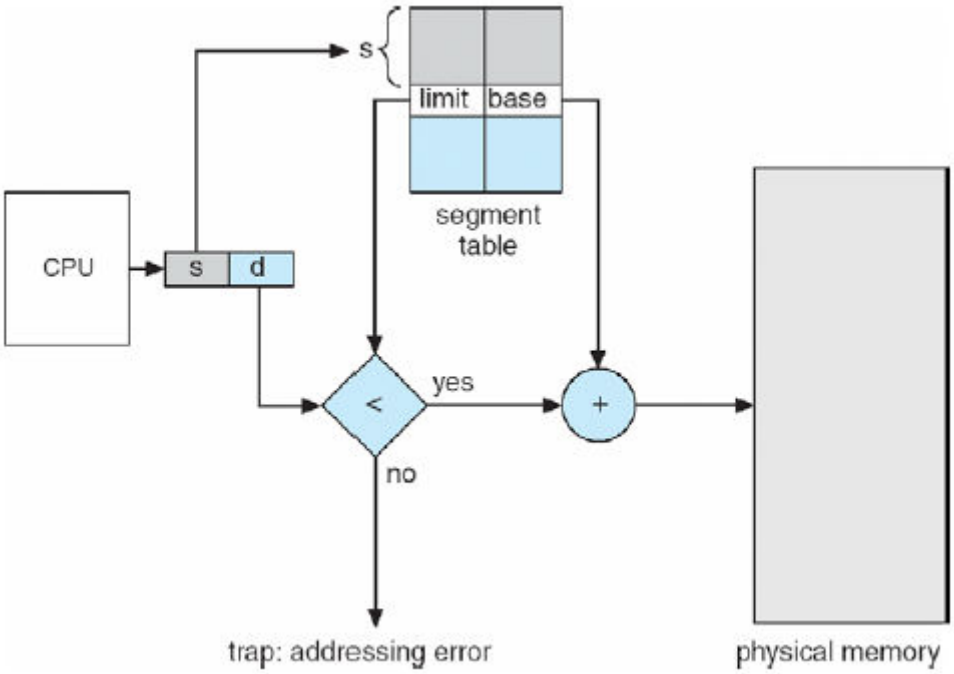
Each inverted page-table entry is a pair <process-id, page-number> where the process-id assumes the role of the address-space identifier. When a memory reference occurs, part of the virtual address, consisting of <process-id, pagenumber>, is presented to the memory subsystem. The inverted page table is then searched for a match. If a match is found—say; at entry /—then the physical address <*i,* or'fset> is generated. If no match is found, then an illegal address access has been attempted.

Segmentation: Segmentation is the process of dividing a program into fixed sized blocks called segments. A Segment is called as a logical grouping of information such as subroutins, arrays, atructues etc…

Basic Method: A logical address space is a collection of segments. Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. segments are numbered and are referred to by a segment number, rather than by a segment name. Tluis, a logical address consists of a *two tuple:*

< segment-number, offset >.

Hardware: the following diagram represents segmentation hardware.

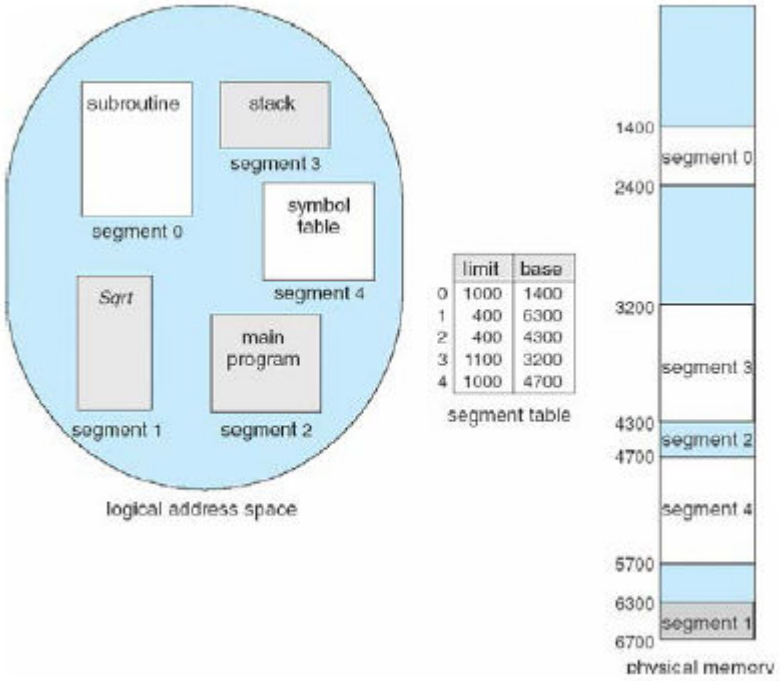


Segmentation hardware.

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. 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, whereas the segment limit specifies the length of the segment.

The offset *d* of the logical address must be between 0 and the segment limit. If it is not, a trap message will be sent to the operating system. 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, We have five segments numbered from 0 through 4. The segments are stored in physical memory as shown below.



Exampie of segmentation.

The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit). For example, segment

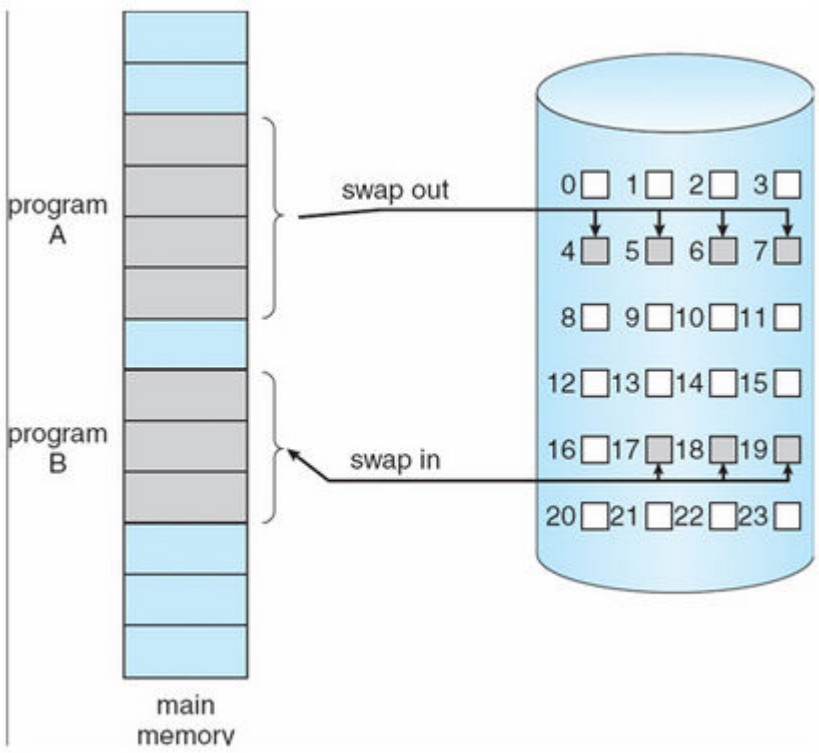
2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353. A reference to segment 3r byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052. A reference to byte 1222 of segment 0 would result in a trap to the operating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

Differences between Segmentation and Paging:

S.No	Segmentation	Paging
1	Program is divided into variable size segments.	Program is divided into fixed size pages.
2	User is responsible for dividing program.	Operating system take care of dividing program.
3	Segmentation is slower than paging.	Paging is faster than segmentation.
4	Segmentation is visible to the user	Paging is invisible to the user.
5	Segmentation eliminates internal fragmentation.	Paging suffers from internal fragmentation.
6	Segmentation suffers from external fragmentation	Paging eliminates internal fragmentation.
7	Processor uses page number, offset to calculate absolute address.	Processor uses segment number, offset to calculate absolute address.

Virtual Memory: Virtual Memory alloews execution of partially loaded processes. Virtual memory can be implemented by using Demand Paging.

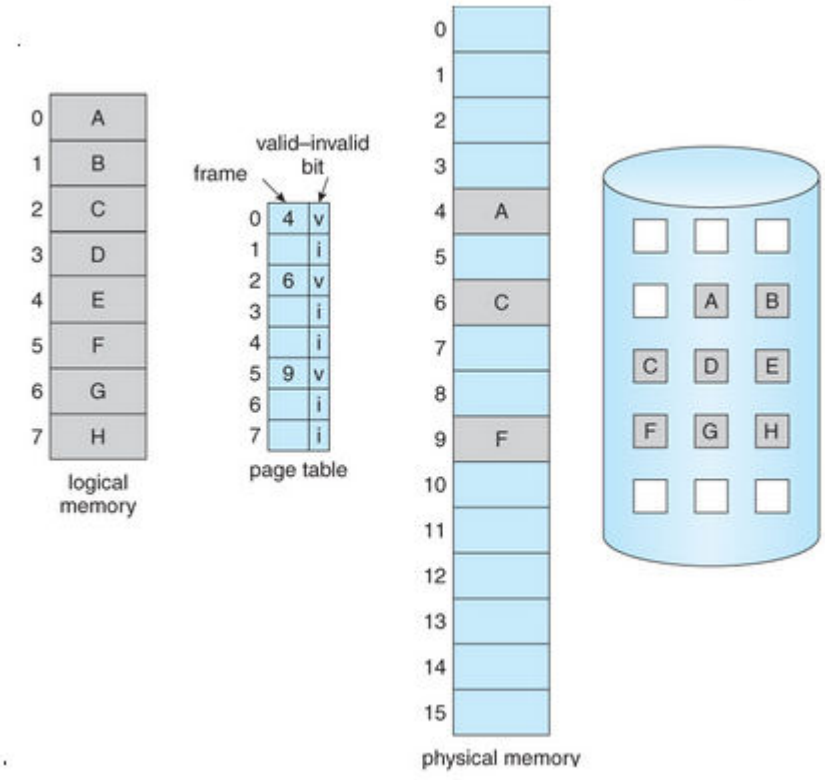
Demand Paging: In Demand Paging, pages are only loaded when they are demanded during program execution; pages that are never accessed are thus never loaded into physical memory. A demand-paging system is similar to a paging system with swapping as shown below. Here processes reside in secondary memory (usually a disk). To execute a process, swap it into memory. Rather than swapping the entire process into memory, however, use a **lazy swapper.** A lazy swapper never swaps a page into memory unless that page will be needed.



Transfer of a paged memory to contiguous disk space.

When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those necessary pages into memory. Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap rime and the amount of physical memory needed.

With this scheme, we need some form of hardware support to distinguish between the pages that are in memory and the pages that are on the disk. The valid-invalid bit scheme is used as shown below.



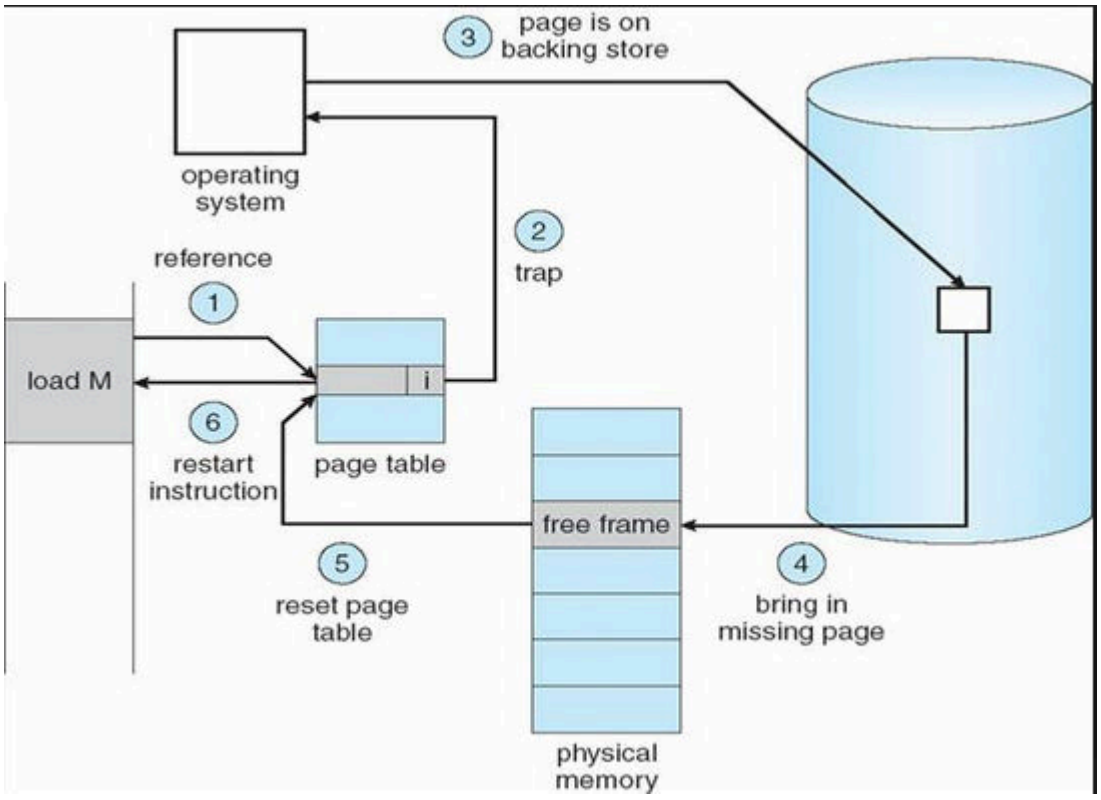
Page table when some pages are not in main memory.

When this bit is set to "valid/" the associated page is both legal and in memory. If the bit is set to "invalid," the page either is not valid (that is, not in the logical address space of the process) or is valid but is currently on the disk.

Access to a page marked invalid causes a **page-fault trap.** his trap is the result of the operating system's failure to bring the desired page into memory. The procedure for handling this page fault is as shown below.

- First check whether the reference was a valid or an invalid memory access.
- If the reference was invalid, terminate the process. If it was valid, but not yet brought in that page, now page it in.
- Find a free frame
- Schedule a disk operation to read the desired page into the newly allocated frame.
- When the disk read is complete, modify the internal table kept with the process and the page table to indicate that the page is now in memory.
- Restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

The following diagram shows steps for handling Page fault.



Steps in handling a page fault.

A crucial requirement for demand paging is the need to be able to restart any instruction after a page fault. A page fault may occur at any memory reference. If the page fault occurs on the instruction fetch, restart byfetching the instruction again. If a page fault occurs during fetching an operand, fetch and decode the instruction again and then fetch the operand.

Performance of Demand Paging: Demand paging can significantly affect the performance of a computer system. This can be calculated in terms of effective access time.

Let p be the probability of a page fault ($0 \le p \le 1$). The effective access time is then

effective access time = $(1 - p) * ma + p *$ page fault time.

Where, ma is memory access time.

Pafefault time is the time waited by CPU due to page fault.

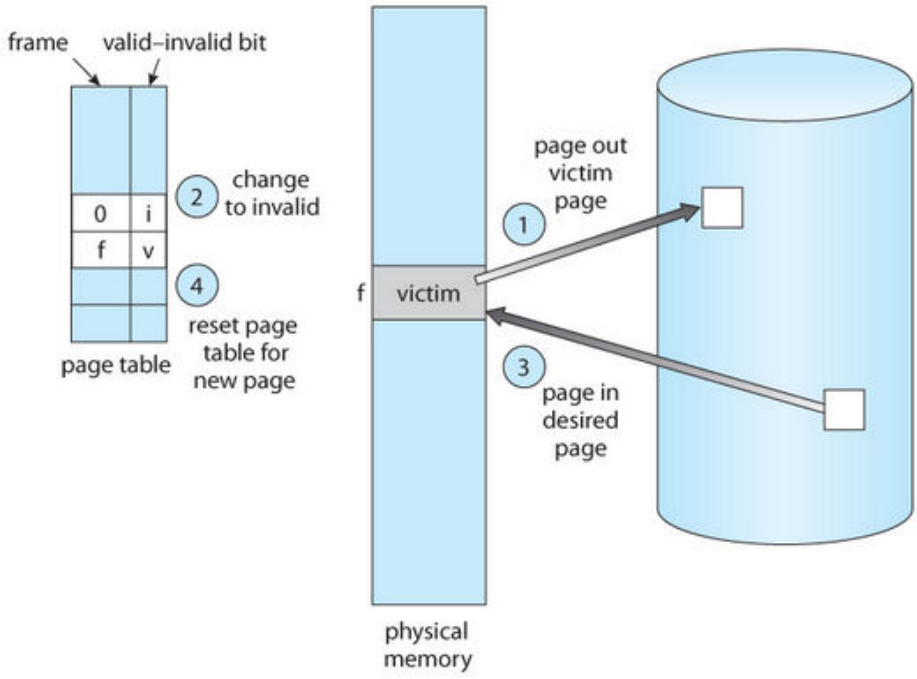
Effective access time is directly promotional to the page fault rate. It is important to keep the page fault rate low in a demand paging system. Otherwise, the effective access time increases, skowing process execution dramatically.

Advantages of Demand Paging:

1. Large virtual memory
2. More efficient use of memory
3. There is no limit on degree of multiprogramming.

Page Replacement Algorithms: Page replacement policy deals with the selection of a page in memory to be replaced when a new page must be brought in. When a page fault occurs, operating system performs the following.

- Find the location of the desired page on the disk.
- Find a free frame:
  - If there is a free frame, use it.
  - If there is no free frame, use a page-replacement algorithm toselect a victim frame.
  - Write the victim frame to the disk; change the page and frame tables accordingly.
- Read the desired page into the newly freed frame; change the page and frame tables.
- Restart the user process.



Page replacement

The following are different types of Page Replacement algorithms used to select victim page.

1. **FIFO Page Replacement:** The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. This technique, creates a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.
   For example, consider the following reference string with 3 frames.

7, 0,1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2,1, 2, 0, 1, 7, 0,1

Initial	E E E	---------
After inserting 7	7 E E	Page Fault
After inserting 0	7 0 E	Page Fault
After inserting 1	7 0 1	Page Fault
After inserting 2	2 0 1	Page Fault
After inserting 0	2 0 1	--------

After inserting 3	2 3 1	Page Fault
After inserting 0	2 3 0	Page Fault
After inserting 4	4 3 0	Page Fault
After inserting 2	4 2 0	Page Fault
After inserting 3	4 2 3	Page Fault
After inserting 0	0 2 3	Page Fault
After inserting 3	0 2 3	----------
After inserting 2	0 2 3	----------
After inserting 1	0 1 3	Page Fault
After inserting 2	0 1 2	Page Fault
After inserting 0	0 1 2	----------
After inserting 1	0 1 2	----------
After inserting 7	7 1 2	Page Fault
After inserting 0	7 0 2	Page Fault
After inserting 1	7 0 1	Page Fault

Total number of page faults: 15

The first three references (7,0,1) cause page faults and are brought into these empty frames. The next reference (2) replaces page 7, because page 7 was brought in first. Since 0 is the next reference and 0 is already in memory, we have no fault for this reference. The first reference to 3 results in replacement of page 0, since it is now first in line. Because of this replacement, the next reference, to 0, will fault. Page 1 is then replaced by page 0. This process continues till last page.

2. **LRU Page Replacement**: The key distinction between the FIFO and OPT algorithms (other than looking backward versus forward in time) is that the FIFO algorithm uses the time when a page was brought into memory, whereas the OPT algorithm uses the time when a page is to be *used.* LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time.
   For example, consider the following reference string with 3 frames.

7, 0,1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2,1, 2, 0, 1, 7, 0,1

Initial	E E E	---------
After inserting 7	7 E E	Page Fault
After inserting 0	7 0 E	Page Fault
After inserting 1	7 0 1	Page Fault
After inserting 2	2 0 1	Page Fault
After inserting 0	2 0 1	----------
After inserting 3	2 0 3	Page Fault
After inserting 0	2 0 3	----------
After inserting 4	4 0 3	Page Fault
After inserting 2	4 0 2	Page Fault
After inserting 3	4 3 2	Page Fault
After inserting 0	0 3 2	Page Fault
After inserting 3	0 3 2	----------
After inserting 2	0 3 2	----------
After inserting 1	1 3 2	Page Fault
After inserting 2	1 3 2	
After inserting 0	1 0 2	Page Fault
After inserting 1	1 0 2	----------
After inserting 7	1 0 7	Page Fault

After inserting 0	1 0 7	----------
After inserting 1	1 0 7	----------

Total number of page faults: 12

The LRU algorithm produces 12 faults. Notice that the first 5 faults are the same as those for optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used. When it then faults for page *2,* the LRU algorithm replaces page 3, since it is now the least recently used of the three pages in memory. Despite these problems, LRU replacement with 12 faults is much better than FIFO replacement with 15.

3. **Optimal Page Replacement:** An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms. In this technique, Replace the page that will not be used for the longest period of time. Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

   For example, consider the following reference string with 3 frames.

   7, 0,1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2,1, 2, 0, 1, 7, 0,1

Initial	E E E	---------
After inserting 7	7 E E	Page Fault
After inserting 0	7 0 E	Page Fault
After inserting 1	7 0 1	Page Fault
After inserting 2	2 0 1	Page Fault
After inserting 0	2 0 1	----------
After inserting 3	2 0 3	Page Fault
After inserting 0	2 0 3	----------
After inserting 4	2 4 3	Page Fault
After inserting 2	2 4 3	----------
After inserting 3	2 4 3	----------
After inserting 0	2 0 3	Page Fault
After inserting 3	2 0 3	----------
After inserting 2	2 0 3	----------
After inserting 1	2 0 1	Page Fault
After inserting 2	2 0 1	----------
After inserting 0	2 0 1	----------
After inserting 1	2 0 1	----------
After inserting 7	7 0 1	Page Fault
After inserting 0	7 0 1	----------
After inserting 1	7 0 1	----------

Total number of page faults: 09

The first three references cause faults that fill the three empty frames. The reference to page 2 replaces page 7, because 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With only nine page faults, optimal replacement is much better than other algorithms.

4. **Counting-Based Page Replacement(LFU):** The **least frequently used (LFU) page-replacement algorithm** requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count. A problem arises, however, when a page is used heavily during the initial phase of a process but then is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed.

For example, consider the following reference string with 3 frames.

7, 0,1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2,1, 2, 0, 1, 7, 0,1

Initial	E E E	---------
After inserting 7	7 E E	Page Fault
After inserting 0	7 0 E	Page Fault
After inserting 1	7 0 1	Page Fault
After inserting 2	2 0 1	Page Fault
After inserting 0	2 0 1	----------
After inserting 3	2 0 3	Page Fault
After inserting 0	2 0 3	----------
After inserting 4	4 0 3	Page Fault
After inserting 2	4 0 2	Page Fault
After inserting 3	3 0 2	Page Fault
After inserting 0	3 0 2	----------
After inserting 3	3 0 2	----------
After inserting 2	3 0 2	----------
After inserting 1	3 1 2	Page Fault
After inserting 2	3 1 2	----------
After inserting 0	0 1 2	Page Fault
After inserting 1	0 1 2	----------
After inserting 7	0 1 7	Page Fault
After inserting 0	0 1 7	----------
After inserting 1	0 1 7	----------

Total number of page faults: 11

Allocation of Frames: Consider a single-user system with 128 KB of memory composed of pages 1 KB in size. This system has 128 frames. The operating system may take 35 KB, leaving 93 frames for the user process. Under pure demand paging, all 93 frames would initially be put on the free-frame list. When a user process started execution, it would generate a sequence of page faults. The first 93 page faults would all get free frames from the free-frame list. When the free-frame list was exhausted, a page-replacement algorithm would he used to select one of the 93 in-memory pages to be replaced with the 94th, and so on. When the process terminated, the 93 frames would once again be placed on the free-frame list.

There are many variations on this simple strategy.

- **Minimum Number of Frames:** The minimum number of frames is defined by the computer architecture.
- **Maximum Number of Frames:** Whereas the minimum number of frames per process is defined by the architecture, the maximum number is defined by the amount of available physical memory.
- **Equal Allocation:** Whereas the minimum number of frames per process is defined by the architecture, the maximum number is defined by the amount of available physical memory.

  Consider a system with a 1-KB frame size. If a small student process of 10 KB and an interactive database of 127 KB are the only two processes running in a system with 62 free frames, it does not make much sense to give each process 31 frames. The student process does not need more than 10 frames, so the other 21 are, strictly speaking, wasted.
- **Proportional Allocation:** To solve this problem, use **proportional allocation**, which allocate available memory to each process according to its size. Let the size of the virtual memory for process P_i be S_i and define

  $$S = \sum S_i$$

Then, if the total number of available frames is *m,* allocate a_i frames to process P_i where a_i is approximately

$$a_i = S_i / S * m$$

For proportional allocation, split 62 frames between two processes, one of 10 pages and one of 127 pages, by allocating 4 frames and *57* frames, respectively, since

10/137 x 62 = 04, and
127/137 x62 = 57.

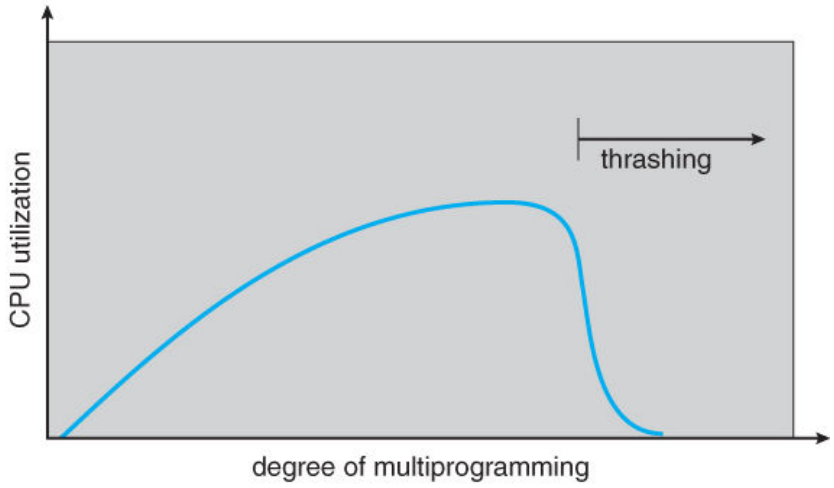
In this way, both processes share the available frames according to their "needs," rather than equally.

- **Global versus Local Allocation:** With multiple processes competing for frames, classify page-replacement algorithms into two broad categories: **global replacement** and **local replacement.** Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process; that is, one process can take a frame from another. Local replacement requires that each process select from only its own set of allocated frames.

Thrashing: The phenomenon of execessively moving pages between memory and external disk is called as Thrashing. A process is thrashing if it is spending more time paging than executing.

Cause of Thrashing: The operating system monitors CPU utilization. If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system. A global page-replacement algorithm is used; it replaces pages without regard to the process to which they belong. Now suppose that a process enters a new phase in its execution and needs more frames. It starts faulting and taking frames away from other processes.

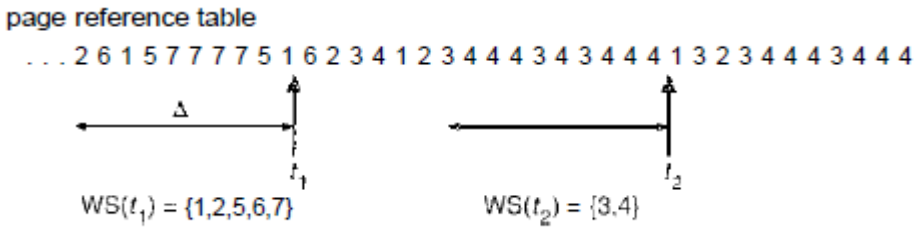
The CPU scheduler sees the decreasing CPU utilization and *increases* the degree of multiprogramming as a result. The new process tries to get started by taking frames from running processes, causing more page faults and a longer queue for the paging device. As a result, CPU utilization drops even further, and the CPU scheduler tries to increase the degree of multiprogramming even more. Thrashing has occurred, and system throughput plunges. The pagefault rate increases tremendously As a result, the effective memory-access time increases. No work is getting done, because the processes are spending all their time paging.



To prevent thrashing, provide a process with as many frames as it needs. But how do we know how many frames it "needs"? There are several techniques. The working-set strategy starts by looking at how many frames a process is actually using.

Working-Set Model: The idea is to examine the most recent ▲ page references. The set of pages in the most recent A page references is the 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 A time units after its last reference.

For example, given the sequence of memory references shown in Figure



If ▲= 10 memory references, then the working set at time *t* is {1, *2,* 5, 6, *7}*. By time *h,* the working set has changed to {3, 4}. The accuracy of the working set depends on the selection of A. If A is too small, it will not encompass the entire locality; if A is too large, it may overlap several localities. In the extreme, if A 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, *WSSj,* 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 *WSSj* 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.

FREQUENTLY ASKED QUESTIONS

1. Distinguish between Logical and Physical address space?
2. What is the purspose of Paging?
3. What is a Virtual Memory? Discuss the benefits of virtual memory technique?
4. What is Thrashing? What can the system do to eliminate this problem?
5. What is a Pagefault? Explain the steps involved in handling a pagefault with a neat sketch?
6. Explain the need of page replacement and explain with an example?
7. Suggest the best algorithm for the given reference string.
   7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 with three page frames.
8. Compute the number of page faults for optimal page replacement strategy for the given reference string 1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 7, 8, 9, 5, 4, 5, 4, 2 with 4 page frames.
9. Consider the following reference string:
   1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6
   How many page faults would occur for the optimal page replacement algorithm, assuming three frames and all frames are initially empty.
10. Discuss the hardware support required to support demand paging?
11. What factors effecting the performance of demand paging? Compute the performance of demand paging when page fault service rate is 8 m/sec and memory access time is 200nsec.
12. Explain how demand paging effects the performance of a computer system?
13. Explain the difference between internal and external fragmentation?
14. Discuss various issues related to the allocation of frames to processes?

Syllubus: **Principles of deadlock** – system model, deadlock characterization, deadlock prevention, detection and avoidance, recovery form deadlock.

Deadlock: In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; and if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock.

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources are partitioned into several types, each consisting of some number of identical instances.

Under the normal mode of operation, a process may utilize a resource in only the following sequence:

- **Request:** If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
- **Use:** The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
- **Release:** The process releases the resource.

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

Necessary Conditions: A deadlock 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 non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. **Hold and wait.** A process must be holding at least one resource and 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 only voluntarily by the process holding it, after that process has completed its task.
4. **Circular wait:** A set *{P$, Pi, ..., Pn* of waiting processes must exist such that *P-0* is waiting for a resource held by *P\, P* is waiting for a resource held by P?, •••, P.,--i is waiting for a resource held by *Pn,* and *P,,* is waiting for a resource held by Pn.

Resource-Allocation Graph: Deadlocks can be described more precisely in terms of a directed graph called a **system resource-allocation** graph. This graph consists of a set of vertices *V* and a set of edges *E.* The set of vertices *V* is partitioned into two different types of nodes: P={P1, P2, P3…..Pn}, the set consisting of all the active processes in the system, and *R = {R1, R2, • • • Rm},* the set consisting of all resource types in the system.

A directed edge from process P_i to resource type *Rj* is denoted by $P_i \rightarrow Rj$; it signifies that process P_i has requested an instance of resource type Rj and is currently waiting for that resource. A directed edge from resource type *Rj* to process P_i is denoted by $Rj \rightarrow P_i$; it signifies that an instance of resource type *Rj* has been allocated to process P_i. A directed edge $P_i \rightarrow Rj$ is called a **request edge;** a directed edge $Rj \rightarrow P_i$; is called an **assignment edge.**

Pictorially, represent each process P_i as a circle and each resource type *Rj* as a rectangle. Since resource type *Rj* may have more than one instance, represent each such instance as a dot within the rectangle.

When process P_i requests an instance of resource type *Rj,* a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is *instantaneously* transformed to an assignment edge. When the process no longer needs access to the resource, it releases the resource; as a result, the assignment edge is deleted.

Consider the following resource allocation graph, containing

➔ The sets P, *R,* and £:

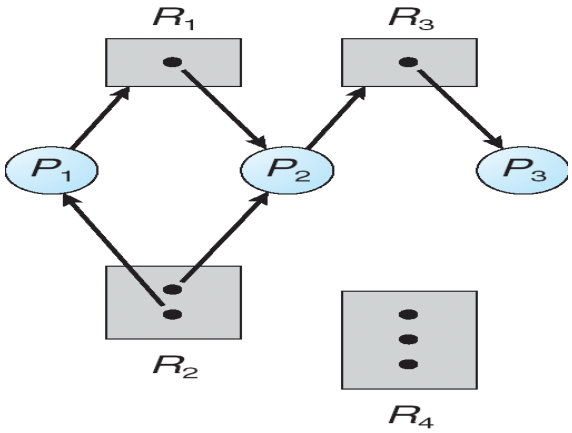
- P={P1, P2, ……PN}
- R= {R1, R2 … RM}
- £ = {P1→ R1, P2→ R3, R1→P2, R2→ P2, R2→P1,….. R3 -> P3 }

➔ 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

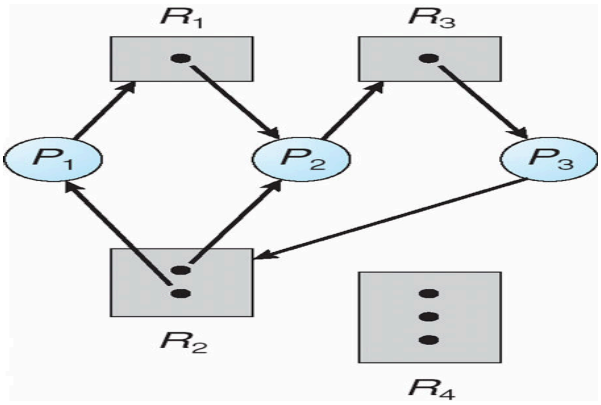
➔ Process states:

- 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.



If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

Suppose that process P3 requests an instance of resource type *R2.* Since no resource instance is currently available, a request edge P3→R2 is added to the graph as shown below.



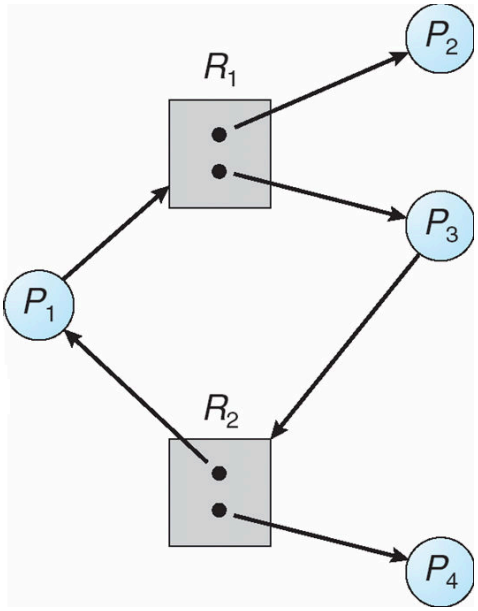
At this point, two minimal cycles exist in the svstem:

P1→R1→P2→R3→P3→R2→P1

P2→R3→P3→R2→P2

Here, 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 P2 to release resource R2. In addition, process P1 is waiting for process P2 to release resource R1.

Now consider the resource-allocation graph as shonw below, having a cycle (P1→R1→P3→R2→P1)with no dead locks. Observe that process P4 may release its instance of resource type R2. That resource can then be allocated to P3, breaking the cycle.



Deadlock Prevention: By ensuring that at least one of the following conditions cannot hold, we can *prevent* the occurrence of a deadlock.

1. **Mutual Exclusion:** The mutual-exclusion condition must hold for nonsharable resources. For example, a printer cannot be simultaneously shared by several processes. Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A process never needs to wait for a sharable resource.
2. **Hold and Wait:** To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources. For this, release all the resources held by it whenever it requests a resource which is not available.
3. No Preemption: this condition can be prevented in several ways.
    - If a process holding certain resources is denied a further request. That process must release its original resources and if necessary request them again, together with an additional resource.
    - If a process requests a resource that is currently held by another process, the operating system may preempt the second process and require it to release its resources.
4. Circular Wait: One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration. For example, process hold resource type R1, then it can only request resource of class R2 or R3 etc…

Advantages of Deadlock Prevention:

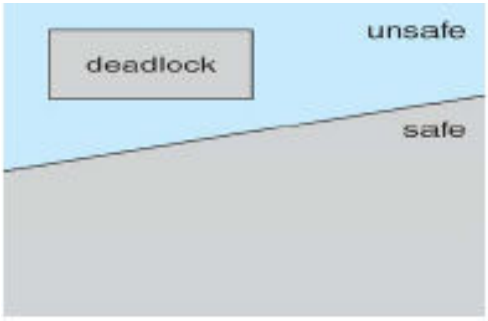
- No preemption necessary
- Needs no runtime computation
- Feasible to enforce via compile time checks.
- Works well processes that perform a single burst of activity.

Disadvantages of Deadlock Prevention:

- Inefficient
- Delays process initiation
- Disallows incremental resource requests.

Deadlock Avoidance: A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circularwait 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: A state is *safe* if the system can allocate resources to each process in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence. A sequence of processes <P1, P2, ..., Pn> is a safe sequence for the current allocation state if, for each *Pi,* the resource requests that Pi can be satisfied by the currently available resources plus the resources held by all *Pi,* with j < i. In this situation, if the resources that *Pi* needs are not immediately available, then Pi can wait until all *Pj* have finished. When they have finished, Pj can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When Pi terminates, Pi+l can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be *unsafe.*



Safe, unsafe, and deadlock state spaces.

A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks, however. An unsafe state *may* lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe (and deadlocked) states. In an unsafe state, the operating system cannot prevent processes from requesting resources such that a deadlock occurs: The behavior of the processes controls unsafe states.

Consider a system with 12 magnetic tape drives and three processes: P0, P1, and P2. Process P0 requires 10 tape drives, process P1 may need as many as 4 tape drives, and process P2 may need up to 9 tape drives. Suppose that, at time *to,* process P0 is holding 5 tape drives, process *P1* is holding *2* tape drives, and process *P2* is holding 2 tape drives. (Thus, there are 3 free tape drives.)

	Maximum Needs	Current Needs
Po	10	5
P_1	4	2
P_2	9	2

At time T0, the system is in a safe state. The sequence < P1, *P0, P2*> satisfies the safety condition. Process P1 can immediately be allocated all its tape drives and then return them (the system will then have 5 available tape drives); then process P0 can get all its tape drives and return them (the system will then have 10 available tape drives); and finally process P2 can get all its tape drives and return them (the system will then have all 12 tape drives available).

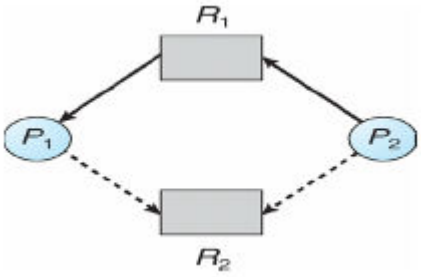
A system can go from a safe state to an unsafe state. Suppose that, at time T1, process *P2* requests and is allocated one more tape drive. The system is no longer in a safe state. At this point, only process P1 can be allocated all its tape drives. When it returns them, the system will have only 4 available tape drives. Since process P0 is allocated 5 tape drives but has a maximum of 10, it may request 5 more tape drives. Since they are unavailable, process P0 must wait. Similarly, process P2 may request an additional 6 tape drives and have to wait, resulting in a deadlock. Here, mistake was in granting the request from process P2 for one more tape drive. If we had made P2 wait until either of the

other processes had finished and released its resources, then we could have avoided the deadlock. The following are two different types of deadlock avoidance algorithms.

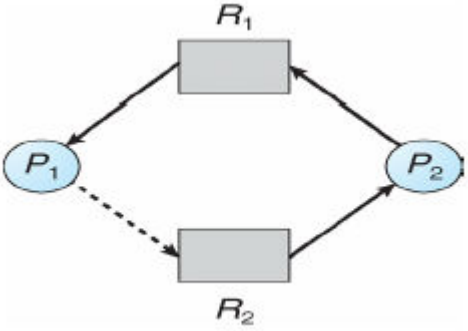
1. **Resource-Allocation-Graph Algorithm:** A claim edge Pi → *Rj* indicates that process Pi may request resource *Rj* at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line. When process Pi requests resource *Rj,* the claim edge Pi→ *Rj* is converted to a request edge. Similarly, when a resource *Rj* is released by *Pj,* the assignment edge *Rj* → Pi is reconverted to a claim edge *Pi* → *Rj.*

   Suppose that process Pi requests resource *Rj.* The request can be granted only if converting the request edge Pi → *Rj* to an assignment edge *Rj* → *Pi* does not result in the formation of a cycle in the resource-allocation graph. An algorithm for detecting a cycle in this graph requires an order of *n2* operations, where *n* is the number of processes in the system.

   If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state. Therefore, process *Pi* will have to wait for its requests to be satisfied. For example,



Resource-allocation graph for deadlock avoidance.



An unsafe state in a resource-allocation graph

   Suppose that P2 requests *R2.* Although *R2* is currently free, cannot allocate it to P2, since this action will create a cycle in the graph. A cycle indicates that the system is in an unsafe state. If P1 requests *R2,* and *P2* requests *R1,* then a deadlock will occur.

2. **Banker's Algorithm:** This algorithm is suitable for resource types with multiple instances. When, a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

   Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. Let *n* be the number of processes in the system and *m* be the number of resource types. Data structures are

   - **Available:** A vector of length *m* indicates the number of available resources of each type. If *Available[j]* equals *k,* there are *k* instances of resource type Rj available.
   - **Max:** An *n* x *m* matrix defines the maximum demand of each process. If Max[i][j] equals *k,* then process *Pi* may request at most *k* instances of resource type Rj.
   - **Allocation**: An *n* x m matrix defines the number of resources of each type currently allocated to each process. If *Allocation[i][j]* equals *k,* then process *Pi* is currently allocated *k* instances of resource type Rj.

- **Need:** An *n* x *m* matrix indicates the remaining resource need of each process. If *Need[i][j]* equals *k,* then process Pi may need *k* more instances of resource type *Rj* to complete its task. Note that *Need*[i][j] equals *Max[i][j]- Allocntion[i][j]*.

Safety Algorithm:

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize: *Work* = *Available*
   Finish [*i*] = *false* for *i* = 0, 1, …, *n*- 1
2. Find an index *i* such that both: *Finish* [*i*] = *false* and $Need_i <= Work$
   If no such *i* exists, go to step 4.
3. $Work=Work+Allocation_i$
   Finish[*i*]=*true*
   go to step 2.
4. If *Finish* [*i*] == true for all *i*, then the system is in a safe state

Resource-Request Algorithm: Let $Request_i$ be the request vector for process P_i. If *Request[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 $Request_i < Need,$ go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If $Request_i < Available,$ go to step 3. Otherwise, P_i must wait, since the resources are not available.
3. $Available = Available – Reques_i;$
   $Allocation_i = Allocation_i + Request_i$
   $Need_i = Necd_i – Request_i$

Consider a system with five processes *P0* through P4 and three resource types *A, B,* and C. Resource type A has 10 instances, resource type *B* has 5 instances, and resource type *C* has 7 instances. Suppose that, at time T_0 the following snapshot of the system has been taken:

	Allocation	*Max*	*Available*
	ABC	*A B C*	*ABC*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

The content of the matrix *Need* is defined to be *Max* - *Allocation* and is as follows:

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

Suppose now that process *P0* requests 7 additional instance of resource type *A* , 4 instances of resource type B and 4 instances of resource type B. Now check for Need <= Work(Avaialable) for all the values of i.

→For process P0, Since (7, 4, 3) <= (3, 3, 2) is false. Make process P0 to wait.

→For process P1, Since (1, 2, 2) <= (3, 3, 2) is true. Process it. After completion of P1, it releases all the releases it held. So, available is updated as follows:
Aaialable = available + $Allocation_i$

Aaialable = (3, 3, 2) + (2, 0, 0) = (5, 3, 2)

→For process P2, Since (6, 0, 0) <= (5, 3, 2) is False. Make P2 to wait.

→For process P3, Since (0, 1, 1) <= (5, 3, 2) is True. Process it. After completion of P3, it releases all the resources it held. So, available is updated as follows.

Aaialable = (5, 3, 2) + (2, 1, 1) = (7, 4, 3)

→For process P4, Since (4, 3, 1) <= (5, 3, 4) is True. Process it. After completion of P4, it releases all the resources it held. So, available is updated as follows.

Aaialable = (7, 4, 3) + (0, 0, 2) = (7, 4, 5)

→For process P0, Since (7, 4, 3) <= (7, 4, 5) is True. After completion of P0, it releases all the resources it held. So, available is updated as follows.

Aaialable = (7, 4, 5) + (0, 1, 0) = (7, 5, 5)

→For process P2, Since (6, 0, 0) <= (7, 5, 5) is True. After completion of P2, it releases all the resources it held. So, available is updated as follows.

Aaialable = (7, 5, 5) + (3, 0, 2) = (10, 5, 7)

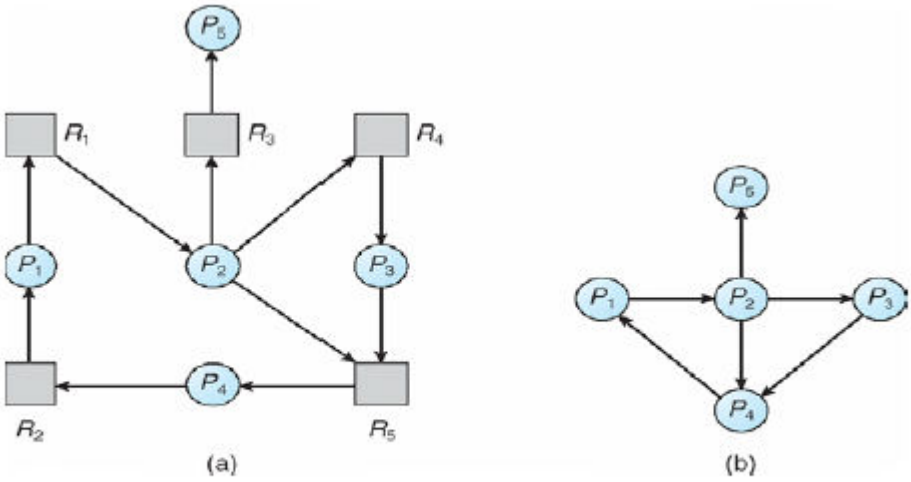
Safe sequence is <P1, P3, P4, P0, P2>

Deadlock Detection: If a system does not employ either a deadlock-prevention or a deadlockavoidance algorithm, then a deadlock situation may occur. In this environment, the system must provide:

- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock.

The following are different types of deadlock detection algorithms.

1. **Single Instance of Each Resource Type(Wait-For Graph):** Wait-For graph is an sibling of Resource-Allocation Graph algorithm. More precisely, an edge from P_i to P_j in a wait-for graph implies that process P_i is waiting for process P_j to release a resource that P_i needs. An edge $P_i \rightarrow P_j$ exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges $P_i \rightarrow R_j$ and $R_j \rightarrow P_j$ for some resource. For example,



Resource-Allocation Graph Corresponding wait-for graph

A deadlock exists in the system if and only if the wait-for graph contains a cycle. To detect deadlocks, the system needs to *maintain* the wait-for graph and periodically *invoke an*

algorithm that searches for a cycle in the graph. An algorithm to detect a cycle in a graph requires an order of n^2 operations, where *n* is the number of vertices in the graph.

2. **Several Instances of a Resource Type (Banker's Algorithm):** The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type. The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm.
   - **Available:** A vector of length *m* indicates the number of available resources of each type.
   - **Allocation:** An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.
   - **Request:** An *n* x m matrix indicates the current request of each process. If *Request[i][j]* equals *k,* then process P_i is requesting *k* more instances of resource type *Rj.*

Algorithm:

1. Let *Work* and *Finish* be vectors of length *in* and *n,* respectively. Initialize *Work = Available.* For *i* = 0 , 1 , . . . , *n-1,* if $Allocation_i \neq 0$, then *Finish[i]= false;* otherwise, *Finisli[i] = true.*
2. Find an index *i* such that both
   a. *Finish[i] -=false*
   b. *Requesti < Work*
   If no such i exists, go to step 4.
3. *Work - Work + Allocation!*
   Finish[i] = true
   Go to step 2.
4. If *Finish[i] == false,* for some /', 0 < / < *n,* then the system is in a deadlocked state. Moreover, if *Finish[i] == false,* then process *P;* is deadlocked.

This algorithm requires an order of *in* x *n2* operations to detect whether the system is in a deadlocked state. consider a system with five processes *P0* through P4 and three resource types *A, B,* and C. Resource type *A* has seven instances, resource type B has two instances, and resource type C has six instances. Suppose that, at time To, we have the following resource-allocation state:

	Allocation	*Request*	*Available*
	A B C	***ABC***	***ABC***
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

For above request algorithm leaves system in safe state with safe sequence <P0, P1, P2, P3, P4> Suppose now that process *Pj* makes one additional request for an instance of type C. The *Request* matrix is modified as follows, which leaves system in unsafe state.

Request
ABC
0 0 0
2 0 2
0 0 1
1 0 0
0 0 2

Recovery From Deadlock: When a detection algorithm determines that a deadlock exists, several alternatives are available to recover from deadlock. One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually. Another possibility is to let the system *recover* from the deadlock automatically. There are two options for breaking a deadlock. One is simply to abort one or more processes to break the circular wait. The other is to preempt some resources from one or more of the deadlocked processes.

1. **Process Termination:**
   - **Abort all deadlocked processes.** This method clearly will break the deadlock cycle, but at great expense; the deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.
   - **Abort one process at a time until the deadlock cycle is eliminated.** This method incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.
2. **Resource Preemption:** To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

FREQUENTLY ASKED QUESTIONS

1. What are the necessary conditions for the occurrence of deadlock?
2. What is a deadlock? How to detect deadlock?
3. Explain Resource – allocation Grapg algorithm for deadlock avoidance?
4. What are the vrious methods for handling deadlocks?
5. Discuss various techniques to recover from deadlock?
6. Explain the various ways of aborting a process in order to eliminate deadlocks?
7. Explain abour deadlock avoidance algorithms in detail?
8. Is it possible to have a deadlock involving only a single process? Explain?
9. Write about Resource – Allocation Graph?
10. Discuss various methods for the prevention of deadlocks?
11. Consider the snapshot of the system processes P1, P2, P3, P4, P5, resources A, B, C, D.
    Allocation {0 0 1 2, 1 0 0 0, 1 3 5 4, 0 6 3 2, 0 0 1 4}
    Max {0 0 1 2, 1 7 5 0, 2 3 5 6, 0 6 5 2, 0 6 5 6}
    Available {1 5 2 0}
    What is the content of Need matrix? And find out is the system in safe state?
12. Write about characterization of deadlock by resource allocation graph?
13. State and explain two ways for handling deadlocks?
14. Consider the snapshot of the system processes P0, P1, P2, P3, P4, resources A, B, C. Allocation {0 1 0, 2 0 0, 3 0 3, 2 1 1, 0 0 2} Max {0 0 0, 2 0 2, 0 0 0, 1 0 0, 0 0 2} Available {0 0 0}
    What is the content of Need matrix? And find out is the system in safe state?

Syllubus: File system Interface- the concept of a file, Access Methods, Directory structure, File system mounting, file sharing, protection.

File System implementation- File system structure, allocation methods, free-space management **Mass-storage structure** overview of Mass-storage structure, Disk structure, disk attachment, disk scheduling.

File: A file is a named collection of related information that is recorded on secondary storage. Commonly, files represent programs and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free form, such as text files, or may be formatted rigidly. In general, a file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user.

The information in a file is defined by its creator. Many different types of information may be stored in a file—source programs, object programs, executable programs, numeric data, text, payroll records, graphic images, sound recordings, and so on. A file has a certain defined structure, which depends on its type. A *text* file is a sequence of characters organized into lines (and possibly pages). A *source* file is a sequence of subroutines and functions, each of which is further organized as declarations followed by executable statements. An *object* file is a sequence of bytes organized into blocks understandable by the system's linker. An *executable* file is a series of code sections that the loader can bring into memory and execute.

File Attributes: A file's attributes vary from one operating system to another but typically consist of these:

- Name: The symbolic file name is the only information kept in humanreadable form,.
- Identifier: This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.
- Type: This information is needed for systems that support different types of files.
- Location: This information is a pointer to a device and to the location of the file on that device.
- Size: The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
- Protection: Access-control information determines who can do reading, writing, executing, and so on.
- Time, date, and user identification: This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

File Operations: The following are different types of operations that can be performed on files:

- **Creating a file:** Two steps are necessary to create a file. First, space in the file system must be found for the file. Second, an entry for the new file must be made in the directory.
- **Writing a file:** To write a file, make a system call specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the file's location. The system must keep a *write* pointer to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.
- **Reading a file:** To read from a file, use a system call that specifies the name of the file and where (in memory) the next block of the file should be put. Again, the directory is searched for the associated entry, and the system needs to keep a *read* pointer to the location in the file where the next read is to take place. Once the read has taken place, the read pointer is updated. Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process **currentfile- position pointer.** Both the read and write operations use this same pointer, saving space and reducing system complexity.
- **Repositioning within** a **file:** The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value. Repositioning within a file need not involve any actual I/O. This file operation is also known as a file *seek.*
- **Deleting a file:** To delete a file, search the directory for the named file. Having found the associated directory entry, release all file space, so that it can be reused by other files, and erase the directory entry.

- **Truncating a file:** The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged—except for file length.

File Types: A common technique for implementing file types is to include the type as part of the file name. The name is split into two parts—a name and an *extension,* usually separated by a period character. The following table lists different types of files.

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

File structure: Four terms are in common use when discussing files:

- Field
- Record
- File
- Database

A **field** is the basic element of data. An individual field contains a single value, such as an employee's last name, a date, or the value of a sensor reading. It is characterized by its length and data type (e.g., ASCII string, decimal). Depending on the file design, fields may be fixed length or variable length. In the latter case, the field often consists of two or three subfields: the actual value to be stored, the name of the field, and, in some cases, the length of the field. In other cases of variable-length fields, the length of the field is indicated by the use of special demarcation symbols between fields.

A **record** is a collection of related fields that can be treated as a unit by some application program.

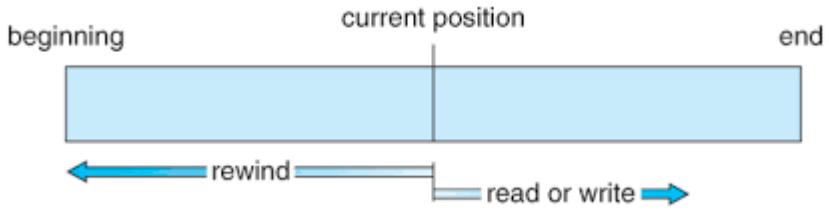
For example, an employee record would contain such fields as name, social security number, job classification, date of hire, and so on. Again, depending on design, records may be of fixed length or variable length. A record will be of variable length if some of its fields are of variable length or if the number of fields may vary. In the latter case, each field is usually accompanied by a field name. In either case, the entire record usually includes a length field.

A **file** is a collection of similar records. The file is treated as a single entity by users and applications and may be referenced by name. Files have file names and may be created and deleted. Access control restrictions usually apply at the file level. That is, in a shared system, users and programs are granted or denied access to entire files. In some more sophisticated systems, such controls are enforced at the record or even the field level. Some file systems are structured only in terms of fields, not records. In that case, a file is a collection of fields.

A **database** is a collection of related data. The essential aspects of a database are that the relationships that exist among elements of data are explicit and that the database is designed for use by a number of different applications. A database may contain all of the information related to an organization or project, such as a business or a scientific study. The database itself consists of one or more types of files. Usually, there is a separate database management system that is independent of the operating system, although that system may make use of some file management programs.

Access Methods: Information in the files must be accessed and read into computer memory when it is needed. The information in the files can be accessed in several ways.

1. **Sequential Access:** The simplest access method is sequential access. Information in the file is processed in order, one record after the other. This mode of access is by far the most common; for example, editors and compilers usually access files in this fashion.



Reads and writes make up the bulk of the operations on a file. A read operation—*read next*—reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location. Similarly, the write operation—*write next*—appends to the end of the file and advances to the end of the newly written data.

2. **Direct Access:** A file is made up of fixedlength **logical records** that allow programs to read and write records rapidly in no particular order. The direct-access method is based on a disk model of a file, since disks allow random access to any file block. For direct access, the file is viewed as a numbered sequence of blocks or records. Thus, we may read block 14, then read block 53, and then write block 7. There are no restrictions on the order of reading or writing for a direct-access file.

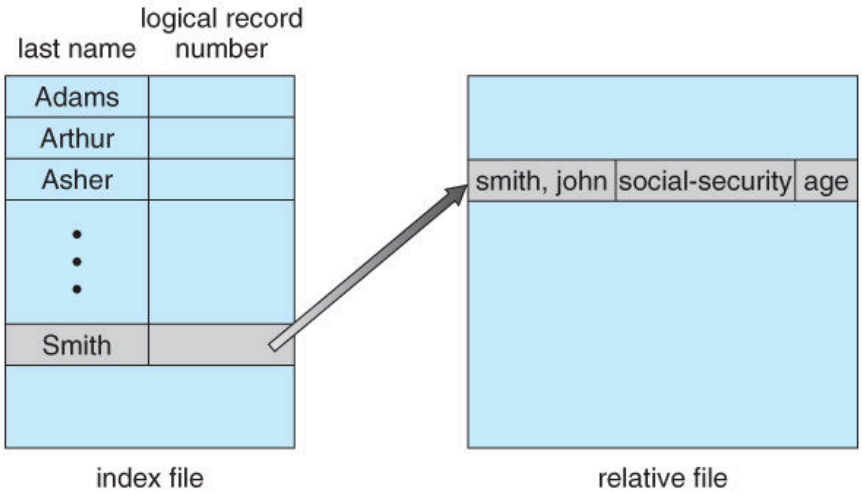
   Direct-access files are of great use for immediate access to large amounts of information. Databases are often of this type.

   For the direct-access method, the file operations must be modified to include the block number as a parameter. Thus, we have *read n,* where *n* is the block number, rather than *read next,* and *write n* rather than *write next.*

   The block number provided by the user to the operating system is normally a **relative block number.** A relative block number is an index relative to the beginning of the file. Thus, the first relative block of the file is 0, the next is 1, and so on, even though the actual absolute disk address of the block may be 14703 for the first block and 3192 for the second.

3. **Other Access Methods:** These methods generally involve the construction of an index for the file. The index, like an index in the back of a book, contains pointers to the various blocks. To find a record in the file, we first search the index and then use the pointer to access the file directly and to find the desired record.

For example, a retail-price file might list the universal product codes (UPCs) for items, with the associated prices. Each record consists of a 10-digit UPC and a 6-digit price, for a 16-byte record, if our disk has 1,024 bytes per block, we can store 64 records per block. A file of 120,000 records would occupy about 2,000 blocks (2 million bytes). By keeping the file sorted by UPC, we can define an index consisting of the first UPC in each block. This index would have 2,000 entries of 10 digits each, or 20,000 bytes, and thus could be kept in memory To find the price of a particular item, we can make a binary search of the index.



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

For example, IBM's indexed sequential-access method (ISAM) uses a small master index that points to disk blocks of a secondary index. The secondary index blocks point to the actual file blocks. The file is kept sorted on a defined key. To find a particular item, first make a binary search of the master index, which provides the block number of the secondary index. This block is read in, and again a binary search is used to find the block containing the desired record. Finally, this block is searched sequentially. In this way, any record can be located from its key by at most two direct-access reads.

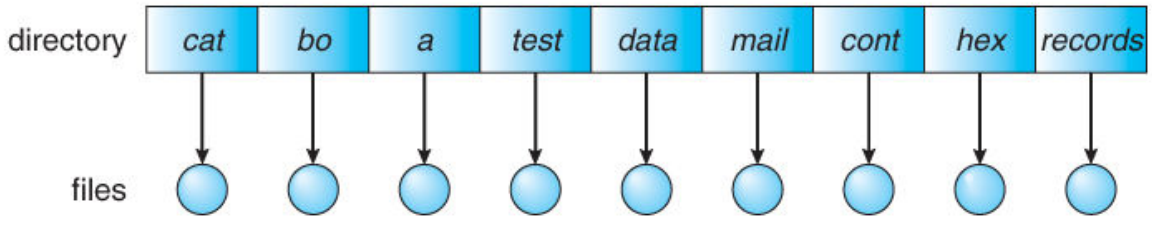
Directory Structure: To organize huge amount of data properly directories are used. The directory can be viewed as a symbol table that translates file names into their directory entries.

The following are different types of directory operations.

1. **Search** for a file: We cansearch a directory structure to find the entry for a particular file. Since files have symbolic names and similar names may indicate a relationship between files, we may want to be able to find all files whose names match a particular pattern.
2. **Create a file:** New files can be created and added to the directory.
3. Delete a file: When a file is no longer needed, we can remove it from the directory.
4. List a directory: We canlist the files in a directory and the contents of the directory entry for each file in the list.
5. Rename a file: Because the name of a file represents its contents to its users, we can 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.
6. Traverse the file system: We can 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.

The following are most common schemes for defining the logical structure of a directory.

1. **Single Level Directory:** The simplest directory structure is the single-level directory. In this type of directory system, there is a root directory which has all files.



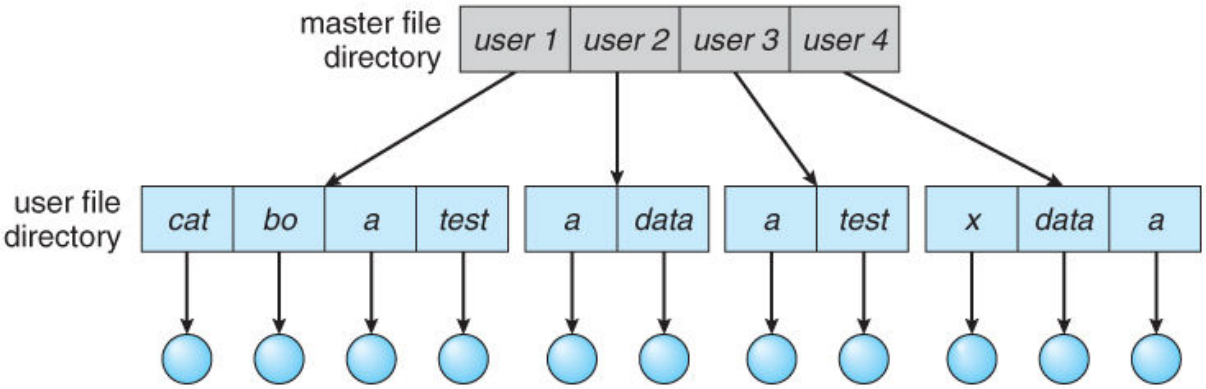
It has a simple architecture and there are no sub directories. Advantage of single level directory system is that it is easy to find a file in the directory. This type of directory system is

used in cameras and phones.All files are contained in the same directory, which is easy to support and understand.

Limitations:

i. Since all files are in the same directory, they must have unique name.
ii. If two users call their data file test, then the unique name rule is violated.
iii. Files are limited in length.
iv. Even a single user may find it difficult to remember names of all files.
v. Keeping track of all files is daunting task.

2. **Two Level Directory:** In the two-level directory structure, each user has his own user file directory (LTD). The UFDs have similar structures, but each lists only the files of a single user. When a user job starts or a user logs in, the system's master file directory (MFD) is searched. The MFD is indexed by user name or account number, and each entry points to the UFD for that user as shown below..



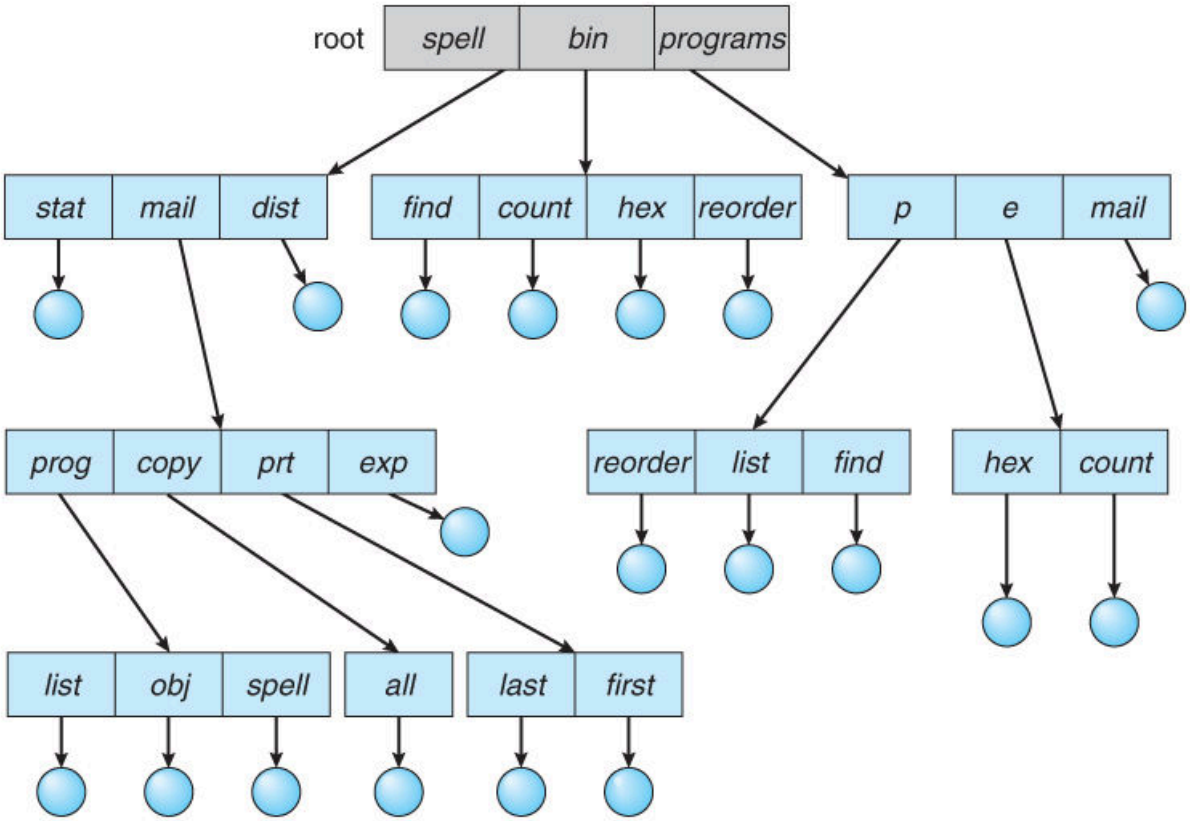
When a user refers to a particular file, only his own UFD is searched. Thus, different users may have files with the same name, as long as all the file names within each UFD are unique. To create a file for a user, the operating system searches only that user's UFD to ascertain whether another file of that name exists. To delete a file, the operating system confines its search to the local UFD; thus, it cannot accidentally delete another user's file that has the same name.

This structure effectively isolates one user from another. Isolation is an advantage wrhen the users are completely independent but is a disadvantage when the users *want* to cooperate on some task and to access one another's files. Some systems simply do not allow local user files to be accessed by other users.

If access is to be permitted, one user must have the ability to name a file in another user's directory. To name a particular file uniquely in a two-level directory, we must give both the user name and the file name. A two-level directory can be thought of as a tree, or an inverted tree, of height 2. The root of the tree is the MFD. Its direct descendants are the UFDs. The descendants of the UFDs are the files themselves. The files are the leaves of the tree. Specifying a user name and a file name defines a path in the tree from the root (the MFD) to a leaf (the specified file). Thus, a user name and a file name define a *path name.* Every file in the system has a path name. To name a file uniquely, a user must know the path name of the file desired.

For example, if user A wishes to access her own test file named *test,* she can simply refer to *test.* To access the file named *test* of user B (with directory-entry name *userb),* however, she might have to refer to */userb/test.* Every system has its own syntax for naming files in directories other than the user's own.

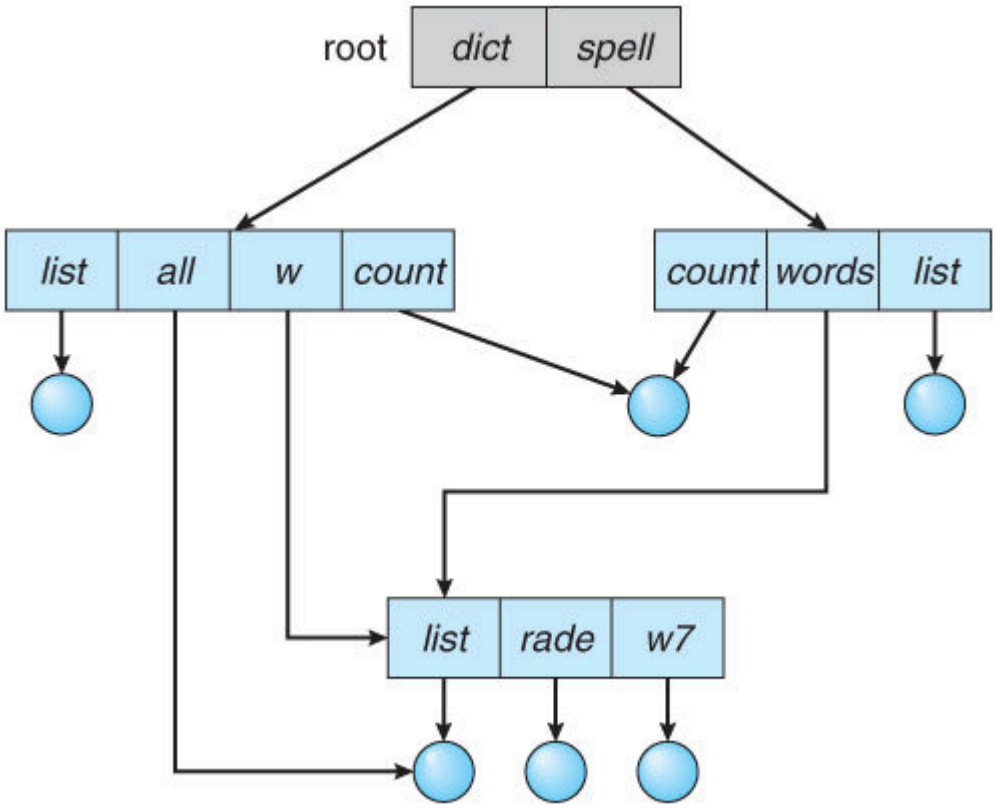
3. **Tree Structured Directories:** A tree is the most common directory structure. The tree has a root directory, and every file in the system has a unique path name. A directory (or subdirectory) contains a set of files or subdirectories. A directory is simply another file, but it is treated in a special way. All directories have the same internal format. One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1). Special system calls are used to create and delete directories.



Each process has a current directory. Current directory should contain most of the files that are of current interest to the process. When a reference is made to a file, the current directory is searched. The user can change his current directory whenever he desires. If a file is not needed in the current directory then the user usually must either specify a path name or change the current directory.

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

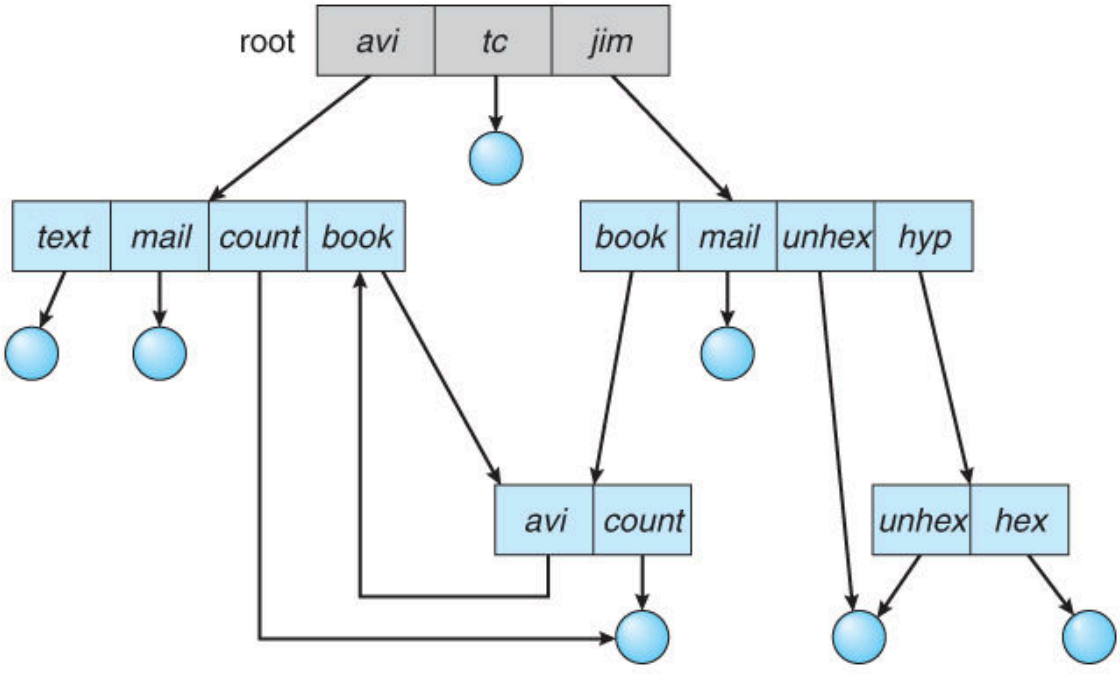
4. **Acyclic Graph Directories:** When the same files need to be accessed in more than one place in the directory structure, it can be useful to provide an acyclic graph directories.



UNIX provides two types of links for implementing the acyclic graph structure.

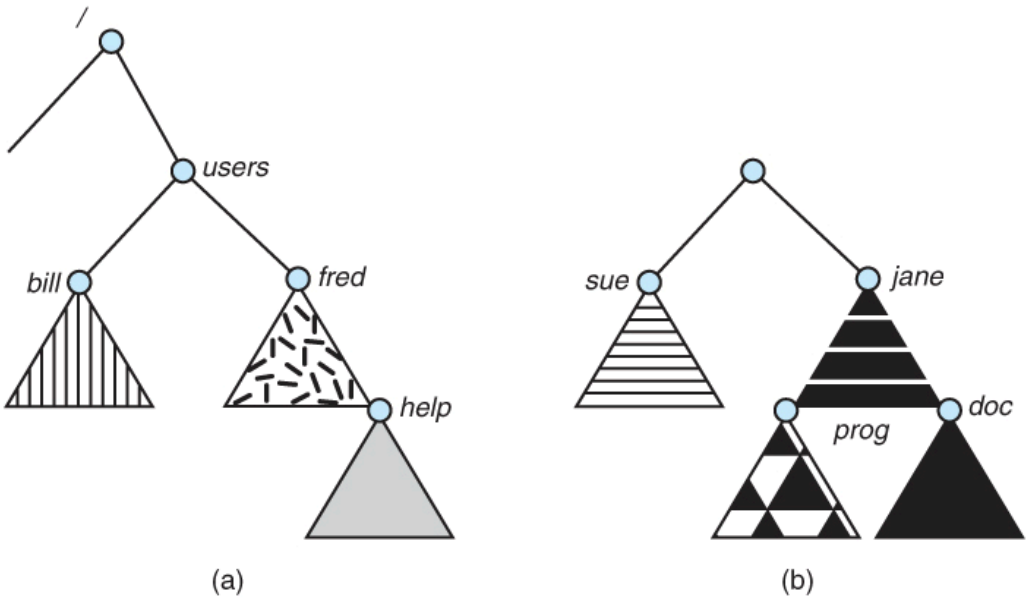
- A *Hard Link* involves multiple directory entries that both refer to the same file. Hard links are valid for ordinary files in the ordinary file system.
- A *Symbolic Link,* that involves a special file, containing information about where to find the linked file. Symbolic links may be used to link directoriesand files in other file systems, as well as ordinary files in the current file system. Hard link requires a reference count ot link count for each file, keeping track of how many entries are currently referring to this file. Whenever one of the references is removed the link count is decreased, and when it reaches zero, the disk space can be reclaimed.

5. General Graph Directory:



File System Mounting: The idea behind mouting file systems is to combine multiple file systems into one large tree structure. The mount procedure is straightforward. The operating system is given the name of the device and the **mount point**—the location within the file structure where the file system is to be attached. Typically, a mount point is an empty directory. For instance, on a UNIX system, a file system containing a user's home directories might be mounted as */home;* then, to access the directory structure within that file system, we could precede the directory names with *ftiome,* as in */homc/janc.* Mounting that file system under */users* would result in the path name */users/jane.*

Consider the following file system where the triangles represent subtrees of directories.



File System : a) Existing System b) Unmounted volume