MODULE 1

INTRODUCTION TO OPERATING SYSTEM

What is an Operating System?

An operating system is system software that acts as an intermediary between a user of a computer and the computer hardware. It is software that manages the computer hardware and allows the user to execute programs in a convenient and efficient manner.

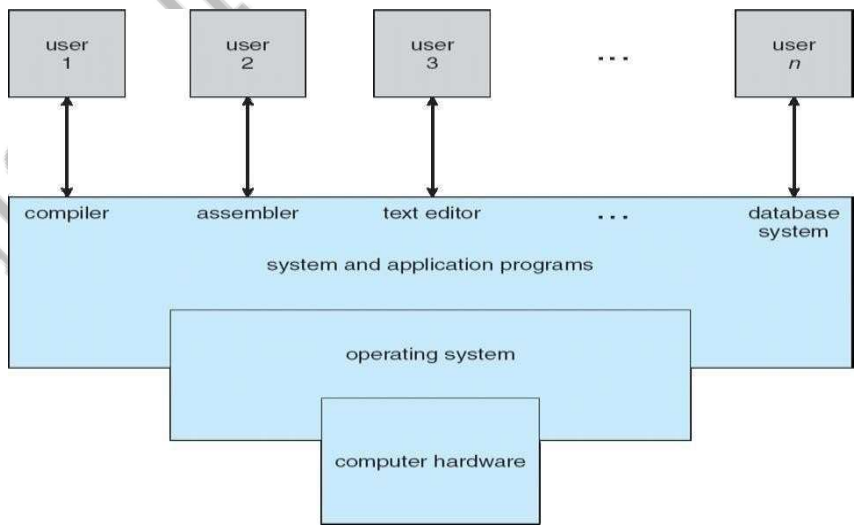
Operating system goals:

- Make the computer system convenient to use. It hides the difficulty in managing thehardware.
- Use the computer hardware in an efficient manner
- Provide and environment in which user can easily interface with computer.
- It is a resource allocator

Computer System Structure (Components of Computer System)

Computer system mainly consists of four components-

- ***Hardware*** – provides basic computing resources CPU, ALU, Peripheral Devices, memory, I/O devices &storage Devices
- ***Operating system*** - Controls and coordinates use of hardware among various applications and users
- ***Application programs*** – define the ways in which the system resources are used to solve the computing problems of the users, Word processors, compilers, web browsers, database systems, video games.
- ***Users*** - People, machines, other computers

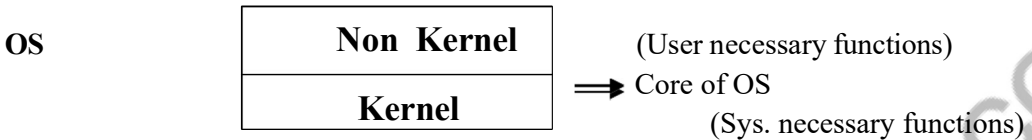


The basic hardware components comprises of CPU, memory, I/O devices. The application program uses these components. The OS controls and co-ordinates the use of hardware, among various application programs (like compiler, word processor etc.) for

various users.

The OS allocates the resources among the programs such that the hardware is efficiently used.

The operating system is the program running at all the times on the computer. It is usually called as the kernel.



Kernel functions are used always in system, so always stored in memory. Non kernel functions are stored in hard disk, and it is retrieved whenever required.

List out the User Views and System views of OS

Operating System can be viewed from two viewpoints– User views & System views

User Views: - The user's view of the operating system depends on the type of user.

- If the user is using **standalone** system, then OS is designed for ease of use and high performances. Here resource utilization is not given importance.
- If the users are at different **terminals** connected to a mainframe or minicomputers, by sharing information and resources, then the OS is designed to maximize resource utilization. OS is designed such that the CPU time, memory and i/o are used efficiently and no single user takes more than the resource allotted to them.
- If the users are in **workstations**, connected to networks and servers, then the user have a system unit of their own and shares resources and files with other systems. Here the OSis designed for both ease of use and resource availability (files).
- Other systems like embedded systems used in home device (like washing m/c) & automobiles do not have any user interaction. There are some LEDs to show the status ofits work
- Users of **hand-held** systems, expects the OS to be designed for ease of use and performance per amount of battery life

System Views: - Operating system can be viewed as a **resource allocator** and **control program**.

- **Resource allocator** – The OS acts as a manager of hardware and software resources. CPU time, memory space, file-storage space, I/O devices, shared files etc. are the different resources required during execution of a program. There can be conflicting request for these resources by different programs running in same system. The OSassigns the resources to the requesting program depending on the priority.

- **Control Program** – The OS is a control program and manage the execution of user program to prevent errors and improper use of the computer.

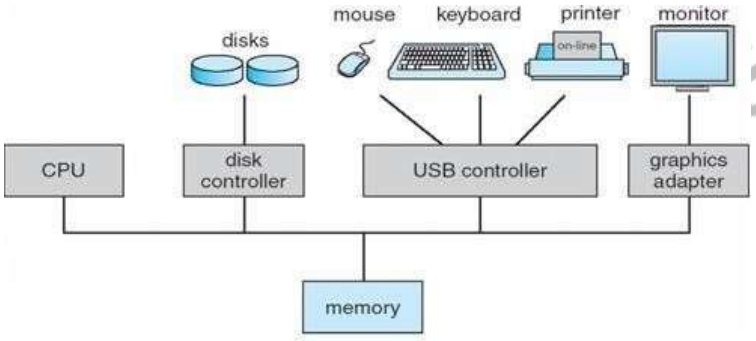
Computer System Organization

Computer - system operation

One or more CPUs, device controllers connect through common bus providing access to shared memory.

Each device controller is in-charge of a specific type of device. To ensure orderly access to the shared memory, a memory controller is provided whose function is to synchronize access to the memory.

The CPU and other devices execute concurrently competing for memory cycles. Concurrent execution of CPUs and devices competing for memory cycles



- When system is switched on, '**Bootstrap'** program is executed. It is the initial program torun in the system. This program is stored in read-only memory (ROM) or in electrically erasable programmable read-only memory (EEPROM).
- It initializes the CPU registers, memory, device controllers and other initial setups. The program also locates and loads, the OS kernel to the memory. Then the OS starts with thefirst process to be executed (ie. 'init' process) and then wait for the interrupt from the user.

Switch on → 'Bootstrap' program

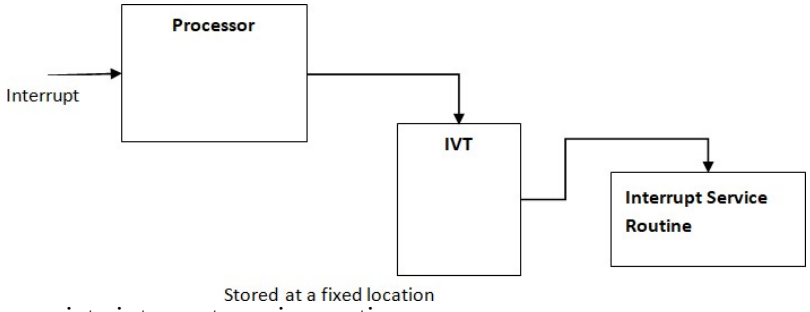
- Initializes the registers, memory and I/O devices
- Locates & loads kernel into memory
- Starts with 'init' process
- Waits for interrupt from user.

Interrupt handling –

- The occurrence of an event is usually signaled by an interrupt. The interrupt can either be from the hardware or the software. Hardware may trigger an interrupt at any time by sending asignal to the CPU. Software triggers an interrupt by executing a special operation called a system call (also called a monitor call).

- When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location. The fixed location (Interrupt Vector Table) contains the starting address where the service routine for the interrupt is located. After the execution of interrupt service routine, the CPU resumes the interrupted computation.

- Interrupts are an important part of computer architecture. Each computer design has its own interrupt mechanism, but several functions are common. The interrupt must transfer control



to the appropriate interrupt service routine.

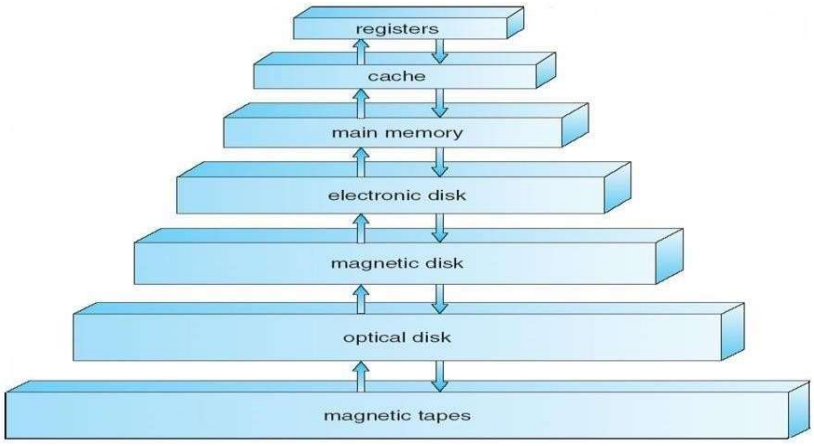
Storage Structure

- Computer programs must be in main memory (**RAM)** to be executed. Main memory is the large memory that the processor can access directly. It commonly is implemented in a semiconductor technology called **dynamic random-access memory (DRAM).** Computers provide Read Only Memory (ROM), whose data cannot be changed.
- All forms of memory provide an array of memory words. Each word has its own address.Interaction is achieved through a sequence of load or store instructions to specificmemory addresses.
- A typical instruction-execution cycle, as executed on a system with a **Von Neumann** architecture, first fetches an instruction from memory and stores that instruction in the **instruction register.** The instruction is then decoded and may cause operands to be fetched from memory and stored in some internal register. After the instruction on the operands has been executed, the result may be stored back in memory.
- Ideally, we want the programs and data to reside in main memory permanently. This arrangement usually is not possible for the following two reasons:

  1.Main memory is usually too small to store all needed programs and data permanently.2.Main memory is a *volatile* storage device that loses its contents when power is turned off.

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

- The most common secondary-storage device is a **magnetic disk,** which provides storage for both programs and data. Most programs are stored on a disk until they are loaded into memory. Many programs then use the disk as both a source and a destination of the information for their processing.

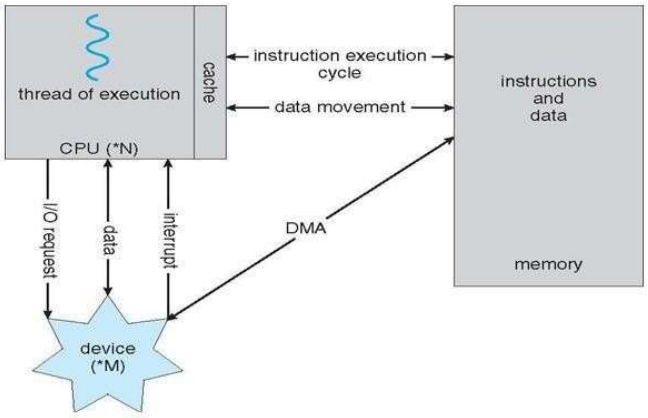


- The wide variety of storage systems in a computer system can be organized in a hierarchy as shown in the figure, according to speed, cost and capacity. The higher levelsare expensive, but they are fast. As we move down the hierarchy, the cost per bit generally decreases, whereas the access time and the capacity of storage generally increases.
- In addition to differing in speed and cost, the various storage systems are either volatileor nonvolatile. V**olatile storage** loses its contents when the power to the device is removed. In the absence of expensive battery and generator backup systems, data must be written to nonvolatile storage for safekeeping. In the hierarchy shown in figure, the storage systems above the electronic disk are volatile, whereas those below are nonvolatile.
- An **electronic disk** can be designed to be either volatile or nonvolatile. During normal operation, the electronic disk stores data in a large DRAM array, which is volatile. But many electronic-disk devices contain a hidden magnetic hard disk and a battery for backup power. If external power is interrupted, the electronic-disk controller copies the data from RAM to the magnetic disk. Another form of electronic disk is flash memory.

I/O Structure

- A large portion of operating system code is dedicated to managing I/O, both because of its importance to the reliability and performance of a system and because of the varying nature of the devices.
- Every device has a device controller, maintains some local buffer and a set of special-purpose registers. The device controller is responsible for moving the data between the peripheral devices. The operating systems have a **device driver** for each device controller.
- Interrupt-driven I/O is well suited for moving small amounts of data but can produce high overhead when used for bulk data movement such as disk I/O. To solve this problem, direct memory access (DMA) is used.
- After setting up buffers, pointers, and counters for the I/O device, the device controller transfers an entire block of data directly to or from its own buffer storage to memory, with no intervention by the CPU. Only one interrupt is generated per block, to tell the device

driver thatthe operation has completed.



Computer System Architecture

Categorized roughly according to the number of general-purpose processors used.

Single-Processor Systems –

- The variety of single-processor systems range from PDAs through mainframes. On a single-processor system, there is one main CPU capable of executing instructions from user processes. It contains special-purpose processors, in the form of device-specific processors, for devices such as disk, keyboard, and graphics controllers.
- All special-purpose processors run limited instructions and do not run user processes. These aremanaged by the operating system; the operating system sends them information about their next task and monitors their status.
- For example, a disk-controller processor, implements its own disk queue and scheduling algorithm, thus reducing the task of main CPU. Special processors in the keyboard, convertsthe keystrokes into codes to be sent to the CPU.
- The use of special-purpose microprocessors is common and does not turn a single-processor system into a multiprocessor. If there is only one general-purpose CPU, then the system is a single-processor system.

Multi -Processor Systems (parallel systems or **tightly coupled systems)**

Systems that have two or more processors in close communication, sharing the computer bus, theclock, memory, and peripheral devices are the multiprocessor systems.

Multiprocessor systems have three main advantages:

1. **Increased throughput** - In multiprocessor system, as there are multiple processors execution of different programs take place simultaneously. Even if the number of processors is increased the performance cannot be simultaneously increased. This is due to the overhead incurred in keeping all the parts working correctly and also due to the competition for the shared resources. The speed-up ratio with *N* processors is not *N,* rather, it is less than N. Thus the speed of the system is not has expected.

2. **Economy of scale** - Multiprocessor systems can cost less than equivalent number of many single-processor systems. As the multiprocessor systems share peripherals, mass storage, and power supplies, the cost of implementing this system is economical. If several processes are working on the same data, the data can also be shared among them.

3. **Increased reliability**- In multiprocessor systems functions are shared among several processors. If one processor fails, the system is not halted, it only slows down. The job of the failed processor is taken up, by other processors.

Two techniques to maintain 'Increased Reliability' - graceful degradation & fault tolerant

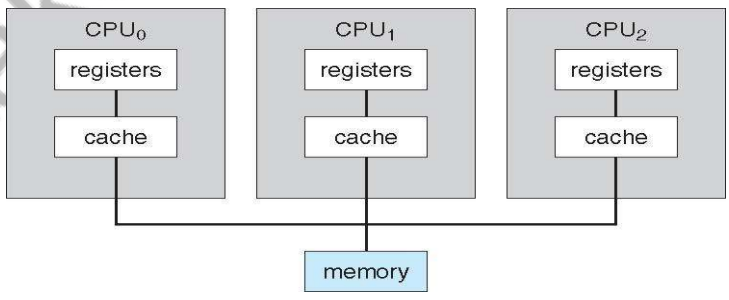
1. **Graceful degradation –** As there are multiple processors when one processor fails otherprocess will take up its work and the system go down slowly.
2. **Fault tolerant –** When one processor fails, its operations are stopped, the system failureis then detected, diagnosed, and corrected.

Different types of multiprocessor systems

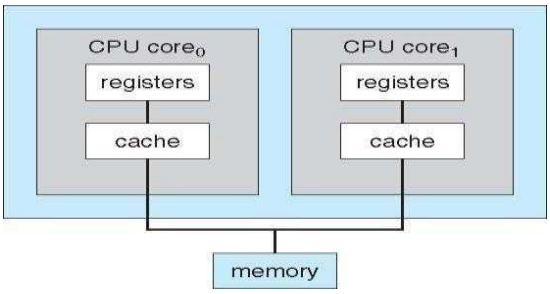
1. Asymmetric multiprocessing
2. Symmetric multiprocessing

1) **Asymmetric multiprocessing – (Master/Slave architecture)** Here each processor is assigned a specific task, by the master processor. A master processor controls the other processors in the system. It schedules and allocates work to the slave processors.

2) **Symmetric multiprocessing (SMP) –** All the processors are considered peers. There is no master-slave relationship. All the processors have their own registers and CPU, only memory is shared.



The benefit of this model is that many processes can run simultaneously. *N* processes can run if there are *N* CPUs—without causing a significant deterioration of performance. Operating systems like Windows, Windows XP, Mac OS X, and Linux—now provide support for SMP. A recent trend in CPU design is to include multiple compute **cores** on a single chip. The communication between processors within a chip is faster than communication between two single processors.



Clustered Systems

Clustered systems are two or more individual systems connected together via a network and sharing software resources. Clustering provides high availability of resources and services. The service will continue even if one or more systems in the cluster fail. High availability is generally obtained by storing a copy of files (s/w resources) in the system.

There are two types of Clustered systems – **asymmetric** and **symmetric**

1. **Asymmetric clustering** – one system is in **hot-standby mode** while the others are running the applications. The hot-standby host machine does nothing but monitor the activeserver. If that server fails, the hot-standby host becomes the active server.

2. **Symmetric clustering** – two or more systems are running applications, and are monitoring each other. This mode is more efficient, as it uses all of the available hardware. If any system fails, its job is taken up by the monitoring system.

Other forms of clusters include parallel clusters and clustering over a wide-area network (WAN). Parallel clusters allow multiple hosts to access the same data on the shared storage. Clustertechnology is changing rapidly with the help of **SAN (storage-area networks)**. Using SANresources can be shared with dozens of systems in a cluster, that are separated by miles.

Operating System Structure

Explain multiprogramming and multitasking systems.

Multiprogramming

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 by organizing jobs, so that the CPU always has one to execute.

- The operating system keeps several jobs in memory simultaneously as shown in figure. Thisset of jobs is a subset of the jobs kept in the job pool. Since the number of jobs that can be keptsimultaneously in memory is usually smaller than the number of jobs that can be kept in thejob pool (in secondary memory). The operating system picks and begins to execute one of the jobs in memory. Eventually, the job may have to wait for some tasks, such as an I/O operation,to complete. In a non-multiprogram system, the CPU would sit idle.

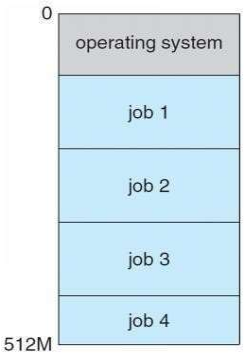
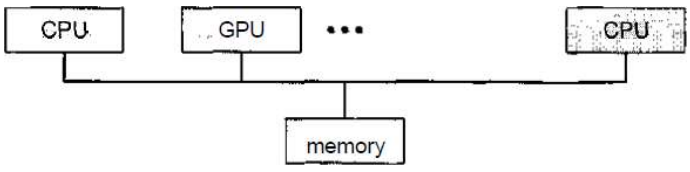


Fig - Memory layout for a multiprogramming system

- In a multiprogrammed 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. Thus, the CPU is never idle.
- Multiprogrammed systems provide an environment in which the various system resources (for example, CPU, memory, and peripheral devices) are utilized effectively, but they do not provide for user interaction with the computer system.

Multitasking Systems

- In **Time sharing** (or **multitasking) systems,** a single 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. The user feels that all the programs are being executed at the same time.
- 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. As the system switches rapidly from one user to the next, each user is given the impression that theentire computer system is dedicated to his use only, even though it is being shared among manyusers.
- A **multiprocessor system** is a computer system having two or more CPUs within a single computer system, each sharing main memory and peripherals. Multiple programs are executed by multiple processors parallel.



Operating-System Operations

Modern operating systems are **interrupt driven.** If there are no processes to execute, no I/O devices

toservice, and no users to whom to respond, an operating system will sit quietly, waiting for somethingto happen. Events are signaled by the occurrence of an interrupt or a trap. A **trap (or an exception)** isa software-generated interrupt. For each type of interrupt, separate segments of code in the operating system determine what action should be taken. An interrupt service routine is provided that is responsible for dealing with the interrupt.

Explain dual mode operation in operating system with a neat block diagramDual-Mode Operation

Since the operating system and the user programs share the hardware and software resources of the computer system, it has to be made sure that an error in a user program cannot cause problems to other programs and the Operating System running in the system.

The approach taken is to use a hardware support that allows us to differentiate among various modes of execution.

The system can be assumed to work in two separate **modes** of operation:

1. User mode
2. Kernel mode (supervisor mode, system mode, or privileged mode).

- A hardware bit of the computer, called the **mode bit**, is used to indicate the current mode: kernel (0) or user (1). With the mode bit, we are able to distinguish between a task that is executed by the operating system and one that is executed by the user.
- When the computer system is executing a user application, the system is in user mode. When a user application requests a service from the operating system (via a system call), the transition from user to kernel mode takes place.

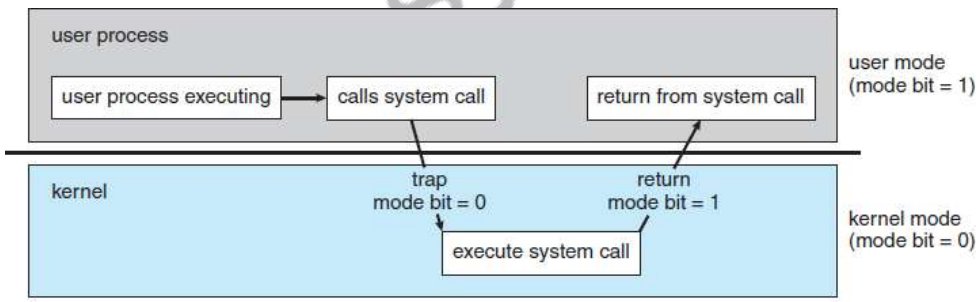


Figure Transition from user to kernel mode.

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

The dual mode of operation provides us with the means for protecting the operating system fromerrant users—and errant users from one another.

- The hardware allows privileged instructions to be executed only in kernel mode. If an

attemptis 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. The instruction to switch to user mode is an example of a privileged instruction.

- Initial control is within the operating system, where instructions are executed in kernel mode. When control is given to a user application, the mode is set to user mode. Eventually, control isswitched back to the operating system via an interrupt, a trap, or a system call.

Process Management

- A program under execution is a process. A process needs resources like CPU time, memory, files, and I/O devices for its execution. These resources are given to the process when it is created or at run time. When the process terminates, the operating system reclaims the resources.

- The program stored on a disk is a **passive entity** and the program under execution is an **active entity**. A single-threaded process has one **program counter** specifying the next instruction to execute. The CPU executes one instruction of the process after another, until the process completes. A multithreaded process has multiple program counters, each pointing to the next instruction to execute for a given thread.

- The operating system is responsible for the following activities in connection with processmanagement:
  - Scheduling process and threads on the CPU
  - Creating and deleting both user and system processes
  - Suspending and resuming processes
  - Providing mechanisms for process synchronization
  - Providing mechanisms for process communication

Memory Management

Main memory is a large array of words or bytes. Each word or byte has its own address.
Main memory is the storage device which can be easily and directly accessed by the CPU. As the program executes, the central processor reads instructions and also reads and writes data from main memory.

To improve both the utilization of the CPU and the speed of the computer's response to its users, general-purpose computers must keep several programs in memory, creating a need for memory management.

The operating system is responsible for the following activities in connection with memory management:

- Keeping track of which parts of memory are currently being used by user.

- Deciding which processes and data to move into and out of memory.
- Allocating and deallocating memory space as needed.

Storage Management

There are three types of storage management

i) File system management
ii) Mass-storage management
iii) Cache management.

File-System Management

- File management is one of the most visible components of an operating system. Computers canstore information on several different types of physical media. Magnetic disk, optical disk, and magnetic tape are the most common. Each of these media has its own characteristics and physical organization. Each medium is controlled by a device, such as a disk drive or tape drive, that also has its own unique characteristics.
- A file is a collection of related information defined by its creator. Commonly, files represent programs and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free-form (for example, text files), or they may be formatted rigidly (for example, fixed fields).
- The operating system implements the abstract concept of a file by managing mass storage media. Files are normally organized into directories to make them easier to use. When multipleusers have access to files, it may be desirable to control by whom and in what ways (read, write, execute) files may be accessed.

The operating system is responsible for the following activities in connection with file management:

- Creating and deleting files
- Creating and deleting directories to organize files
- Supporting primitives for manipulating files and directories
- Mapping files onto secondary storage
- Backing up files on stable (nonvolatile) storage media

Mass-Storage Management

- As the main memory is too small to accommodate all data and programs, and as the data that it holds are erased when power is lost, the computer system must provide secondary storage to back up main memory. Most modern computer systems use disks as the storage medium for both programs and data.

- Most programs—including compilers, assemblers, word processors, editors, and formatters— are stored on a disk until loaded into memory and then use the disk as both the source and destination of their processing. Hence, the proper management of disk storage is of central importance to a computer system.

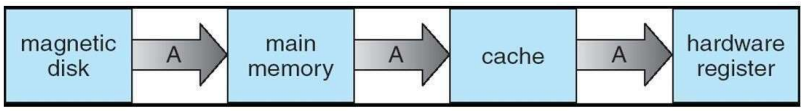
The operating system is responsible for the following activities in connection with disk management:

- Free-space management
- Storage allocation
- Disk scheduling

As the secondary storage is used frequently, it must be used efficiently. The entire speed of operation of a computer may depend on the speeds of the disk. Magnetic tape drives and their tapes, CD, DVD drives and platters are **tertiary storage** devices. The functions that operating systems provides includemounting and unmounting media in devices, allocating and freeing the devices for exclusive use by processes, and migrating data from secondary to tertiary storage.

Caching

- **Caching** is an important principle of computer systems. Information is normally kept in some storage system (such as main memory). As it is used, it is copied into a faster storage system— the cache—as temporary data. When a particular piece of information is required, first we check whether it is in the cache. If it is, we use the information directly from the cache; if it is not in cache, we use the information from the source, putting a copy in the cache under the assumption that we will need it again soon.
- Because caches have limited size, **cache management** is an important design problem. Carefulselection of the cache size and page replacement policy can result in greatly increased performance.
- The movement of information between levels of a storage hierarchy may be either explicit or implicit, depending on the hardware design and the controlling operating-system software. For instance, data transfer from cache to CPU and registers is usually a hardware function, with no operating-system intervention. In contrast, transfer of data from disk to memory is usually controlled by the operating system.
- In a hierarchical storage structure, the same data may appear in different levels of the storage system. For example, suppose to retrieve an integer A from magnetic disk to the processing program. The operation proceeds by first issuing an I/O operation to copy the disk block on which A resides to main memory. This operation is followed by copying A to the cache and to an internal register. Thus, the copy of A appears in several places: on the magnetic disk, inmain memory, in the cache, and in an internal register.



- In a multiprocessor environment, in addition to maintaining internal registers, each of theCPUs also contains a local cache. In such an environment, a copy of A may exist simultaneously in several caches. Since the various CPUs can all execute concurrently, any update done to the value of A in one cache is immediately reflected in all other caches where Aresides. This situation is called **cache coherency,** and it is usually a hardware problem (handled below the operating-system level).

I/O Systems

One of the purposes of an operating system is to hide the peculiarities of specific hardware devicesfrom the user. The I/O subsystem consists of several components:

- A memory-management component that includes buffering, caching, and spooling
- A general device-driver interface
- Drivers for specific hardware devices

Only the device driver knows the peculiarities of the specific device to which it is assigned.

Protection and Security

- If a computer system has multiple users and allows the concurrent execution of multiple processes, then access to data must be regulated. For that purpose, mechanisms ensure thatfiles, memory segments, CPU, and other resources can be operated on by only those processes that have gained proper authorization from the operating system.
- For example, memory-addressing hardware ensures that a process can execute only within its own address space. The timer ensures that no process can gain control of the CPU for a long time. Device-control registers are not accessible to users, so the integrity of the various peripheral devices is protected.
- **Protection** is a mechanism for controlling the access of processes or users to the resources defined by a computer system. This mechanism must provide means for specification of the controls to be imposed and means for enforcement.
- Protection improves reliability. A protection-oriented system provides a means to distinguish between authorized and unauthorized usage. A system can have adequate protection but still beprone to failure and allow inappropriate access.
- Consider a user whose authentication information is stolen. Her data could be copied or
  deleted, even though file and memory protection are working. It is the job of **security** to defend a system from external and internal attacks. Such attacks spread across a huge range and include viruses and worms, denial-of service attacks etc.
- Protection and security require the system to be able to distinguish among all its users. Most operating systems maintain a list of user names and associated **user identifiers (user IDs).** When a user logs in to the system, the authentication stage determines the appropriate user ID for the user.

Distributed Systems

- A distributed system is a collection of systems that are networked to provide the users with access to the various resources in the network. Access to a shared resource increases computation speed, functionality, data availability, and reliability.
- A **network** is a communication path between two or more systems. Networks vary by the protocols used(TCP/IP,UDP,FTP etc.), the distances between nodes, and the transport

media(copper wires, fiber-optic,wireless).

- TCP/IP is the most common network protocol. The operating systems support of protocols alsovaries. Most operating systems support TCP/IP, including the Windows and UNIX operating systems.
- 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. A global company may have a WAN to connect its offices worldwide. These networks may run one protocol or several protocols. A **metropolitan-area network** (MAN) connects buildings within a city. BlueTooth and 802.11 devices use wireless technology to communicate over a distance of several feet, in essence creating a **small-area network** such as might be found in a home.
- The transportation media to carry networks are also varied. They include copper wires, fiber strands, and wireless transmissions between satellites, microwave dishes, and radios. When computing devices are connected to cellular phones, they create a network.

Special Purpose Systems

Multimedia Systems

- Multimedia data consist of audio and video files as well as conventional files. These data differfrom conventional data in that multimedia data—such as frames of video—must be delivered (streamed) according to certain time restrictions (for example, 30 frames per second).
- Multimedia describes a wide range of applications like audio files - MP3, DVD movies, video conferencing, and short video clips of movie previews or news. Multimedia applications may also include live webcasts of speeches or sporting events and even live webcams. Multimedia applications can be either audio or video or combination of both. For example, a movie may consist of separate audio and video tracks.

Handheld Systems

- Handheld systems include personal digital assistants (PDAs), such as Palm and Pocket-PCs, and cellular telephones. Developers of these systems face many challenges, due to the limited memory, slow processors and small screens in such devices.
- The amount of physical memory in a handheld depends upon the device, the operating system and applications must manage memory efficiently. This includes returning all allocated memory back to the memory manager when the memory is not being used. 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 faster speed than the processor in a PC. Faster processors require more power and so, a larger battery is required. Another issue is the usageof I/O devices.
- Generally, the limitations in the functionality of PDAs are balanced by their convenience and portability. Their use continues to expand as network connections become more available and other options, such as digital cameras and MP3 players, expand their utility.

Computing Environments

The different computing environments are -

Traditional Computing

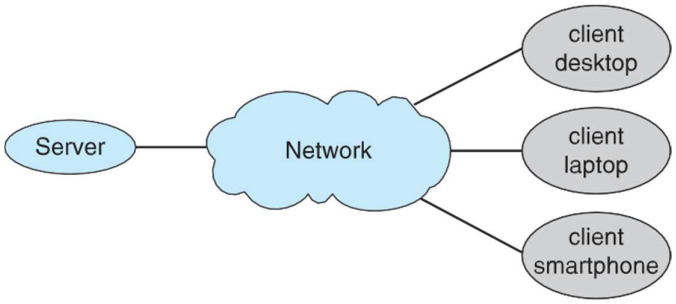
- The current trend is toward providing more ways to access these computing environments. Web technologies are stretching the boundaries of traditional computing. Companies establish **portals,** which provide web accessibility to their internal servers. **Network computers** are essentially terminals that understand web-based computing. Handheld computers cansynchronize with PCs to allow very portable use of company information. Handheld PDAs can also connect to **wireless networks** to use the company's web portal. The fast data connections are allowing home computers to serve up web pages and to use networks. Some homes even have **firewalls** to protect their networks.

- In the latter half of the previous century, computing resources were scarce. Years before, systems were either batch or interactive. Batch system processed jobs in bulk, with predetermined input (from files or other sources of data). Interactive systems waited for input from users. To optimize the use of the computing resources, multiple users shared time onthese systems. Time-sharing systems used a timer and scheduling algorithms to rapidly cycle processes through the CPU, giving each user a share of the resources.

- Today, traditional time-sharing systems are used everywhere. The same scheduling techniqueis still in use on workstations and servers, but frequently the processes are all owned by the same user (or a single user and the operating system). User processes, and system processes that provide services to the user, are managed so that each frequently gets a slice of computer time.

Client-Server Computing

Designers shifted away from centralized system architecture to - terminals connected to centralized systems. As a result, many of today's systems act as **server systems to** satisfy requests generated by **client systems.** This form of specialized distributed system, called **client- server** system.



General Structure of Client – Server System

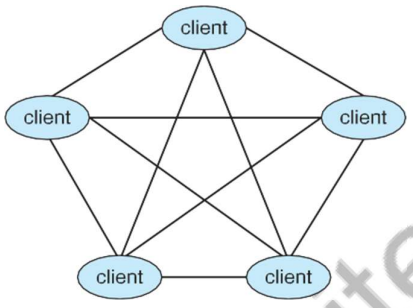
Server systems can be broadly categorized as compute servers and file servers:

- The **compute-server system** provides an interface to which a client can send a request to perform an action (for example, read data); in response, the server executes the action and sends back results to the client. A server running a database that responds to client requests for data is an example of such a system.
- The **file-server system** provides a file-system interface where clients can create, update, read, and delete files. An example of such a system is a web server that delivers files to clients running the web browsers.

Peer-to-Peer Computing

- In this model, clients and servers are not distinguished from one another; here, all nodes within the system are considered peers, and each may act as either a client or a server, depending on whether it is requesting or providing a service.
- In a client-server system, the server is a bottleneck, because all the services must be served by the server. But in a peer-to-peer system, services can be provided by several nodes distributed throughout the network.
- To participate in a peer-to-peer system, a node must first join the network of peers. Once a node has joined the network, it can begin providing services to—and requesting services from—other nodes in the network.

Determining what services are available is accomplished in one of two general ways:



- When a node joins a network, it registers its service with a centralized lookup service on thenetwork. Any node desiring a specific service first contacts this centralized lookup serviceto determine which node provides the service. The remainder of the communication takes place between the client and the service provider.
- A peer acting as a client must know, which node provides a desired service by broadcasting a request for the service to all other nodes in the network. The node (or nodes) providingthat service responds to the peer making the request. To support this approach, a *discovery protocol* must be provided that allows peers to discover services provided by other peers in the network.

Web-Based Computing

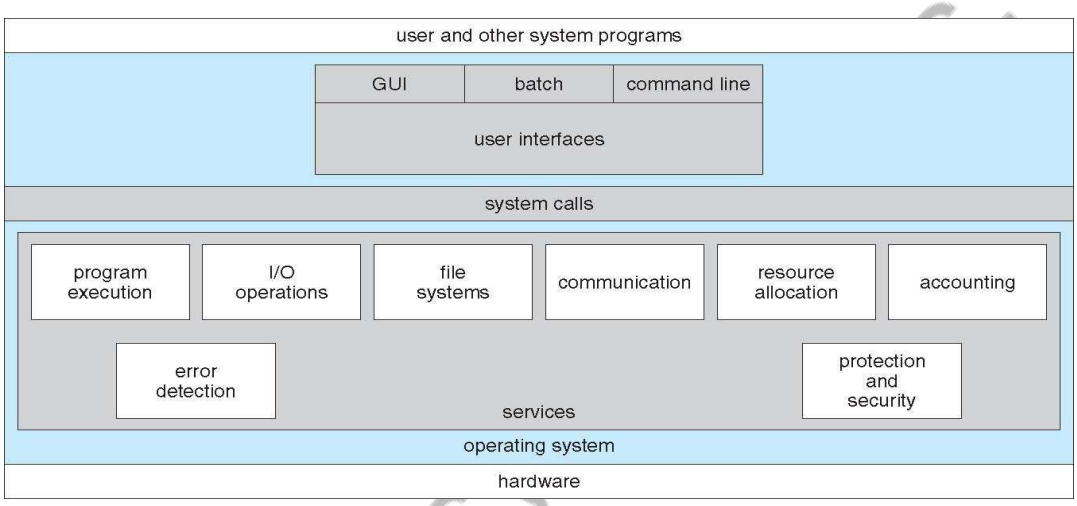
- Web computing has increased the importance on networking. Devices that were not previously networked now include wired or wireless access.
- The implementation of web-based computing has given rise to new categories of devices, such as **load balancers,** which distribute network connections among a pool of similar servers. Operating systems like Windows 95, which acted as web clients, have evolved into Linux and Windows XP, which can act as web servers as well as clients. Generally, the Web hasincreased the complexity of devices, because their users require them to be web-enabled.
- The design of an operating system is a major task. It is important that the goals of the new system be well defined before the design of OS begins. These goals form the basis for choices among various algorithms and strategies.

OPERATING SYSTEM SERVICES

Operating-System Services

Q) List and explain the services provided by OS for the user and efficientoperation of system.

An operating system provides an environment for the execution of programs. It provides certainservices to programs and to the users of those programs.



OS provide services for the users of the system, including:

- **User Interfaces** - Means by which users can issue commands to the system. Depending on the operating system these may be a **command-line interface** (e.g. sh, csh, ksh, tcsh, etc.), a **Graphical User Interface** (e.g. Windows, X-Windows, KDE, Gnome, etc.), or a **batch command systems**.
  In Command Line Interface (CLI)- commands are given to the system.
  In Batch interface – commands and directives to control these commands are put in a file andthen the file is executed.
  In GUI systems- windows with pointing device to get inputs and keyboard to enter the text.
- **Program Execution** - The OS must be able to load a program into RAM, run the program, andterminate the program, either normally or abnormally.
- **I/O Operations** - The OS is responsible for transferring data to and from I/O devices, including keyboards, terminals, printers, and files. For specific devices, special functions are provided (device drivers) by OS.
- **File-System Manipulation** – Programs need to read and write files or directories. The servicesrequired to create or delete files, search for a file, list the contents of a file and change the file permissions are provided by OS.

- **Communications** - Inter-process communications, IPC, either between processes running on the same processor, or between processes running on separate processors or separate machines.May be implemented by using the service of OS- like shared memory or message passing.
- **Error Detection** - Both hardware and software errors must be detected and handled appropriately by the OS. Errors may occur in the CPU and memory hardware (such as power failure and memory error), in I/O devices (such as a parity error on tape, a connection failureon a network, or lack of paper in the printer), and in the user program (such as an arithmetic overflow, an attempt to access an illegal memory location).

OS provide services for the efficient operation of the system, including:

- **Resource Allocation** – Resources like CPU cycles, main memory, storage space, and I/O devices must be allocated to multiple users and multiple jobs at the same time.
- **Accounting** – There are services in OS to keep track of system activity and resource usage, either for billing purposes or for statistical record keeping that can be used to optimize future performance.
- **Protection and Security** – The owners of information (file) in multiuser or networked computer system may want to control the use of that information. When several separate processes execute concurrently, one process should not interfere with other or with OS. Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders must also be done, by means of a password.

User Operating-System Interface

There are several ways for users to interface with the operating system.

i) Command-line interface, or command interpreter, allows users to directly enter commandsto be performed by the operating system.
ii) Graphical user interface (GUI), allows users to interface with the operating system usingpointer device and menu system.

Command Interpreter

- Command Interpreters are used to give commands to the OS. There are multiple command interpreters known as shells. In UNIX and Linux systems, there are several different shells, likethe *Bourne shell, C shell, Bourne-Again shell, Korn shell,* and others.
- The main function of the command interpreter is to get and execute the user-specified command. Many of the commands manipulate files: create, delete, list, print, copy, execute,and so on.

The commands can be implemented in two general ways-

i) The command interpreter itself contains the code to execute the command. For example, a command to delete a file may cause the command interpreter to jump to a particular sectionof its code that sets up the parameters and makes the appropriate system call.
ii) The code to implement the command is in a function in a separate file. The interpreter

searches for the file and loads it into the memory and executes it by passing the parameter.

Thus by adding new functions new commands can be added easily to the interpreterwithout disturbing it.

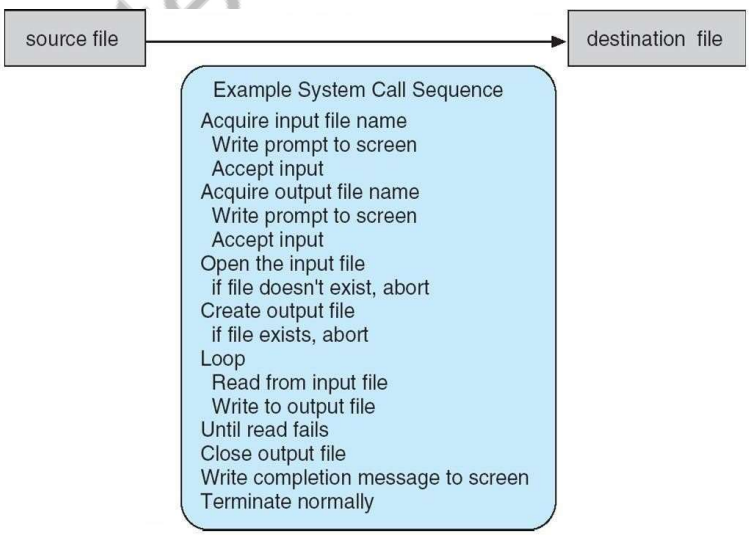
Graphical User Interfaces

- A second strategy for interfacing with the operating system is through a userfriendly graphical user interface or GUI. Rather than having users directly enter commands via a command-line interface, a GUI allows provides a mouse-based window-and-menu system as an interface.
- A GUI provides a **desktop** metaphor where the mouse is moved to position its pointer on images, or **icons,** on the screen (the desktop) that represent programs, files, directories, and system functions.
- Depending on the mouse pointer's location, clicking a button on the mouse can invoke a program, select a file or directory—known as a **folder**— or pull down a menu that contains commands.

System Calls

Q) What are system calls? Briefly point out its types.

- System calls provides an interface to the services of the operating system. These are generallywritten in C or C++, although some are written in assembly for optimal performance.
- The below figure illustrates the sequence of system calls required to copy a file content fromone file (input file) to another file (output file).



An example to illustrate how system calls are used: writing a simple program to read data from onefile and copy them to another file

- There are number of system calls used to finish this task. The first system call is to write a message on the screen (monitor). Then to accept the input filename. Then another

system call to write message on the screen, then to accept the output filename.

- When the program tries to open the input file, it may find that there is no file of that nameor that the file is protected against access. In these cases, the program should print a message on the console (another system call) and then terminate abnormally (another system call) and create a new one (another system call).
- Now that both the files are opened, we enter a loop that reads from the input file (another system call) and writes to output file (another system call).
- Finally, after the entire file is copied, the program may close both files (another systemcall), write a message to the console or window (system call), and finally terminate normally (final system call).

✓ Application developers design programs according to an Application Program Interface (API). API specifies the set of functions that are available to an application programmer including the parameter that are passed to each function and returns values the programmer can expect.

✓ Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM).

✓ The runtime support system(a set of functions built into libraries included with a compiler) for most programming languages provides a system call interface that serves as the link to system calls made available by the OS.

✓ The system call interface intercepts function call in the API and invokes the necessary system call within the OS.

✓ 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 invokes intended system call in OS kernel and returns status of the system call and any return values.

✓ The caller needs to know nothing about how the system call is implemented or what it does during execution.

✓ The below figure illustrates how the OS handles a user application which is invoking open () system call.

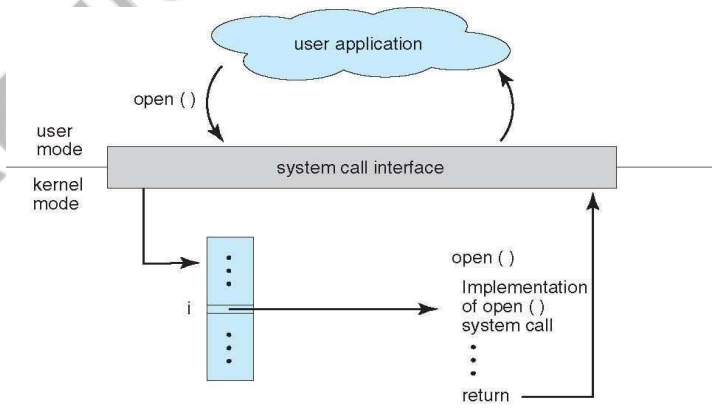


Figure: The handling of a user application invoking the open() system call.

Three general methods used to pass parameters to OS are –

i) To pass parameters in registers

ii) If parameters are large blocks, address of block (where parameters are stored in

memory) issent to OS in the register. (Linux & Solaris).

iii) Parameters can be pushed onto the stack by program and popped off the stack by OS.

iv) Some OS prefer the **block or stack** methods, because those approaches do not limit the number or length of parameters being passed.

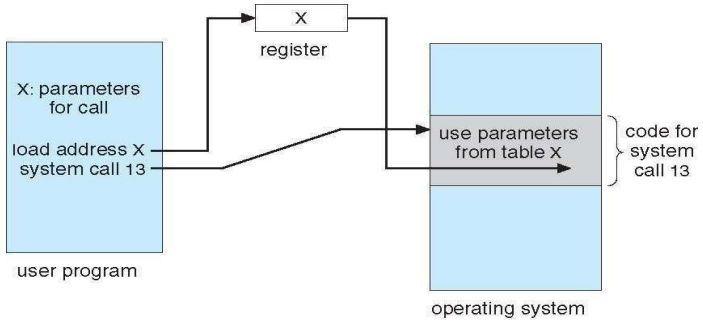


Figure: Passing of parameters as a table.

Types of System Calls

The system calls can be categorized into six major categories:

1. Process Control
2. File management
3. Device management
4. Information management
5. Communications
6. Protection

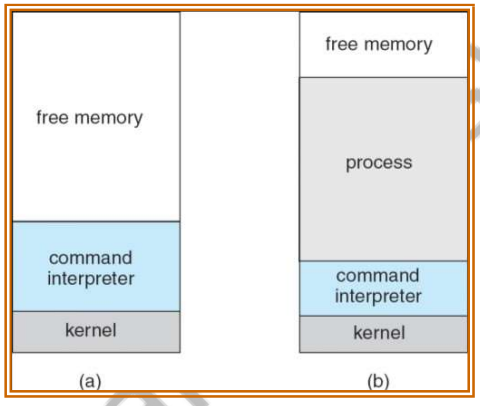
1. Process Control

- **end, abort**
  - ✓ A running program needs to be able to halt its execution either **normally** or **abnormally**. In an abnormal termination a dump of memory is taken and an error message is generated. Dump is written to disk and examined by the debugger to determine the cause of problem.
  - ✓ In normal or abnormal situations the OS must transfer the control to the command interpreter system, which then reads the next command given by the user.
  - ✓ In batch system the command interpreter terminates the execution of job & continues with the next job. Batch-systems uses **control cards** to indicate the special recovery action to be taken in case of errors. It is a command to manage the execution of a process.
  - ✓ More severe errors can be indicated by a higher level error parameter. Normal & abnormal termination can be combined by defining normal termination as an error at **level 0**.The command interpreter uses this error level to determine next action automatically.
- **load, execute**
  - ✓ A process executing one program may want to **load and execute** another program. This feature allows the command interpreter to execute programs as directed by the user.

- ✓ The question of where to return the control when the loaded program terminates is related to the problem of whether the existing program is lost, saved or allowed to continue execution concurrently with the new program. There is a system call for this purpose (**create or submit process**).

- **create process, terminate process**
  - ✓ If we create a new job or process, it should be able to control its execution.
  - ✓ This control requires the ability to determine and reset the attributes of a process, including the process priority, its maximum allowable execution time, and so on (get process attributes and set process attributes).
  - ✓ We may also want to terminate a process that we created (terminate process)
- **wait event, signal event**
  - ✓ When new jobs have been created, we may want to wait for certain amount of time using **wait time** system call.
  - ✓ When a job has to wait for a certain event to occur **wait event** system call is used.
  - ✓ When the event has occurred the job should signal the occurrence through the **signal event** system call.
- **get process attributes, set process attributes**
- **wait for time**
- **allocate and free memory**

- **In MS-DOS**

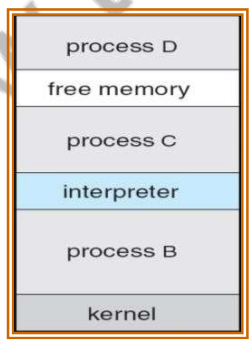


✓ MS-DOS is an example of single tasking system, which has command interpreter system that is invoked when the computer is started as shown in **figure (a)**.

✓ To run a program MS-DOS uses simple method. It does not create a newprocess when one process is running.

✓ It loads the program into memory and gives the program as much memory as possible as shown in **figure (b)**.

In FreeBSD



✓ Free BSD is an example of multitasking system.

✓ In free BSD the command interpreter may continue running while other program is executed as shown in below **figure**.

✓ fork () is a system call used to create new process.

✓ Then, the selected program is loaded into memory via an exec () system call and then program is executed.

2. File management

- create file, delete file

- ✓ System calls can be used to create & delete files. System calls may require the name of the files and attributes for creating & deleting of files.

- open, close file
  - ✓ Opens the file for usage and finally we need to close the file.
- read, write, reposition
  - ✓ Other operation may involve the reading of the file, write & reposition the file after it is opened.
- get and set file attributes
  - ✓ For directories, some set of operation are to be performed. Sometimes it is required to reset some of the attributes on files & directories. The system call **get file attribute** & **set file attribute** are used for this type of operation.

3. Device management

- request device, release device
- read, write, reposition
- get device attributes, set device attributes
- logically attach or detach devices

- ✓ The system calls are also used for accessing devices.
- ✓ Many of the system calls used for files are also used for devices.
- ✓ A system with multiple users may require us to first request the device, to ensure exclusive use of it.
- ✓ After using the device, it must be released using **release** system call. These functions are similar to open & close system calls of files.
- ✓ Read, write & reposition system calls may be used with devices.
- ✓ MS-DOS & UNIX merge the I/O devices & the files to form **file-device structure**.

4. Information maintenance

- get time or date, set time or date
- get system data, set system data
- get and set process, file, or device attributes
  - ✓ Many system calls exist for the purpose of transferring information between the user program and the operating system.
  - ✓ For example, most systems have a system call to return the current **time** and **date**.
  - ✓ Other system calls may return information about the system, such as the number of current users, the version number of the operating system, the amount of free memory or disk space, and so on.
  - ✓ The operating system also keeps information about all its processes, and system calls are used to access this information.
  - ✓ System calls are also used to reset the process information (**get process attributes and set process attributes**).

5. Communications

- create, delete communication connection

- send, receive messages
- transfer status information
- attach and detach remote devices

There are two models of inter-process communication:

❖ **Message Passing model**

- ✓ In message passing model, the communicating processes exchange messages with one another to transfer information. Messages can be exchanged between the processes either directly or indirectly through a common **mailbox**.
- ✓ Each computer in a network will have a **host name.** Similarly, each process has a **process name**, and this name is translated into an identifier by which the operating system can refer to the process. The **get hostid** and **get processid** system calls do this translation.
- ✓ The identifiers are then passed to the general purpose **open** and **close** calls provided by the file system or to specific **open connection** and **close connection** system calls, depending on the system's model of communication.
- ✓ The recipient process must give its permission for communication to take place with an **accept connection** system call.
- ✓ The receiving daemons execute a **wait for connection** call and are awakened when a connection is made.
- ✓ The source of the communication, known as the **client**, and the receiving daemon, known as a **server**, exchange messages by using **read message** and **write message** system calls.
- ✓ The **close connection** call terminates the communication.

❖ **Shared Memory model**

- ✓ In shared memory model, processes use shared memory create and shared memory attach system calls to create and gain access to regions of memory owned by other processes.
- ✓ The OS tries to prevent one process from accessing another process's memory, so several processes have to agree to remove this restriction. Then they exchange information by reading and writing in the shared areas.
- ✓ The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

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

6. Protection

- Protection provides mechanisms for controlling which users / processes have access to which system resources.
- System calls allow the access mechanisms to be adjusted as needed, and for non-privileged users to be granted elevated access permissions under carefully controlled temporary circumstances.

System Programs

Q) List and explain the different categories of system program?

A collection of programs that provide a convenient environment for program development and execution (other than OS) are called system programs or system utilities.

System programs may be divided into five categories:

1. **File management** - programs to create, delete, copy, rename, print, list, and generally manipulate files and directories.
2. **Status information** - Utilities to check on the date, time, number of users, processes running, data logging, etc. System registries are used to store and recall configuration information for particular applications.
3. **File modification** - e.g. text editors and other tools which can change file contents.
4. **Programming-language support** - E.g. Compilers, linkers, debuggers, profilers, assemblers, library archive management, interpreters for common languages, and support for make.
5. **Program loading and execution** - loaders, dynamic loaders, overlay loaders, etc., as well as interactive debuggers.
6. **Communications** - Programs for providing connectivity between processes and users, including mail, web browsers, remote logins, file transfers, and remote command execution.

Operating-System Design and Implementation

Design Goals

- The first problem in designing a system is to define goals and specifications. At the highest level, the design of the system will be affected by the choice of hardware and the type of system: batch, time shared, single user, multiuser, distributed, real time, or general purpose.
- Beyond this highest design level, the requirements may be much harder to specify. The requirements can, however, be divided into two basic groups
    1. User goals (User requirements)
    2. System goals (system requirements)

- **User requirements** are the features that user care about and understand like system should be convenient to use, easy to learn, reliable, safe and fast.
- **System requirements** are written for the developers, ie. People who design the OS. Their requirements are like easy to design, implement and maintain, flexible, reliable, error free and efficient.

Mechanisms and Policies

- Policies determine *what* is to be done. Mechanisms determine *how* it is to be implemented.
- Example: in timer- counter and decrementing counter is the mechanism and deciding how longthe time has to be set is the policies.
- Policies change overtime. In the worst case, each change in policy would require a change in the underlying mechanism.
- If properly separated and implemented, policy changes can be easily adjusted without re-writing the code, just by adjusting parameters or possibly loading new data / configuration files.

Implementation

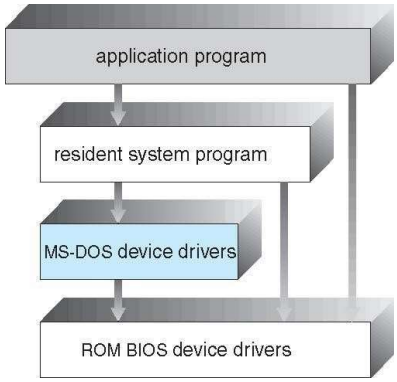
- Traditionally OS were written in assembly language.
- In recent years, OS are written in C, or C++. Critical sections of code are still written inassembly language.
- The first OS that was not written in assembly language, it was the Master ControlProgram (MCP).
- The advantages of using a higher-level language for implementing operating systems are:The code can be written faster, more compact, easy to port to other systems and is easier to understand and debug.
- The only disadvantages of implementing an operating system in a higher-level language are reduced speed and increased storage requirements.

Operating-System Structure

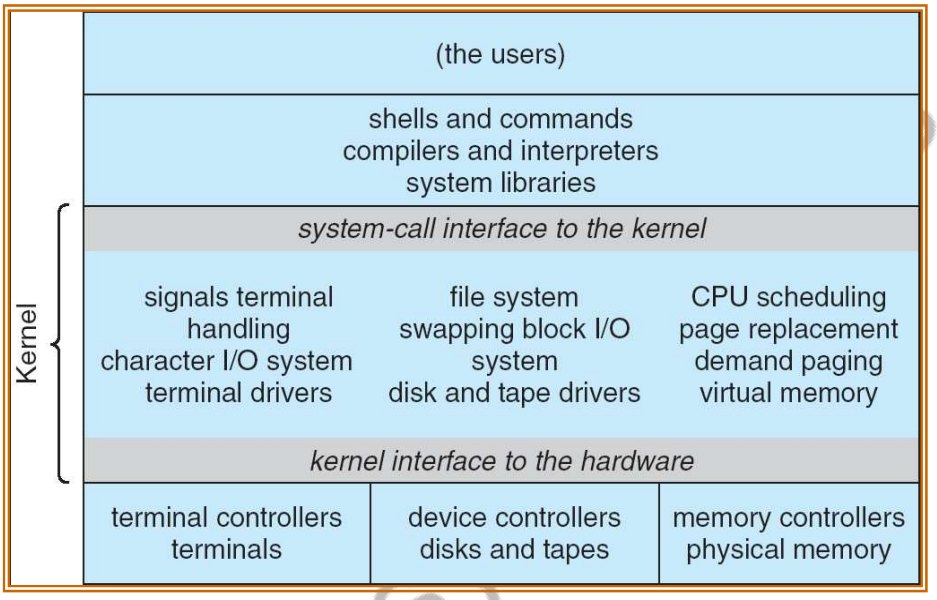
- ✓ Modern OS is large & complex. It consists of different types of components. These components are interconnected & melded into kernel.
- ✓ For designing the system, different types of structures are used. They are,
  - Simple structures.
  - Layered Approach.
  - Micro kernels.

Simple Structure

- ✓ Simple structure OS are small, simple & limited systems. The structure is not well defined.
- ✓ MS-DOS is an example of simple structure OS.MS-DOS layer structure is shown in below **figure**.



- ✓ 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.
- ✓ **UNIX** is another example for simple structure. Initially it was limited by hardware functions.
  - It consists of **two** separable parts: the kernel and the system programs.
  - The kernel is further separated into series of interfaces & device drivers.
  - We can view the traditional UNIX operating system as being layered, as shown in below **figure**.



- ✓ Everything below the system-call interface and above the physical hardware is the kernel.
- ✓ Kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls.
- ✓ This **monolithic** structure was difficult to implement and maintain.

Layered Approach

- The OS is broken into number of layers (levels). Each layer rests on the layer below it, and relies on the services provided by the next lower layer.
- Bottom layer (layer 0) is the hardware and the topmost layer is the user interface.
- A typical layer, consists of data structure and routines that can be invoked by higher-levellayer.
- **Advantage** of layered approach is simplicity of construction and debugging.
- The layers are selected so that each uses functions and services of only lower-level layers. So simplifies debugging and system verification. The layers are debugged one by one from the lowest and if any layer doesn't work, then error is due to that layer only, as the lower layers arealready debugged. Thus, the design and implementation are simplified.

A layer need not know how its lower-level layers are implemented. Thus hides the operations from higher layers.

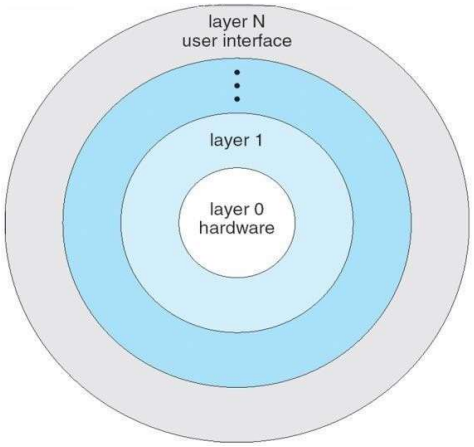


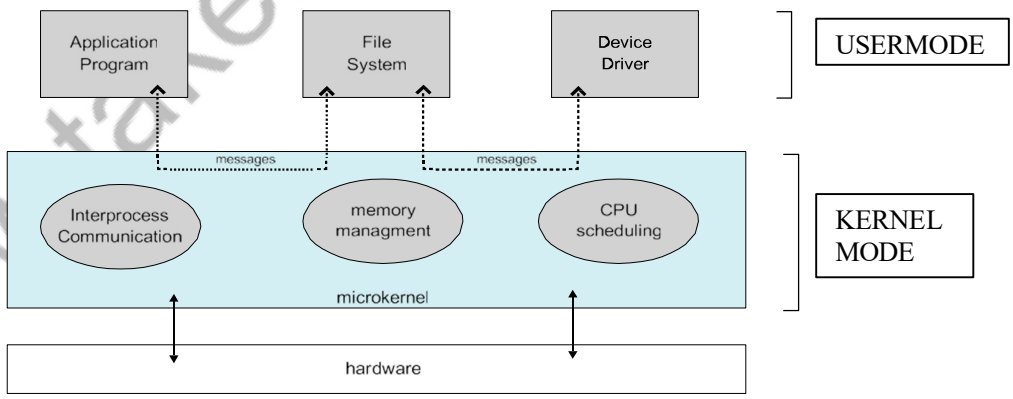
Figure: A layered Operating System

Disadvantages of layered approach:

- The various layers must be appropriately defined, as a layer can use only lower-level layers.
- Less efficient than other types, because any interaction with layer 0 required from top layer.The system call should pass through all the layers and finally to layer 0. This is an overhead.

Microkernels

- This method structures the operating system by removing all nonessential components from thekernel and implementing them as system and user-level programs thus making the kernel as small and efficient as possible.
- The removed services are implemented as system applications.
- Most microkernels provide basic process and memory management, and message passing between other services.
- The main function of the microkernel is to provide a communication facility between the clientprogram and the various services that are also running in user space.



Benefit of microkernel –

- System expansion can also be easier, because it only involves adding more system applications, not rebuilding a new kernel.
- Mach was the first and most widely known microkernel, and now forms a major component Mac OSX.

Modules

- Modern OS development is object-oriented, with a relatively small core kernel and a set of
  modules which can be linked in dynamically.
- Modules are similar to layers in that each subsystem has clearly defined tasks and interfaces, but any module is free to contact any other module, eliminating the problems of going through multiple intermediary layers.
- The kernel is relatively small in this architecture, similar to microkernels, but the kernel does not have to implement message passing since modules are free to contact each other directly. Eg: Solaris, Linux and MacOSX.

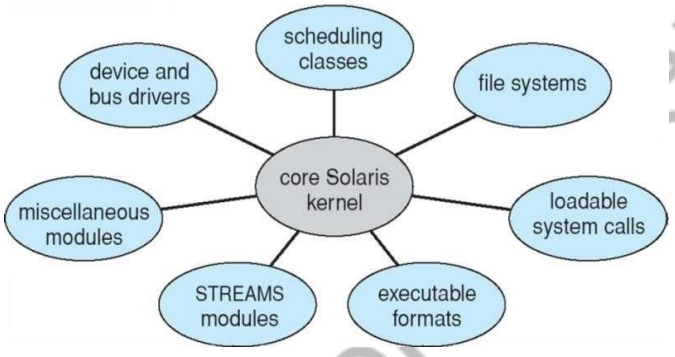


Figure: Solaris loadable modules

- The Max OSX architecture relies on the Mach microkernel for basic system management services, and the BSD kernel for additional services. Application services and dynamically loadable modules (kernel extensions) provide the rest of the OS functionality.
- Resembles layered system, but a module can call any other module.
- Resembles microkernel, the primary module has only core functions and the knowledge of howto load and communicate with other modules.

Virtual Machines

Q) Demonstrate the concept of virtual machine with an example

- The fundamental idea behind a virtual machine is to abstract the hardware of a single computer(the CPU, memory, disk drives, network interface cards, and so forth) into several different execution environments, thereby creating the illusion that each separate execution environmentis running its own private computer.
- Creates an illusion that a process has its own processor with its own memory.
- Host OS is the main OS installed in system and the other OS installed in the system are called guest OS.

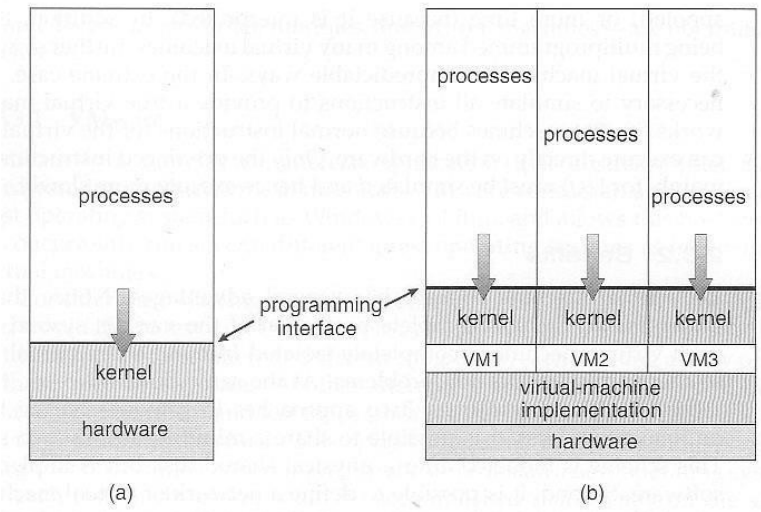


Figure: System modes. (A) Non-virtual machine (b) Virtual machine

Virtual machines first appeared as the VM Operating System for IBM mainframes in 1972.

Implementation

- The virtual-machine concept is useful, it is difficult to implement.
- Work is required to provide an exact duplicate of the underlying machine. Remember that theunderlying machine has two modes: user mode and kernel mode.
- The virtual-machine software can run in kernel mode, since it is the operating system. Thevirtual machine itself can execute in only user mode.

Benefits

- Able to share the same hardware and run several different execution environments (OS).
- Host system is protected from the virtual machines and the virtual machines are protected fromone another. A virus in guest OS, will corrupt that OS but will not affect the other guest systems and host systems.
- Even though the virtual machines are separated from one another, software resources can be shared among them. Two ways of sharing s/w resource for communication are:
  - To share a file system volume (part of memory).
  - To develop a virtual communication network to communicate between the virtual machines.
- The operating system runs on and controls the entire machine. Therefore, the current system must be stopped and taken out of use while changes are made and tested. This period is commonly called *system development time.* In virtual machines such problem is eliminated. User programs are executed in one virtual machine and system development is done in another environment.
- Multiple OS can be running on the developer's system **concurrently**. This helps in rapid porting and testing of programmer's code in different environments.
- **System consolidation** – two or more systems are made to run in a single system.

Simulation –

Here the host system has one system architecture and the guest system is compiled in different architecture. The compiled guest system programs can be run in an emulator that translates each instructions of guest program into native instructions set of host system.

Para-Virtualization –

This presents the guest with a system that is similar but not identical to the guest's preferred system. The guest must be modified to run on the para-virtualized hardware.

Examples

VMware

- VMware is a popular commercial application that abstracts Intel 80X86 hardware into isolated virtual machines. The virtualization tool runs in the user-layer on top of the host OS. Thevirtual machines running in this tool believe they are running on bare hardware, but the fact is that it is running inside a user-level application.
- VMware runs as an application on a host operating system such as Windows or Linux and allows this host system to concurrently run several different **guest operating systems** as independent virtual machines.

In below scenario, Linux is running as the host operating system; FreeBSD, Windows NT, and Windows XP are running as guest operating systems. The virtualization layer is the heart of VMware, as it abstracts the physical hardware into isolated virtual machines running as guest operating systems. Each virtual machine has its own virtual CPU, memory, disk drives, network interfaces, and so forth.

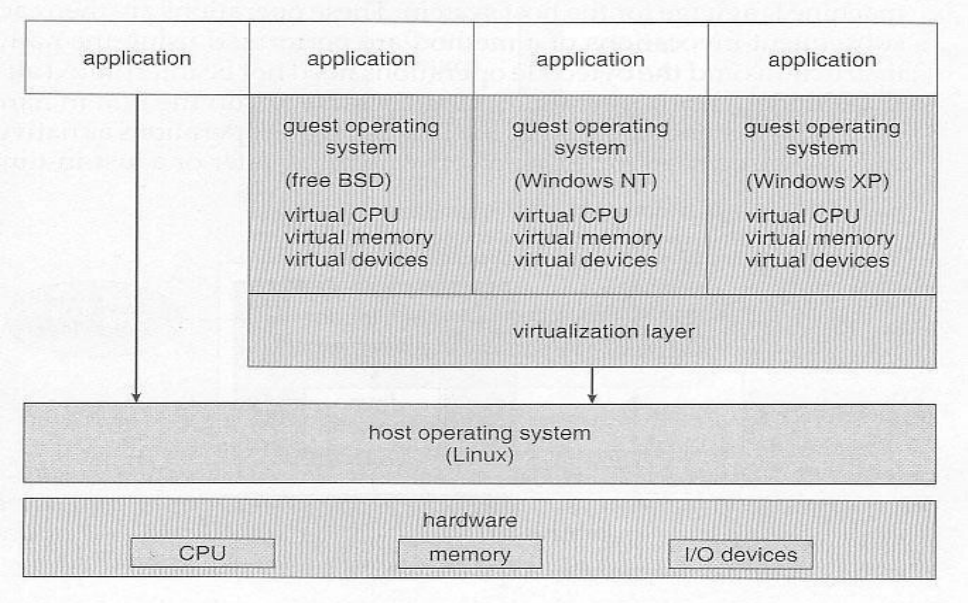


Figure: VMware architecture

The Java Virtual Machine

- Java was designed from the beginning to be platform independent, by running Java only on a Java Virtual Machine, JVM, of which different implementations have been developed for numerous different underlying HW platforms.
- Java source code is compiled into Java byte code in .class files. Java byte code is binary instructions that will run on the JVM.
- The JVM implements memory management and garbage collection.
- JVM consists of class loader and Java Interpreter. Class loader loads compiled .class files from both Java program and Java API for the execution of Java interpreter. Then it checks the .class file for validity.

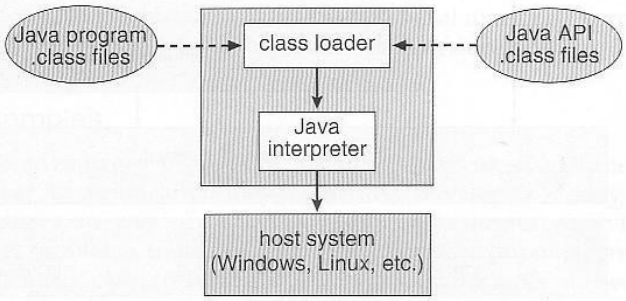


Figure: The JVM

Operating System Generation

- Operating systems are designed to run on any of a class of machines; the system must beconfigured for each specific computer site.
- SYSGEN program obtains information concerning the specific configuration of the hardwaresystem.
    - *Booting* – starting a computer by loading the kernel.
    - *Bootstrap program* – code stored in ROM that is able to locate the kernel, load it intomemory, 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 itSometimes two-step process where **boot block** at fixed location loads bootstrap loader.
- When power initialized on system, execution starts at a fixed memory location Firmware usedto hold initial boot code

MODULE: 2

CONTENTS:

- **Process Management**:
  - Process concept;
  - Process scheduling;
  - Operations on processes;
  - Inter process communication
- **Multi-threaded Programming**:
  - Multithreading models
  - Thread Libraries
  - Threading issues
- **Process Scheduling**:
  - Basic concepts
  - Scheduling Criteria
  - Scheduling Algorithms
  - Multiple processor scheduling
  - Thread scheduling

MODULE 2

PROCESS MANAGEMENT

Process Concept

- A process is a program under execution.
- Its current activity is indicated by PC (Program Counter) and the contents of the processor'sregisters.

The Process: ***Process memory is divided into four sections as shown in the figure below:***

- The **stack** is used to store temporary data such as local variables, function parameters, functionreturn values, return address etc.
- The **heap** which is memory that is dynamically allocated during process run time
- The **data** section stores global variables.
- The **text** section comprises the compiled program code.
- Note that, there is a free space between the stack and the heap. When the stack is full, it growsdownwards and when the heap is full, it grows upwards.

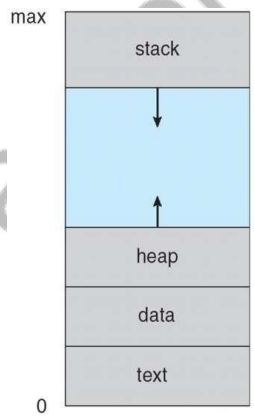


Figure: Process in memory.

Process State

Q) Illustrate with a neat sketch, the process states and process control block.

Process State

A Process has 5 states. Each process may be in one of the following states –

1. **New** - The process is in the stage of being created.
2. **Ready** - The process has all the resources it needs to run. It is waiting to be assigned tothe processor.
3. **Running** – Instructions are being executed.
4. **Waiting** - The process is waiting for some event to occur. For example, the process may be waiting for keyboard input, disk access request, inter-process messages, a timer to go off, or a child process to finish.
5. **Terminated** - The process has completed its execution.

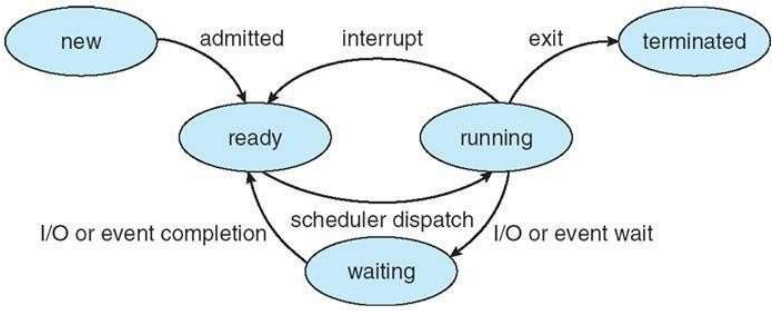


Figure: Diagram of process state

Process Control Block

For each process there is a Process Control Block (PCB), which stores the process-specificinformation as shown below –

- **Process State** – The state of the process may be new, ready, running, waiting, and so on.
- **Program counter** – The counter indicates the address of the next instruction to be executed forthis process.
- **CPU registers** - The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.
- **CPU scheduling information**- This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Memory-management information** – This includes information such as the value of the base and limit registers, the page tables, or the segment tables.
- **Accounting information** – This information includes the amount of CPU and real time used, time limits, account numbers, 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.

The PCB simply serves as the repository for any information that may vary from process to process.

Figure: Process control block (PCB)

Context switching

- The task of switching a CPU from one process to another process is called context switching. Context-switch times are highly dependent on hardware support (Number of CPU registers).
- Whenever an interrupt occurs (hardware or software interrupt), the state of the currentlyrunning process is saved into the PCB and the state of another process is restored from the PCBto the CPU.
- Context switch time is an overhead, as the system does not do useful work while switching.

CPU Switch from Process to Process

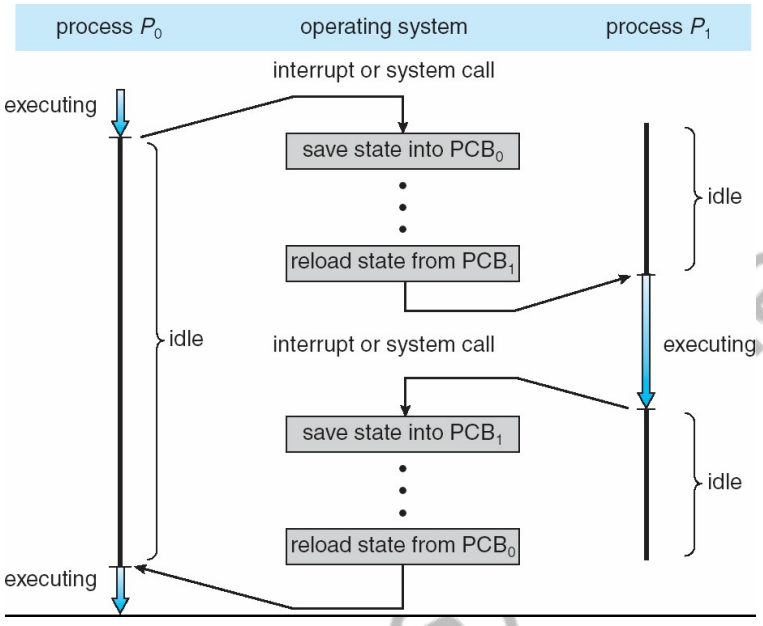


Figure: Diagram showing CPU switch from process to process.

Process Scheduling

Scheduling Queues

- As processes enter the system, they are put into a job queue, which consists of all processes inthe system.
- The processes that are residing in main memory and are ready and waiting to execute are kepton 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 PCBincludes a pointer field that points to the next PCB in the ready queue.

Ready Queue and Various I/O Device Queues

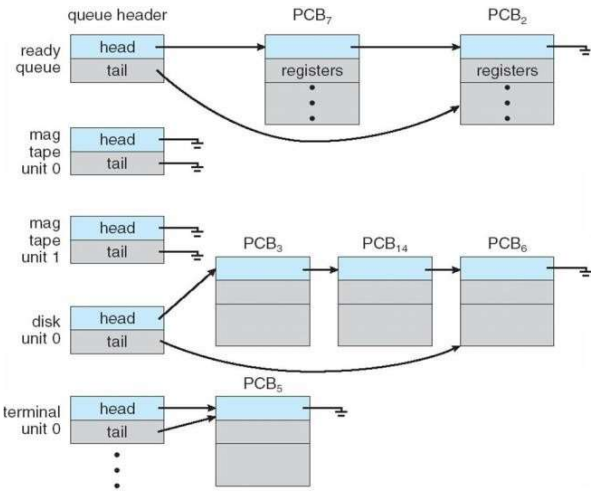


Figure: The ready queue and various I/O device queues

A common representation of process scheduling is a ***queueing diagram***. Each rectangular box in the diagram represents a queue. Two types of queues are present: the ready queue and a setof 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 in the ready queue until it is selected for execution and is given the CPU. 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 its termination.
    - The process could be removed forcibly from the CPU, as a result of an interrupt,and be put back in the ready queue.

In the first two cases, the process eventually switches from the waiting state to the ready state, and is then put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues.

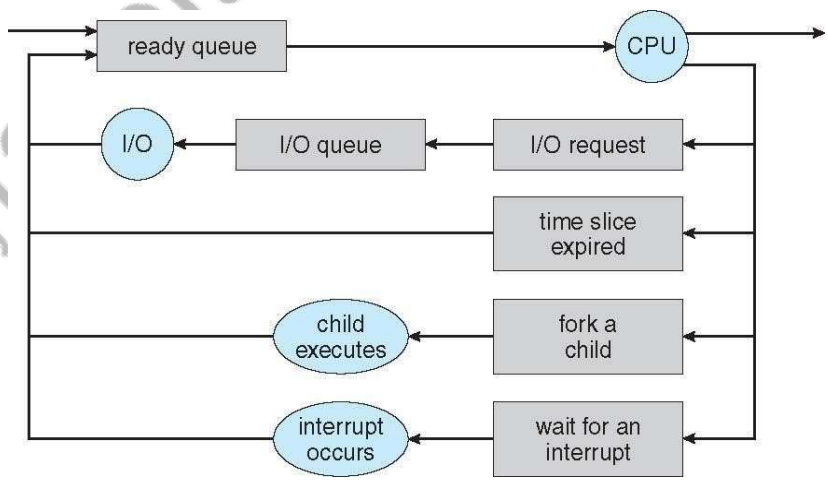


Figure: Queueing-diagram representation of process scheduling.

Schedulers

Schedulers are software which selects an available program to be assigned to CPU.

- A **long-term scheduler or Job scheduler** – selects jobs from the job pool (of secondary

memory, disk) and loads them into the memory.
If more processes are submitted, than that can be executed immediately, such processes will be in secondary memory. It runs infrequently, and can take time to select the next process.

- **The short-term scheduler, or CPU Scheduler** – selects job from memory and assigns theCPU to it. It must select the new process for CPU frequently.
- **The medium-term scheduler** - selects the process in ready queue and reintroduced into the memory.

Processes can be described as either:

- I/O-bound process – spends more time doing I/O than computations,
- CPU-bound process – spends more time doing computations and few I/O operations.

An efficient scheduling system will select a good mix of **CPU-bound** processes and **I/O bound** processes.

- If the scheduler selects **more I/O bound process**, then I/O queue will be full and readyqueue will be empty.
- If the scheduler selects **more CPU bound process**, then ready queue will be full and I/Oqueue will be empty.

Time sharing systems employ a **medium-term scheduler**. It swaps out the process from ready queue and swap in the process to ready queue. When system loads get high, this scheduler will swap one or more processes out of the ready queue for a few seconds, in order to allow smaller faster jobs to finish up quickly and clear the system.

Advantages of medium-term scheduler –

- To remove process from memory and thus reduce the degree of multiprogramming(number of processes in memory).
- To make a proper mix of processes (CPU bound and I/O bound)

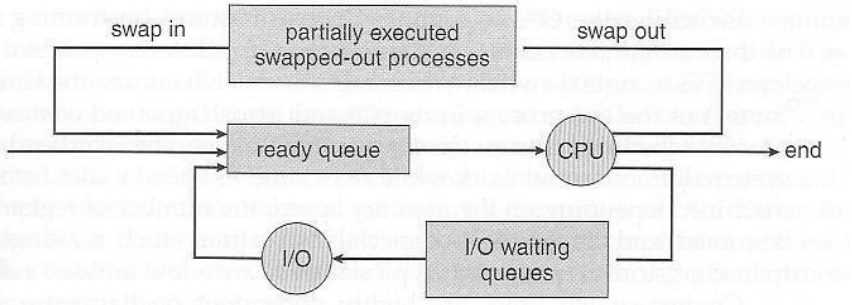


Figure 3.8 Addition of medium-term scheduling to the queueing diagram.

Operations on Processes

Q) Demonstrate the operations of process creation and process termination in UNIXProcess Creation

- A process may create several new processes. The creating process is called a **parent** process, and the new processes are called the **children** of that process. Each of these newprocesses may in turn create other processes. Every process has a unique process ID.

- On typical Solaris systems, the process at the top of the tree is the '**sched**' process with PID of 0. The '**sched**' process creates several children processes – **init**, **pageout** and **fsflush**. Pageout and fsflush are responsible for managing memory and file systems. The init process with a PID of 1, serves as a parent process for all user processes.

A process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its

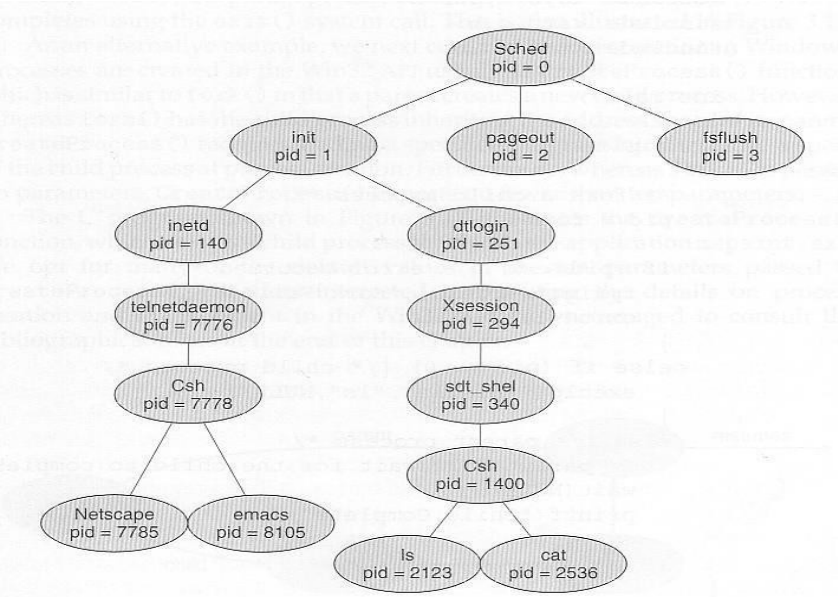


Figure 3.9 A tree of processes on a typical Solaris system.

task. When a process creates a subprocess, the subprocess may be able to obtain its resources in two ways:

- directly from the operating system
- Subprocess may take the resources of the parent process. The resource can be taken from parent in two ways –
  - The parent may have to partition its resources among its children
  - Share the resources among several children.

There are two options for the parent process after creating the child:

- Wait for the child process to terminate and then continue execution. The parent makes a wait()system call.
- Run concurrently with the child, continuing to execute without waiting.

Two possibilities for the address space of the child relative to the parent:

- The child may be an exact duplicate of the parent, sharing the same program and data segments in memory. Each will have their own PCB, including program counter, registers, and PID. This is the behaviour of the **fork** system call in UNIX.
- The child process may have a new program loaded into its address space, with all new code and data segments. This is the behaviour of the **spawn** system calls in Windows.

```
#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 */
       execlp("/bin/ls","ls",NULL);
    }
    else {/* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
       exit(0);
    }
}
```

Figure 3.10 C program forking a separate process.

In UNIX OS, a child process can be created by **fork()** system call. The **fork** system call, if successful, returns the PID of the child process to its parents and returns a zero to the child process. If failure, it returns -1 to the parent. Process IDs of current process or its direct parent can be accessed using the getpid( ) and getppid( ) system calls respectively.

The parent waits for the child process to complete with the wait() system call. When the child process completes, the parent process resumes and completes its execution.

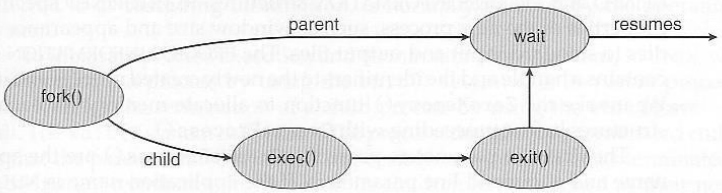


Figure 3.11 Process creation.

In windows the child process is created using the function **createprocess( )**. The createprocess( ) returns 1, if the child is created and returns 0, if the child is not created.

Process Termination

- A process terminates when it finishes executing its last statement and asks the operating systemto delete it, by using the **exit** () system call. All of the resources assigned to the process like memory, open files, and I/O buffers, are deallocated by the operating system.
- A process can cause the termination of another process by using appropriate system call. The parent process can terminate its child processes by knowing of the PID of the child.
- A parent may terminate the execution of children for a variety of reasons, such as:

- The child has exceeded its usage of the resources, it has been allocated.
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system terminates all the children. This is called **cascading termination**.

Interprocess Communication

Q) What is interprocess communication? Explain types of IPC.

Interprocess Communication- Processes executing may be either co-operative or independent processes.

- ***Independent Processes*** – processes that cannot affect other processes or be affected by other processes executing in the system.
- ***Cooperating Processes*** – processes that can affect other processes or be affected by other processes executing in the system.

Co-operation among processes are allowed for following reasons –

- **Information Sharing** - There may be several processes which need to access the same file. Sothe information must be accessible at the same time to all users.
- **Computation speedup** - Often a solution to a problem can be solved faster if the problem canbe broken down into sub-tasks, which are solved simultaneously (particularly when multiple processors are involved.)
- **Modularity** - A system can be divided into cooperating modules and executed by sending information among one another.
- **Convenience** - Even a single user can work on multiple tasks by information sharing.

Cooperating processes require some type of inter-process communication. This is allowed by two models:

1. Shared Memory systems
2. Message passing systems.

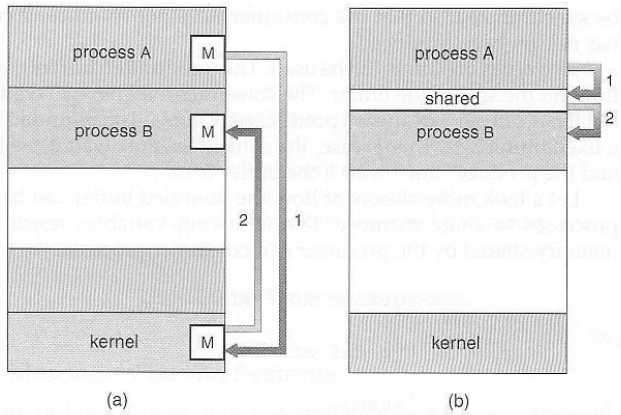


Figure 3.13 Communications models. (a) Message passing. (b) Shared memory.

Sl No	Shared Memory	Message passing

1.	A region of memory is shared by communicating processes, into which the information is written and read	Message exchange is done among the processes by using objects.
2.	Useful for sending large block of data	Useful for sending small data.
3.	System call is used only to create shared memory	System call is used during every read and write operation.
4.	Message is sent faster, as there are no system calls	Message is communicated slowly.

- **Shared Memory** is faster once it is set up, because no system calls are required and access occurs at normal memory speeds. Shared memory is generally preferable when large amounts of information must be shared quickly on the same computer.
- **Message Passing** requires system calls for every message transfer, and is therefore slower, but it is simpler to set up and works well across multiple computers. Message passing is generally preferable when the amount and/or frequency of data transfers is small.

Shared-Memory Systems

- A region of shared-memory is created within the address space of a process, which needs to communicate. Other process that needs to communicate uses this shared memory.
- The form of data and position of creating shared memory area is decided by the process. Generally, a few messages must be passed back and forth between the cooperating processes first in order to set up and coordinate the shared memory access.
- The process should take care that the two processes will not write the data to the shared memory at the same time.

Producer-Consumer Example Using Shared Memory

- This is a classic example, in which one process is producing data and another process is consuming the data.
- The data is passed via an intermediary buffer (shared memory). The producer puts the data to the buffer and the consumer takes out the data from the buffer. 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. In this situation, the consumer must wait until an item is produced.
- There are two types of buffers into which information can be put –
  - Unbounded buffer
  - Bounded buffer

- **With Unbounded buffer**, there is no limit on the size of the buffer, and so on the data produced by producer. But the consumer may have to wait for new items.

- **With bounded-buffer** – As the buffer size is fixed. The producer has to wait if the buffer isfull and the consumer has to wait if the buffer is empty.

This example uses shared memory as a circular queue. The **in** and **out** are two pointers to the array. Note in the code below that only the producer changes "in", and only the consumer changes "out".

First the following data is set up in the shared memory area:

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
}item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

The producer process –

Note that the buffer is full when [(in+1) % BUFFER_SIZE == out]

```
item nextProduced;

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

Figure The producer process.

The consumer process –

Note that the buffer is empty when [in == out]

```
item nextConsumed;

while (true) {
     while (in == out)
        ; //do nothing

     nextConsumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     /* consume the item in nextConsumed */
}
```

Figure The consumer process.

Message-Passing Systems

A mechanism to allow process communication without sharing address space. It is used in distributed systems.

- Message passing systems uses system calls for "send message" and "receive message".

- A communication link must be established between the cooperating processes before messagescan be sent.
- There are three methods of creating the link between the sender and the receiver-
  - Direct or indirect communication (naming)
  - Synchronous or asynchronous communication (Synchronization)
  - Automatic or explicit buffering.

1. Naming

Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.

a) Direct communication the sender and receiver must explicitly know each other's name. The syntaxfor send() and receive() functions are as follows-

- **send** (*P, message*) – send a message to process P
- **receive**(*Q, message*) – receive a message from process Q

Properties of communication link:

- A link is established automatically between every pair of processes that wants tocommunicate. The processes need to know only each other's identity to communicate.
- A link is associated with exactly one pair of communicating processes
- Between each pair, there exists exactly one link.

Types of addressing in direct communication –

- Symmetric addressing – the above-described communication is symmetric communication.Here both the sender and the receiver processes have to name each other to communicate.
- Asymmetric addressing – Here only the sender's name is mentioned, but the receiving datacan be from any system.

  send (P, message) --- Send a message to process ***P***
  receive (id, message). Receive a message from any process

Disadvantages of direct communication – any changes in the identifier of a process, may have to change the identifier in the whole system (sender and receiver), where the messages are sent and received.

b) Indirect communication uses shared mailboxes, or ports.

A mailbox or port is used to send and receive messages. Mailbox is an object into which messages can be sent and received. It has a unique ID. Using this identifier messages are sent and received. Two processes can communicate only if they have a shared mailbox. The send and receive functions are –

- **send** (*A, message*) – send a message to mailbox A
- **receive** (*A, message*) – receive a message from mailbox A

Properties of communication link:

- A link is established between a pair of processes only if they have a shared mailbox
- A link may be associated with more than two processes
- Between each pair of communicating processes, there may be any number of links, each linkis associated with one mailbox.
- A mail box can be owned by the operating system. It must take steps to –
  - create a new mailbox
  - send and receive messages from mailbox
  - delete mailboxes.

2. Synchronization

The send and receive messages can be implemented as either **blocking** or **non-blocking**.

Blocking (synchronous) send - sending process is blocked (waits) until the message isreceived by receiving process or the mailbox.

Non-blocking (asynchronous) send - sends the message and continues (does not wait)

Blocking (synchronous) receive - The receiving process is blocked until a message isavailable

Non-blocking (asynchronous) receive - receives the message without block. Thereceived message may be a valid message or null.

3. Buffering

When messages are passed, a temporary queue is created. Such queue can be of three capacities:

Zero capacity – The buffer size is zero (buffer does not exist). Messages are not stored in the queue. The senders must block until receivers accept the messages.

Bounded capacity- The queue is of fixed size(n). Senders must block if the queue is full. After sending 'n' bytes the sender is blocked.

Unbounded capacity - **The queue is of infinite capacity. The sender never blocks.**

MULTITHREADED PROGRAMMING

- A thread is a basic unit of CPU utilization.
- It consists of
  - thread ID
  - PC
  - register-set and
  - stack.
- It shares with other threads belonging to the same process its code-section &data-section.
- A traditional (or heavy weight) process has a single thread of control.
- If a process has multiple threads of control, it can perform more than one task at a time.such a process is called **multithreaded process**

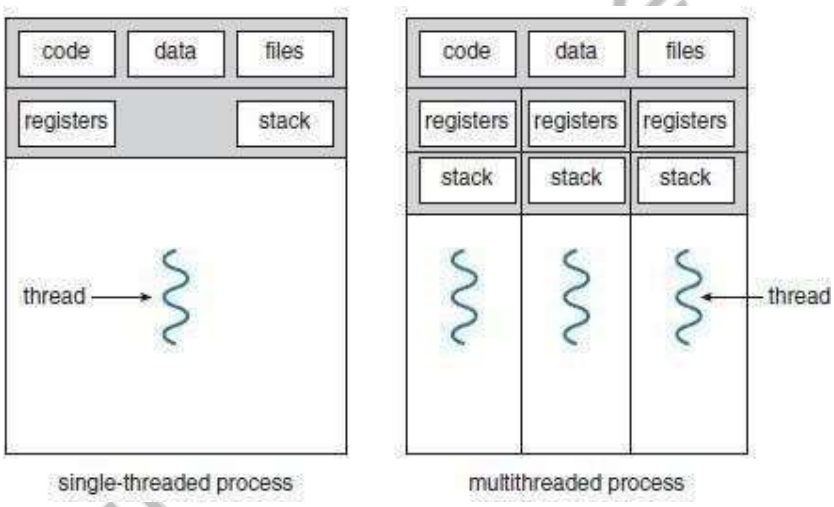


Fig: Single-threaded and multithreaded processes

Motivation for Multithreaded Programming

1. The software-packages that run on modern PCs are multithreaded. An application is implemented as a separate process with several threads of control. For ex: A word processor may have

- First thread for displaying graphics
- Second thread for responding to keystrokes and
- Third thread for performing grammar checking.

✓ As **process creation** takes **more time** than **thread creation** it is more efficient to use process that contains multiple threads. So, that the amount of time that a client have to wait for its request to be serviced from the web server will be less.

✓ Threads also play an important role in **remote procedure call.**

2. In some situations, a single application may be required to perform several similar tasks. For ex:A web-server may create a separate thread for each client requests. This allows the server to service several concurrent requests.

3. RPC servers are multithreaded.

- When a server receives a message, it services the message using separateconcurrent threads.

4. Most OS kernels are multithreaded;

- Several threads operate in kernel, and each thread performs a specific task,such as managing devices or interrupt handling.

Benefits of Multithreaded Programming

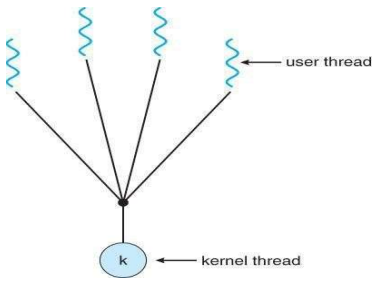
- ***Responsiveness*** A program may be allowed to continue running even if part of it is blocked. Thus, increasing responsiveness to the user.
- ***Resource Sharing*** By default, threads share the memory (and resources) of the process to which they belong. Thus, an application is allowed to have severaldifferent threads of activity within the same address-space.
- ***Economy*** Allocating memory and resources for process-creation is costly. Thus, it is more economical to create and context-switch threads.
- ***Utilization of Multiprocessor Architectures*** In a multiprocessor architecture, threads may be running in parallel on different processors. Thus, parallelism will be increased.

MULTITHREADING MODELS

- Support for threads may be provided at either,
  1. The user level, for **user threads** or
  2. By the kernel, for **kernel threads**.
- User-threads are supported above the kernel and are managed without kernel support. Kernel-threads are supported and managed directly by the OS.
- Three ways of establishing relationship between user-threads &kernel-threads:
  1. Many-to-one model
  2. One-to-one model and
  3. Many-to-many model.

Many-to-One Model

- Many user-level threads are mapped to one kernel thread.



Advantages:

- Thread management is done by the thread library in user space, so it is efficient.

Disadvantages:

- The entire process will block if a thread makes a blocking system-call.
- Multiple threads are unable to run in parallel on multiprocessors.

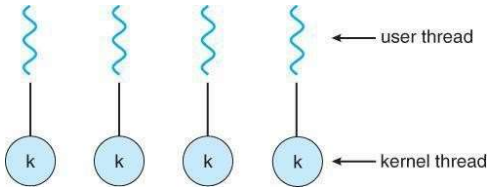
- For example:
  - Solaris green threads
  - GNU portable threads.

One-to-One Model

- Each user thread is mapped to a kernel thread.

Advantages:

- It provides more concurrency by allowing another thread to run when a threadmakes a blocking system-call.
- Multiple threads can run in parallel on multiprocessors.



Disadvantage:

- Creating a user thread requires creating the corresponding kernel thread.

- For example:
  - Windows NT/XP/2000, Linux

Many-to-Many Model

- Many user-level threads are multiplexed to a smaller number of kernel threads.

Advantages:

- Developers can create as many user threads as necessary
- The kernel threads can run in parallel on a multiprocessor.
- When a thread performs a blocking system-call, kernel can schedule another threadfor execution.

Two Level Model

- A variation on the many-to-many model is the two level-model
- Similar to M:N, except that it allows a user thread to be bound to kernel thread.
- For example:
  - HP-UX
  - Tru64 UNIX

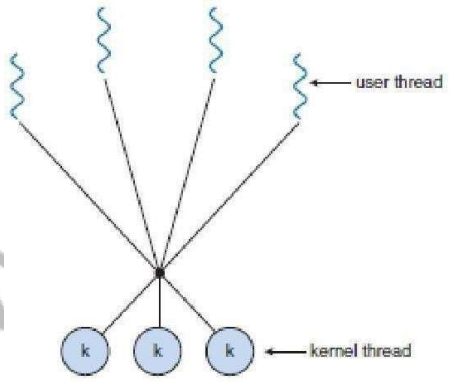


Fig: Many-to-many model

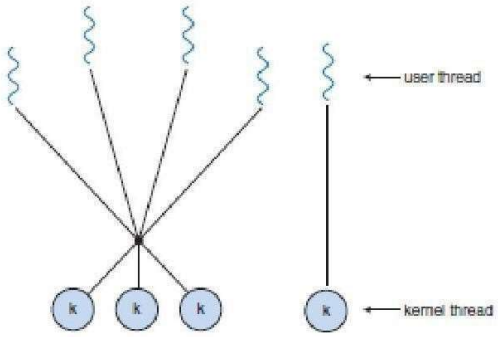


Fig: Two-level model

Thread Libraries

- It provides the programmer with an API for the creation and management of threads.
- Two ways of implementation:

1. *First Approach:*

   Provides a library entirely in user space with no kernel support. All code and data structuresfor the library exist in the user space.

2. *Second Approach*

   Provides a library entirely in user space with no kernel support. All code and data structuresfor the library exist in the user space.

Three main thread libraries:

1. **POSIX Pthreads**: extension of posix standard, they may be provided as either a user nor kernel library.
2. **Win32:** is a kernel level library available on windows systems.
3. **Java threads**: API allows creation and management directly in Java programs. However, on windows java threads are implemented using win32 and on UNIX and Linux using Pthreads.

Pthreads

- This is a POSIX standard API for thread creation and synchronization.
- This is a specification for thread-behavior, not an implementation.
- OS designers may implement the specification in any way they wish.
- Commonly used in: UNIX and Solaris.

Multithreaded C program using the Pthreads API

```
#include <pthread.h>
#include <stdio.h>
int sum;                                /* this data is shared by the thread(s) */
void *runner(void *param);               /* the thread */
int main(int argc, char *argv[])
{
pthread_t  tid;                         /* the thread identifier */
pthread_attr_t attr;                   /* set of thread attributes */

if (argc != 2)
{
fprintf(stderr,"usage: a.out <integer value>\n");
return -1;
}
if (atoi(argv[1]) < 0)
{
fprintf(stderr,"%d must be>= 0\n",atoi(argv[1]));
return -1;
}
pthread_attr_init(&attr);                        /* get the default attributes */
pthread_create(&tid,&attr,runner,argv[1]);     /* create the thread */
pthread_join(tid,NULL);                         /* wait for the thread to exit */
printf("sum = %d\n",sum);
}
```

```
void *runner(void *param) /* The thread will begin control in this function */
{
int i, upper= atoi(param);
sum = 0;
for (i = 1; i <= upper; i++)
      sum += i;
pthread_exit(0) ;
}
```

Win32 threads

- The Win32 thread library is a kernel-level library available on Windows systems.
- Implements the one-to-one mapping
- Each thread contains
  - A thread id
  - Register set
  - Separate user and kernel stacks
  - Private data storage area
- The register set, stacks, and private storage area are known as the context of the threads The primary data structures of a thread include:
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)

✓ The technique for creating threads using the Win32 thread library is similar to the Pthreads technique in several ways. We must include the windows.h header file when using the Win32 API.

✓ Threads are created in the Win32 API using the CreateThread() function and a set of attributes for the thread is passed to this function.

✓ These attributes include security information, the size of the stack, and a flag that can be set to indicate if the thread is to start in a suspended state.

✓ The parent thread waits for the child thread using the WaitForSingleObject() function, which causes the creating thread to block until the summation thread has exited.

Multithreaded C program using the Win32 API

```
#include <Windows.h>
#include <stdio.h>
DWORD Sum;      /* data is shared by the thread(s) */
               /* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
DWORD Upper = *(DWORD*)Param;
for (DWORD i = 0; i <= Upper; i++)
      Sum += i;
      return 0;
}

int main(int argc, char *argv[])
{
DWORD ThreadId;
```

```
 HANDLE ThreadHandle;
 int Param;

 if (argc != 2)          /* perform some basic error checking */
 {
       fprintf(stderr," An integer parameter is required\n");
       return -1;
 }
 Param = atoi(argv[1]);
 if (Param < 0)
 {
      fprintf(stderr,"An integer>= 0 is required\n");
      return -1;
 }

 /*create the thread*/
 ThreadHandle = CreateThread(NULL, 0, Summation, &Param, 0, &ThreadId);

 // NULL: default security attributes
 // 0: default stack size
 // Summation:  thread function
// &Param: parameter to thread function
// 0: default creation flags
//ThreadId:  returns the thread identifier

 if (ThreadHandle != NULL)
  {
   WaitForSingleObject(ThreadHandle,INFINITE); //now wait for the thread to finish
   CloseHandle(ThreadHandle);                  // close the thread handle
    printf("sum = %d\n" ,Sum);
 }
 }
```

Java Threads

- Threads are the basic model of program-execution in
  - Java program and
  - Java language.
- The API provides a rich set of features for the creation and management of threads.
- All Java programs comprise at least a single thread of control.
- Two techniques for creating threads:

1. Create a new class that is derived from the Thread class and override its run() method.

Define a class that implements the Runnable interface. The Runnable interface is defined as follows:

```
public interface Runnable
{
    public abstract void run();
}
```

- ✓ When a class implements Runnable, it must define a **run()** method. The code implementing the run() method runs as a separate thread.
- ✓ Creating a Thread object does not specifically create the new thread but it is the **start()** method that actually creates the new thread. Calling the **start()** method for the new object does two things:
  - It allocates memory and initializes a new thread in the JVM.
  - It calls the **run()** method, making the thread eligible to be run by the JVM.
- ✓ As Java is a **pure object-oriented** language, it has **no notion of global data.** If two or more threads have to share data means then the sharing occurs by passing reference to the shared object to the appropriate threads.
- ✓ This shared object is referenced through the appropriate **getSum() and setSum()** methods.
- ✓ As the Integer class is **immutable**, that is, once its value is set it cannot change, a new **sum** class is designed.
- ✓ The parent threads in Java uses join() method to wait for the child threads to finish before proceeding.

Java program for the summation of a non-negative integer.

```
class Sum
{
private int sum;
          public int getSum()
               {
                   return sum;
                }

         public void setSum(int sum)
              {
                  this.sum = sum;
                }
}

class Summation implements Runnable
{
private int upper;
private Sum  sumValue;

   public Summation(int upper, Sum sumValue)
        {
            this.upper = upper;
            this.sumValue = sumValue;
        }

  public void run()
       {
            int sum = 0;
            for (int i = 0; i <= upper; i++)
            sum += i;
            sumValue.setSum(sum);
       }
}

public class Driver
 {
         public static void main(String[]  args)
          {
```

```
                if (args.length > 0)
                   {
                        if (Integer.parseint(args[O]) < 0)
                                  System.err.println(args[O] + "must be>= 0.");
                        else
                             {
                          Sum sumObject = new Sum();       //create the object to be
shared
                                 int upper= Integer.parseint(args[O]);
                                 Thread thrd =new Thread(new Summation(upper,
sumObject));
                                 thrd.start();
                                 try
                                {
                     thrd. join () ;
                                  System.out.println("The sum of "+upper+" is
"+sumObject.getSum());
                                }
                                 catch (InterruptedException ie) { }
                             }
            }
                   else
                   System.err.println("Usage: Summation <integer value>");
 }
```

THREADING ISSUES

fork() and exec() System-calls

- fork() is used to create a separate, duplicate process.
- If one thread in a program calls fork(),then
    1. Some systems duplicate all threads and
    2. Other systems duplicate only the thread that invoked the fork().
- If a thread invokes the exec(), the program specified in the parameter to exec() will replace the entire process including all threads.

Thread Cancellation

- This is the task of terminating a thread before it has been completed.
- Target thread is the thread that is to be cancelled.
- Thread cancellation occurs in two different cases:
    1. ***Asynchronous cancellation***: One thread immediately terminates the target thread.
    2. ***Deferred cancellation***: The target thread periodically checks whether it should be terminated.

Signal Handling

- In UNIX, a signal is used to notify a process that a particular event has occurred.
- All signals follow this pattern:

1. A signal is generated by the occurrence of a certain event.
2. A generated signal is delivered to a process.
3. Once delivered, the signal must be handled.

- A signal handler is used to process signals.
- A signal may be received either synchronously or asynchronously, depending on the source.

1. ***Synchronous signals***
   - Delivered to the same process that performed the operation causing the signal.
   - E.g. illegal memory access and division by 0.
2. ***Asynchronous signals***
   - Generated by an event external to a running process.
   - E.g. user terminating a process with specific keystrokes<ctrl><c>.

- Every signal can be handled by one of two possible handlers:

1. ***A Default Signal Handler***
   - Run by the kernel when handling the signal.
2. ***A User-defined Signal Handler***
   - Overrides the default signal handler.

- In ***single-threaded programs***, delivering signals is simple (since signals are always delivered to a process).
- In ***multithreaded programs***, delivering signals is more complex. Then, the following options exist:
  1. Deliver the signal to the thread to which the signal applies.
  2. Deliver the signal to every thread in process
  3. Deliver the signal to certain threads in the process.
  4. Assign a specific thread to receive all signals for the process.

THREAD POOLS

- The basic idea is to
  - create a no. of threads at process-startup and
  - place the threads into a pool (where they sit and wait for work).
- Procedure:
  1. When a server receives a request, it awakens a thread from the pool.
  2. If any thread is available, the request is passed to it for service.
  3. Once the service is completed, the thread returns to the pool.
- Advantages:
  - Servicing a request with an existing thread is usually faster than waiting tocreate a thread.
  - The pool limits the no. of threads that exist at any one point.

- No. of threads in the pool can be based on actors such as
  - no. of CPUs
  - amount of memory and
  - expected no. of concurrent client-requests.

THREAD SPECIFIC DATA

- Threads belonging to a process share the data of the process.
- this sharing of data provides one of the benefits of multithreaded programming.
- In some circumstances, each thread might need its own copy of certain data. We will call suchdata **thread-specific data**.
- For example, in a transaction-processing system, we might service each transaction in a separate thread.
- Furthermore, each transaction may be assigned a unique identifier. To associateeach thread with its unique identifier, we could use thread-specific data.

SCHEDULER ACTIVATIONS

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application.
- Scheduler activations provide **upcalls** a communication mechanism from the kernel to the thread library
- This communication allows an application to maintain the correct number kernel threads
- One scheme for communication between the user-thread library and the kernel isknown as **scheduler activation.**

➢ Differences

✓ The differences between process and thread are,

	Process	Thread
1.	It is called heavyweight process.	It is called lightweight process.
2.	Process switching needs interface with OS.	Thread switching does not need interface with OS.
3.	Multiple processes use more resources than multiple threads.	Multiple threaded processes use fewer resources than multiple processes.
4.	In multiple process implementations each process executes same code but has its own memory and file resources.	All threads can share same set of open files.
5.	If one server process is blocked no other server process can execute until the first process unblocked.	While one server thread is blocked and waiting, second thread in the same task could run.
6.	In multiple processes each process operates independently of others.	One thread can read, write or even completely wipeout another threads stack.

PROCESS SCHEDULING

Basic Concepts

- In a single-processor system,
  - Only one process may run at a time.
  - Other processes must wait until the CPU is rescheduled.
- Objective of multiprogramming:
  - To have some process running at all times, in order to maximize CPUutilization.

CPU-I/0 Burst Cycle

- Process execution consists of a cycle of
  - CPU execution and
  - I/O wait
- Process execution begins with a CPU burst, followed by an I/O burst, then another CPU burst, etc…
- Finally, a CPU burst ends with a request to terminate execution.
- An I/O-bound program typically has many short CPU bursts.
- A CPU-bound program might have a few long CPU bursts.

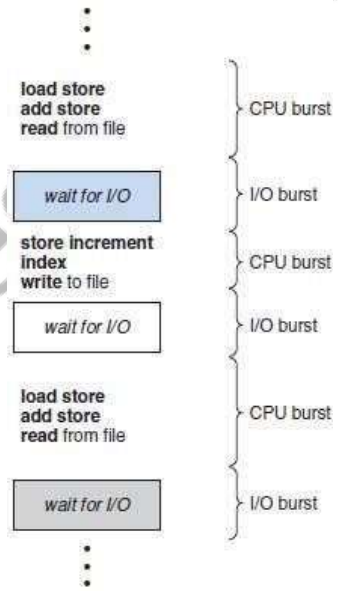


Fig Alternating sequence of CPU and I/O bursts

✓ The duration of CPU bursts vary from process to process and from computer to computer.

✓ The **frequency curve** is as shown below **figure.**

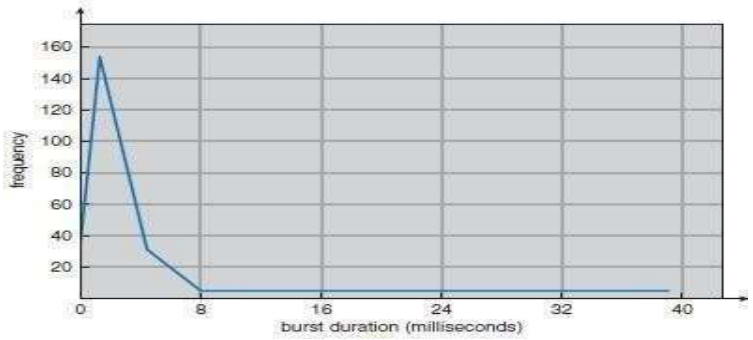


Fig: Histogram of CPU-burst durations

CPU Scheduler

- This scheduler
  - selects a waiting-process from the ready-queue and
  - allocates CPU to the waiting-process.
- The ready-queue could be a FIFO, priority queue, tree and list.
- The records in the queues are generally process control blocks (PCBs) of the processes.

CPU Scheduling

- Four situations under which CPU scheduling decisions take place:
  1. When a process switches from the running state to the waiting state. For ex; I/Orequest.
  2. When a process switches from the running state to the ready state. For ex:when an interrupt occurs.
  3. When a process switches from the waiting state to the ready state. For ex:completion of I/O.
  4. When a process terminates.
- Scheduling under 1 and 4 is non- preemptive. Scheduling under 2 and 3 is preemptive.

Non Preemptive Scheduling

- Once the CPU has been allocated to a process, the process keeps the CPU until itreleases the CPU either
  - by terminating or
  - by switching to the waiting state.

Preemptive Scheduling

- This is driven by the idea of prioritized computation.
- Processes that are runnable may be temporarily suspended
- Disadvantages:
  1. Incurs a cost associated with access to shared-data.
  2. Affects the design of the OS kernel.

Dispatcher

- It gives control of the CPU to the process selected by the short-term scheduler.
- The function involves:
  1. Switching context
  2. Switching to user mode&
  3. Jumping to the proper location in the user program to restart that program
- It should be as fast as possible, since it is invoked during every process switch.
- ***Dispatch latency*** means the time taken by the dispatcher to
  - stop one process and
  - start another running.

SCHEDULING CRITERIA:

In choosing which algorithm to use in a particular situation, depends upon the properties of the various algorithms. Many criteria have been suggested for comparing CPU- scheduling algorithms. The criteria include the following:

1. **CPU utilization:** We want to keep the CPU as busy as possible. Conceptually, 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).
2. **Throughput:** If the CPU is busy executing processes, then work is being done. 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 ten processes per second.
3. **Turnaround time.** This is the important criterion which tells how long it takes to execute that process. 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/0.
4. **Waiting time**: The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/0, it affects only the amount of time thata process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.
5. **Response time:** In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first responseis produced. This measure, called 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.

SCHEDULING ALGORITHMS

- CPU scheduling deals with the problem of deciding which of the processes inthe ready-queue is to be allocated the CPU.

- Following are some scheduling algorithms:
    1. FCFS scheduling (First Come First Served)
    2. Round Robin scheduling
    3. SJF scheduling (Shortest Job First)
    4. SRT scheduling
    5. Priority scheduling
    6. Multilevel Queue scheduling
    7. Multilevel Feedback Queue scheduling

FCFS Scheduling

- The process that requests the CPU first is allocated the CPU first.
- The implementation is easily done using a FIFO queue.
- Procedure:
    1. When a process enters the ready-queue, its PCB is linked onto the tail ofthe queue.
    2. When the CPU is free, the CPU is allocated to the process at the queue's head.
    3. The running process is then removed from the queue.
- Advantage:
    1. Code is simple to write & understand.
- Disadvantages:
    1. **Convoy effect:** All other processes wait for one big process to get off the CPU.
    2. Non-preemptive (a process keeps the CPU until it releases it).
    3. Not good for time-sharing systems.
    4. The average waiting time is generally not minimal.

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Example: Suppose that the processes arrive in the order P1, P2,P3.
- The Gantt Chart for the schedule is as follows:

- Waiting time for *P1* = 0; *P2* = 24; *P3* =27

  Average waiting time: (0 + 24 + 27)/3 = 17ms
- Suppose that the processes arrive in the order P2, P3,P1.
- The Gantt chart for the schedule is as follows:

- Waiting time for P1 = 6;P2 = 0; P3 =3
- Average waiting time: (6 + 0 + 3)/3 = 3ms

SJF Scheduling

- The CPU is assigned to the process that has the smallest next CPU burst.
- If two processes have the same length CPU burst, FCFS scheduling is used to breakthe tie.
- For long-term scheduling in a batch system, we can use the process time limitspecified by the user, as the 'length'
- SJF can't be implemented at the level of short-term scheduling, because there isno way to know the length of the next CPU burst
- Advantage:
    1. The SJF is optimal, i.e. it gives the minimum average waiting time for a given set of processes.
- Disadvantage:
    1. Determining the length of the next CPU burst.
- SJF algorithm may be either 1) non-preemptive or 2) preemptive.

1. Non preemptive SJF

The current process is allowed to finish its CPU burst.

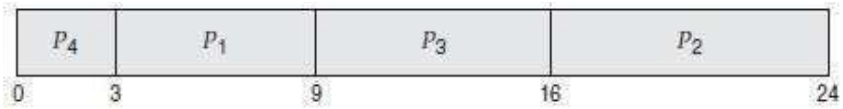
2. Preemptive SJF

If the new process has a shorter next CPU burst than what is left of the executing process, that process is preempted. It is also known as **SRTF** scheduling (**Shortest-Remaining-Time-First**).

- Example (for non-preemptive SJF): Consider the following set of processes,with the length of the CPU-burst time given in milliseconds.

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- For non-preemptive SJF, the Gantt Chart is as follows:



- Waiting time for P1 = 3; P2 = 16; P3 = 9; 4=0
- Average waiting time: (3 + 16 + 9 +0)/4= 7

preemptive SJF/SRTF: Consider the following set of processes, with the length

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

of the CPU- burst time given in milliseconds.

- For preemptive SJF, the Gantt Chart is as follows:

- The average waiting time is ((10 - 1) + (1 - 1) + (17 - 2) + (5 - 3))/4 = 26/4 =6.5.

Priority Scheduling

- A priority is associated with each process.
- The CPU is allocated to the process with the highest priority.
- Equal-priority processes are scheduled in FCFS order.
- Priorities can be defined either internally or externally.
    1. ***Internally-defined*** priorities.
        - Use some measurable quantity to compute the priority of a process.
        - For example: time limits, memory requirements, no. f open files.
    2. ***Externally-defined*** priorities.
        - Set by criteria that are external to the OS Forexample:
        - importance of the process, political factors
- Priority scheduling can be either preemptive or non-preemptive.1.***Preemptive***

  The CPU is preempted if the priority of the newly arrived process ishigher than the priority of the currently running process.

  2. ***Non Preemptive***

  The new process is put at the head of the ready-queue

- Advantage: Higher priority processes can be executed first.
- Disadvantage:
    - 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 priority scheduling algorithm

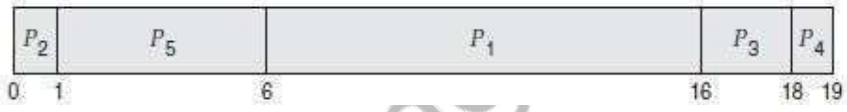
can leave some low- priority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.

- Solution: ***Aging*** is a technique of increasing priority of processes that wait in system for a long time.

• Example: 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 time 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

• The Gantt chart for the schedule is as follows:



• The average waiting time is 8.2milliseconds.

Round Robin Scheduling

- Designed especially for time sharing systems.
- It is similar to FCFS scheduling, but with preemption.
- A small unit of time is called a ***time quantum****(or time slice).*
- Time quantum is ranges from 10 to 100ms.
- The ready-queue is treated as a **circular queue**.
- The CPUscheduler
  - goes around the ready-queue and
  - allocates the CPU to each process for a time interval of up to 1 time quantum.
- To implement:

  The ready-queue is kept as a FIFO queue of processes
- CPUscheduler
  1. Picks the first process from the ready-queue.
  2. Sets a timer to interrupt after 1 time quantum and
  3. Dispatches the process.
- One of two things will then happen.
  1. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily.

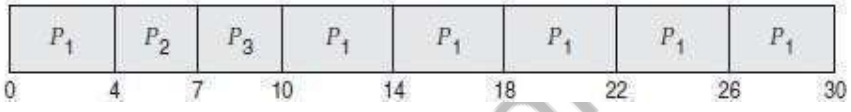
2. If the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the OS. The process will be put at the tail of the ready-queue.

- Advantage:
  - Higher average turnaround than SJF.
- Disadvantage:
  - Better response time than SJF.

- Example: Consider the following set of processes that arrive at time 0, with thelength of the CPU-burst time given in milliseconds.

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart for the schedule is as follows:



- The average waiting time is 17/3 = 5.66milliseconds.

- *The RR scheduling algorithm is preemptive.*

  No process is allocated the CPU for more than *1* time quantum in a row.If a process' CPU burst exceeds *1* time quantum, that process is preempted and is put back in the ready- queue.

- The performance of algorithm depends heavily on the size of the time quantum.
  1. If time quantum=very large, RR policy is the same as the FCFS policy.
  2. If time quantum=very small, RR approach appears to the users as though each of n processes has its own processor running at l/n the speed of the real processor.
- In software, we need to consider the effect of context switching on the performance of RR scheduling
  1. Larger the time quantum for a specific process time, less time is spend on context switching.
  2. The smaller the time quantum, more overhead is added for the purpose of context- switching.

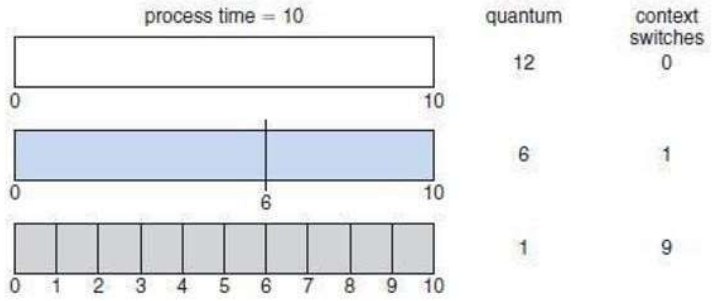


Fig: How a smaller time quantum increases context switches

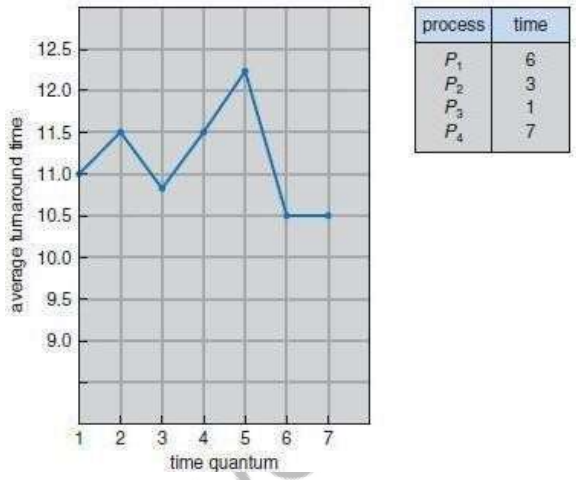


Fig: How turnaround time varies with the time quantum

Multilevel Queue Scheduling

- Useful for situations in which processes are easily classified into different groups.
- For example, a common division is made between
  - foreground (or interactive) processes and
  - background (or batch) processes.
- The ready-queue is partitioned into several separate queues (Figure2.19).
- The processes are permanently assigned to one queue based on some property like
  - memory size
  - process priority or
  - process type.
- Each queue has its own scheduling algorithm.

  For example, separate queues might be used for foreground and background processes.

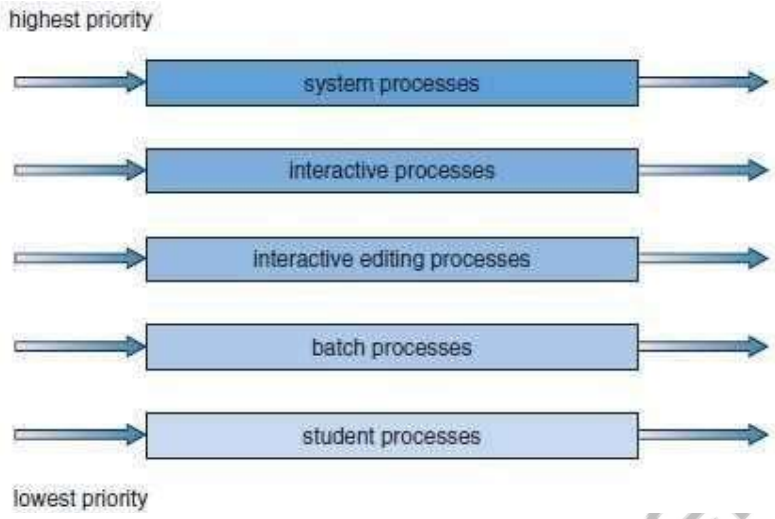


Fig Multilevel queue scheduling

- 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.
- **Time slice**: each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR 20% to backgroundin FCFS

Multilevel Feedback Queue Scheduling

- A process may move between queues
- The basic idea: Separate processes according to the features of their CPU bursts. For example
  1. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues.
  2. If a process waits too long in a lower-priority queue, it may be moved to a higher-priority queue This form of aging prevents starvation.

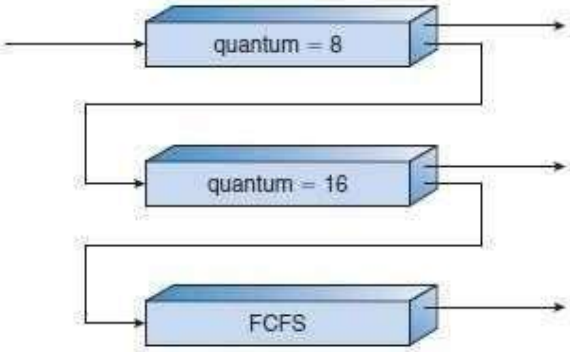


Figure Multilevel feedback queues

In general, a multilevel feedback queue scheduler is defined by the following parameters:

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

MULTIPLE PROCESSOR SCHEDULING

- If multiple CPUs are available, the scheduling problem becomes more complex.
- Two approaches:

Asymmetric Multiprocessing

The basic idea is:

- A master server is a single processor responsible for all scheduling decisions, I/Oprocessing and other system activities.
- The other processors execute only user code.
- Advantage: This is simple because only one processor accesses the system data structures, reducing the need for data sharing.

Symmetric Multiprocessing

The basic idea is:

- Each processor is self-scheduling.
- To do scheduling, the scheduler for each processor
- Examines the ready-queue and
- Selects a process to execute.

Restriction: We must ensure that two processors do not choose the same process and that processes are not lost from the queue.

Some of the issues related to SMP are:

a. Processor Affinity

- In SMP systems,
  1. Migration of processes from one processor to another are avoided and
  2. Instead processes are kept running on the same processor. This is known asprocessor affinity.
- Two forms:
  1. *Soft Affinity*
     - When an OS try to keep a process on one processor because ofpolicy, but cannot guarantee it will happen.
     - It is possible for a process to migrate between processors.
  2. *Hard Affinity*

- When an OS have the ability to allow a process to specify that it is not tomigrate to other processors. Eg. : Solaris OS

b. Load Balancing

- This attempts to keep the workload evenly distributed across all processors in anSMP system.
- Two approaches:

1. Push Migration

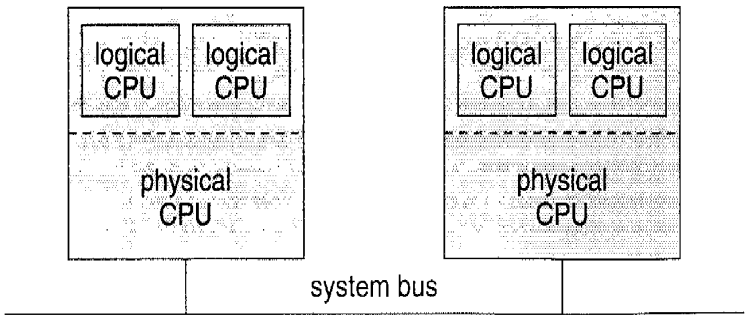
A specific task periodically checks the load on each processor and if it finds an imbalance, it evenly distributes the load to idle processors.

2. Pull Migration

An idle processor pulls a waiting task from a busy processor.

c. Symmetric Multithreading

- The basic idea:
  1. Create multiple logical processors on the same physical processor.
  2. Present a view of several logical processors to the OS.
- Each logical processor has its own architecture state, which includes general-purpose and machine-state registers.
- The following figure illustrates a typical SMT architecture with two physical processors, each housing two logical processors. From the operating system's perspective, four processors are available for work on this system.



- Each logical processor is responsible for its own interrupt handling.
- SMT is a feature provided in hardware, not software.

THREAD SCHEDULING

- On OSs, it is kernel-level threads but not processes that are being scheduled by the OS.
- User-level threads are managed by a thread library, and the kernel is unaware of them.
- To run on a CPU, user-level threads must be mapped to an associated kernel-level thread.

Contention Scope

- Two approaches:
  1. ***Process-Contention scope***
     - On systems implementing the many-to-one and many-to-many models, thethread library schedules user-level threads to run on an available LWP.
     - Competition for the CPU takes place among threads belonging to the same process.
  2. ***System-Contention scope***
     - The process of deciding which kernel thread to schedule on the CPU.
     - Competition for the CPU takes place among all threads in the system.
     - Systems using the one-to-one model schedule threads using only SCS.

Pthread Scheduling

- Pthread API that allows specifying either PCS or SCS during thread creation.
- Pthreads identifies the following contention scope values:
  1. PTHREAD_SCOPE_ PROCESS schedules threads using PCS scheduling.
  2. PTHREAD-SCOPE_SYSTEM schedules threads using SCS scheduling.
- Pthread IPC provides following two functions for getting and setting the contentionscope policy:
  1. pthread_attr_setscope(pthread_attr_t *attr, int scope)
  2. pthread_attr_getscope(pthread_attr_t *attr, int*scope)

MODULE: 3

CONTENTS:

- **Deadlocks:**
  - Deadlocks;
  - System model;
  - Deadlock characterization;
  - Methods for handling deadlocks;
  - Deadlock prevention;
  - Deadlock avoidance;
  - Deadlock detection and recovery from deadlock.
- **Memory Management:**
  - Memory management strategies: Background;
  - Swapping;
  - Contiguous memory allocation;
  - Paging;
  - Structure of page table;
  - Segmentation.

MODULE 3
PROCESS SYNCHRONIZATION

- A cooperating process is one that can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages.
- Concurrent access to shared data may result in data inconsistency. To maintain data consistency, various mechanisms is required to ensure the orderly execution of cooperating processes that share a logical address space.

Producer- Consumer Problem

- A Producer process produces information that is consumed by consumer process.
- To allow producer and consumer process to run concurrently, A Bounded Buffer can be used where the items are filled in a buffer by the producer and emptied by the consumer.
- The original solution allowed at **BUFFER_SIZE - 1** item in the buffer at the same time. To overcome this deficiency, an integer variable ***counter***, initialized to 0 is added.
- ***counter*** is incremented every time when a new item is added to the buffer and is decremented every time when one item removed from the buffer.

The code for the ***producer process*** can be modified as follows:

```
while (true) {

     /* produce an item and put in
        nextProduced*/ while (counter ==
        BUFFER_SIZE)
                    ; // do
                    nothing
               buffer [in] = nextProduced;
               in = (in + 1) % BUFFER_SIZE;
               counter++;
}
```

The code for the ***consumer process*** can be modified as follows:

```
while (true){
           while (counter ==0)
                ; // do nothing
              nextConsumed =buffer[out];
               out = (out + 1) % BUFFER_SIZE;
              counter--;
                /* consume the item in nextConsumed */
     }
```

- **Race Condition**
  When the producer and consumer routines shown above are correct separately, theymay not function correctly when executed concurrently.
- *Illustration:*
  Suppose that the value of the variable ***counter*** is currently 5 and that the producer and consumer processes execute the statements "counter++" and "counter--" concurrently. The value of the variable counter may be 4, 5, or 6 but the only correct result is counter == 5, which is generated correctly if the producer and consumer execute separately.

```
The value of counter may be incorrect as shown below:
The statement counter++ could be
            implemented as register1=
            counter
            register1 =
            register1 + 1
            counter =register1

The statement counter-- could be
            implemented as register2
            =counter
            register2 = register2 - 1
            count = register2
```

The concurrent execution of "counter++" and "counter--" is equivalent to a sequentialexecution in which the lower-level statements presented previously are interleaved in some arbitrary order. One such interleaving is

Consider this execution interleaving with "count = 5" initially:

S0: producer execute register1=counter	{register1 = 5}
S1: producer execute register1 = register1+1	{register1 = 6}
S2: consumer execute register2=counter	{register2 = 5}
S3: consumer execute register2 = register2-1	{register2 = 4}
S4: producer execute counter=register1	{count =6}
S5: consumer execute counter=register2	{count =4}

- *Note:* It is arrived at the incorrect state "counter == 4", indicating that four buffers are full, when, in fact, five buffers are full. If we reversed the order of the statementsat T4 and T5, we would arrive at the incorrect state "counter==6".
- *Definition* ***Race Condition***: A situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **Race Condition.**
- To guard against the race condition, ensure that only one process at a time can be manipulating the variable counter. To make such a guarantee, ***the processes are synchronized*** in some way.

The Critical Section Problems

- Consider a system consisting of n processes {Po, P1 , ... ,Pn-1}.
- 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 soon
- 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.
- The critical-section problem is to design a protocol that the processes can use to cooperate.

The general structure of a typical process Pi is shown in the figure below:

- 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 **reminder section**.

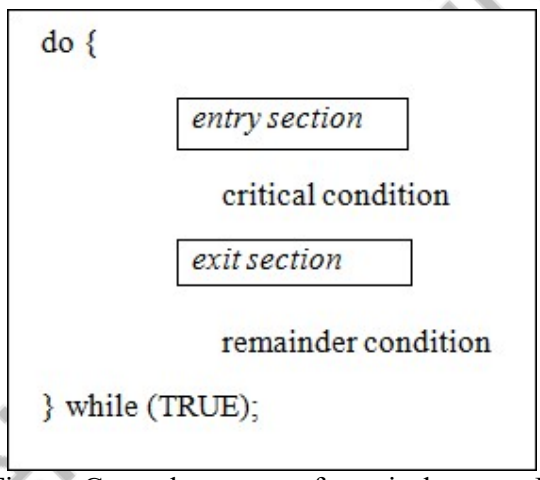


Figure: General structure of a typical process Pi

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

1. **Mutual exclusion:** If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.

2. **Progress:** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

3. **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

- This is a classic software-based solution to the critical-section problem. There are no guarantees that Peterson's solution will work correctly on modern computer architectures
- Peterson's solution provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting.

Peterson's solution is restricted to two processes that alternate execution between theircritical sections and remainder sections. The processes are numbered P_o and P_1 or Pi and P_j where j = 1-i

Peterson's solution requires the two processes to share two data items:

```
Int         turn;
boolean flag[2];
```

- **turn:** The variable turn indicates whose turn it is to enter its critical section. **Ex:** if turn == i, then process P_i is allowed to execute in its critical section
- **flag:** The flag array is used to indicate if a process is ready to enter its critical section. **Ex:** if flag [i] is true, this value indicates that P_i is ready to enter its critical section.

P_i	P_j
do { flag[i] = true; turn = j; while (flag[j] && turn == j); /* critical section */ flag[i] = false; /* remainder section */ } while (true);	do { flag[j] = true; turn = i; while (flag[i] && turn == i); /* critical section */ flag[j] = false; /* remainder section */ } while (true);

Figure: The structure of process P_i and P_j in Peterson's solution

- To enter the critical section, process *Pi 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, it can do so.
- 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 eventual *value of turn determines* which of the two processes is allowed toenter its critical section first

To prove that solution is correct, then we need to show that,

1. Mutual exclusion is preserved.
2. Progress requirement is satisfied.
3. Bounded-waiting requirement is met.

1. **To prove Mutual exclusion**

- Each pi enters its critical section only if either flag [j] == false or turn ==i.
- If both processes are executed in their critical sections at the same time, the flag [0] == flag [1]==true.
- These two observations imply that Pi and Pj 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. Hence, one of the processes (Pj) must have successfully executed the while statement, whereas Pi had to execute at least one additional statement ("turn==j").
- However, at that time, flag [j] == true and turn == j, and this condition will persist as long as Pi is in its critical section, as a result, mutual exclusion is preserved.

2. **To prove Progress and Bounded-waiting**

- A process Pi can be prevented from entering the critical section only if it is stuck inthe while loop with the condition flag [j] ==true and turn=== j; this loop is the onlyone possible.
- If Pj is not ready to enter the critical section, then flag [j] ==false, and Pi can enter itscritical section.
- If Pj has set flag [j] = true and is also executing in its while statement, then eitherturn === 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 flag [j] = false, allowing Pi toenter its critical section.
- If Pj resets flag [j] to true, it must also set turn to i.
- Thus, since Pi does not change the value of the variable turn while executing the while statement, Pi will enter the critical section (progress) after at most one entry byPj (bounded waiting).

SYNCHRONIZATION HARDWARE

- The solution to the critical-section problem requires a simple tool-a **lock.**
- **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 and it releases the lock when it exits the critical section.

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

Figure: Solution to the critical-section problem using locks.

- The critical-section problem could be solved simply in a uniprocessor environment if interrupts are prevented from occurring while a shared variable was being modified. In this manner, the current sequence of instructions would be allowed to be executed in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable.
- But this solution is not as feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming, as the message is passed to all the processors. This message passing delays entry into each critical section,and system efficiency decreases.

TestAndSet ( ) and Swap( ) instructions

- Many modern computer systems provide special hardware instructions that allow to **test** and **modify** the content of a word or to **swap** the contents of two words **atomically**, that is, as one uninterruptible unit.
- Special instructions such as TestAndSet () and Swap() instructions are used to solve the critical-section problem
- The TestAndSet () instruction can be defined as shown in Figure. The important characteristic of this instruction is that it is executed atomically.

Definition:

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

Figure: The definition of the TestAndSet () instruction.

- Thus, if two TestAndSet () instructions are executed simultaneously, they will be executed sequentially in some arbitrary order. If the machine supports the TestAndSet () instruction, then implementation of mutual exclusion can be done by declaring a Boolean variable **lock,** initialized to **false**.

P_i	P_j
do { while (TestAndSet(&lock)) ; // do nothing //critical section lock =FALSE; //remainder section } while (TRUE);	do { while (TestAndSet(&lock)); // do nothing //critical section lock =FALSE; //remainder section } while (TRUE);

Figure: Mutual-exclusion implementation with TestAndSet ()

Definition:

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

Figure: The definition of the Swap ( ) instruction

- The Swap() instruction, operates on the contents of two words, it is defined as shown above Swap() it is executed atomically. If the machine supports the Swap() instruction, then mutual exclusion can be provided as follows.

P_i	P_j
do { key = TRUE; while (key = = TRUE) Swap(&lock, &key}; // critical section lock = FALSE; // remainder section }while (TRUE);	do { key = TRUE; while (key = = TRUE) Swap(&lock, &key}; // critical section lock = FALSE; // remainder section }while (TRUE);

- A global Boolean variable lock is declared and is initialized to false. In addition, eachprocess has a local Boolean variable key.
- These algorithms satisfy the mutual-exclusion requirement, they do not satisfy thebounded-waiting requirement.
- Below algorithm using the TestAndSet () instruction that satisfies all the critical-section requirements. The common data structures are

boolean waiting[n];and ***boolean lock;*** These data structures are initialized to false.

```
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);
```

Figure: Bounded-waiting mutual exclusion with Testament ()

1. **To prove the mutual exclusion requirement**

- 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.

2. **To prove the progress requirement**

- Note that, the arguments presented for mutual exclusion also apply here, since a process exiting the critical section either sets lock to false or sets waiting[j] to false. Both allow a process that is waiting to enter its critical section to proceed.

3. **To prove the bounded-waiting requirement**

- Note that, when a process leaves its critical section, it scans the array waiting in thecyclic ordering (i + 1, i + 2, ... , n 1, 0, ... , i 1).
- It designates the first process in this ordering that is in the entry section (waiting[j] ==true) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within n - 1 turns.

SEMAPHORE

- Software solution to process synchronization.
- Proposed by Edsger Dijkstra
- A semaphore is a synchronization tool is used solve various synchronization problem and can be implemented efficiently.
- Semaphore do not require busy waiting.
  - A semaphore **S** is an integer variable that is accessed only through two standard atomic operations: wait () and signal ().
  - The wait () operation was originallytermed **P** (Dutch word *PROBEREN* meaning "to test")
  - signal() was called **V** (Dutch word *VERHOGEN* meaning "to increment") .

Definition of wait (): to test if any other process is in critical section/ using shared resources.

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

Definition of signal (): process has completed its critical section or shared resources has to be released

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

- All 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.

Binary semaphore

- The value of a binary semaphore can range only between 0 and1.
- Binary semaphores are known as mutex locks, as they are locks that provide mutual exclusion. Binary semaphores to deal with the critical-section problem for multiple processes. Then processes share a semaphore, mutex, initialized to1

Each process Pi is organized as shown in below figure

Figure: Mutual-exclusion implementation with semaphores

```
do {
      wait (mutex);
              // Critical
        Section signal
        (mutex);
            // remainder section
} while (TRUE);
```

Counting semaphore

- The value of a counting semaphore can range over an unrestricted domain.
- 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. When a process releases a resource, it performs a signal()operation.
- 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.

Implementation

- The main disadvantage of the semaphore definition requires busy waiting.
- While a process is in its critical section, any other process that tries to enter itscritical 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.

Semaphore implementation with no busy waiting

- The definition of the wait() and signal() semaphore operations is modified.
- 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.

- To implement semaphores under this definition, we define a semaphore as a "C' struct:

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

- Each semaphore has an integer value and a list of processes list. When a process mustwait on a semaphore, it is added to the list of processes. A signal() operation removesone 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. These two operations are provided by the operating system as basic system calls.

- In this implementation semaphore values may be negative. If a semaphore valueis negative, its magnitude is the number of processes waiting on that semaphore.

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. The event in question is the execution of a signal( ) operation. When such a state is reached, these processes are said to be deadlocked.
- To illustrate this, consider a system consisting of two processes, Po and P1, each accessing two semaphores, S and Q, set to the value 1

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

```
.                          .
.                          .
signal(S);                 signal(Q);
signal(Q);                 signal(S);
```

Suppose that Po executes wait (S) and then P1 executes wait (Q). When Poexecutes wait (Q), it must wait until P1 executes signal (Q). Similarly, when P1 executes wait (S), it must wait until Po executes signal(S). Since these signal () operations cam1ot be executed, Po and P1 are deadlocked.

- Another problem related to deadlocks is indefinite blocking or starvation: Asituation in which processes wait indefinitely within the semaphore.
- Indefinite blocking may occur if we remove processes from the list associated witha semaphore in LIFO (last-in, first-out) order.

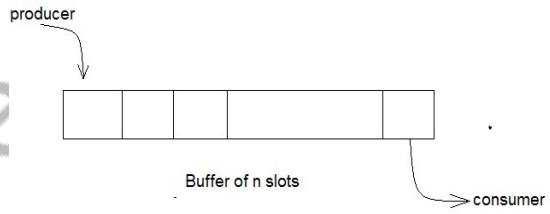
CLASSICAL PROBLEMS OF SYNCHRONIZATION

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

What is the Problem Statement?

There is a buffer of n slots and each slot is capable of storing one unit of data. There are two processes running, namely, producer and consumer, which are operating on the buffer.



A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the buffer.

Challenges:

Producer must not insert data when buffer is full. Consumer must not remove data when buffer is empty. Producer and consumer should not insert and remove data simultaneously.

Solution to bounded buffer problem is : 3 semaphores are used

- Binary Semaphore **mutex** initialized to the value 1
- Counting Semaphore **full** initialized to the value0
- Counting Semaphore **empty** initialized to the value N.

The structure of the producer process:	The structure of the consumer process:
do { wait(empty); //wait until empty>0 wait(mutex); // acquire lock **// add data to buffer** signal(mutex); // release lock signal(full); // increment *full* }while(TRUE)	do { wait(full); //wait until full>0 wait(mutex); //acquire lock **// remove data from buffer** signal(mutex); //release lock signal(empty); //increment *empty* }while(TRUE)

• a producer first **waits** until there is atleast one empty slot. • Then it *decrements* the **empty** semaphore because, there will now be one less empty slot, since the producer is going to insert data in one of those slots. • Then, it **acquires lock** on the buffer, so that the consumer cannot access the buffer until producer completes its operation. • After performing the insert operation, the **lock is released** and the value of **full** is *incremented* because the producer has just filled a slot in the buffer.	• The consumer **waits** until there is atleast one full slot in the buffer. • Then it *decrements* the **full** semaphore because the number of occupied slots will be decreased by one, after the consumer completes its operation. • After that, the consumer **acquires lock** on the buffer and completes the removal operation so that the data from one of the full slots is removed. • Then, the consumer **releases the lock.** • Finally, the **empty** semaphore is *incremented* by 1, indicating consumer has just removed data from an occupied slot.

Readers-Writers Problem

The Problem Statement

There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context. They are ***reader and writer***..

- Readers – Any number of readers can read from the shared resource simultaneously. They can only read the data set; they do **not** perform any updates.
- Writers – can both read and write.
- **Challenges**
  - Allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.
  - When a writer is writing data to the resource, no other process can access the resource.
  - A writer cannot write to the resource if there are non zero number of readers accessing the resource at that time

Solution using semaphore:

- Binary Semaphore **mutex** initialized to 1.
- Binary Semaphore **w** initialized to1.
- Integer variable **readcount** is used to maintain the number of readers currently accessing the resource and is initialized to 0.

Writer

```
while(TRUE)
{
  wait(w);

  /* perform the write operation */

  signal(w);
}
```

Reader

```
while(TRUE)
{
    //acquire lock
    wait(m);
    readcount++;
    if(readcount == 1)
        wait(w);
    //release lock
    signal(m);
    /* perform the reading operation */
    // acquire lock
    wait(m);
```

```
        readcount--;
        if(readcount == 0)
            signal(w);
           // release lock
        signal(m);
}
```

- As seen above in the code for the writer, the writer just **waits on the w semaphore** until it gets a chance to write to the resource.
- After performing the write operation, it **increments w** so that the next writer can access the resource.
- The code for the reader, the **lock is acquired** whenever the readcount is updated by a process.
- When a reader wants to access the resource, first it ***increments the readcount*** value, then accesses the resource and then ***decrements the readcount*** value.
- The **semaphore w** is used by the first reader which enters the critical section and the last reader which exits the critical section. The reason for this is, when the first readers enters the critical section, the writer is blocked from the resource. Only new readers can access the resource now.
- Similarly, when the last reader exits the critical section, it signals the writer using the w semaphore because there are zero readers now and a writer can have the chance to access the resource.

Dining-Philosophers Problem

The dining philosophers' problem is another classic synchronization problem which is used to evaluate situations where there is a need of allocating multiple resources to multiple processes.

What is the Problem Statement?

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 chop sticks.

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. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again.

It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

Solution: 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 stick[5];

where all the elements of chopstick (*a binary semaphore*) are initialized to 1. The structure of philosopher ***i*** is shown

```
while(TRUE)
{
     wait(stick[i]);
     wait(stick[(i+1) % 5]);

/*to find the adjacent stick mod is used because if i=5, next chopstick is 1 (dining table is circular)*/

     Critical section /* eat */

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

     Remainder Section /* think */
}
```

Several possible remedies to the deadlock problem are replaced by:

- Allow at most four philosophers to sit simultaneously at the table.
- Allow a philosopher to pick up her sticks only if both chop sticks are available.
- Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even numbered philosopher picks up her right chopstick and then her left chopstick.

Problems with Semaphores

Correct use of semaphore operations:

- signal (mutex) …. wait (mutex) : Replace signal with wait and vice-versa
- wait (mutex) … wait(mutex)
- Omitting of wait (mutex) or signal (mutex) (or both)

Monitor

- Highlevel abstraction provides a convenient and effective mechanism for process synchronization.
- An **abstract data type**—or **ADT**—encapsulates data with a set of functions to operate on that data that are independent of any specific implementation of the ADT.
- A ***monitor type*** is an ADT that includes a set of programmer defined operations that are provided with mutual exclusion within the monitor. The monitor type alsodeclares the variables whose values define the state of an instance of that type, alongwith the bodies of functions that operate on those variables.
- The monitor construct ensures that only one process at a time is active within the monitor.
- Procedure within a monitor can access only those variables declared locally within the monitor.
- Local variables of a monitor can be accessed by only the local procedures
- Monitor ensures that only one process at a time can be

```
monitor monitor name
{
   // shared variable declarations

   procedure P1 ( . . . ) {
      . . .
   }

   procedure P2 ( . . . ) {
      . . .
   }

         .
         .
         .

   procedure Pn ( . . . ) {
      . . .
   }

   initialization code ( . . . ) {
      . . .
   }
}
```

Syntax of the monitor

active within the monitor.

- To have a powerful Synchronization schemes a *condition* construct is added to the Monitor. So synchronization scheme can be defined with one or more variables of type *condition* Two operations on a condition variable:

Condition x, y

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

  The operation

  x.wait () – a process that invokes the operation is suspended.

  x.signal () – resumes one of processes (if any) that invoked x.wait ()

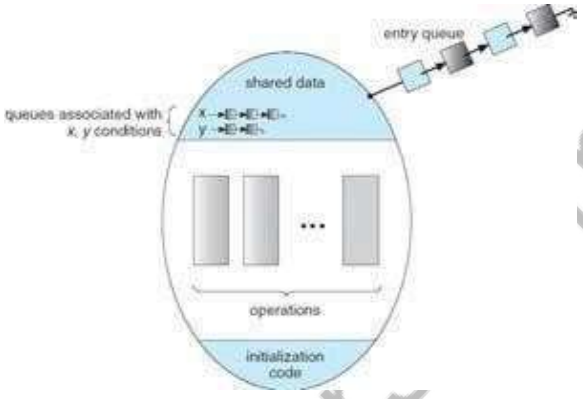


Fig: Monitor with Condition Variables

Solution to Dining Philosophers

Each philosopher *i* invokes the operations **pickup()** and **putdown()** in the following

- For each monitor, a semaphore mutex (initialized to 1) is provided. A process must execute wait(mutex) before entering the monitor and must execute signal(mutex) after leaving the monitor.

```
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[i] = THINKING;
        // test left and right neighbors
        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;
}
}
```

- Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore, next, is introduced, initialized to 0. The signaling processes can use next to suspend themselves. An integer variable next_count is also provided to count the number of processes suspended on next.Thus, each external function F is replaced by

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

- For each condition x, we introduce a semaphore x sem and an integer variable xcount, 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 to determine which of the suspended processes should be resumed next, one simple solution is to use a first-come, first-served (FCFS)ordering, so that the process that has been waiting the longest is resumed first. For thispurpose, the **conditional-wait** construct can be used. This construct has theform

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 theprocess that is suspended. When x.signal() is executed, the process with the smallest priority number is resumed next.

```
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;
   }
}
```

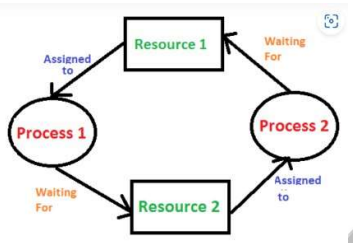
- The Resource Allocator monitor shown in the above Figure, which controls the allocation of a single resource among competing processes.
- A process that needs to access the resource in question must observe the following sequence:

```
R.acquire(t);
...
access the resource;
...
R.release();
```

where R is an instance of type Resource Allocator.

- The monitor concept cannot guarantee that the preceding access sequence will be observed. In particular, the following problems can occur:
- A process might access a resource without first gaining access permission to the resource.
- A process might never release a resource once it has been granted access to the resource.
- A process might attempt to release a resource that it never requested.
- A process might request the same resource twice (without first releasing theresource).

DEADLOCKS



A process requests resources, 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 Deadlock**.**

SYSTEM MODEL

- 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. Memory space, CPU cycles, files, and I/0 devices are examples of resource types.
- A process must request a resource before using it and must release the resource after using it. A process may request as many resources as it requires carrying out its designated task. The number of resources requested may not exceed the total number of resources available in the system.

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

1. **Request:** The process requests the resource. If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.
2. **Use:** The process can operate on the resource.
3. **Release:** The process releases the resource.

A set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are resource acquisition and release. The resources may be either physical resources or logical resources.

To illustrate a deadlocked state, consider a system with three CD RW drives.

Suppose each of three processes holds one of these CD RW drives. If each process now requests another drive, the three processes will be in a deadlocked state.

Each is waiting for the event "CD RW is released," which can be caused only by one of the other waiting processes. This example illustrates a deadlock involving the same resource type. Deadlocks may also involve different resource types. For example, consider a system with one printer and one DVD drive. Suppose that process Pi is holding the DVD and process Pj is holding the printer. If Pi requests the printer and Pj requests the DVD drive, a deadlock occurs.

DEADLOCK CHARACTERIZATION

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 {P0, Pl, ... , Pn} of waiting processes must exist such that Po is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ... , Pn-1 is waiting for a resource held by Pn and Pn is waiting for a resource held by Po.

Resource-Allocation Graph

Deadlocks can be described in terms of a directed graph called **System Resource-Allocation Graph**

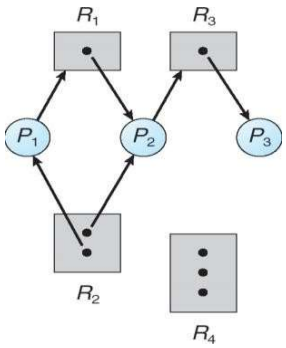
The graph consists of a set of vertices V and a set of edges E. The set of vertices V ispartitioned into two different types of nodes:

- $P = \{P_1, P_2, ..., P_n\}$, the set consisting of all the active processes in the system.
- $R = \{R_1, R_2, ..., R_m\}$ the set consisting of all resource types in the system.

A directed edge from process P_i to resource type R_j is denoted by $P_i \rightarrow R_j$ it signifies that process P_i has requested an instance of resource type R_j and is currently waiting for that resource.
A directed edge from resource type R_j to process P_i is denoted by $R_j \rightarrow P_i$ it signifies that an instance of resource type R_j has been allocated to process P_i.

- A directed edge $P_i \rightarrow R_j$ is called a **Request Edge**.
- A directed edge $R_j \rightarrow P_i$ is called an **Assignment Edge**.

Pictorially each process P_i as a circle and each resource type R_j as a rectangle. Since resource type R_j may have more than one instance, each instance is represented as a ***dot*** within the rectangle.
A request edge points to only the rectangle R_j, whereas an *assignment edge must also designate one of the dots in the rectangle.*
When process P_i requests an instance of resource type R_j, 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.
The resource-allocation graph shown in Figure depicts the following situation.



The sets P, K and E:

- $P = \{P_1, P_2, P_3\}$
- $R = \{R_1, R_2, R_3, R_4\}$
- $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$

Resource instances:

- One instance of resource type R_1
- Two instances of resource type R_2
- One instance of resource type R_3
- Three instances of resource type R_4

Process states:

- Process P_1 is holding an instance of resource type R_2 and is waiting for an instance of resource type R_1.
- Process P_2 is holding an instance of R_1 and an instance of R_2 and is waiting for an instance of R_3.
- Process P_3 is holding an instance of R_3.

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

- If each resource type has exactly **one instance**, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked.
- 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.

To illustrate this concept, the resource-allocation graph depicted in below figure:

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. At this point, two minimal cycles exist in the system:

1. P1 →R1 → P2 → R3 → P3 → R2→P1
2. P2 →R3 → P3 → R2 → P2

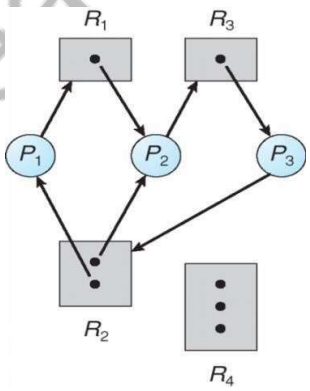


Figure: Resource-allocation graph with a deadlock.

Processes P1, P2, and P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by process P3. Process P3 is waiting for either process P1 or process P2 to release resourceR2. In addition, process P1 is waiting for process P2 to release resource R1.

Consider the resource-allocation graph as depicted in below Figure. In this example also have a cycle:

P1→R1→P3→R2→P1

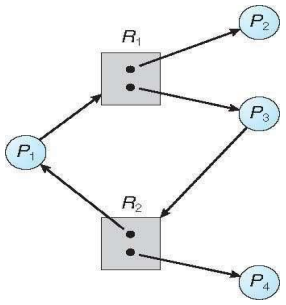


Figure: Resource-allocation graph with a cycle but no deadlock

However, there is no deadlock. Observe that process P4 may release its instance of resource type R2. That resource can then be allocated to P3, breaking the cycle.

METHODS FOR HANDLING DEADLOCKS

The deadlock problem can be handled in one of three ways:

1. Use a protocol to **prevent or avoid deadlocks**, ensuring that the system will never enter a deadlocked state.
2. Allow the system to enter a deadlocked state, **detect it**, and recover.
3. Ignore the problem altogether and **pretend that deadlocks never occur** in the system.

To ensure that deadlocks never occur, the system can use either deadlock prevention or a deadlock-avoidance scheme.

Deadlock prevention provides a set of methods for ensuring that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made.

Deadlock-avoidance requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. Withthis additional knowledge, it can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may arise. In this environment, the system can provide an **algorithm** that examines the state of the system to determine whether a deadlock has occurred and an algorithm to recover from the deadlock.

In the absence of algorithms to detect and recover from deadlocks, the system is in a deadlock state yet has no way of recognizing what has happened. In this case, the undetected deadlock will result in deterioration of the system's performance, because resources are being held by processes that cannot run and because more and more processes, as they make requests for resources, will enter a deadlocked state. Eventually, the system will stop functioning and will need to be restarted manually.

DEADLOCK PREVENTION

Deadlock can be prevented by ensuring that at least one of the four necessary conditions cannot hold.

Mutual Exclusion

- The mutual-exclusion condition must be held for non-sharable resources. Sharable resources do not require mutually exclusive access and thus cannot be involved in a deadlock.
- Ex: Read-only files are 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.
- Deadlocks cannot prevent by denying the mutual-exclusion condition because some resources are intrinsically non-sharable.

Hold and Wait

To ensure that the hold-and-wait condition never occurs in the system, then guarantee that, whenever a process requests a resource, it does not hold any other resources.

- One protocol that can be used requires each process to request and be allocated all its resources before it begins execution.
- Another protocol allows a process to request resources only when it has none. A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.

Ex:

- Consider a process that copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer. If all resources must be requested at the beginning of the process, then the process must initially request the DVD drive, disk file, and printer. It will hold the printer for its entire execution, even though it needs the printer only at the end.
- The second method allows the process to request initially only the DVD drive and disk file. It copies from the DVD drive to the disk and then releases both the DVD drive and the disk file. The process must then again request the disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminates.

The two main disadvantages of these protocols:

- Resource utilization may be low since resources may be allocated but unused for a long period.
- Starvation is possible.

No Preemption

- The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated.
- To ensure that this condition does not hold, the following protocols can be used:
- If a process is holding some resources and requests another resource that cannot be immediately allocated to it, then all resources the process is currently holding are preempted.
- The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

If a process requests some resources, first check whether they are available. If they are, allocatethem.

If they are not available, check whether they are allocated to some other process that is waiting for additional resources. If so, preempt the desired resources from the waiting process and allocate them to the requesting process.
If the resources are neither available nor held by a waiting process, the requesting process must wait. While it is waiting, some of its resources may be preempted, but only if another process requests them. A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting.

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.

To illustrate, let R = {R1, R2, ... , Rm} be the set of resource types. Assign a unique integer number to each resource type, which allows to compare two resources and to determinewhether one precedes another in ordering. Formally, it is defined as a one-to-one function.
F: R ->N, where N is the set of natural numbers.

Example: if the set of resource types R includes tape drives, disk drives, and printers, then the function F might be defined as follows:

F (tape drive) = 1
F (disk drive) = 5
F (printer) = 12

Now consider the following protocol to prevent deadlocks. Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type -R_i. After that, the process can request instances of resource typeR_j if and only if $F(R_j) > F(R_i)$.

DEADLOCK AVOIDANCE

- To avoid deadlocks additional information is required about how resources are to be requested. With the knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock.
- Each request requires that in making this decision the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.
- The various algorithms that use this approach differ in the amount and type of information required. The simplest model requires that each process declare the *maximum number* of resources of each type that it may need. Given this a prior information, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state. Such an algorithm defines the ***deadlock-avoidance approach.***

Safe State

- **Safe state:** A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. A system is in a safe state only if there exists a safe sequence.

- **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 still make can be satisfied by the currently available resources plus the resources held by all Pj, 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, Pi can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When Pi terminates, Pi+1 can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be unsafe.

A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks as shown in figure. An unsafe state may lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe states.

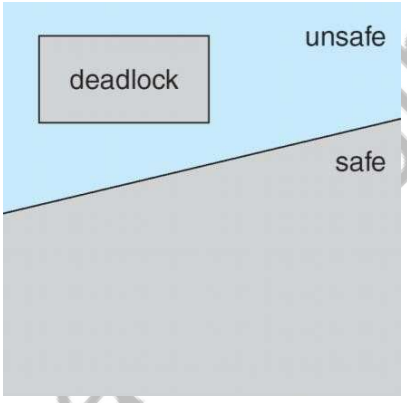


Figure: Safe, unsafe, and deadlocked state spaces.

Resource-Allocation-Graph Algorithm

- If a resource-allocation system has only one instance of each resource type, then a variant of the resource-allocation graph is used for deadlock avoidance.
- In addition to the request and assignment edges, a new type of edge is introduced, called a **claim edge.**
- 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. When a resource *Rj* is released by *Pi* the assignment edge *Rj->Pi* is reconverted to a claim edge *Pi->Rj.*

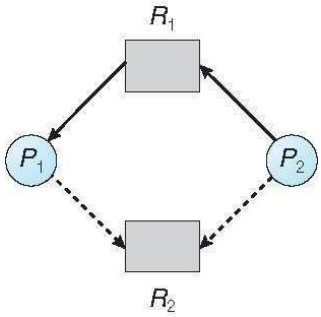


Figure: Resource-allocation graph for deadlock avoidance.

Note that the resources must be claimed a prior in the system. That is, before process *Pi* starts executing, all its claim edges must already appear in the resource-allocation graph.
We can relax this condition by allowing a claim edge *Pi ->Rj* to be added to the graph only if all the edges associated with process *Pi* are claim edges.

Now 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.

There is a need to check for safety by using a cycle-detection algorithm. An algorithm for detecting a cycle in this graph requires an order of n^2 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. In that case,process Pi will have to wait for its requests to be satisfied.

To illustrate this algorithm, consider the resource-allocation graph as shown above. Suppose that P2 requests R2. Although R2 is currently free, we 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.

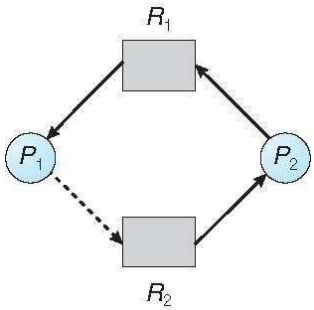


Figure: An unsafe state in a resource-allocation graph

Banker's Algorithm

The Banker's algorithm is applicable to a resource allocation system with multiple instances of each resource type.

- 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.

To implement the banker's algorithm the following data structures are used.

Let n = number of processes, and m = number of resources types

Available: A vector of length *m* indicates the number of available resources of each type. If available [j] = 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] = k, thenprocess 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 toeach process. If Allocation[i,j] = k then Pi is currently allocated k instances of Rj

Need: An n x m matrix indicates the remaining resource need of each process. If Need[i,j] = k,then Pi may need k more instances of Rj to complete its task.

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

Safety Algorithm

The algorithm for finding out whether or not a system is in a safe state. This algorithm can be described as follows:

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:
   (a) Finish[i] = false
   (b) $Need_i \leq$ 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

This algorithm may require an order of m x n^2 operations to determine whether a state is safe.

Resource-Request Algorithm

The algorithm for determining whether requests can be safely granted.
Let $Request_i$ be the request vector for process P_i. If $Request_i[j] == k$, then process P_i wants k instances of resource type R_j. When a request for resources is made by process Pi, the followingactions are taken:

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceededits maximum claim

2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available

3. Have the system pretend to allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

If safe ⇒ the resources are allocated to Pi
If unsafe ⇒ Pi must wait, and the old resource-allocation state is restored

Example

Consider a system with five processes P_o through P_4 and three resource types *A, B,* and C. Resource type *A* has ten instances, resource type *B* has five instances, and resource type C has seven instances. Suppose that, at time T_0 the following snapshot of the system has been taken:

	Allocation	Max	Available
	A B C	A B C	A B C
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*

	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

The system is currently in a safe state. Indeed, the sequence <P_1, P_3, P_4, P_2, P_0> satisfies the safety criteria.

Suppose now that process P_1 requests one additional instance of resource type A and two instances of resource type C, so $Request_1 = (1,0,2)$. Decide whether this request can be immediately granted.

Check that Request ≤ Available

$(1,0,2) \leq (3,3,2) \Rightarrow$ true

Then pretend that this request has been fulfilled, and the following new state has arrived.

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

Executing safety algorithm shows that sequence <P_1, P_3, P_4, P_0, P_2> satisfies safety requirement.

DEADLOCK DETECTION

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

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

Single Instance of Each Resource Type

- If all resources have only a single instance, then define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a ***wait-for*** graph**.**
- This graph is obtained from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.
- 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 $P_i \rightarrow R_q$ and $R_q \rightarrow P_i$ for some resource R_q.

Example: In below Figure, a resource-allocation graph and the corresponding wait-for graph is presented.

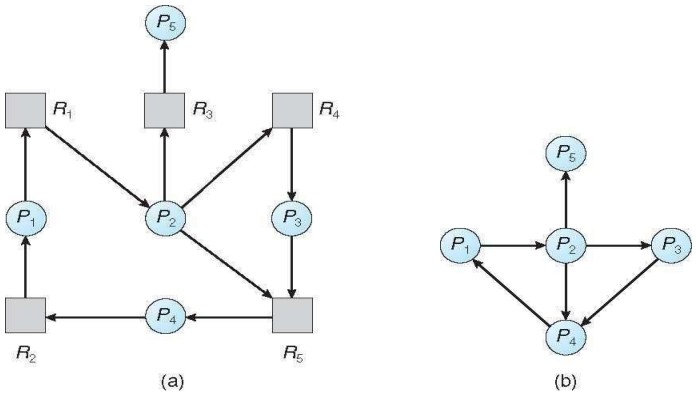


Figure: (a) Resource-allocation graph. (b) Corresponding wait-for graph.

- A deadlock exists in the system if and only if the wait-for graph contains a cycle. Todetect 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 isthe number of vertices in the graph.

Several Instances of a Resource Type

A deadlock detection algorithm that is applicable to several instances of a 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 eachtype.
- **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;* is requesting *k* more instances of resource type *Rj.*

Algorithm:

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:
   (a) Work = Available
   (b) For $i = 1,2, \ldots, n$, if $Allocation_i \neq 0$, then *Finish*[i] = false; otherwise, *Finish*[i] = *true*

2. Find an index *i*such that both:
   (a) Finish[*i*] == *false*
   (b) $Request_i \leq Work$

   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*] == false, for some *i*, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked

Algorithm requires an order of O(m x n^2) operations to detect whether the system is in deadlocked state

Example of Detection Algorithm

Consider a system with five processes *Po* 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 Chas six instances. Suppose that, at time T_0, the following resource-allocation state:

	Allocation	Request	Available
	A B C	A B C	A B C
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	

After executing the algorithm, Sequence <P_0, P_2, P_3, P_1, P_4> will result in *Finish*[*i*] = true forall *i*

Suppose now that process P2 makes one additional request for an instance of type C. TheRequest matrix is modified as follows:

	Request
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

The system is now deadlocked. Although we can reclaim the resources held by process Po, thenumber of available resources is not sufficient to fulfill the requests of the other processes.
Thus, a deadlock exists, consisting of processes P1, P2, P3, and P4.

Detection-Algorithm Usage

The detection algorithm can be invoked on two factors:

1. How *often* is a deadlock likely to occur?
2. How *many* processes will be affected by deadlock when it happens?

If deadlocks occur frequently, then the detection algorithm should be invoked frequently.Resources allocated to deadlocked processes will be idle until the deadlock can be broken.

If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes "caused" the deadlock.

RECOVERY FROM DEADLOCK

The system recovers 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 topreempt some resources from one or more of the deadlocked processes.

Process Termination

To eliminate deadlocks by aborting a process, use one of two methods. In both methods, thesystem reclaims all resources allocated to the terminated processes.

1. **Abort all deadlocked processes:** This method clearly will break the deadlock cycle, butat 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.
2. **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-detectionalgorithm must be invoked to determine whether any processes are still deadlocked.

If the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated. Many factors may affect which process is chosen, including:

1. What the priority of the process is
2. How long the process has computed and how much longer the process will computebefore completing its designated task
3. How many and what types of resources the process has used.
4. How many more resources the process needs in order to complete
5. How many processes will need to be terminated?
6. Whether the process is interactive or batch

Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resourcesfrom processes and give these resources to other processes until the deadlock cycle is broken.

If preemption is required to deal with deadlocks, then three issues need to be addressed:

1. **Selecting a victim.** Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed during its execution.
2. **Rollback.** If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some neededresource. We must roll back the process to some safe state and restart it from that state. Since it is difficult to determine what a safe state is, the simplest solution is a total rollback: abort the process and then restart it.
3. **Starvation.** How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

MODULE: 4

CONTENTS:

- **Virtual Memory Management**:
  - Background
  - Demand paging
  - Copy-on-write.
  - Page replacement
  - Allocation of frames
  - Thrashing
- **File System**
  - File concept
  - Access methods
  - Directory structure
  - File system mounting
  - File sharing
  - Protection
- **Implementation of File System**
  - File system structure
  - File system implementation
  - Directory implementation
  - Allocation methods
  - Free space management

MODULE 4

MEMORY MANAGEMENT

Main Memory Management Strategies

- Memory management is concerned with managing the primary memory.
- Memory consists of array of bytes or words each with its own address.
- Every program to be executed must be in memory. The instruction must be fetched from memory before it is executed.
- In multi-tasking OS memory management is complex, because as processes are swapped in and out of the CPU, their code and data must be swapped in and out of memory.

Basic Hardware

- Main memory, cache and CPU registers in the processors are the only storage spaces that CPU can access directly.
- The program and data must be bought into the memory from the disk, for the process to run. Each process has a separate memory space and must access only this range of legal addresses. Protection of memory is required to ensure correct operation. This prevention is provided by hardware implementation.
- Two registers are used - a base register and a limit register. The base register holds the smallest legal physical memory address; the limit register specifies the size of the range.
- For example, if the base register holds 300040 and limit register is 120900, then the program can legally access all addresses from 300040 through 420940 (inclusive).

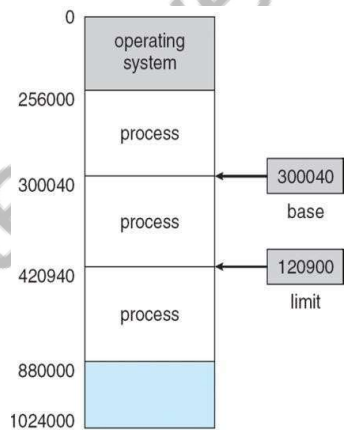


Figure: A base and a limit-register define a logical-address space

- The base and limit registers can be loaded only by the operating system, which uses special privileged instruction. Since privileged instructions can be executed only in kernel mode only the operating system can load the base and limit registers.

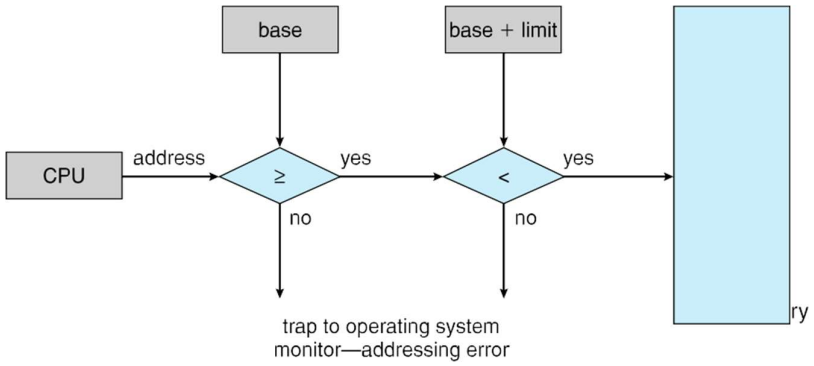


Figure: Hardware address protection with base and limit-registers

- ✓ **Protection of memory space** is accomplished by having the CPU hardware compare every address generated in user mode with the registers.
- ✓ Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a **trap to the operating system**, which treats the attempt as a **fatal error** as shown in below **figure.**
- ✓ This prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.

Address Binding

- User programs typically refer to memory addresses with symbolic names. These symbolic names must be mapped or bound to physical memory addresses.
- Programs are stored on the secondary storage disks as binary executable files.
- When the programs are to be executed, they are brought into the main memory and placed within a process.
- The collection of processes on the disk waiting to enter the main memory forms the **input queue.**
- One of the processes which are to be executed is fetched from the queue and is loaded into main memory.
- During the execution it fetches instruction and data from main memory. After the process terminates it returns the memory space.
- During execution the process will go through several steps as shown in **the figure below.** and in each step the address is represented in different ways.
- In source program the address is symbolic. The compiler **binds** the symbolic address to re-locatable address. The loader will in turn bind this re-locatable address to the absolute address.
- Address binding of instructions to memory-addresses can happen at 3 different stages.
  1. **Compile Time** - If it is known at compile time where a program will reside in physical memory, then absolute code can be generated by the compiler, containing actual physical addresses. However, if the load address changes at some later time, then the program will have to be recompiled.
  2. **Load Time** - If the location at which a program will be loaded is not known at compile time, then the compiler must generate relocatable code, which references addressesrelative to the start of the program. If that starting address changes, then the program must be reloaded but not recompiled.
  3. **Execution Time** - If a program can be moved around in memory during the course of itsexecution, then binding must be delayed until execution time.

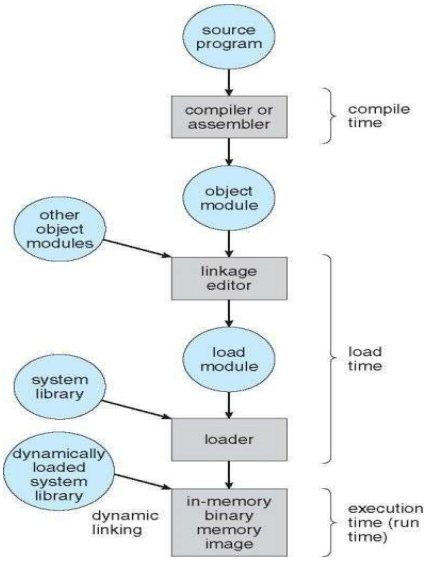


Figure: Multistep processing of a user program

Logical Versus Physical Address Space

- The address generated by the CPU is a logical address, whereas the memory addresswhere programs are actually stored is a physical address.
- The set of all logical addresses used by a program composes the logical address space,and the set of all corresponding physical addresses composes the physical address space.
- The run time mapping of logical to physical addresses is handled by the memory-management unit (MMU).
    - One of the simplest is a modification of the base-register scheme.
    - The base register is termed a relocation register
    - The value in the relocation-register is added to every address generated by auser-process at the time it is sent to memory.
    - The user-program deals with logical-addresses; it never sees the real physical-addresses.

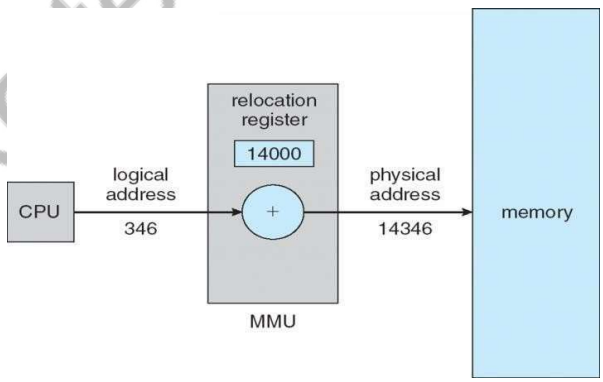


Figure: Dynamic relocation using a relocation-register

- **For example**, if the base is at **14000**, then an attempt by the user to address **location 0** is dynamically relocated to location 14000; an access to location **346** is mapped to location **14346**. The user program never sees the real physical addresses.
- The size of the process is thus limited to the size of the physical memory.

Dynamic Loading

- This can be used to obtain better memory-space utilization.
- A routine is not loaded until it is called.

This works as follows:

1. Initially, all routines are kept on disk in a relocatable-load format.
2. Firstly, the main-program is loaded into memory and is executed.
3. When a main-program calls the routine, the main-program first checks to see whether theroutine has been loaded.
4. If routine has been not yet loaded, the loader is called to load desired routine into memory.
5. Finally, control is passed to the newly loaded-routine.

Advantages:

1. An unused routine is never loaded.
2. Useful when large amounts of code are needed to handle infrequently occurring cases.
3. Although the total program-size may be large, the portion that is used (and hence loaded)may be much smaller.
4. Does not require special support from the OS.

Dynamic Linking and Shared Libraries

- With **static linking** library modules get fully included in executable modules, wasting both disk space and main memory usage, because every program that included a certain routine from the library would have to have their own copy of that routine linked into their executable code.
- With **dynamic linking**, however, only a stub is linked into the executable module, containing references to the actual library module linked in at run time.
    - The stub is a small piece of code used to locate the appropriate memory-residentlibrary-routine.
    - This method saves disk space, because the library routines do not need to be fully included in the executable modules, only the stubs.
    - An added benefit of dynamically linked libraries (DLLs, also known as shared libraries or shared objects on UNIX systems) involves easy upgrades and updates.

Shared libraries

- A library may be replaced by a new version, and all programs that reference the librarywill automatically use the new one.
- Version info. is included in both program & library so that programs won't accidentallyexecute incompatible versions.

Swapping

- A process must be loaded into memory in order to execute.
- If there is not enough memory available to keep all running processes in memory at the same time, then some processes that are not currently using the CPU may have their memory swapped out to a fast local disk called the ***backing store***.
- *Swapping is the process of moving a process from memory to backing store and moving another process from backing store to memory*. Swapping is a very slow process compared to other operations.
- A variant of 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 ***called roll out, roll in.***

- Normally the process which is swapped out will be swapped back to the same memory space that is occupied previously and this depends upon address binding.
- The system maintains a **ready queue** consisting of all the processes whose memory images are on the backing store or in memory and are ready to run.

Swapping depends upon address-binding:

- If binding is done at load-time, then process cannot be easily moved to a different location.
- If binding is done at execution-time, then a process can be swapped into a different memory-space, because the physical-addresses are computed during execution-time.

Major part of swap-time is transfer-time; i.e. total transfer-time is directly proportional to theamount of memory swapped.

Disadvantages:

1. Context-switch time is fairly high.
2. If we want to swap a process, we must be sure that it is completely idle.Two solutions:
   i) Never swap a process with pending I/O.
   ii) Execute I/O operations only into OS buffers.

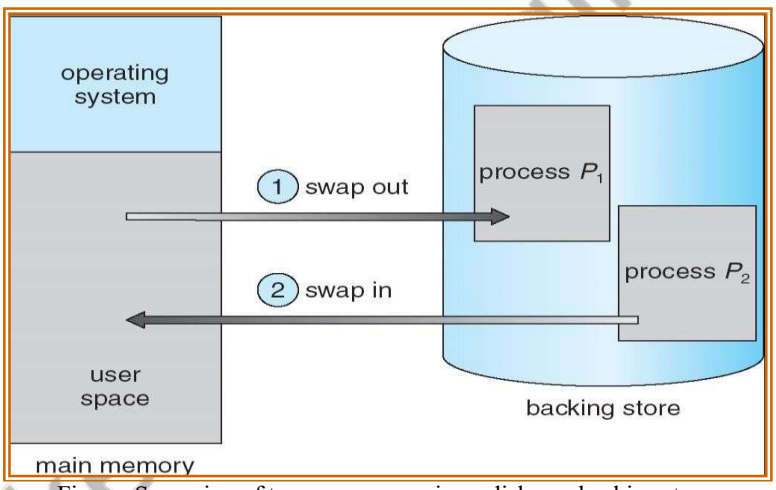


Figure: Swapping of two processes using a disk as a backing store

Example:

Assume that the user process is 10 MB in size and the backing store is a standard hard disk witha transfer rate of 40 MB per second.

The actual transfer of the 10-MB process to or from main memory takes10000 KB/40000 KB per second = 1/4 second

= 250 milliseconds.

Assuming that no head seeks are necessary, and assuming an average latency of 8 milliseconds, the swap time is 258 milliseconds. Since we must both swap out and swap in, the total swap time is about 516 milliseconds.

Contiguous Memory Allocation

- The main memory must accommodate both the operating system and the various user processes. Therefore we need to allocate the parts of the main memory in the most efficient way possible.
- Memory is usually divided into 2 partitions: One for the resident OS. One for the user processes.

- Each process is contained in a single contiguous section of memory.

1. Memory Mapping and Protection

- Memory-protection means protecting OS from user-process and protecting user-processes from one another.
- Memory-protection is done using
  - Relocation-register: contains the value of the smallest physical-address.
  - Limit-register: contains the range of logical-addresses.
- Each logical-address must be less than the limit-register.
- The MMU maps the logical-address dynamically by adding the value in the relocation-register. This mapped-address is sent to memory
- When the CPU scheduler selects a process for execution, the dispatcher loads therelocation and limit-registers with the correct values.
- Because every address generated by the CPU is checked against these registers, we canprotect the OS from the running-process.
- The relocation-register scheme provides an effective way to allow the OS size to change dynamically.
- Transient OS code: Code that comes & goes as needed to save memory-space and overhead for unnecessary swapping.

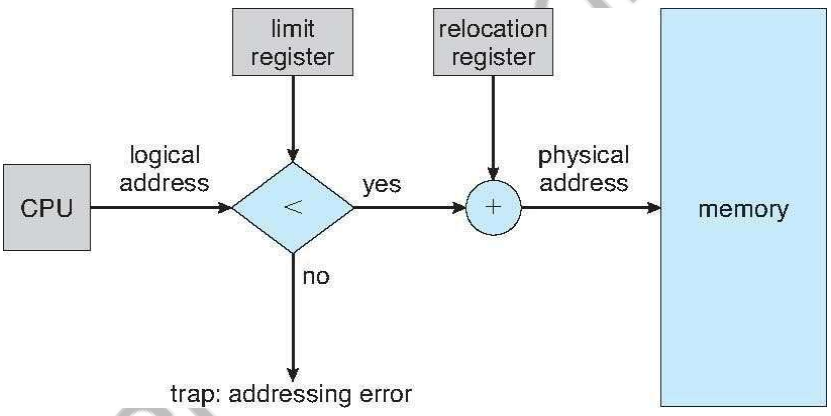


Figure: Hardware support for relocation and limit-registers

2. Memory Allocation

Two types of memory partitioning are:

1. Fixed-sized partitioning
2. Variable-sized partitioning

1. Fixed-sized Partitioning

- The memory is divided into fixed-sized partitions.
- Each partition may contain exactly one process.
- The degree of multiprogramming is bound by the number of partitions.
- When a partition is free, a process is selected from the input queue and loaded into thefree partition.
- When the process terminates, the partition becomes available for another process.

2. Variable-sized Partitioning

- The OS keeps a table indicating which parts of memory are available and which parts are

occupied.

- A hole is a block of available memory. Normally, memory contains a set of holes ofvarious sizes.
- Initially, all memory is available for user-processes and considered one large hole.
- When a process arrives, the process is allocated memory from a large hole.
- If we find the hole, we allocate only as much memory as is needed and keep the remaining memory available to satisfy future requests.

Three strategies used to select a free hole from the set of available holes:

1. **First Fit:** Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended.
2. **Best Fit:** Allocate the smallest hole that is big enough. We must search the entire list,unless the list is ordered by size. This strategy produces the smallest leftover hole.
3. **Worst Fit:** Allocate the largest hole. Again, we must search the entire list, unless it issorted by size. This strategy produces the largest leftover hole.

First-fit and best fit are better than worst fit in terms of decreasing time and storage utilization.

a. What are the drawbacks of contiguous memory allocation? Given five memory partitions of 100KB, 500KB, 200KB, 300KB and 600KB (in order), how would each of the first fit, best fit and worst fit algorithms place processes of 212 KB, 417 KB, 112KB and 426 KB (in order)? Which algorithm makes the most efficient use of memory? **(06 Marks)**

1a. Solution:

First Fit

100K	
	212K
500K	112K
200K	
300K	
600K	417K

426 must wait

Best Fit

100K	
500K	417K
200K	112K
300K	212K
600K	426K

Worst Fit

100K	
500K	417K
200K	
300K	212K
600K	112K

426 must wait

The Best Fit is the efficient algorithm.

3. Fragmentation

Two types of memory fragmentation:

1. Internal fragmentation
2. External fragmentation

1. Internal Fragmentation

- The general approach is to break the physical-memory into fixed-sized blocks andallocate memory in units based on block size.
- The allocated-memory to a process may be slightly larger than the requested-memory.

- The difference between requested-memory and allocated-memory is called internalfragmentation i.e. Unused memory that is internal to a partition.

2. External Fragmentation

- External fragmentation occurs when there is enough total memory-space to satisfy a request but the available-spaces are not contiguous. (i.e. storage is fragmented into alarge number of small holes).
- Both the first-fit and best-fit strategies for memory-allocation suffer from external fragmentation.
- Statistical analysis of first-fit reveals that given N allocated blocks, another 0.5 N blocks will be lost to fragmentation. This property is known as the 50-percent rule.

Two solutions to external fragmentation:

- Compaction: The goal is to shuffle the memory-contents to place all free memorytogether in one large hole. Compaction is possible only if relocation is dynamic and doneat execution-time
- Permit the logical-address space of the processes to be non-contiguous. This allows a process to be allocated physical-memory wherever such memory is available. Twotechniques achieve this solution: 1) Paging and 2) Segmentation.

Paging

- Paging is a memory-management scheme.
- This permits the physical-address space of a process to be non-contiguous.
- This also solves the considerable problem of fitting memory-chunks of varying sizesonto the backing-store.
- Traditionally: Support for paging has been handled by hardware.
- Recent designs: The hardware & OS are closely integrated.

Basic Method of Paging

- 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 framesfrom 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 Figure.

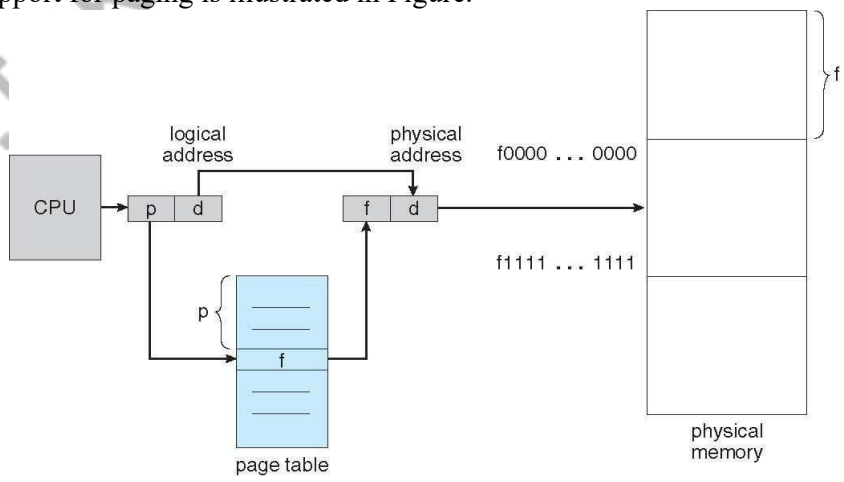


Figure : Paging hardware

- Address generated by CPU is divided into 2 parts (Figure 2):
  1. Page-number (p) is used as an index to the page-table. The page-table contains thebase-address of each page in physical-memory.
  2. Offset (d) is combined with the base-address to define the physical-address. This physical-address is sent to the memory-unit.
- The page table maps the page number to a frame number, to yield a physical address
- The page table maps the page number to a frame number, to yield a physical addresswhich also has two parts: The frame number and the offset within that frame.
- The number of bits in the frame number determines how many frames the system canaddress, and the number of bits in the offset determines the size of each frame.

The paging model of memory is shown in Figure

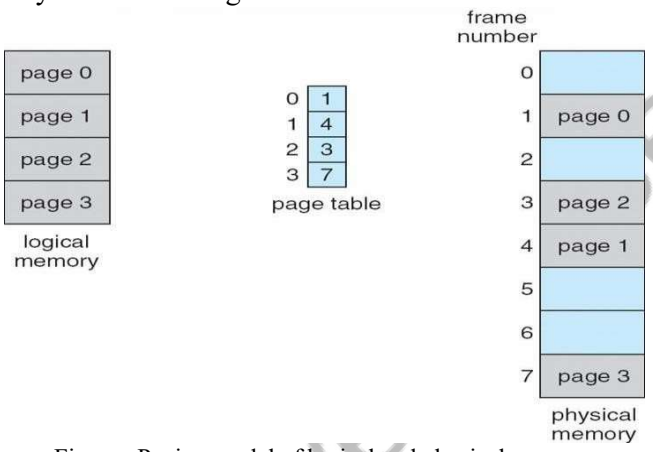


Figure : Paging model of logical and physical memory.

- 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.
- If the size of logical address space is 2^m and a page size is 2^n 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:

page number	page offset
p	d
m -n	n

✓ **Ex:**

✓ Consider a logical address space of 64 pages of 1,024 words each, mapped onto a physical memory of 32 frames.
a. How many bits are there in the logical address?
Solution: 64 pages of 1,024 words each= $2^6 * 2^{10} = 2^{16}$ hence we need 16 bits
b. How many bits are there in the physical address?
Solution: 32 frames of 1,024 words each= $2^5 * 2^{10} = 2^{15}$ hence we need 15 bits

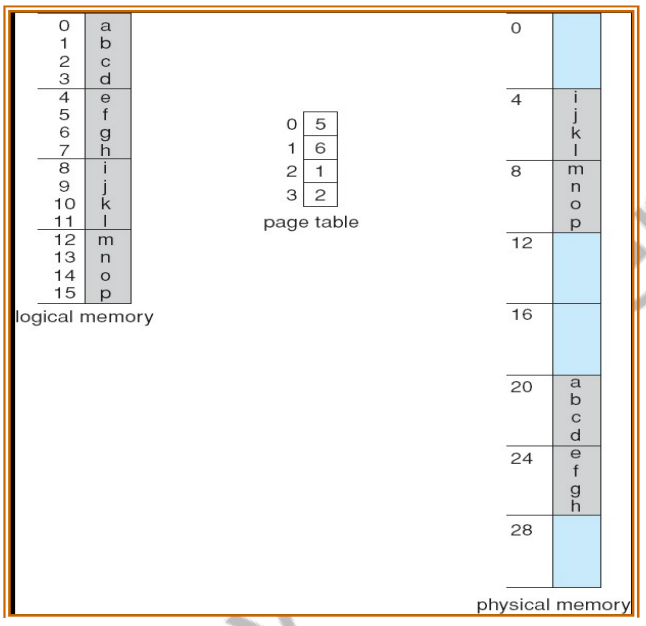
✓ To show how to map logical memory into physical memory, consider a page size of 4 bytes and physical memory of 32 bytes (8 pages) as shown in below **figure**.

a. Logical address 0 is page 0 and offset 0 and Page 0 is in frame 5. The **logical address 0** maps to physical address **[(5*4) + 0]=20.**

b. **Logical address 3** is page 0 and offset 3 and Page 0 is in frame 5. The **logical address 3** maps to **physical address [(5*4) + 3]= 23.**

c. **Logical address 4** is page 1 and offset 0 and page 1 is mapped to frame 6. So logical address 4 maps to **physical address [(6*4) + 0]=24**.

d. **Logical address 13** is page 3 and offset 1 and page 3 is mapped to frame 2. So logical address 13 maps to **physical address [(2*4) + 1]=9.**



- ✓ In paging scheme, there is **no external fragmentation**. Any free frame can be allocated to a process that needs it. But there may exist **internal fragmentation**.
- ✓ If the memory requirements of a process do not happen to coincide with page boundaries, the last frame allocated may not be completely full.
- ✓ **For example**, if page size is 2,048 bytes, a process of 72,766 bytes will need 35 pages plus 1,086 bytes. It will be allocated 36 frames, resulting in **internal fragmentation** of 2,048 - 1,086= **962 bytes**.
- ✓ When a process arrives in the system to be executed, its size expressed in pages is examined. Each page of the process needs one frame. Thus, if the process requires **n** pages, at least **n** frames must be available in memory. If n frames are available, they are allocated to this arriving process.
- ✓ The first page of the process is loaded in to one of the allocated frames, and the frame number is put in the page table for this process. The next page is loaded into another frame and its frame number is put into the page table and so on, as shown in below **figure. (a) before allocation, (b) after allocation**.

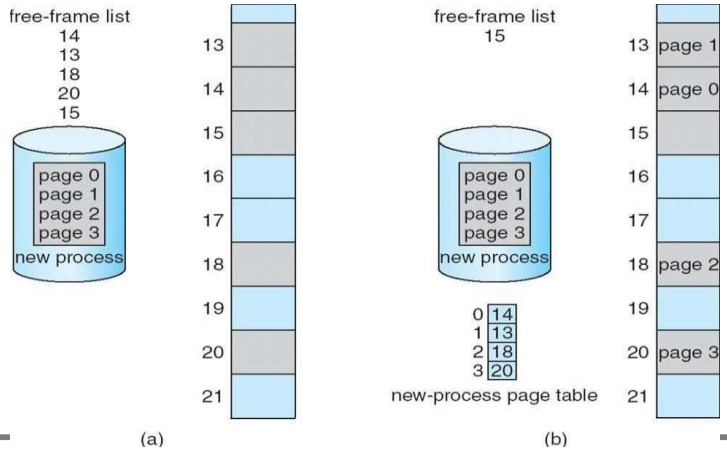


Figure: Free frames (a) before allocation and (b) after allocation.

Hardware Support

Translation Look aside Buffer

- A special, small, fast lookup hardware cache, called a translation look-aside buffer(TLB).
- 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 search is fast; the hardware, however, is expensive. Typically, the number of entries in a TLB is small, often numbering between 64 and 1,024.
- The TLB contains only a few of the page-table entries.

Working:

- When a logical-address is generated by the CPU, its page-number is presented to theTLB.
- If the page-number is found (TLB hit), its frame-number is immediately available and used to access memory
- If page-number is not in TLB (TLB miss), a memory-reference to page table must be made. The obtained frame-number can be used to access memory (Figure 1)

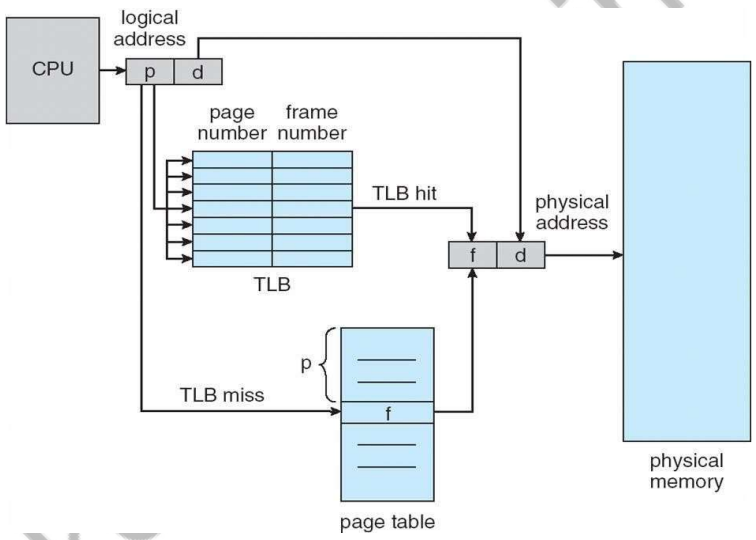


Figure : Paging hardware with TLB

- In addition, we add the page-number and frame-number to the TLB, so that they will befound quickly on the next reference.
- If the TLB is already full of entries, the OS must select one for replacement.
- Percentage of times that a particular page-number is found in the TLB is called hit ratio.

Advantage: Search operation is fast.

Disadvantage: Hardware is expensive.

- Some TLBs have wired down entries that can't be removed.
- Some TLBs store ASID (address-space identifier) in each entry of the TLB that uniquelyidentify each process and provide address space protection for that process.

✓ **For example,** 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**. Thus the effective access time is,

Effective Access Time (EAT) = 0.80 x 120 + 0.20 x 220
= **140 nanoseconds.**

In this example, we suffer a **40-percent slowdown** in memory-access time (from 100 to 140 nanoseconds).

✓ For a **98-percent hit ratio** we have
Effective Access Time (EAT) = 0.98 x 120 + 0.02 x 220
= **122 nanoseconds.**

✓ This increased hit rate produces only a **22 percent slowdown** in access time.

Protection

- Memory-protection is achieved by protection-bits for each frame.
- The protection-bits are kept in the page-table.
- One protection-bit can define a page to be read-write or read-only.
- Every reference to memory goes through the page-table to find the correct frame-number.
- Firstly, the physical-address is computed. At the same time, the protection-bit is checkedto verify that no writes are being made to a read-only page.
- An attempt to write to a read-only page causes a hardware-trap to the OS (or memory protection violation).

Valid Invalid Bit

- This bit is attached to each entry in the page-table.
- Valid bit: "valid" indicates that the associated page is in the process' logical addressspace, and is thus a legal page
- Invalid bit: "invalid" indicates that the page is not in the process' logical address space
- Illegal addresses are trapped by use of valid-invalid bit.
- The OS sets this bit for each page to allow or disallow access to the page.

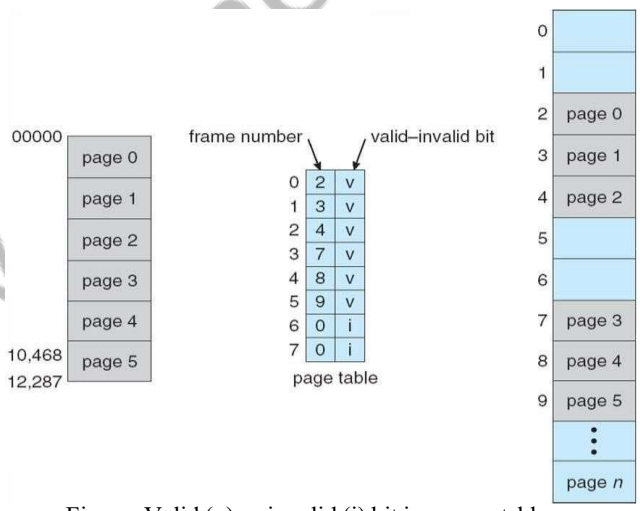


Figure: Valid (v) or invalid (i) bit in a page-table

Shared Pages

- An advantage of paging is the possibility of sharing common code.
- Re-entrant code (Pure Code) is non-self-modifying code, it never changes duringexecution.
- Two or more processes can execute the same code at the same time.
- Each process has its own copy of registers and data-storage to hold the data for the process's execution.
- The data for 2 different processes will be different.
- 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 aremapped onto different frames.

Disadvantage:

Systems that use inverted page-tables have difficulty implementing shared-memory.

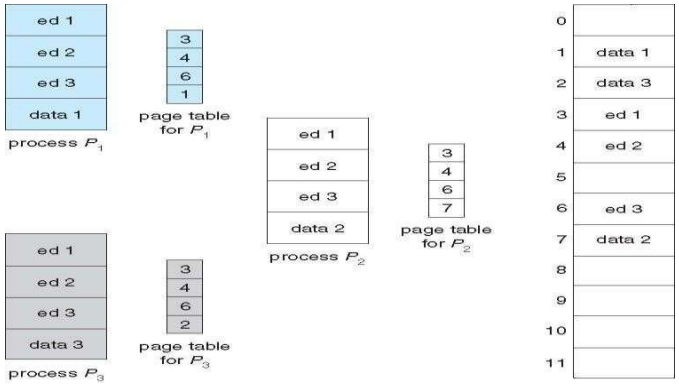


Figure: Sharing of code in a paging environment

Structure of the Page Table

The most common techniques for structuring the page table:

1. Hierarchical Paging
2. Hashed Page-tables
3. Inverted Page-tables

1. Hierarchical Paging

- Problem: Most computers support a large logical-address space (232 to 264). In these systems, the page-table itself becomes excessively large.
- Solution: Divide the page-table into smaller pieces.

Two Level Paging Algorithm:

- The page-table itself is also paged.
- This is also known as a forward-mapped page-table because address translation worksfrom the outer page-table inwards.

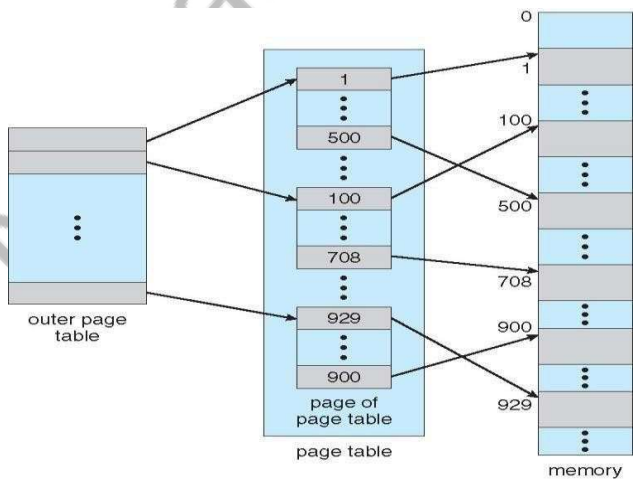


Figure: A two-level page-table scheme

For example:

Consider the system with a 32-bit logical-address space and a page-size of 4 KB.A logical-address is divided into

→ 20-bit page-number and

→ 12-bit page-offset.

Since the page-table is paged, the page-number is further divided into

→ 10-bit page-number and

→ 10-bit page-offset.

Thus, a logical-address is as follows:

page number		page offset
p_1	p_2	d
12	10	10

- where p_1 is an index into the outer page table, and p_2 is the displacement within thepage of the inner page table

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

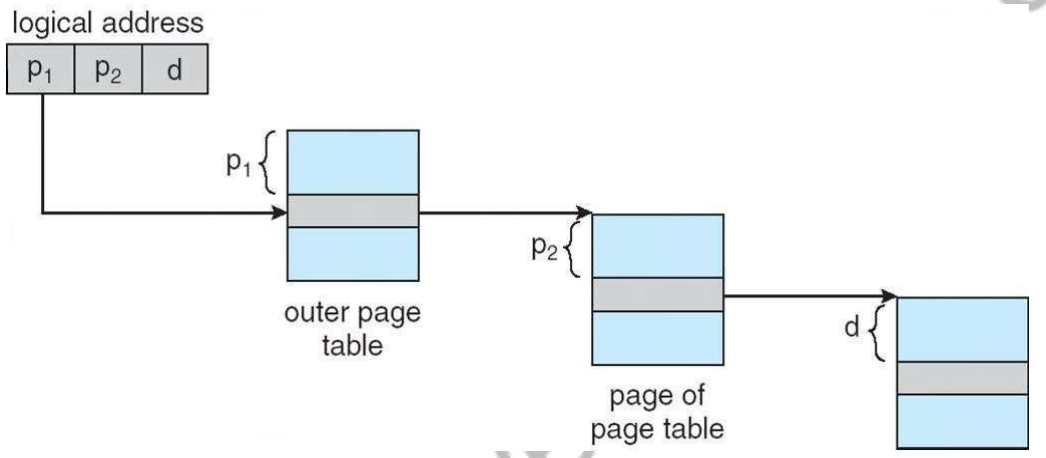


Figure: Address translation for a two-level 32-bit paging architecture

2. Hashed Page Tables

- This approach is used for handling address spaces larger than 32 bits.
- The hash-value is the virtual page-number.
- Each entry in the hash-table contains a linked-list of elements that hash to the samelocation (to handle collisions).
- Each element consists of 3 fields:
  1. Virtual page-number
  2. Value of the mapped page-frame and
  3. Pointer to the next element in the linked-list.

The algorithm works as follows:

- The virtual page-number is hashed into the hash-table.
- The virtual page-number is compared with 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 matchingvirtual page-number.

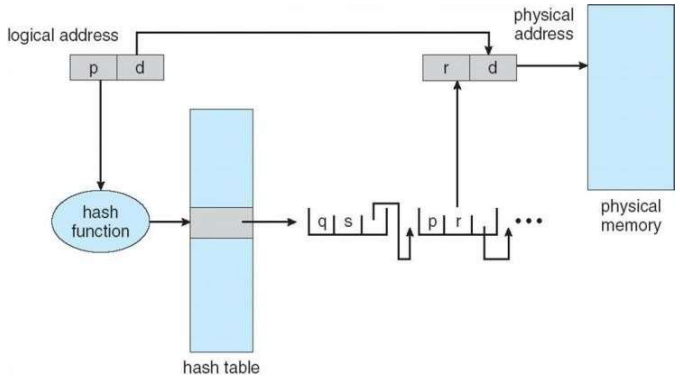


Figure: Hashed page-table

3. Inverted Page Tables

- Has one entry for each real page of memory.
- Each entry consists of virtual-address of the page stored in that real memory-locationand information about the process that owns the page.
- Each virtual-address consists of a triplet <process-id, page-number, offset>.
- Each inverted page-table entry is a pair <process-id, page-number>

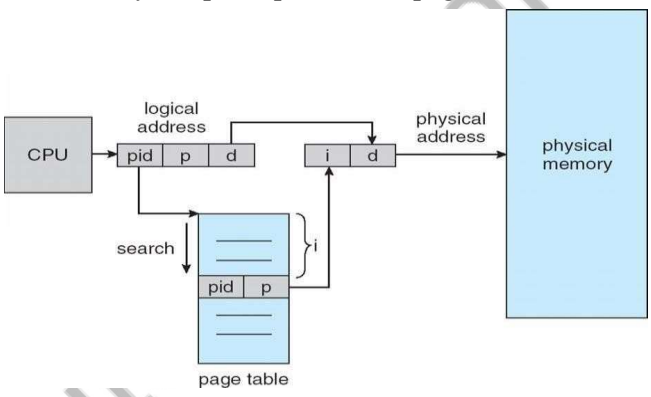


Figure: Inverted page-table

The algorithm works as follows:

1. When a memory-reference occurs, part of the virtual-address, consisting of <process-id,page-number>, is presented to the memory subsystem.
2. The inverted page-table is then searched for a match.
3. If a match is found, at entry i-then the physical-address <i, offset> is generated.
4. If no match is found, then an illegal address access has been attempted.

Advantage:

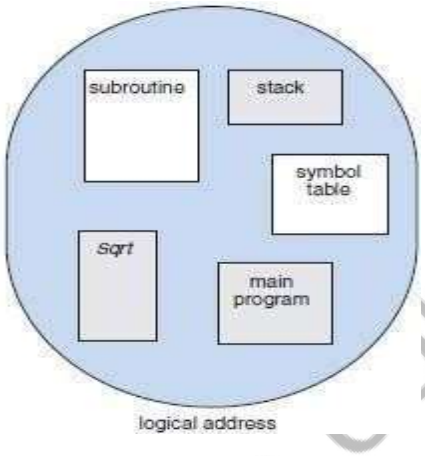
1. Decreases memory needed to store each page-table

Disadvantages:

1. Increases amount of time needed to search table when a page reference occurs.
2. Difficulty implementing shared-memory

Segmentation

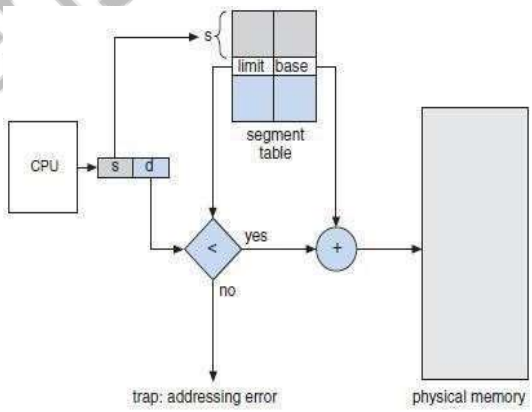
Basic Method of Segmentation



- This is a memory-management scheme that supports user-view of memory (Figure 1).
- A logical-address space is a collection of segments.
- Each segment has a name and a length.
- The addresses specify both segment-name and offset within the segment.
- Normally, the user-program is compiled, and the compiler automatically constructssegments reflecting the input program.
- For ex: The code, Global variables, The heap, from which memory is allocated, Thestacks used by each thread, The standard C library

Hardware support for Segmentation

- Segment-table maps 2 dimensional user-defined addresses into one-dimensional physical addresses.
- In the segment-table, each entry has following 2 fields:
  1. Segment-base contains starting physical-address where the segment resides inmemory.
  2. Segment-limit specifies the length of the segment (Figure 2).
- A logical-address consists of 2 parts:
  1. Segment-number(s) is used as an index to the segment-table
  2. Offset(d) must be between 0 and the segment-limit.
- If offset is not between 0 & segment-limit, then we trap to the OS(logical-addressing attempt beyond end of segment). If offset is legal, then it is added to the segment-base to produce the physical-memory address



✓ **For example**, consider the below **figure.** We have **five segments** numbered from 0 through 4. 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 byte 852 of segment 3, 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.

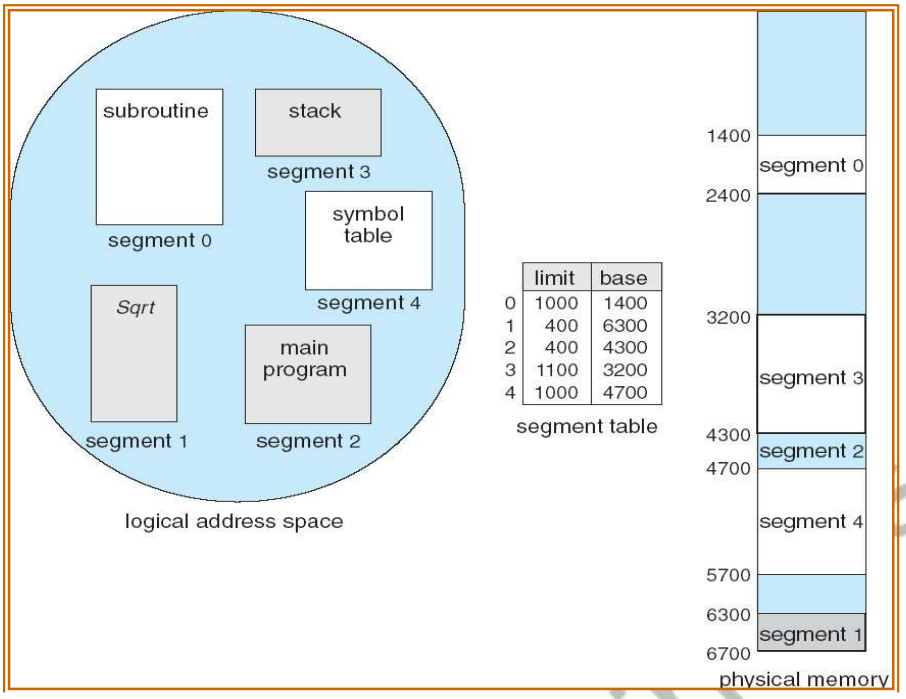


Figure: Segmentation hardware

VIRTUAL MEMORYMANAGEMENT

- Virtual memory is a technique that allows for the execution of partially loaded process.
- Advantages:
  - A program will not be limited by the amount of physical memory that is available user can be able to write into large virtual space.
  - Since each program takes less amount of physical memory, more than one program could be run at the same time which can increase the throughput andCPU utilization.
  - Less i/o operation is needed to swap or load user program into memory. So, each user program could run faster.

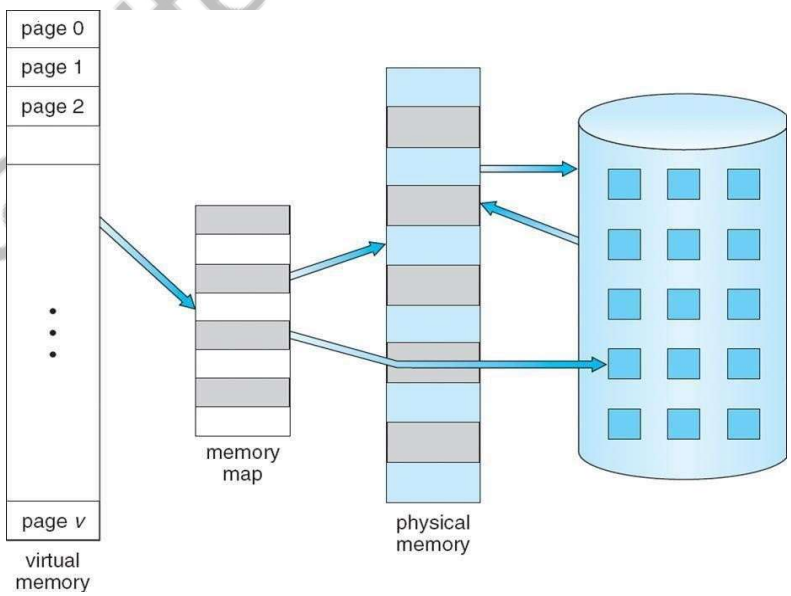


Fig: Virtual memory that is larger than physical memory.

- Virtual memory is the separation of users' logical memory from physical memory. This separation allows an extremely large virtual memory to be provided when there is less physical memory.

- Separating logical memory from physical memory also allows files and memory to be shared by several different processes through page sharing.

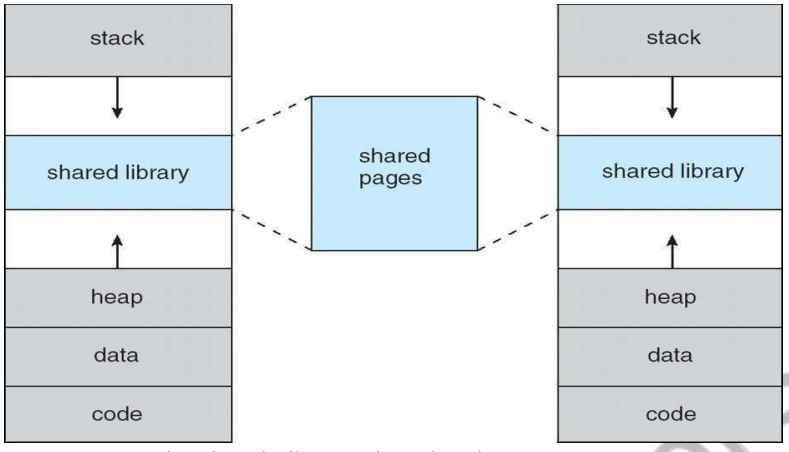


Fig: Shared Library using Virtual Memory

- Virtual memory is implemented using Demand Paging.
- Virtual address space: Every process has a virtual address space i.e used as the stack orheap grows in size.

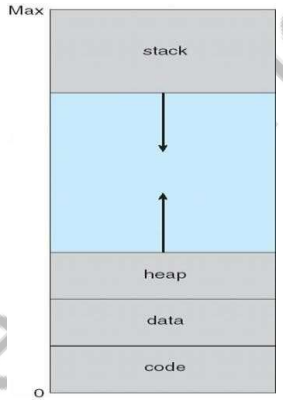


Fig: Virtual address space

DEMAND PAGING

- A demand paging is similar to a paging system with swapping. When we want to execute a process, we swap the process into memory otherwise it will not be loaded into memory.
- A swapper manipulates the entire processes, whereas a pager manipulates individual pages of the process.
  - Bring a page into memory only when it is needed.
  - Less I/O needed.
  - Less memory needed.
  - Faster response
  - More users
  - Page is needed ⇒ reference to it.
  - invalid reference ⇒abort
  - not-in-memory ⇒ bring to memory.
  - ***Lazy swapper***– never swaps a page into memory unless page will be needed.
  - Swapper that deals with pages is a **pager.**

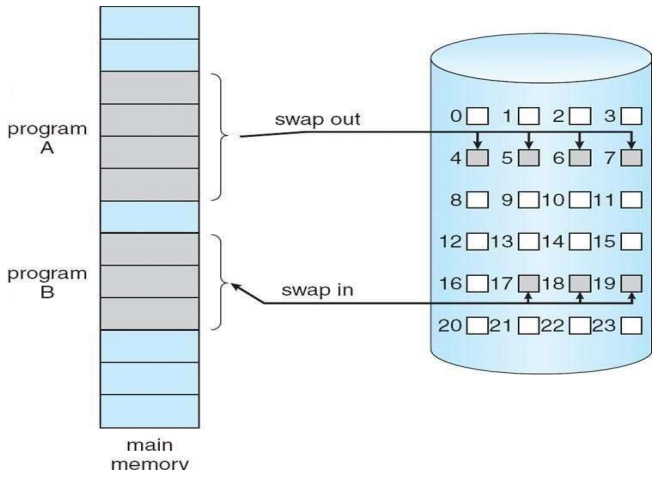


Fig: Transfer of a paged memory into continuous disk space

- **Basic concept:** Instead of swapping the whole process the pager swaps only the necessary pages into memory. Thus, it avoids reading unused pages and decreases the swap time and amount of physical memory needed.
- The valid-invalid bit scheme can be used to distinguish between the pages that are on the diskand that are in memory.
  - With each page table entry, a valid–invalid bit is associated.
  - (**v** ⇒ in-memory, **i**⇒not-in-memory)
  - Initially valid–invalid bit is set to **i**on all entries.
  - Example of a page table snapshot:

Frame #	valid-invalid bit
	v
	v
	v
	v
	i
....	
	i
	i

page table

- During address translation, if valid–invalid bit in page table entry is **I** ⇒ page fault.
- If the bit is valid then the page is both legal and is in memory.
- If the bit is invalid, then either page is not valid or is valid but is currently on the disk. Marking a page as invalid will have no effect if the processes never access that page. Suppose if it accesses the page which is marked invalid, causes a page fault trap. This may result in failure of OS to bring the desired page into memory.

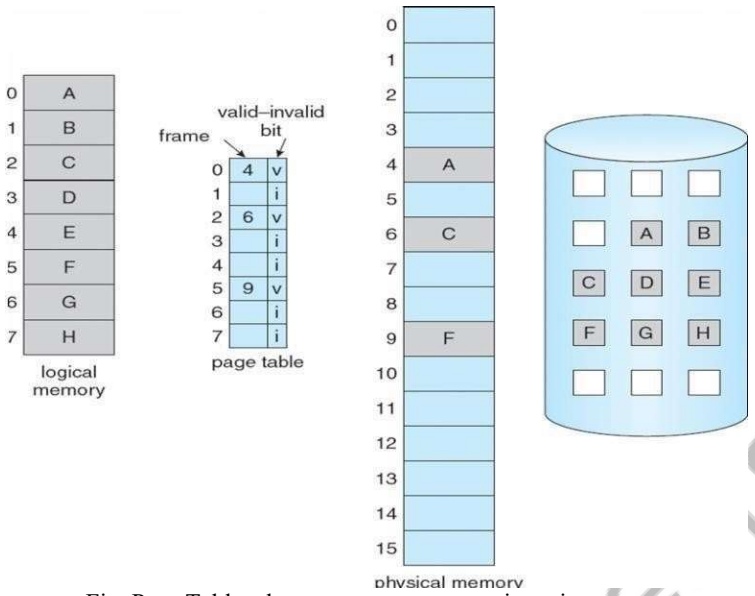


Fig: Page Table when some pages are not in main memory

Page Fault

If a page is needed that was not originally loaded up, then a ***page fault trap*** is generated.

Steps in Handling a Page Fault

1. The memory address requested is first checked, to make sure it was a valid memoryrequest.
2. If the reference is to an invalid page, the process is terminated. Otherwise, if the page isnot present in memory, it must be paged in.
3. A free frame is located, possibly from a free-frame list.
4. A disk operation is scheduled to bring in the necessary page from disk.
5. After the page is loaded to memory, the process's page table is updated with the new frame number, and the invalid bit is changed to indicate that this is now a valid page reference.
6. The instruction that caused the page fault must now be restarted from the beginning.

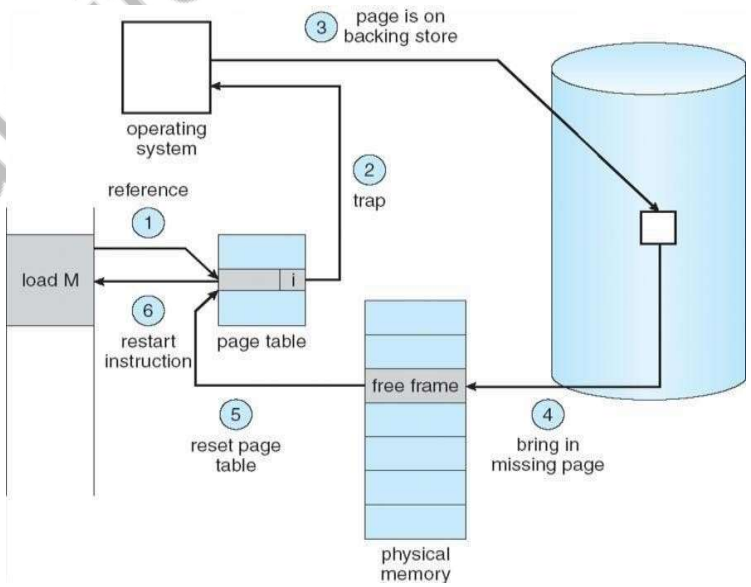


Fig: steps in handling page fault

Pure Demand Paging: Never bring a page into main memory until it is required.

- We can start executing a process without loading any of its pages into main memory.

- Page fault occurs for the non-memory resident pages.
- After the page is brought into memory, process continues to execute.
- Again, page fault occurs for the next page.

Hardware support: For demand paging the same hardware is required as paging and swapping.

1. Page table: -Has the ability to mark an entry invalid through valid-invalid bit.
2. Secondary memory: -This holds the pages that are not present in main memory.

Performance of Demand Paging: Demand paging can have significant effect on the performanceof the computer system.

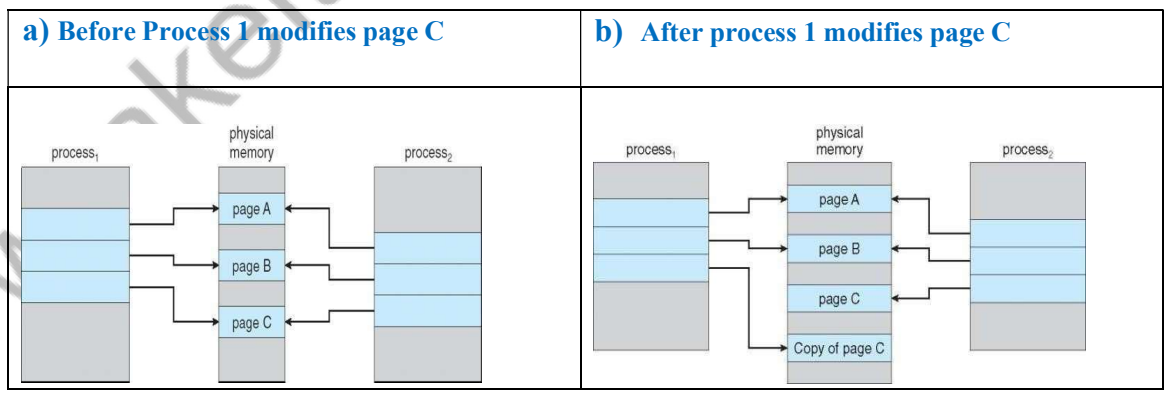
- ✓ Let P be the probability of the page fault (0<=P<=1)
- ✓ **Effective access time = (1-P) * ma + P * page fault.**
  - Where P = page fault and ma = memory access time.
- ✓ Effective access time is directly proportional to page fault rate. It is important to keep pagefault rate low in demand paging.

Demand Paging Example

- ✓ Memory access time = 200 nanoseconds
- ✓ Average page-fault service time = 8milliseconds
- ✓ EAT = (1 – p) x 200 + p (8milliseconds)
  - = (1 – p x 200 + p x 8,000,000
  - = **200 + p x 7,999,800**
- ✓ If one access out of 1,000 causes a page fault, then EAT = 8.2 microseconds. This is aslowdown by a factor of 40.

COPY-ON-WRITE

- ✓ Technique initially allows the parent and the child to share the same pages. These pages are marked as copy on- write pages i.e., if either process writes to a shared page,a copy of shared page is created.
- ✓ *Eg*:-If a child process tries to modify a page containing portions of the stack; the OS recognizes them as a copy-on-write page and create a copy of this page and maps it on to the address space of the child process. So, the child process will modify its copied page and not the page belonging to parent. The new pages are obtained from the poolof free pages.
- ✓ The previous contents of pages are erased before getting them into main memory.
- ✓ This is called **Zero – on fill demand**.



PAGE REPLACEMENT

- Page replacement policy deals with the solution of pages in memory to be replaced by a new page that must be brought in. When a user process is executing a page fault occurs.
- The hardware traps to the operating system, which checks the internal table to see that thisis a page fault and not illegal memory access.
- The operating system determines where the derived page is residing on the disk, and this finds that there are no free frames on the list of free frames.

- When all the frames are in main memory, it is necessary to bring a new page to satisfy the page fault, replacement policy is concerned with selecting a page currently in memory to be replaced.
- The page i, e to be removed should be the page i,e least likely to be referenced in future.

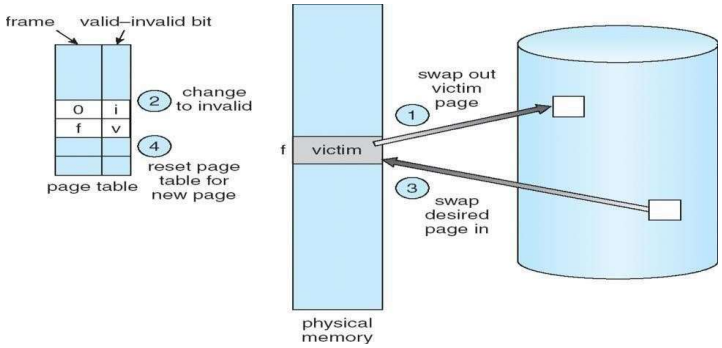


Fig: Page Replacement

Working of Page Replacement Algorithm

a. Find the location of the derived page on the disk.
b. Find a free frame x If there is a free frame, use it. x Otherwise, use a replacement algorithm to select a victim.
   i. Write the victim page to the disk.
   ii. Change the page and frame tables accordingly.
c. Read the desired page into the free frame; change the page and frame tables.
d. Restart the user process.

Victim Page

- The page that is supported out of physical memory is called victim page.
- If no frames are free, the two-page transforms come (out and one in) are read. This will seethe effective access time.
- Each page or frame may have a dirty (modify) bit associated with the hardware. Themodify bit for a page is set by the hardware whenever any word or byte in the page is written into, indicating that the page has been modified.
- When we select the page for replacement, we check its modify bit. If the bit is set, then the page is modified since it was read from the disk.
- If the bit was not set, the page has not been modified since it was read into memory. Therefore, if the copy of the page has not been modified, we can avoid writing the memory page to the disk, if it is already there. Sum pages cannot be modified.

✓ Modify bit/ Dirty bit:

- Each page/frame has a modify bit associated with it.
- If the page is not modified (read-only) then one can discard such page without writing itonto the disk. Modify bit of such page is set to0.
- Modify bit is set to 1, if the page has been modified. Such pages must be written to thedisk.
- Modify bit is used to reduce overhead of page transfers – only modified pages are written to disk.

PAGE REPLACEMENT ALGORITHMS

- Want lowest page-fault rate.
- Evaluate algorithm by running it on a particular string of memory references (referencestring) and computing the number of page faults on that string.
- In all our examples, the reference string is

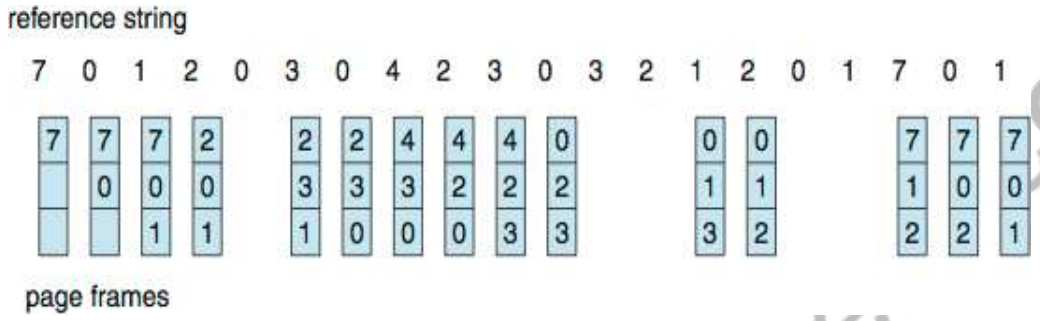
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

FIFO Algorithm:

oThis is the simplest page replacement algorithm. A FIFO replacement algorithm associates each page

with the time when that page was brought into memory.

- oWhen a Page is to be replaced the oldest one is selected.
- oWe replace the queue at the head of the queue. When a page is brought into memory, weinsert it at the tail of the queue.
- oIn the following example, a reference string is given and there are 3 free frames. There are 20 page requests, which results in 15 page faults.



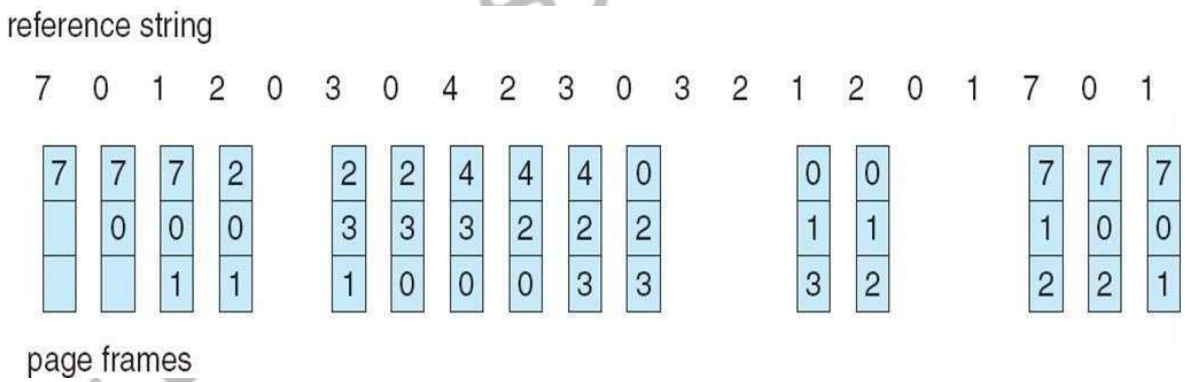
Belady's Anomaly

- For some page replacement algorithms, the page fault may increase as the number of allocated frames increases. FIFO replacement algorithm may face this problem.

more frames ⇒ more page faults

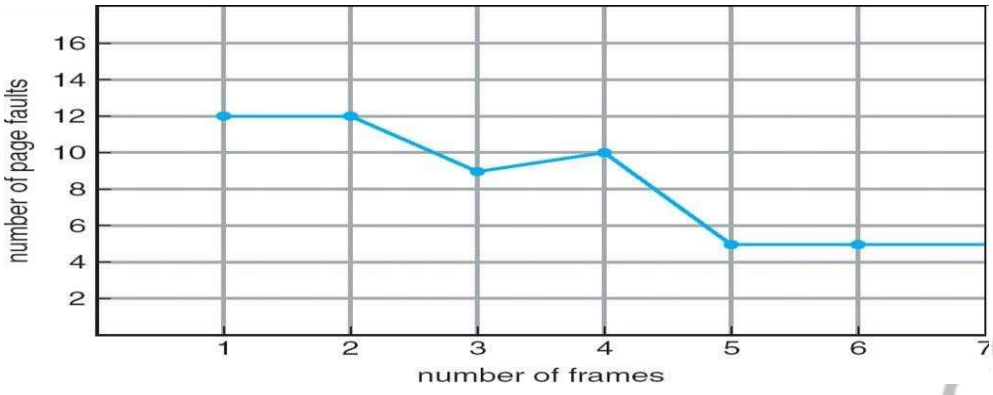
Example: Consider the following references string with frames initially empty.

- The first three references (7,0,1) cases page faults and are brought into the empty frames.
- The next reference 2 replaces page 7 because page 7 was brought in first. x Since 0 isthe next references and 0 is already in memory e has no page faults.
- The next references 3 results in page 0 being replaced so that the next references to 0causer



page fault. This will continue till the end of string. There are 15 faults altogether.

FIFO Illustrating Belady's Anomaly



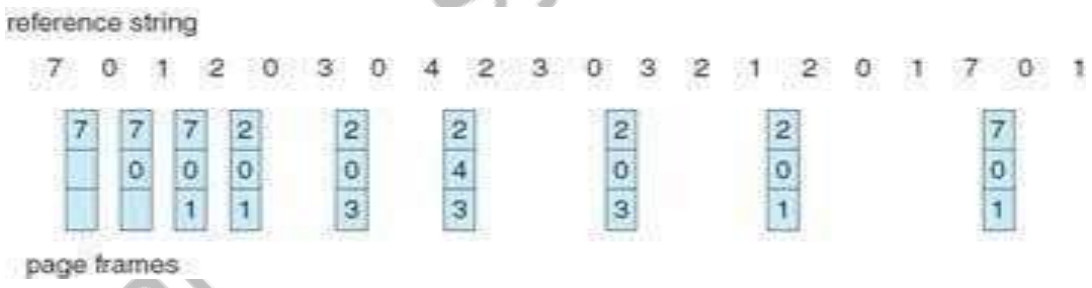
Optimal Algorithm

- Optimal page replacement algorithm is mainly to solve the problem of Belady's Anomaly.
- The Optimal page replacement algorithm has the lowest page fault rate of all algorithms.
- An optimal page replacement algorithm exists and has been called OPT.

The working is simple ***"Replace the page that will not be used for the longest period of time"***Example: consider the following reference string

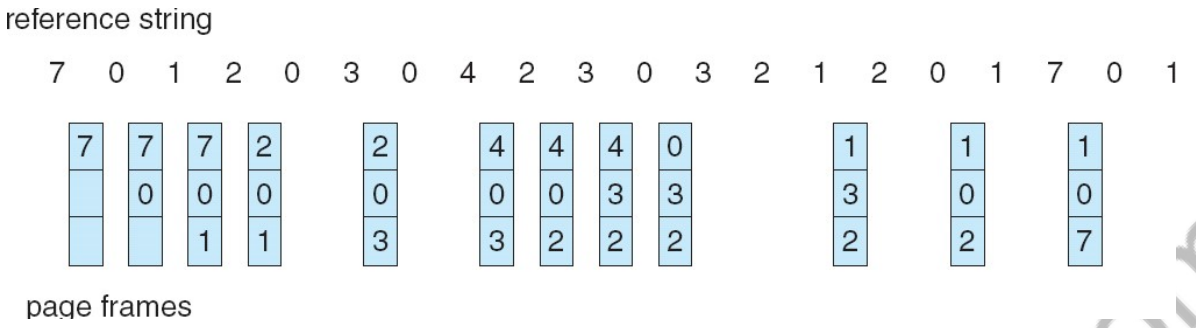
- The first three references cause faults that fill the three empty frames.
- The references to page 2 replace page 7, because 7 will not be used until reference 18. The page 0 will be used at 5 and page 1 at 14.
- With only 9 page faults, optimal replacement is much better than a FIFO, which had 15 faults. This algorithm is difficult to implement because it requires future knowledge of reference strings.
- Replace page that will not be used for longest period of time.

Optimal Page Replacement



Least Recently Used (LRU) Algorithm

- The ***LRU (Least Recently Used)*** algorithm predicts that the page that has not been usedin the longest time is the one that will not be used again in the near future.
- Some view LRU as analogous to OPT, but here we look backwards in time instead offorwards.



The main problem is how to implement LRU is the LRU requires additional h/w assistance.

Two implementations are possible:

1. **Counters:** In this we associate each page table entry a time -of -use field and add to the CPU a logical clock or counter. The clock is incremented for each memory reference. When a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page table entry for that page. In this way wehave the time of last reference to each page we replace the page with smallest time value. The time must also be maintained when page tables are changed.
2. **Stack:** Another approach to implement LRU replacement is to keep a stack of page numbers when a page is referenced it is removed from the stack and put on to the top of stack. In this way the top of stack is always the most recently used page and the bottom in least recently used page. Since the entries are removed from the stack it is best implement by a doubly linked list. With a head and tail pointer.

Note: Neither optimal replacement nor LRU replacement suffers from Belady's Anamoly. These are called stack algorithms.

LRU-Approximation Page Replacement

- Many systems offer some degree of hardware support, enough to approximate LRU.
- In particular, many systems provide a ***reference bit*** for every entry in a page table, whichis set anytime that page is accessed. Initially all bits are set to zero, and they can also all becleared at any time. One bit distinguishes pages that have been accessed since the last clearfrom those that have not been accessed.

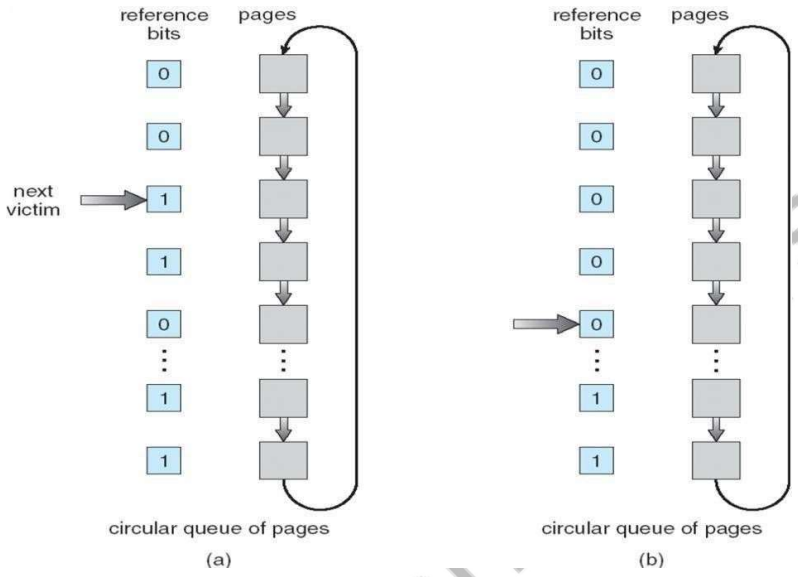
Additional-Reference-Bits Algorithm

- An 8-bit byte (reference bit) is stored for each page in a table in memory.
- At regular intervals (say, every 100 milliseconds), a timer interrupt transfers control to theoperating system. The operating system shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right by 1 bit and discarding the low-order bit.
- These 8-bit shift registers contain the history of page use for the last eight time periods.
- If the shift register contains 00000000, then the page has not been used for eight time periods.
- A page with a history register value of 11000100 has been used more recently than one with a value of 01110111.

Second- chance (clock) page replacement algorithm

- The ***second chance algorithm*** is a FIFO replacement algorithm, except the reference bitis used to give pages a second chance at staying in the page table.
- When a page must be replaced, the page table is scanned in a FIFO (circular queue)manner.
- If a page is found with its reference bit as '0', then that page is selected as the nextvictim.

- If the reference bit value is '1', then the page is given a second chance and its reference bitvalue is cleared (assigned as '0').
- Thus, a page that is given a second chance will not be replaced until all other pages havebeen replaced (or given second chances). In addition, if a page is used often, then it sets its reference bit again.
- This algorithm is also known as the ***clock*** algorithm.



Enhanced Second-Chance Algorithm

- The **enhanced second chance algorithm** looks at the reference bit and the modify bit (dirtybit) as an ordered page, and classifies pages into one of four classes:
  1. (0, 0) - Neither recently used nor modified.
  2. (0, 1) - Not recently used, but modified.
  3. (1, 0) - Recently used, but clean.
  4. (1, 1) - Recently used and modified.
- This algorithm searches the page table in a circular fashion, looking for the first page it can find in the lowest numbered category. i.e. it first makes a pass looking for a (0, 0), and thenif it can't find one, it makes another pass looking for a(0,1), etc.
- The main difference between this algorithm and the previous one is the preference for replacing clean pages if possible.

Count Based Page Replacement

There are many other algorithms that can be used for page replacement, we can keep a counterof the number of references that has made to a page.

a) **LFU** (least frequently used):

This causes the page with the smallest count to be replaced. The reason for this selection is that an actively used page should have a large reference count.

This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process but never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed.

b) **MFU** Algorithm:

based on the argument that the page with the smallest count was probably just brought in and has yet to be used

ALLOCATION OF FRAMES

- The absolute minimum number of frames that a process must be allocated is dependent on system architecture.
- The maximum number is defined by the amount of available physical memory.

Allocation Algorithms

After loading of OS, there are two ways in which the allocation of frames can be done to the processes.

Equal Allocation- If there are m frames available and n processes to share them, each process gets m / n frames, and the left overs are kept in a free-frame buffer pool.

Proportional Allocation - Allocate the frames proportionally depending on the size of the process. If the size of process i is S_i, and S is the sum of size of all processes in the system, then the allocation for process P_i is $a_i = m * S_i / S$. where m is the free frames available in the system.

- Consider a system with a 1KB 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.
- with proportional allocation, we would split 62 frames between two processes, as follows:
  m=62, S = (10+127) =137
  Allocation for process 1 = 62 X 10/137 ~ 4 Allocation for process 2 = 62 X 127/137 ~57
  Thus allocates 4 frames and 57 frames to student process and databaserespectively.
- Variations on proportional allocation could consider priority of process rather than justtheir size.

Global versus Local Allocation

- Page replacement can occur both at the local or global level.
- With local replacement, the number of pages allocated to a process is fixed, and page replacement occurs only amongst the pages allocated to this process.
- With global replacement, any page may be a potential victim, whether it currently belongs to the process seeking a free frame or not.
- Local page replacement allows processes to better control their own page fault rates and leads to more consistent performance of a given process over different system load levels.
- Global page replacement is overall more efficient and is the more commonly used approach.

Non-Uniform Memory Access (New)

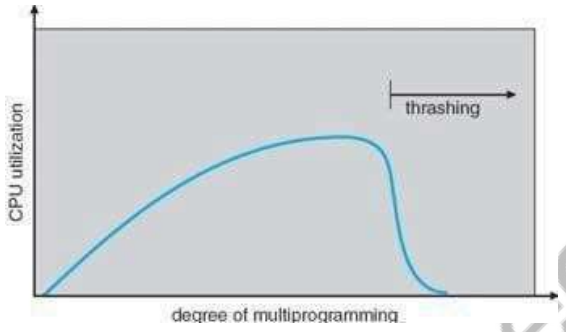
- Usually, the time required to access all memory in a system is equivalent.
- This may not be the case in multiple-processor systems, especially where each CPU is physically located on a separate circuit board which also holds some portion of the overall system memory.
- In such systems, CPU s can access memory that is physically located on the same boardmuch faster than the memory on the other boards.
- The basic solution is akin to processor affinity - At the same time that we try to schedule processes on the same CPU to minimize cache misses, we also try to allocate memory for those processes on the same boards, to minimize access times.

THRASHING

- If the number of frames allocated to a low-priority process falls below the minimum number required by the computer architecture, then we suspend the process execution.
- A process is thrashing if it is spending more time in paging than executing.
- If the processes do not have enough number of frames, it will quickly page fault. During this it must replace some pages that are not currently in use. Consequently, it quickly faults again and again.
- The process continues to make a fault, replacing pages for which it then faults and brings back. This high paging activity is called thrashing. The phenomenon of excessively moving pages back and forth b/w memory and secondary has been called ***thrashing***.

Cause of Thrashing

- Thrashing results in severe performance problems.
- The operating system monitors the CPU utilization is low. We increase the degree ofmulti programming by introducing new processes to the system.
- A global page replacement algorithm replaces pages with no regard to the process towhich they belong.



The figure shows the thrashing

- As the degree of multi programming increases, CPU utilization increases, although more slowly until a maximum is reached.
- If the degree of multi programming is increased even further, **thrashing sets in** and the CPU utilization drops sharply.
- At this point, to increase CPU utilization and stop thrashing, we must decrease the degree of multiprogramming.
- We can limit the effect of thrashing by using a local replacement algorithm. To prevent thrashing, we must provide a process with as many frames as it needs.

Locality of Reference:

- As the process executes, it moves from locality to locality.
- A locality is a set of pages that are actively used together.
- A program may consist of several different localities, which may overlap.
- Locality is caused by loops in code that find reference arrays and other data structures by indices.
- The ordered list of page numbers accessed by a program is called reference string.
- Locality is of two types:
  1. spatial locality
  2. temporal locality

Working set model

- Working set model algorithm uses the current memory requirements to determine the number of page frames to allocate to the process, an informal definition is "thecollection of pages that a process is working with, and which must be resident if the process to avoid thrashing". The idea is to use the recent needs of a process to predict its future reader.
- The working set is an approximation of programs locality. Ex: given a sequence of memory reference, if the working set window size to memory references, then working set at time t1 is {1,2,5,6,7} and at t2 is changed to {3,4}
- At any given time, all pages referenced by a process in its last 4 seconds of execution are considered to compromise its working set.
- A process will never execute until its working set is resident in its main memory.
- Pages outside the working set can be discarded at any movement.

- Working sets are not enough and we must also introduce balance set.
  - If the sum of the working sets of all the run able process is greater than the size of memory the refuse some process for a while.
  - Divide the run able process into two groups, active and inactive. The collection of active set is called the balance set. When a process is made active its working set is loaded.
  - Some algorithms must be provided for moving processes into and out of the balance set. As a working set is changed, corresponding change is made to the balance set.
  - Working set presents thrashing by keeping the degree of multi programming as high as possible. Thus, it optimizes the CPU utilization. The main disadvantage of this is keeping track of the working set.

Page-Fault Frequency

- When page- fault rate is too high, the process needs more frames and when it is too low,the process may have too many frames.
- The upper and lower bounds can be established on the page-fault rate. If the actualpage- fault rate exceeds the upper limit, allocate the process another frame or suspend the process.
- If the page-fault rate falls below the lower limit, remove a frame from the process. Thus, we can directly measure and control the page-fault rate to prevent thrashing.

Solved Exercises (VTU QP problems)

For the following page reference calculate the page faults that occur using FIFO and LRU for 3 and 4 page frames respectively, 5,4,2,1,4,3,5,4,3,2,1,5. (10 marks) **Jan 2015**

Solution:

3 Frames (FIFO) -------- 10 pagefaults

		3	3	3	4	4	4	2	2
	4	4	4	1	1	1	5	5	5
5	5	5	2	2	2	3	3	3	1

3 Frames (FIFO) -------11 page faults

			2	2	2	2	3	3	3	3
		3	3	3	3	4	4	4	4	5
	4	4	4	4	5	5	5	5	1	1
5	5	5	5	1	1	1	1	2	2	2

3 Frames (LRU)---------11 page faults

		3	3	3	4	4	4	4	1	1
	4	4	4	1	1	1	5	2	2	2
5	5	5	2	2	2	3	3	3	3	5

4 Frames (LRU)-------- 9 page faults

			2	2	5	5	1	1
		3	3	3	3	3	3	3
	4	4	4	4	4	4	4	5
5	5	5	5	1	1	2	2	2

MODULE: 5
CONTENTS:

❖ File System

- File concept
- Access methods
- Directory structure
- File system mounting
- File sharing
- Protection

❖ Implementation of File System

- File system structure
- File system implementation
- Directory implementation
- Allocation methods
- Free space management

❖ Secondary Storage Structures

- Mass storage structures
- Disk structure
- Disk attachments
- Disk scheduling
- Disk management
- swap space management

❖ Protection

- Goals of protection
- Principles of protection
- Domain of protection
- Access matrix
- Access control
- Revocation of access rights
- Capability- Based systems

MODULE 5
FILE CONCEPT

FILE:

- *A* file is a named collection of related information that is recorded on secondary storage.
- The information in a file is defined by its creator. Many different types of informationmay be stored in 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.
  - *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 thesystem's linker.
  - An *executable* file is a series of code sections that the loader can bring into memoryand execute.

File Attributes

- A file is named, for the convenience of its human users, and is referred to by itsname. A name is usually a string of characters, such as *example*.c
- When a file is named, it becomes independent of the process, the user, and even thesystem that created it.

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

- **Name:** The symbolic file name is the only information kept in human readable form.
- **Identifier:** This unique tag, usually a number, identifies the file within the filesystem; 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 themaximum 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.

The information about all files is kept in the directory structure, which also resides on secondary storage. Typically, a directory entry consists of the file's name and its unique identifier. The identifier in turn locates the other file attributes.

File Operations

A file is an abstract data type. To define a file properly, we need to consider the operations that can be performed on files.

1. **Creating a file:** Two steps are necessary to create a file,
   a) Space in the file system must be found for the file.
   b) An entry for the new file must be made in the directory.
2. **Writing a file:** To write a file, we make a system call specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the file's location. The system must keep a write pointer to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.
3. **Reading a file:** To read from a file, we use a system call that specifies the name of thefile and where the next block of the file should be put. Again, the directory issearched for the associated entry, and the system needs to keep a read pointer to the location in the file where the next read

is to take place. Once the read has taken place,the read pointer is updated. Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process current file-position pointer.

4. **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 withina file need not involve any actual I/0. This file operation is also known as files seek.
5. **Deleting a file:** To delete a file, search the directory for the named file. Having found the associated directory entry, then release all file space, so that it can be reused by other files, and erase the directory entry.
6. **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 but lets the file be reset to length zero and its file space released.

- Other common operations include appending new information to the end of an existing file and renaming an existing file.
- Most of the file operations mentioned involve searching the directory for the entry associated with the named file.
- To avoid this constant searching, many systems require that an open () system call be made before a file is first used actively.
- The operating system keeps a small table, called the **open file table** containinginformation about all open files. When a file operation is requested, the file isspecified via an index into this table, so no searching is required.
- The implementation of the open() and close() operations is more complicated in an environment where several processes may open the file simultaneously.
- The operating system uses two levels of internal tables:

1. A per-process table
2. A system-wide table

The per-process table:

- Tracks all files that a process has open. Stored in this table is information regarding theuse of the file by the process.
- Each entry in the per-process table in turn points to a system-wide open-file table.

The system-wide table

- contains process-independent information, such as the location of the file on disk, access dates, and file size. Once a file has been opened by one process, the system-wide table includes an entry for the file.

Several pieces of information are associated with an open file.

1. **File pointer:** On systems that do not include a file offset as part of the read() and write() system calls, the system must track the last read write location as a current- file-position pointer. This pointer is unique to each process operating on the file and therefore must be kept separate from the on-disk file attributes.
2. **File-open count:** As files are closed, the operating system must reuse its open- file table entries, or it could run out of space in the table. Because multiple processes mayhave opened a file, the system must wait for the last file to close before removing the open-file table entry. The file-open counter tracks the number of opens and closes andreaches zero on the last close. The system can then remove the entry.
3. **Disk location of the file:** Most file operations require the system to modify data within the file. The information needed to locate the file on disk is kept in memory sothat the system does not have to read it from disk for each operation.
4. **Access rights:** Each process opens a file in an access mode. This information is stored on the per-process table so the operating system can allow or deny subsequent I/0 requests.

File Types

- The operating system should recognize and support file types. If an operating system recognizes the type of a file, it can then operate on the file in reasonable ways.
- 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**, usuallyseparated by a period character.

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

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, 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 com-pressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

File Structure

- File types also can be used to indicate the internal structure of the file. For instance, source and object files have structures that match the expectations of the programs that read them. Certain files must conform to a required structure that is understood by the operating system.
  For example: the operating system requires that an executable file has a specific structure so that it can determine where in memory to load the file and what the location of the first instruction is.
- The operating system supports multiple file structures: the resulting size of the operating system also increases. If the operating system defines five different file structures, it needs to contain the code to support these file structures.
- It is necessary to define every file as one of the file types supported by the operating system. When new applications require information structured in ways not supported by the operating system, severe problems may result.
- Example: The Macintosh operating system supports a minimal number of file structures. Itexpects files to contain two parts: a resource fork and data fork.
  - **The resource fork** contains information of interest to the user.
  - **The data fork** contains program code or data.

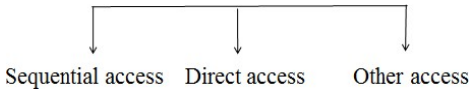
Internal File Structure

- Internally, locating an offset within a file can be complicated for the operating system. Disk systems typically have a well-defined block size determined by the size of a sector.
- All disk I/0 is performed in units of one block (physical record), and all blocks are the same size. It is unlikely that the physical record size will exactly match the length of thedesired logical record.
- Logical records may even vary in length. Packing a number of logical records into physical blocks is a common solution to this problem.

ACCESS METHODS

- Files store information. When it is used, this information must be accessed and readinto computer memory. The information in the file can be accessed in several ways.

- Some of the common methods are:



1. Sequential methods

- The simplest access method is sequential methods. Information in the file is processed in order, one record after the other.
- Reads and writes make up the bulk of the operations on a file.
- A read operation (next-reads) reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location.
- The write operation (write next) appends to the end of the file and advances to theend of the newly written material.
- A file can be reset to the beginning and on some systems, a program may be able toskip forward or backward n records for some integer n-perhaps only for n =1.

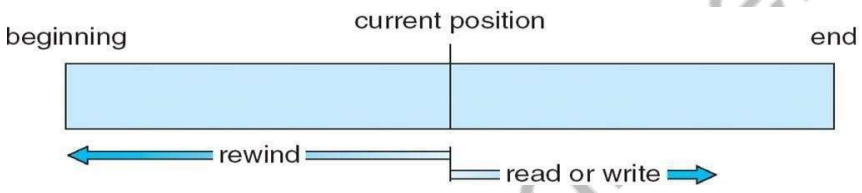


Figure: Sequential-access file.

2. Direct Access

- A file is made up of fixed length 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.
- Example: if we may read block 14, then read block 53, and then write block 7. Thereare 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 such as Databases, where searching becomes easy and fast.
- 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.
- An alternative approach is to retain read next and write next, as with sequential access, and to add an operation position file to n, where n is the block number. Then, to affect a read n, we would position to n and then read next.

sequential access	implementation for direct access
reset	*cp* = 0;
read next	*read cp*; *cp* = *cp* + 1;
write next	*write cp*; *cp* = *cp* + 1;

Figure: Simulation of sequential access on a direct-access file.

3. Other Access Methods:

- Other access methods can be built on top of a direct-access method. These methods generally involve the construction of an index for the file.
- The **Index** is like an index in the back of a book that 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.

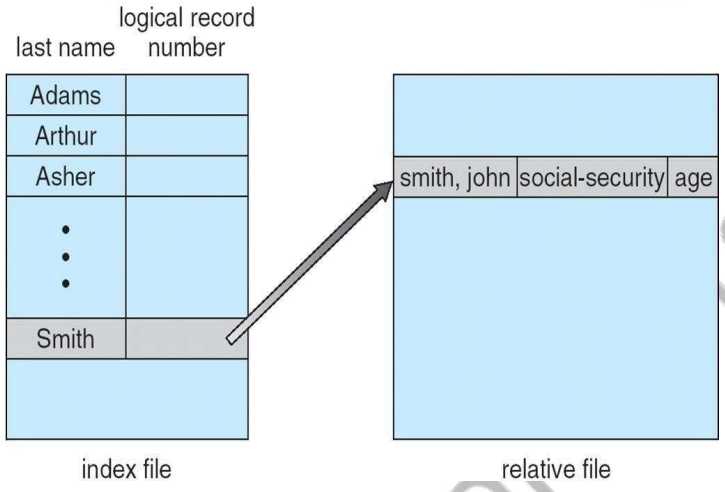


Figure: Example of index and relative files

DIRECTORY AND DISK STRUCTURE

- Files are stored on random-access storage devices, including hard disks, optical disks, and solid state (memory-based) disks.
- A storage device can be used in its entirety for a file system. It can also be subdivided for finer-grained control.
- Disk can be subdivided into partitions. Each disk or partition can be RAID protected against failure.
- Partitions are also known as minidisks or slices. Entity containing file system known asa volume. Each volume that contains a file system must also contain information about the files in the system. This information is kept in entries in a **device directory**or **volume table of contents.**

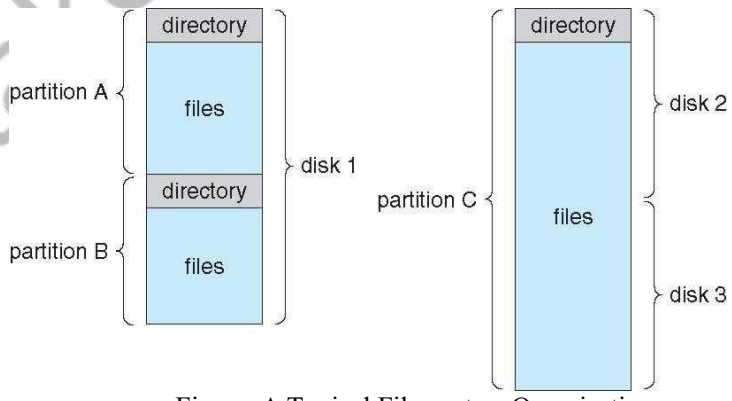


Figure: A Typical File-system Organization

Directory Overview

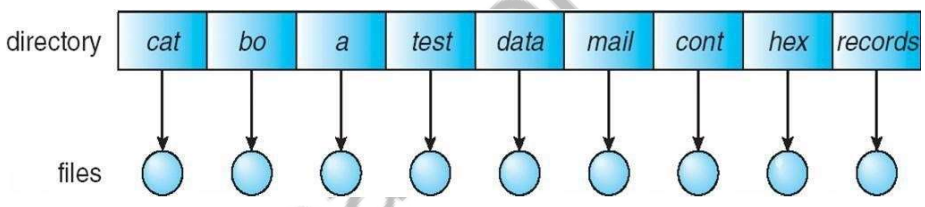
- The directory can be viewed as a symbol table that translates file names into their directory entries. A directory contains information about the files including attributes location and ownership. To consider a particular directory structure, certain operationson the directory have to be considered:

- **Search for a file:** Directory structure is searched for a particular file in directory. Files have symbolic names and similar names may indicate a relationship between files. Using this similarity, it will be easy to find all whose name matches a particular pattern.
- **Create a file:** New files needed to be created and added to the directory.
- **Delete a file:** When a file is no longer needed, then it is able to remove it from the directory.
- **List a directory:** It is able to list the files in a directory and the contents of the directory entry for each file in the list.
- **Rename a file:** Because the name of a file represents its contents to its users, it is possible to change the name when the contents or use of the file changes. Renaminga file may also allow its position within the directory structure to be changed.
- **Traverse the file system:** User may wish to access every directory and every file within a directory structure. To provide reliability the contents and structure of the entire file system is saved at regular intervals.

- The most common schemes for defining the logical structure of a directory are described below.
  1. Single-level Directory
  2. Two-Level Directory
  3. Tree-Structured Directories
  4. Acyclic-Graph Directories
  5. General Graph Directory

1. Single-level Directory

- The simplest directory structure is the single-level directory. All files are contained inthe same directory, which makes it easy to support and understand.



- The entire system contains only one directory, which contains only one entry per file.

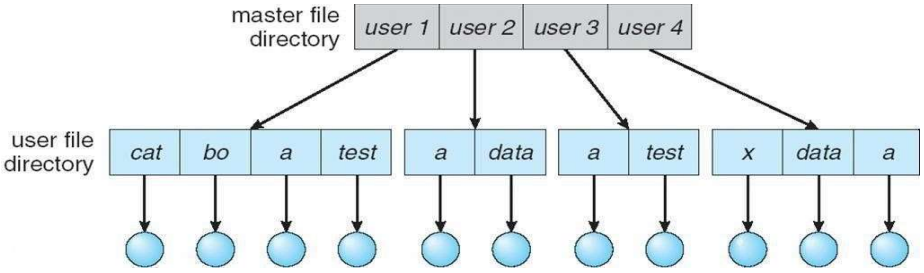
Advantages:
Implementation is easy.
If there are less number of files, searching is easier.
Disadvantages:

- We cannot have 2 files with the same name, because of name collision.
- Searching time increases with the increase in directory size (number of files increases).
- Protection cannot be implemented for multiple users.
- Finding a unique name for every file is difficult when the number of files increases or when the system has more than one user.
- As directory structure is single, uniqueness of file name has to be maintained, which is difficult when there are multiple users.
- Even a single user on a single-level directory may find it difficult to remember the namesof all the files as the number of files increases.
- It is not uncommon for a user to have hundreds of files on one computer system and an equal number of additional files on another system. Keeping track of so many files is a daunting task.

2. Two-Level Directory

- In the two-level directory structure, each user has its own **user file directory** (UFD). The UFDs have similar structures, but each lists only the files of a single user.
- When a user refers to a particular file, only his own UFD is searched. 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'sfile that has the same name.

- 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 pointsto the UFD for that user.



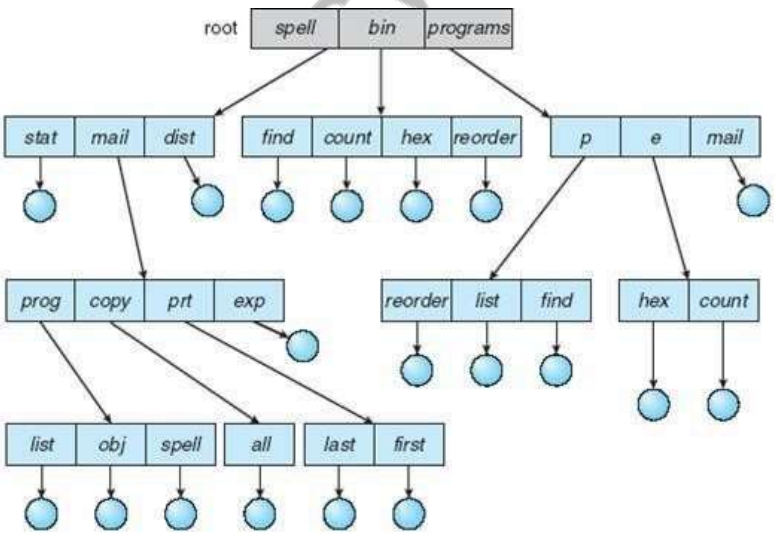
- **Advantage:**
  - No file name-collision among different users.
  - Efficient searching.
- **Disadvantage**
  - Users are isolated from one another and can't cooperate on the same task.

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 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 theentry as a file (0) or as a subdirectory (1). Special system calls are used to create and delete directories.

Two types of path-names:

1. Absolute path-name: begins at the root.
2. Relative path-name: defines a path from the current directory.



How to delete directory?

1. To delete an empty directory: Just delete the directory.
2. To delete a non-empty directory:
   - First, delete all files in the directory.
   - If any subdirectories exist, this procedure must be applied recursively to them.

Advantage:

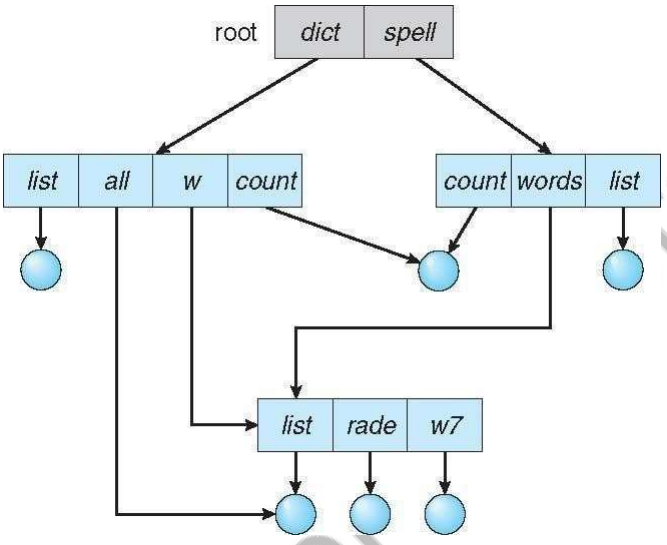
- Users can be allowed to access the files of other users.

Disadvantages:

- A path to a file can be longer than a path in a two-level directory.
- Prohibits the sharing of files (or directories).

4. Acyclic Graph Directories

- The common subdirectory should be shared. A shared directory or file will exist in the file system in two or more places at once. A tree structure prohibits the sharing of files or directories.
- An acyclic graph is a graph with no cycles. It allows directories to share subdirectoriesand files.



The same file or subdirectory may be in two different directories. The acyclic graph is a natural generalization of the tree-structured directory scheme.

Two methods to implement shared files (or subdirectories):

1. Create a new directory-entry called a link. A link is a pointer to another file (or subdirectory).
2. Duplicate all information about shared files in both sharing directories.

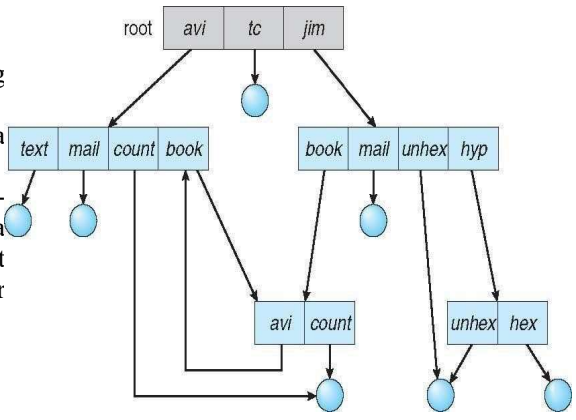
Two problems:

1. A file may have multiple absolute path-names.
2. Deletion may leave dangling-pointers to the non-existent file.

Solution to deletion problem:

1. Use back-pointers: Preserve the file until all references to it are deleted.
2. With symbolic links, remove only the link, not the file. If the file itself isdeleted, the link can be removed.

5. General Graph Directory



- **Problem:** If there are cycles, we want to avoid searching components twice.
- **Solution:** Limit the no. of directories accessed in a search.
- **Problem:** With cycles, the reference-count may be non-zero even when it is no longer possible to refer to a directory (or file). (A value of 0 in the reference count means that there are no more references to the file or directory, and the file can be deleted).
- **Solution:** Garbage-collection scheme can be used to determine when the last reference has been deleted. Garbage collection is involved.
  - First pass traverses the entire file-system and marks everything that can be accessed.
  - A second pass collects everything that is not marked onto a list of free-space.

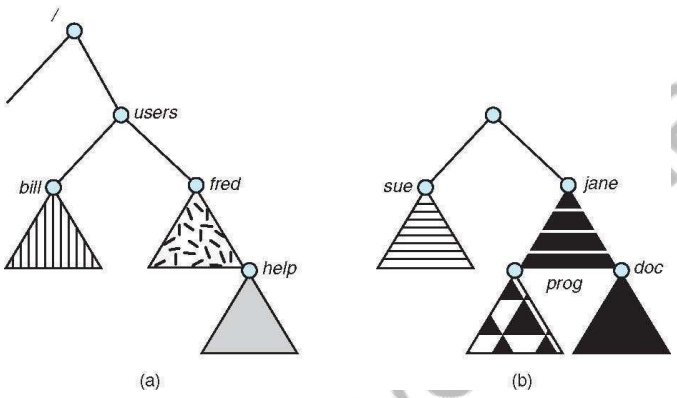
FILE SYSTEM MOUNTING

- A file must be *opened* before it is used, a file system must be *mounted* before it can beavailable to process on the system.
- **Mount Point:** The location within the file structure where the file system is to be attached.

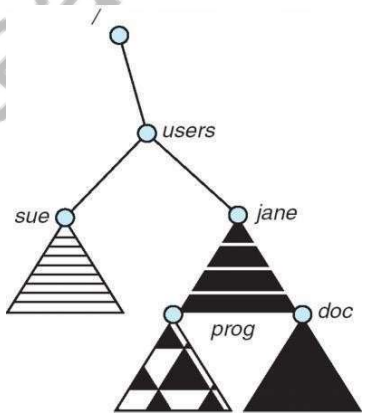
The mounting procedure:

- The operating system is given the name of the device and the mount point.
- The operating system verifies that the device contains a valid file system. It does so by asking the device driver to read the device directory and verifying that the directory has the expected format.
- The operating system notes in its directory structure that a file system is mounted at the specified mount point.

To illustrate file mounting, consider the file system shown in figure. The triangles represent sub-trees of directories that are of interest.



- Figure (a) shows an existing file system,
- while Figure 1(b) shows an un-mounted volume residing on */device/dsk.* At this point,only the files on the existing file system can be accessed.



- Above figure shows the effects of mounting the volume residing on */device/dsk* over */users.*
- If the volume is un-mounted, the file system is restored to the situation depicted in the first Figure.

FILE SHARING

- Sharing of files on multi-user systems is desirable.
- Sharing may be done through a protection scheme.
- On distributed systems, files may be shared across a network.
- Network File-system (NFS) is a common distributed file-sharing method.

Multiple Users

File-sharing can be done in 2 ways:

1. The system can allow a user to access the files of other users by default or
2. The system may require that a user specifically grant access.

To implement file-sharing, the system must maintain more file- & directory-attributes than on asingle-user system.
Most systems use concepts of file owner and group.

1. *Owner*
   - The user who may change attributes & grant access and has the most control over the file(or directory).
   - Most systems implement owner attributes by managing a list of user-names and user IDs
2. *Group*
   - The group attribute defines a subset of users who can share access to the file.
   - Group functionality can be implemented as a system-wide list of group-names and groupIDs.
   - Exactly which operations can be executed by group-members and other users is definableby the file's owner.
   - The owner and group IDs of files are stored with the other file-attributes and can be usedto allow/deny requested operations.

Remote File Systems

It allows a computer to mount one or more file-systems from one or more remote- machines.There are three methods:

1. Manually transferring files between machines via programs like ftp.
2. Automatically DFS (Distributed file-system): remote directories are visible from a localmachine.
3. Semi-automatically via www (World Wide Web): A browser is needed to gain access tothe remote files, and separate operations (a wrapper for ftp) are used to transfer files.

ftp is used for both anonymous and authenticated access. **Anonymous access** allows a user totransfer files without having an account on the remote system.

Client Server Model

- Allows clients to mount remote file-systems from servers.
- The machine containing the files is called the **server**. The machine seeking access tothe files is called the **client**.
- A server can serve multiple clients, and a client can use multiple servers.
- The server specifies which resources (files) are available to which clients.
- A client can be specified by a network-name such as an IP address.

Disadvantage:

- Client identification is more difficult.
- In UNIX and its NFS (network file-system), authentication takes place via the client networking information by default.
- Once the remote file-system is mounted, file-operation requests are sent to the servervia the DFS protocol.

Distributed Information Systems

- Provides unified access to the information needed for remote computing.
- The DNS (domain name system) provides hostname-to-network address translations.
- Other distributed info. systems provide username/password space for a distributed facility.

Failure Modes

- Local file-systems can fail for a variety of reasons such as failure of disk (containing thefile-system), corruption of directory-structure & cable failure.
- Remote file-systems have more failure modes because of the complexity of network-systems.
- The network can be interrupted between 2 hosts. Such interruptions can result fromhardware

failure, poor hardware configuration or networking implementation issues.

- DFS protocols allow delaying of file-system operations to remote-hosts, with the hopethat the remote-host will become available again.
- To implement failure-recovery, some kind of state information may be maintained onboth the client and the server.

Consistency Semantics

- These represent an important criterion of evaluating file-systems that support file- sharing. These specify how multiple users of a system are to access a shared-file simultaneously.
- In particular, they specify when modifications of data by one user will be observed by other users.
- These semantics are typically implemented as code with the file-system.
- These are directly related to the process-synchronization algorithms.
- A successful implementation of complex sharing semantics can be found in theAndrew file-system (AFS).

UNIX Semantics

- UNIX file-system (UFS) uses the following consistency semantics:
    1. Writes to an open-file by a user are visible immediately to other users whohave this file opened.
    2. One mode of sharing allows users to share the pointer of current location intoa file. Thus, the advancing of the pointer by one user affects all sharing users.
- A file is associated with a single physical image that is accessed as an exclusive resource.
- Contention for the single image causes delays in user processes.

Session Semantics

The AFS uses the following consistency semantics:

1. Writes to an open file by a user are not visible immediately to other users that have thesame file open.
2. Once a file is closed, the changes made to it are visible only in sessions starting later.The already open instances of the file do not reflect these changes.

- A file may be associated temporarily with several (possibly different) images at the sametime.
- Consequently, multiple users are allowed to perform both read and write accessesconcurrently on their images of the file, without delay.
- Almost no constraints are enforced on scheduling accesses.

Immutable Shared Files Semantics

- Once a file is declared as shared by its creator, it cannot be modified.
- An immutable file has 2 key properties:
    1. File-name may not be reused.
    2. File-contents may not be altered.
- Thus, the name of an immutable file signifies that the contents of the file are fixed.
- The implementation of these semantics in a distributed system is simple, because the sharing is disciplined.

PROTECTION

- When information is stored in a computer system, we want to keep it safe from physical damage (reliability) and improper access (protection).
- Reliability is generally provided by duplicate copies of files.
- For a small single-user system, we might provide protection by physically removing thefloppy disks and locking them in a desk drawer.
- File owner/creator should be able to control what can be done and by whom.

Types of Access

- Systems that do not permit access to the files of other users do not need protection. This istoo extreme, so controlled access is needed.
- Following operations may be controlled:
  1. ***Read:*** Read from the file.
  2. ***Write:*** Write or rewrite the file.
  3. ***Execute:*** Load the file into memory and execute it.
  4. ***Append:*** Write new information at the end of the file.
  5. ***Delete:*** Delete the file and tree its space for possible reuse.
  6. ***List:*** List the name and attributes of the file.

Access Control

- A common approach to protection problem is to make access dependent on the identity of the user.
- Files can be associated with an ACL (access-control list) which specifies username andtypes of access for each user.

Problems:

1. Constructing a list can be tedious.
2. Directory-entry now needs to be of variable-size, resulting in more complicated space management.

Solution:

- These problems can be resolved by combining ACLs with an 'owner, group, universe'access control scheme.
- To reduce the length of the ACL, many systems recognize 3 classifications of users:
  1. ***Owner:*** The user who created the file is the owner.
  2. ***Group:*** A set of users who are sharing the file and need similar access is agroup.
  3. ***Universe:*** All other users in the system constitute the universe.

Other Protection Approaches

- A password can be associated with each file.
- **Disadvantages:**
  - The no. of passwords you need to remember may become large.
  - If only one password is used for all the files, then all files are accessible if it isdiscovered.
  - Commonly, only one password is associated with all of the user's files, soprotection is all-or nothing.
- In a multilevel directory-structure, we need to provide a mechanism for directoryprotection.
- The directory operations that must be protected are different from the File-operations:

- Control creation & deletion of files in a directory.
- Control whether a user can determine the existence of a file in a directory.

IMPLEMENTATION OF FILE SYSTEM

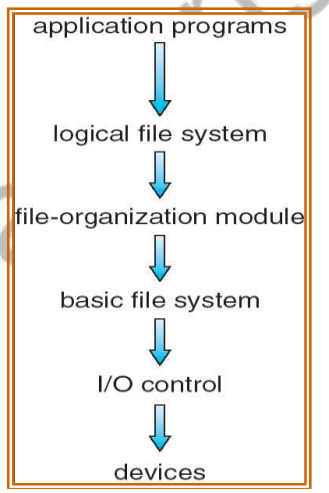
FILE SYSTEM STRUCTURE

- Disks provide the bulk of secondary storage on which a file-system is maintained.

The disk is a suitable medium for storing multiple files.

- This is because of two characteristics
  1. A disk can be rewritten in place; it is possible to read a block from the disk, modify the block, and write it back into the same place.
  2. A disk can access directly any block of information it contains. Thus, it is simple to access any file either sequentially or randomly, and switching from one file to another requires only moving the read-write heads and waiting for the disk to rotate.
- To improve I/O efficiency, I/O transfers between memory and disk are performed in unitsof blocks. Each block has one or more sectors. Depending on the disk drive, sector-size varies from 32 bytes to 4096 bytes. The usual size is 512 bytes.
- File-systems provide efficient and convenient access to the disk by allowing data to be stored, located, and retrieved easily.
- Design problems of file-systems:
  1. Defining how the file-system should look to the user.
  2. Creating algorithms & data-structures to map the logical file-system onto the physical secondary-storage devices.

Layered File Systems:



- The file-system itself is generally composed of many different levels. Every level in design uses features of lower levels to create new features for use by higher levels.
- File system provides efficient and convenient access to the disk by allowing data to be stored, located, and retrieved easily.

A file system poses two quite different design problems.

1. The first problem is defining how the file system should look to the user. This task involves defining a file and its attributes, the operations allowed on a file, and the directory structure for organizing files.
2. The second problem is creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices.

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

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

A device driver can be thought of as a translator. Its input consists of high-level commands such as "retrieve block 123."
Its output consists of low level, hardware-specific instructions that are used by the hardware controller, which interfaces the I/0 device to the rest of the system.

- The device driver usually writes specific bit patterns to special locations in the I/0controller's memory to tell the controller which device location to act on and what actionsto take.
- The **basic file system** needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk. Each physical block is identified by its numeric disk address (for example, drive 1, cylinder 73, track 2, sector10).

This layer also manages the memory buffers and caches that hold various file-system, directory, and data blocks.
A block in the buffer is allocated before the transfer of a disk block can occur. When the bufferis full, the buffer manager must find more buffer memory or free up buffer space to allow a requested I/O to complete.

File organization

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

Logical file system

- Manages metadata information. Metadata includes all of the file- system structure except the actual *data* (or contents of the files). The logical file system manages the directory structure to provide the file organization module with the information the latter needs, given a symbolic file name. It maintains file structure via **file-control blocks (FCB).**
- FCB contains information about the file, including ownership, permissions, and location ofthe file contents.

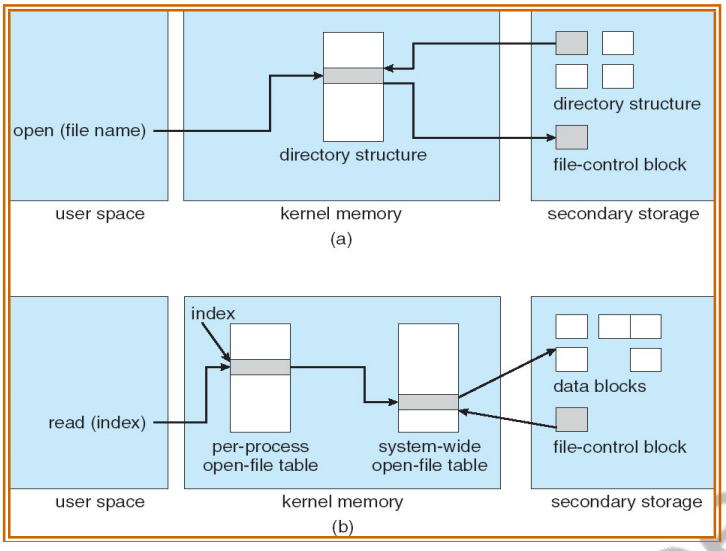
File System Implementation

On disk, the file system may contain information about how to boot an operating system stored there, the total number of blocks, the number and location of free blocks, the directory structure, and individual files.

- **Boot Control Block**: On disk, the file system may contain information about how to boot an operating system stored there, the total number of blocks, the number and location of free blocks, the directory structure, and individual files.
- **Volume Control Block:** (per volume) contains volume (or partition) details, such as the number of blocks in the partition, the size of the blocks, a free-block count and free-blockpointers and a free-FCB count and FCB pointers.
- **A directory structure** (per file system) is used to organize the files.
- **A per-file FCB** contains many details about the file. It has a unique identifier number to allow association with a directory entry.

The in-memory information is used for both file-system management and performance improvement via caching. The data are loaded at mount time, updated during file-system operations, and discarded at dismount. Several types of structures may be included.

- An in-memory mount table contains information about each mounted volume.
- An in-memory directory-structure cache holds the directory information of recentlyaccessed directories.
- The system wide open file table contains a copy of the FCB of each open file, as well asother information.
- The per -process open file table contains a pointer to the appropriate entry in thesystem- wide open-file table, as well as other information.



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

Steps for creating a file:

1) An application program calls the logical file system, which knows the format of thedirectory structures.
2) The logical file system allocates a new file control block (FCB)
   - If all FCBs are created at file-system creation time, an FCB is allocated from the free list.
3) The logical file system then
   - Reads the appropriate directory into memory.
   - Updates the directory with the new file name and FCB.
   - Writes the directory back to the disk.

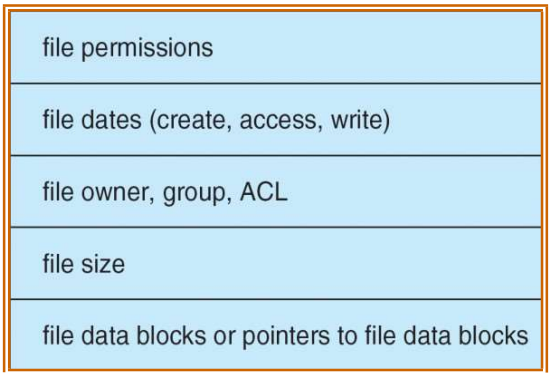
UNIX treats a directory exactly the same as a file by means of a type of field in the i node Windows NT implements separate system calls for files and directories and treats directories asentities separate from files.

Steps for opening a file:

1) The function first searches the system-wide open-file table to see if the file is already inuse by another process.
   - If it is, a per-process open-file table entry is created pointing to the existingsystem-wide open-file table.
   - This algorithm can have substantial overhead; consequently, parts of the directorystructure are usually cached in memory to speed operations.
2) Once the file is found, the FCB is copied into a system-wide open-file table in memory.
   - This table also tracks the number of processes that have the file open.
3) Next, an entry is made in the per-process open-file table, with a pointer to the entry in thesystem-wide open-file table.
4) The function then returns a pointer/index to the appropriate entry in the per-process file-system table
   - All subsequent file operations are then performed via this pointer.
   - UNIX refers to this pointer as the file descriptor.
   - Windows refers to it as the file handle Steps for closing a file:

1) The per-process table entry is removed.
2) The system-wide entry's open count is decremented.
3) When all processes that have opened the file eventually close it

Any updated metadata is copied back to the disk-based directory structure. The system-wide open- file table entry is removed.



Partitions and Mounting

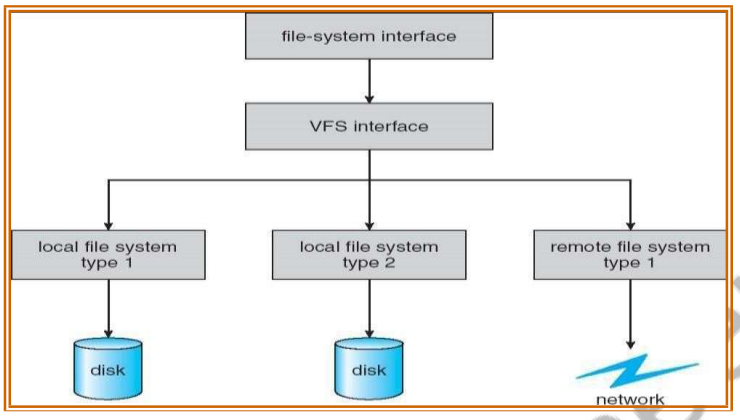
- Each partition can be either "raw," containing no file system, or "cooked," containing a file system.
- Raw disk is used where no file system is appropriate.
- UNIX swap space can use a raw partition, for example, as it uses its own format on disk and does not use a file system.
- Boot information can be stored in a separate partition. Again, it has its own format, because at boot time the system does not have the file-system code loaded and therefore cannot interpret the file-system format.
- Boot information is a sequential series of blocks, loaded as an image into memory.
- Execution of the image starts at a predefined location, such as the first byte. This boot loader in turn knows about the file-system structure to be able to find and load the kernel and start it executing.
- The root partition which contains the operating-system kernel and sometimes other system files, is mounted at boot time.
- Other volumes can be automatically mounted at boot or manually mounted later,depending on the operating system.
- After successful mount operation, the operating system verifies that the device contains avalid file system.
- It is done by asking the device driver to read the device directory and verifying that thedirectory has the expected format.
- If the format is invalid, the partition must have its consistency checked and possiblycorrected, either with or without user intervention.
- Finally, the operating system notes in its in-memory mount table that a file system ismounted, along with the type of the file system.
- The root partition is mounted at boot time.
  - It contains the operating-system kernel and possibly other system files.
- Other volumes can be automatically mounted at boot time or manually mounted later.
- As part of a successful mount operation, the operating system verifies that the storagedevice contains a valid file system.
  - It asks the device driver to read the device directory and verify that the directory hasthe expected format.
  - If the format is invalid, the partition must have its consistency checked and possibly corrected.
  - Finally, the operating system notes in its in-memory mount table structure that a filesystem is mounted along with the type of the file system.

Virtual file Systems

The file-system implementation consists of three major layers, as depicted schematically inFigure. The first layer is the file-system interface, based on the open(), read(), write(), and close() calls and on file descriptors. The second layer is called the virtual file system (vfs) layer. The VFS layer serves two important functions:

- It separates file-system-generic operations from their implementation by defining a clean VFS interface. Several implementations for the VFS interface may coexist on the same machine, allowing transparent access to different types of file systems mounted locally.

- It provides a mechanism for uniquely representing a file throughout a network. The VFS is based on a file-representation structure, called a vnode that contains a numerical designatorfor a network-wide unique file. This network-wide uniqueness is required for support of network file systems.
- The kernel maintains one vnode structure for each active node.
- Thus, the VFS distinguishes local files from remote ones, and local files are furtherdistinguished according to their file-system types.



Directory Implementation

Selection of directory allocation and directory management algorithms significantly affects theefficiency, performance, and reliability of the file system.

One Approach: Direct indexing of a linear list

- Consists of a list of file names with pointers to the data blocks
- Simple to program
- Time-consuming to search because it is a linear search.
- Sorting the list allows for a binary search; however, this may complicate creating and deleting files.
- To create a new file, we must first search the directory to be sure that no existing file hasthe same name.
- Add a new entry at the end of the directory.
- To delete a file, we search the directory for the named file and then release the space allocated to it.
- To reuse the directory entry, we can do one of several things. Mark the entry as unused.
- An alternative is to copy the last entry in the directory into the freed location and to decrease the length of the directory.
- Directory information is used frequently, and users will notice if access to it is slow.

Another Approach: List indexing via a hash function

- Takes a value computed from the file name and returns a pointer to the file name in thelinear list.
- Greatly reduces the directory search time.
- **Can result in collisions** – situations where two file names hash to the same location.
- A hash table are its generally fixed size and the dependence of the hash function on thatsize. (i.e., fixed number of entries).
- Each hash entry can be a linked list instead of an individual value, and we can resolve collisions by adding the new entry to the linked list.

ALLOCATION METHODS

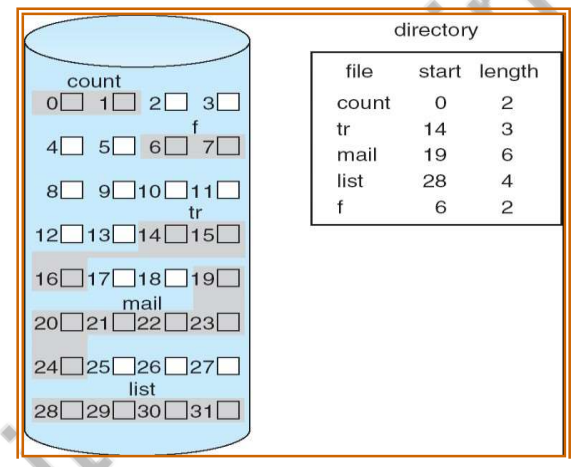
Allocation methods address the problem of allocating space to files so that disk space is utilized effectively, and files can be accessed quickly.

Three methods exist for allocating disk space.

- Contiguous allocation
- Linked allocation
- Indexed allocation

Contiguous allocation:

- Requires that each file occupies a set of contiguous blocks on the disk.
- Accessing a file is easy – only need the starting location (block #) and length (number ofblocks)
- Contiguous allocation of a file is defined by the disk address and length (in block units) ofthe first block. If the file is *n* blocks long and starts at location *b,* then it occupies blocks *b,b* + 1, *b* + 2, ... ,*b* + *n* - 1. The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file.
- Accessing a file that has been allocated contiguously is easy. For sequential access, thefile system remembers the disk address of the last block referenced and when necessary, reads the next block. For direct access to block *i* of a file that starts at block *b,* we can immediately access block *b* + *i.* Thus, both sequential and direct access can be supported by contiguous allocation.



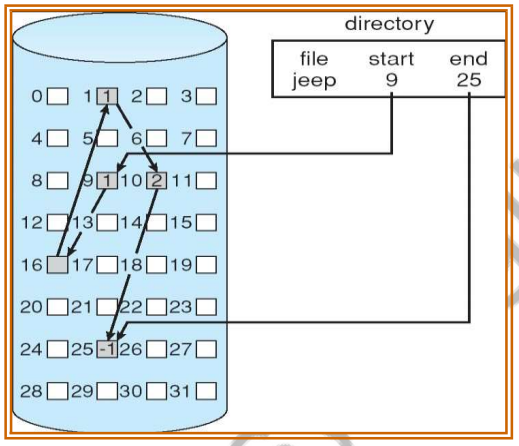
Disadvantages:

1. Finding space for a new file is difficult. The system chosen to manage free space determines how this task is accomplished. Any management system can be used, but some are slower than others.
2. Satisfying a request of size *n* from a list of free holes is a problem. First fit and best fit are the most common strategies used to select a free hole from the set of available holes.
3. The above algorithms suffer from the problem of external fragmentation.
   - As files are allocated and deleted, the free disk space is broken into pieces.
   - External fragmentation exists whenever free space is broken into chunks.
   - It becomes a problem when the largest contiguous chunk is insufficient for a request; storage is fragmented into a number of holes, none of which is large enough to store the data.
   - Depending on the total amount of disk storage and the average file size, external fragmentation may be a minor or a major problem.

- Solves the problems of contiguous allocation.
- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk.
- The directory contains a pointer to the first and last blocks of a file.
- Creating a new file requires only the creation of a new entry in the directory.
- Writing to a file causes the free-space management system to find a free block.
  - ➢ This new block is written to and is linked to the end of the file.
  - ➢ Reading from a file requires only reading blocks by following the pointers from block toblock.

Advantages

- There is no external fragmentation.
- Any free blocks on the free list can be used to satisfy a request for disk space.
- The size of a file need not be declared when the file is created.
- A file can continue to grow as long as free blocks are available.
- It is never necessary to compact disk space for the sake of linked allocation(however, file access efficiency may require it)



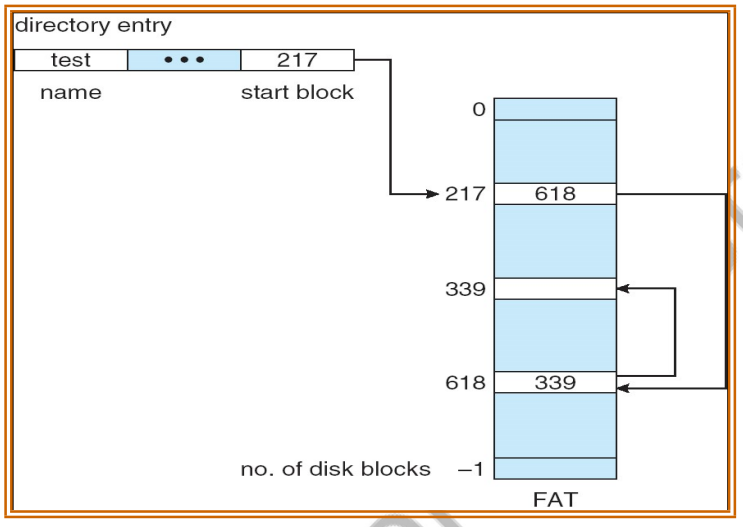
- Each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file.
- For example, a file of five blocks might start at block 9 and continue at block 16, then block 1, then block 10, and finally block 25. Each block contains a pointer to the next block. These pointers are not made available to the user. A disk address (the pointer) requires 4 bytes in the disk.
- To **create** a new file, we simply create a new entry file in the directory. With linked allocation, each directory entry has a pointer to the first disk block of the file. This pointeris initialized to *nil* (the end-of-list pointer value) to signify an empty file. The size field is also set to 0.
- A **write** to the file causes the free-space management system to find a free block, and this new block is written to and is linked to the end of the file.
- To **read** a file, we simply read blocks by following the pointers from block to block. There is no external fragmentation with linked allocation, and any free block on the free- space list can be used to satisfy a request. The size of a file need not be declared when that file is created.
- A file can continue to grow as long as free blocks are available. Consequently, it is never necessary to compact disk space.

a) **Disadvantages:**

1. The major problem is that it can be used effectively only for sequential-access files. To file the i^{th} block of a file, we must start at the beginning of that file and follow the pointers until we get to the i^{th} block.
2. Space required for the pointers. Solution is clusters. Collect blocks into multiplesand allocate clusters rather than blocks.
3. Reliability - the files are linked together by pointers scattered all over the disk and if a pointer were lost or damaged then all the links are lost.

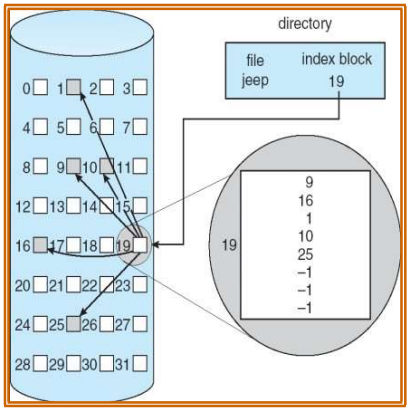
File Allocation Table:

- A section of disk at the beginning of each volume is set aside to contain the table. Thetable has one entry for each disk block and is indexed by block number.
- The FAT is used in much the same way as a linked list. The directory entry contains theblock number of the first block of the file.
- The table entry indexed by that block number contains the block number of the next blockin the file.
- The chain continues until it reaches the last block, which has a special end-of-file value asthe table entry.
- An unused block is indicated by a table value of 0.
- Consider a FAT with a file consisting of disk blocks 217, 618, and 339.



Indexed allocation:

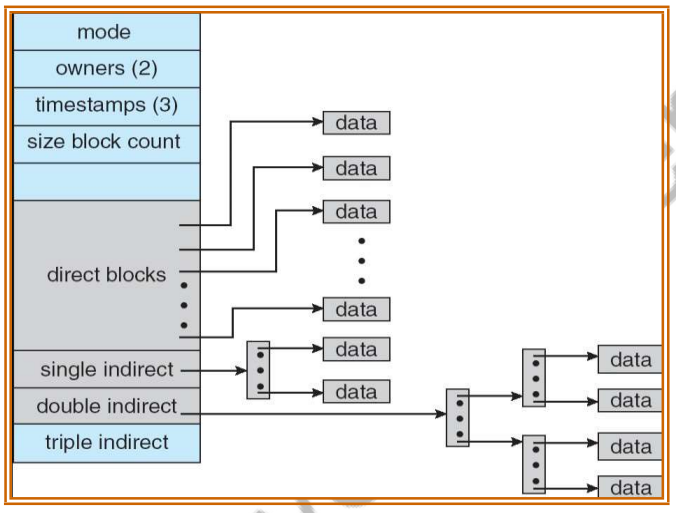
- Brings all the pointers together into one location called an index block.
- Each file has its own index block, which is an array of disk-block addresses.
- The i^{th} entry in the index block points to the i^{th} block of the file. The directory contains the address of the index block. To find and read the i^{th} block, we use the pointer in the i^{th} index- block entry.
- When the file is created, all pointers in the index block are set to *nil*. When the i^{th} block is first written, a block is obtained from the free-space manager and its address is put in the i^{th} index- block entry.
- Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk can satisfy a request for more space.
- **Disadvantages:**
  - Suffers from some of the same performance problems as linked allocation.
  - Index blocks can be cached in memory; however, data blocks may be spread allover the disk volume.
  - Indexed allocation does suffer from wasted space.
  - The pointer overhead of the index block is generally greater than the pointeroverhead of linked allocation.



If the index block is too small, however, it will not be able to hold enough pointers for a large file,and a mechanism will have to be available to deal with this issue. Mechanisms for this purpose include the following:

a) **Linked scheme.** An index block is normally one disk block. Thus, it can be read and written directly by itself. To allow for large files, we can link together several index blocks. For example, an index block might contain a small header giving the name of the file and a set of the first 100 disk-block addresses. The next address (the last word in the index block) is *nil* (for a small file) or is a pointer to another index block (for a large file).

b) **Multilevel index.** A variant of linked representation uses a first-level index block to point to a set of second-level index blocks, which in turn point to the file blocks. To access a block, the operating system uses the first-level index to find a second-level index block and then uses that block to find the desired data block. This approach could be continued to a third or fourth level, depending on the desired maximum file size.



c) **Combined scheme.** For eg. 15 pointers of the index block is maintained in the file's i node. The first 12 of these pointers point to **direct blocks**; that is, they contain addresses of blocks that contain data of the file. Thus, the data for small files (of no more than 12 blocks) do not need a separate index block. If the block size is 4 KB, then up to 48 KB of data can be accessed directly. The next three pointers point to indirect blocks. The first points to a **single indirect block**, which is an index block containing not data but the addresses of blocks that do contain data. The second points to a **double indirect block**, which contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks. The last pointer contains the address of a **triple indirect block**.

Performance

- **Contiguous allocation** requires only one access to get a disk block. Since we can easilykeep the initial address of the file in memory, we can calculate immediately the disk address of the i^{th} block and read it directly.
- For **linked allocation**, we can also keep the address of the next block in memory andread it directly. This method is fine for sequential access. Linked allocation should not be used for an application requiring direct access.
- **Indexed allocation** is more complex. If the index block is already in memory, then the access can be made directly. However, keeping the index block in memory requires considerable space. If this memory space is not available, then we may have to read first the index block and then the desired data block.

Free Space Management

The space created after deleting the files can be reused. Another important aspect of disk management is keeping track of free space in memory. The list which keeps track of free space in memory is called the free-space list. To create a file, search the free-space list for the required amount of space and allocate that space to the new file. This space is then removed from the free- space list. When a file is deleted, its disk space is added to the free-space list. The free-space list, isimplemented in different ways as explained below.

b) **Bit Vector**

- Fast algorithms exist for quickly finding contiguous blocks of a given size.
- One simple approach is to use a ***bit vector***, in which each bit represents a disk block, set to 1 if free or 0 if allocated.

For example, consider a disk where blocks 2,3,4,5,8,9, 10,11, 12, 13, 17and 18 are free, and therest of the blocks are allocated. The free-space bit map would be.
0011110011111100011

- Easy to implement and also very efficient in finding the first free block or 'n'. consecutive free blocks on the disk.
- The downside is that a 40GB disk requires over 5MB just to store the bitmap.

c) **Linked List**

a. A linked list can also be used to keep track of all free blocks.
b. Traversing the list and/or finding a contiguous block of a given size are not easy, butfortunately are not frequently needed operations. Generally, the system just adds and removes single blocks from the beginning of the list.
c. The FAT table keeps track of the free list as just one more linked list on the table.

d) **Grouping**

a. A variation on linked list free lists. It stores the addresses of n free blocks in the first free block. The first n-1 blocks are actually free. The last block contains the addresses of another n free blocks, and so on.
b. The address of a large number of free blocks can be found quickly.

e) **Counting**

a. When there are multiple contiguous blocks of free space then the system can keeptrack of the starting address of the group and the number of contiguous free blocks.
b. Rather than keeping all list of n free disk addresses, we can keep the address of firstfree block and the number of free contiguous blocks that follow the first block.
c. Thus, the overall space is shortened. It is similar to the extent method of allocatingblocks.

f) **Space Maps**

a. Sun's ZFS file system was designed for huge numbers and sizes of files, directories, and even file systems.
b. The resulting data structures could be inefficient if not implemented carefully. For example, freeing up a 1 GB file on a 1 TB file system could involve updating thousands of blocks of free list bit maps if the file was spread across the disk.
c. ZFS uses a combination of techniques, starting with dividing the disk up into (hundreds of) ***Meta slabs*** of a manageable size, each having their own space map.
d. Free blocks are managed using the counting technique, but rather than write the information to a table, it is recorded in a log-structured transaction record. Adjacent free blocks are also coalesced into a larger single free block.
e. An in-memory space map is constructed using a balanced tree data structure, constructed from the log data.
f. The combination of the in-memory tree and the on-disk log provides for very fast andefficient management of these very large files and free blocks.

SECONDARY STORAGE STRUCTURES

OVERVIEW OF MASS-STORAGE STRUCTURE

Web link- https://youtu.be/ZjMwUhapSEM

Magnetic Disks

- Magnetic disks provide the bulk of secondary storage for modern computer systems.
- Each disk platter has a flat circular shape, like a CD. Common platter diameters rangefrom 1.8 to 5.25 inches.
- The two surfaces of a platter are covered with a magnetic material. The informationstored by recording it magnetically on the platters.

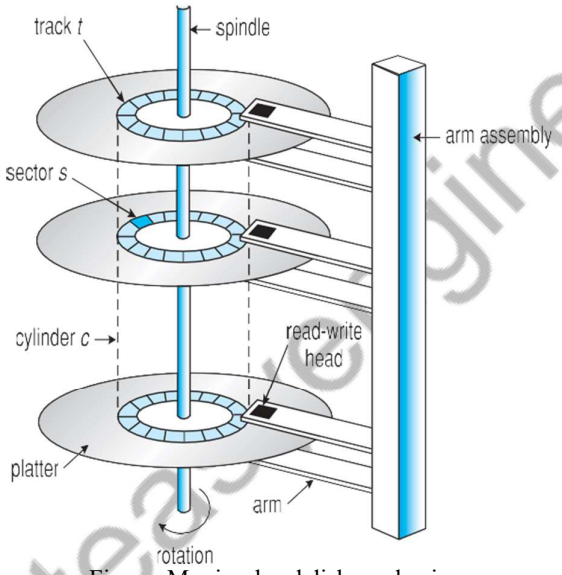


Figure: Moving-head disk mechanism

- The surface of a platter is logically divided into circular tracks, which are subdivided into sectors. Sector is the basic unit of storage. The set of tracks that are at one arm position makes up a cylinder.
- The number of cylinders in the disk drive equals the number of tracks in each platter.
- There may be thousands of concentric cylinders in a disk drive, and each track may contain hundreds of sectors.
    - **Seek Time:-**Seek time is the time required to move the disk arm to the required track.
    - **Rotational Latency (Rotational Delay):-** Rotational latency is the time taken for the disk to rotate so that the required sector comes under the r/w head.
    - **Positioning time or random access time** is the summation of seek time and rotational delay.
    - **Disk Bandwidth:-** Disk bandwidth is the total number of bytes transferred dividedby total time between the first request for service and the completion of last transfer.
    - **Transfer rate** is the rate at which data flow between the drive and the computer.

As the disk head flies on an extremely thin cushion of air, the head will make contact with the disk surface. Although the disk platters are coated with a thin protective layer, sometimes the head will damage the magnetic surface. This accident is called a **head crash**.

Magnetic Tapes

- Magnetic tape is a secondary-storage medium. It is a permanent memory and can hold large quantities of data.
- The time taken to access data (access time) is large compared with that of magnetic disk, because here data is accessed sequentially.
- When the n^{th} data has to be read, the tape starts moving from first and reaches the n^{th} position and then data is read from n^{th} position. It is not possible to directly move to the n^{th} position. So tapes are used mainly for backup, for storage of infrequently used information.
- A tape is kept in a spool and is wound or rewound past a read-write head. Moving to the correct spot on a tape can take minutes, but once positioned, tape drives can write data at speeds comparable to disk drives.

DISK STRUCTURE

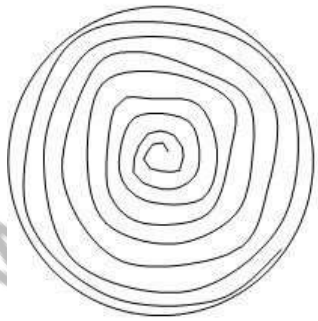
- Modern disk drives are addressed as a large one-dimensional array. The one- dimensional array of logical blocks is mapped onto the sectors of the disk sequentially.
- Sector 0 is the first sector of the first track on the outermost cylinder. The mapping proceeds in order through that track, then through the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

The disk structure (architecture) can be of two types –

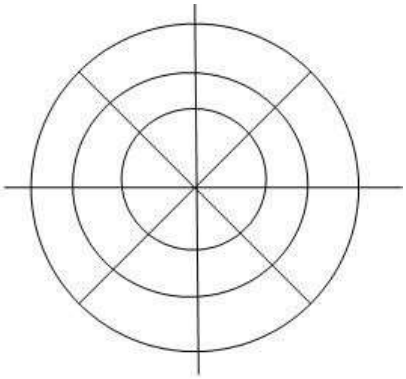
1. Constant Linear Velocity (CLV)
2. Constant Angular Velocity (CAV)

1. **CLV** – The density of bits per track is uniform. The farther a track is from the center of the disk, the greater its length, so the more sectors it can hold. As we move from outer zones to inner zones, the number of sectors per track decreases. This architecture is used in CD-ROM and DVD-ROM.

2. **CAV** – There is same number of sectors in each track. The sectors are densely packed inthe inner tracks. The density of bits decreases from inner tracks to outer tracks to keep the data rate constant.



CLV



CAV

DISK ATTACHMENT

Computers can access data in two ways.

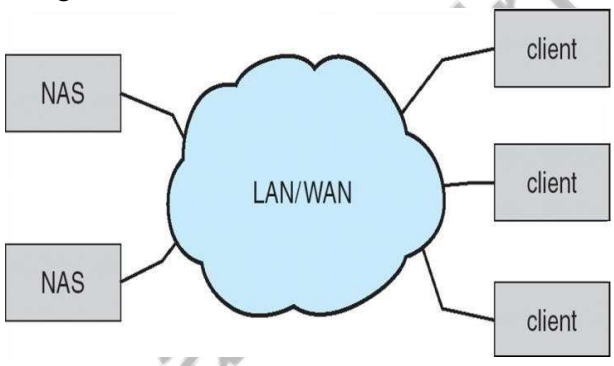
1. via I/O ports (or host-attached storage)
2. via a remote host in a distributed file system (or network-attached storage)

1. **Host-Attached Storage:**

- Host-attached storage is storage accessed through local I/O ports.
- Example: the typical desktop PC uses an I/O bus architecture called IDE or ATA. This architecture supports a maximum of two drives per I/O bus.
- The other cabling systems are – SATA (Serially Attached Technology Attachment), SCSI (Small Computer System Interface) and fiber channel (FC).
- SCSI is a bus architecture. Its physical medium is usually a ribbon cable. FC is a high- speed serial architecture that can operate over optical fiber or over a four- conductor copper cable. An improved version of this architecture is the basis of storage-area networks (SANs).

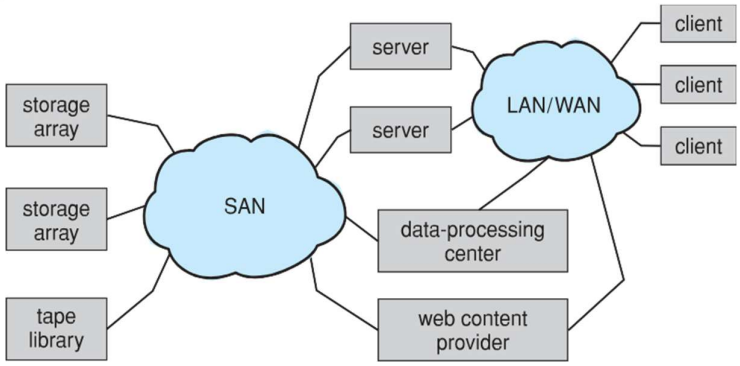
2. Network-Attached Storage

- A network-attached storage (NAS) device is a special-purpose storage system that is accessed remotely over a network as shown in the figure.
- Clients access network-attached storage via a remote-procedure-call interface. The remote procedure calls (RPCs) are carried via TCP or UDP over an IP network usually the same local-area network (LAN) carries all data traffic to the clients.
- Network- attached storage provides a convenient way for all the computers on aLAN to share a pool of storage files.



3. Storage Area Network (SAN)

- A storage-area network (SAN) is a private network connecting servers and storageunits.
- The power of a SAN lies in its flexibility. Multiple hosts and multiple storage arrayscan attach to the same SAN, and storage can be dynamically allocated to hosts.
- A SAN switch allows or prohibits access between the hosts and the storage. Fiber Chanel is the most common SAN interconnect.



DISK SCHEDULING

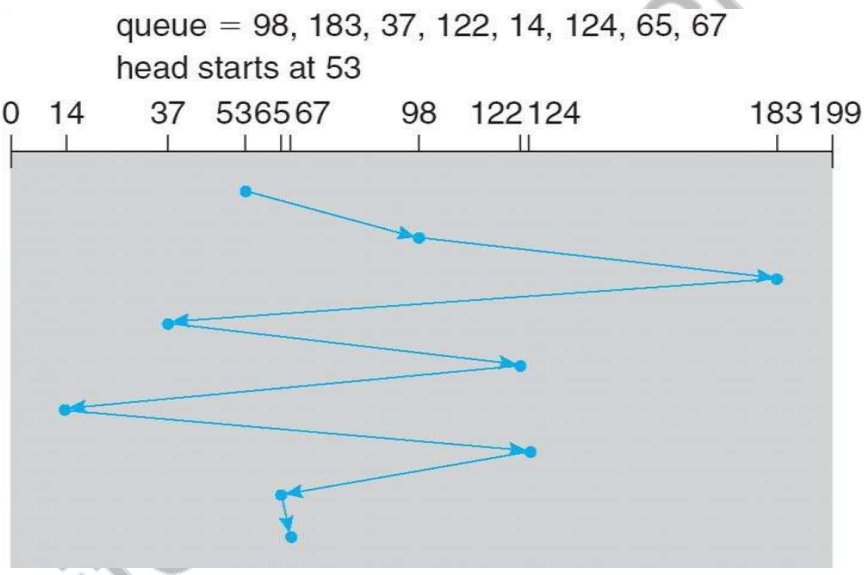
Different types of disk scheduling algorithms are as follows:

1. FCFS (First Come First Serve)
2. SSTF (Shortest Seek Time First)
3. SCAN (Elevator)
4. C-SCAN
5. LOOK
6. C-LOOK

1. **FCFS scheduling algorithm:**

This is the simplest form of disk scheduling algorithm. This services the request in the orderthey are received. This algorithm is fair but do not provide fastest service. It takes no specialcare to minimize the overall seek time.

Eg:- consider a disk queue with request for i/o to blocks on cylinders. 98, 183, 37, 122, 14, 124, 65, 67

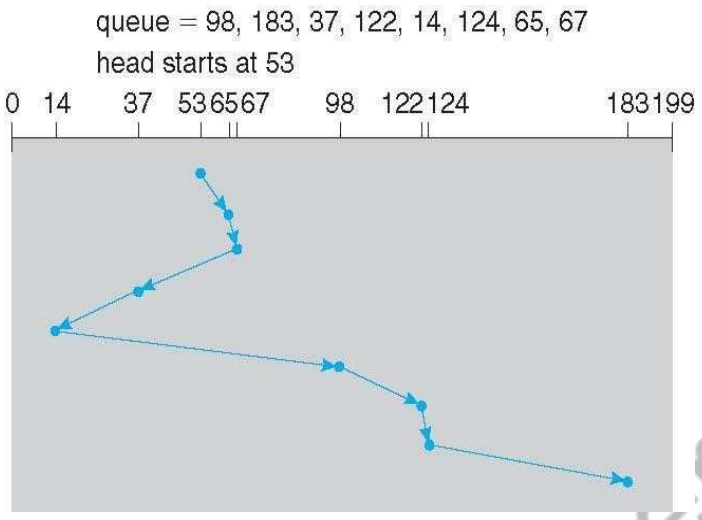


If the disk head is initially at 53, it will first move from 53 to 98 then to 183 and then to 37, 122, 14, 124, 65, 67 for a total head movement of 640 cylinders. The wild swing from 122to 14 and then back to 124 illustrates the problem with this schedule.

2. **SSTF (Shortest Seek Time First) algorithm**:

This selects the request with minimum seek time from the current head position. SSTF chooses the pending request closest to the current head position.

Eg:- consider a disk queue with request for i/o to blocks on cylinders. 98, 183, 37, 122, 14, 124, 65, 67

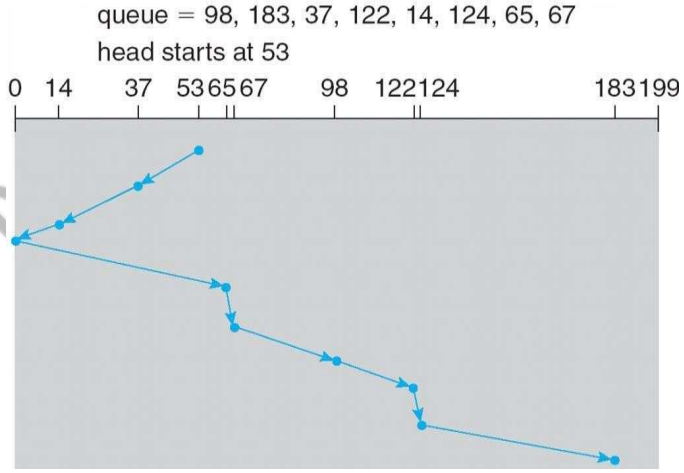


If the disk head is initially at 53, the closest is at cylinder 65, then 67, then 37 is closer than 98 to 67. So it services 37, continuing we service 14, 98, 122, 124 and finally 183. The total head movement is only 236 cylinders. SSTF is a substantial improvement over FCFS, it is not optimal.

3. **SCAN algorithm:**

In this the disk arm starts moving towards one end, servicing the request as it reaches each cylinder until it gets to the other end of the disk. At the other end, the direction of the head movement is reversed and servicing continues. The initial direction is chosen depending upon the direction of the head.

Eg:- consider a disk queue with request for i/o to blocks on cylinders. 98, 183, 37, 122, 14, 124, 65, 67



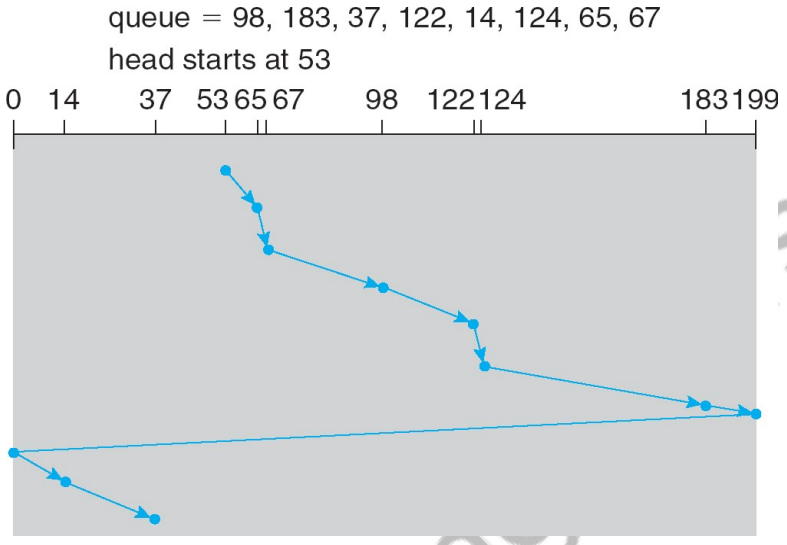
If the disk head is initially at 53 and if the head is moving towards the outer track, itservices 65, 67, 98, 122, 124 and 183. At cylinder 199 the arm will reverse and will move towards the other end of the disk servicing 37 and then 14. The SCAN is also called aselevator algorithm

4. **C-SCAN (Circular scan) algorithm**:

C-SCAN is a variant of SCAN designed to provide a more uniform wait time.

Like SCAN, C-SCAN moves the head from end of the disk to the other servicing the request along the way. When the head reaches the other end, it immediately returns to the beginning of the disk, without servicing any request on the return.

Eg:- consider a disk queue with request for i/o to blocks on cylinders. 98, 183, 37, 122, 14, 124, 65, 67.



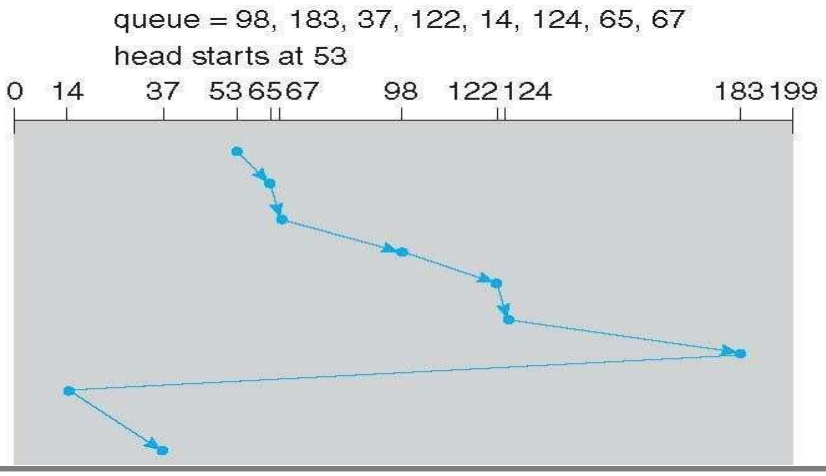
If the disk head is initially at 53 and if the head is moving towards the outer track, it services 65, 67, 98, 122, 124 and 183. At cylinder 199 the arm will reverse and will move immediately towards the other end of the disk, then changes the direction of head and serves14 and then 37.

Note: If the disk head is initially at 53 and if the head is moving towards track 0, it services 37 and 14 first. At cylinder 0 the arm will reverse and will move immediately towards the other end of the disk servicing 65, 67, 98, 122, 124 and 183.

5. **Look Scheduling algorithm:**

Look and C-Look scheduling are different version of SCAN and C-SCAN respectively. Here the arm goes only as far as the final request in each direction. Then it reverses, withoutgoing all the way to the end of the disk. The Look and C-Look scheduling look for a requestbefore continuing to move in a given direction.

Eg:- consider a disk queue with request for i/o to blocks on cylinders. 98, 183, 37, 122, 14, 124, 65, 67



If the disk head is initially at 53 and if the head is moving towards the outer track, itservices 65, 67, 98, 122, 124 and 183. At the final request 183, the arm will reverse and willmove towards the first request 14 and then serves 37.

SELECTION OF A DISK-SCHEDULING ALGORITHM

- SSTF is commonly used and it increases performance over FCFS.
- SCAN and C-SCAN algorithm is better for a heavy load on disk. SCAN and C-SCANhave less starvation problem.
- SSTF or Look is a reasonable choice for a default algorithm.
- Selection of disk scheduling algorithm is influenced by the file allocation method, if contiguous file allocation is chosen, then FCFS is best suitable, because the files are stored in contiguous blocks and there will be limited head movements required.
- A linked or indexed file may include blocks that are widely scattered on the disk, resulting in greater head movement.
- The location of directories and index blocks is also important. Since every file must be opened to be used, and opening a file requires searching the directory structure, the directories will be accessed frequently.
- Suppose that a directory entry is on the first cylinder and a file's data are on the final cylinder. The disk head has to move the entire width of the disk. If the directory entry were on the middle cylinder, the head would have to move, at most, one-half the width. Caching the directories and index blocks in main memory can also help to reduce the disk-arm movement, particularly for read requests.

DISK MANAGEMENT

Disk Formatting

- The process of dividing the disk into sectors and filling the disk with a special data structure is called low-level formatting. Sector is the smallest unit of area that is read / written by the disk controller. The data structure for a sector typically consists of a header, a data area (usually 512 bytes in size) and a trailer. The header and trailer contain information used by the disk controller, such as a sector number and an error- correcting code (ECC).
- When the controller writes a sector of data during normal I/O, the ECC is updated with avalue calculated from all the bytes in the data area. When a sector is read, the ECC is recalculated and is compared with the stored value. If the stored and calculated numbers are different, this mismatch indicates that the data area of the sector has become corrupted and that the disk sector may be bad.
- Most hard disks are low-level- formatted at the factory as a part of the manufacturing process. This formatting enables the manufacturer to test the disk and to initialize the mapping from logical block numbers to defect-free sectors on the disk.
- When the disk controller is instructed for low-level-formatting of the disk, the size of data block of all sector sit can also be told how many bytes of data space to leave between the header and trailer of all sectors. It is of sizes, such as 256, 512, and 1,024 bytes. Formatting a disk with a larger sector size means that fewer sectors can fit on eachtrack; but it also means that fewer headers and trailers are written on each track and more space is available for user data.

The operating system needs to record its own data structures on the disk. It does so in two steps i.e., Partition and logical formatting.

1. **Partition** – is to partition the disk into one or more groups of cylinders. The operating system can treat each partition as though it were a separate disk. For instance, one partition can hold a copy of the operating system's executable code, while another holds user files.
2. **Logical formatting (or creation of a file system)** - Now, the operating system stores the initial file-system data structures onto the disk. These data structures may include maps of free and allocated space (a FAT or modes) and an initial empty directory.

To increase efficiency, most file systems group blocks together into larger chunks, frequentlycalled **clusters.**

Boot Block

When a computer is switched on or rebooted, it must have an initial program to run. This iscalled the bootstrap program.

The bootstrap program –

- Initializes the CPU registers, device controllers, main memory, and then starts theoperating system.
- Locates and loads the operating system from the disk
- Jumps to beginning the operating-system execution.

The bootstrap is stored in read-only memory (ROM). Since ROM is read only, it cannot be infected by a computer virus. The problem is that changing this bootstrap code requires changing the ROM, hardware chips. So most systems store a tiny bootstrap loader program in the boot ROM whose only job is to bring in a full bootstrap program from disk. The full bootstrap program can be changed easily: A new version is simply written onto the disk. The full bootstrap program is stored in "the boot blocks" at a fixed location on the disk. A disk that has a boot partition is called a boot disk or system disk.

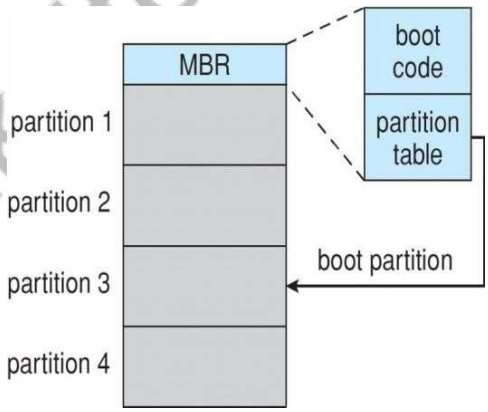


Figure: Booting from disk in Windows 2000.

The Windows 2000 system places its boot code in the first sector on the hard disk (master boot record, or MBR). The code directs the system to read the boot code from, the MBR. In addition to containing boot code, the MBR contains a table listing the partitions for the hard disk and a flag indicating which partition the system is to be booted from.

Bad Blocks

Disks are prone to failure of sectors due to the fast movement of r/w head. Sometimes thewhole disk will be changed. Such group of sectors that are defective are called as **bad blocks**.

Different ways to overcome bad blocks are -

- Some bad blocks are handled manually, eg. In MS-DOS.
- Some controllers replace each bad sector logically with one of the spare sectors (extra sectors). The schemes used are sector sparing or forwarding and sector slipping.

In MS-DOS format command, scans the disk to find bad blocks. If format finds a bad block, it writes a special value into the corresponding FAT entry to tell the allocation routines not to use that block.
In SCSI disks, bad blocks are found during the low-level formatting at the factory and is updated over the life of the disk. Low-level formatting also sets aside spare sectors not visibleto the operating system. The controller can be told to replace each bad sector logically with one of the spare sectors. This scheme is known as **sector sparing or forwarding**.

A typical bad-sector transaction might be as follows:

- The operating system tries to read logical block 87.
- The controller finds that the sector is bad. It reports this finding to the operating system.
- The next time the system is rebooted, a special, command is run to tell the SCSI controller to replace the bad sector with a spare.
- After that, whenever the system requests logical block 87, the request is translated intothe replacement sector's (spare) address by the controller.

Some controllers replace bad blocks by sector slipping.

Example: Suppose that logical block 17 becomes defective and the first available spare followssector 202. Then, sector slipping remaps all the sectors from 17 to 202, moving them all down one spot. That is, sector 202 is copied into the spare, then sector 201 into 202, and then 200 into201, and so on, until sector 18 is copied into sector 19. Slipping the sectors in this way frees up the space of sector 18, so sector 17 can be mapped to it.

SWAP-SPACE MANAGEMENT

- Swap-space management is another low-level task of the operating system.
- Swapping occurs when the amount of physical memory reaches a critically low pointand processes are moved from memory to swap space to free available memory.

Swap-Space Use

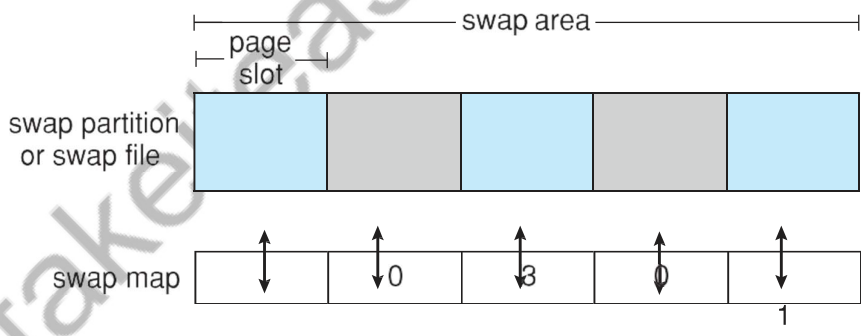
- The amount of swap space needed on a system can vary depending on the amount of physical memory, the amount of virtual memory it is backing, and the way in which the virtual memory is used. It can range from a few megabytes of disk space to gigabytes.
- The swap space can overestimate or underestimated. It is safer to overestimate than to underestimate the amount of swap space required. If a system runs out of swap spacedue to underestimation of space, it may be forced to abort processes or may crash entirely. Overestimation wastes disk space that could otherwise be used for files, but it does no other harm.

Swap-Space Location

- A swap space can reside in one of two places: It can be carved out of the normal file system, or it can be in a separate disk partition. If the swap space is simply a large file within the file system, normal file-system routines can be used to create it, name it, and allocate its space.
- External fragmentation can greatly increase swapping times by forcing multiple seeks during reading or writing of a process image. We can improve performance by caching the block location information in physical memory.
- Alternatively, swap space can be created in a separate raw partition. A separate swap- space storage manager is used to allocate and deallocate the blocks from the rawpartition.

Swap-Space Management: An Example

- Solaris allocates swap space only when a page is forced out of physical memory, rather than when the virtual memory page is first created.
- Linux is similar to Solaris in that swap space is only used for anonymous memory or for regions of memory shared by several processes. Linux allows one or more swap areas to be established.
- A swap area may be in either a swap file on a regular file system or a raw swap partition.Each swap area consists of a series of 4-KB page slots, which are used to hold swapped pages. Associated with each swap area is a swap map—an array of integer counters, each corresponding to a page slot in the swap area.
- If the value of a counter is 0, the corresponding page slot is available. Values greaterthan 0 indicate that the page slot is occupied by a swapped page. The value of the counter indicates the number of mappings to the swapped page; for example, a value of3 indicates that the swapped page is mapped to three different processes.
- The data structures for swapping on Linux systems are shown in below figure.



PROTECTION

Web Link: https://youtu.be/O_WbprDZMDw

GOALS OF PROTECTION

- Protection is a mechanism for controlling the access of programs, processes, or users to the resources defined by a computer system. Protection ensures that only processes that have gained proper authorization from the operating system can operate on the files, memory segments, CPU, and other resources of a system.
- Protection is required to prevent mischievous, intentional violation of an access restriction by a user.

PRINCIPLES OF PROTECTION

- A key, time-tested guiding principle for protection is the 'principle of least privilege'. It dictates that programs, users, and even systems be given just enough privileges toperform their tasks.
- An operating system provides mechanisms to enable privileges when they are neededand to disable them when they are not needed.

DOMAIN OF PROTECTION

- A computer system is a collection of processes and objects. Objects are both hardware objects (such as the CPU, memory segments, printers, disks, and tape drives) andsoftware objects (such as files, programs, and semaphores). Each object (resource) has a unique name that differentiates it from all other objects in the system.
- The operations that are possible may depend on the object. For example, a CPU can onlybe executed on. Memory segments can be read and written, whereas a CD-ROM or DVD-ROM can only be read. Tape drives can be read, written, and rewound. Data files can be created, opened, read, written, closed, and deleted; program files can be read, written, executed, and deleted.
- A process should be allowed to access only those resources for which it has authorization and currently requires to complete process

Domain Structure

- A domain is a set of objects and types of access to these objects. Each domain is an ordered pair of <object-name, rights-set>.
- Example, if domain D has the access right <file F,{read,write}>, then all process executing in domain D can both read and write file F, and cannot perform any other operation on that object.
- Domains do not need to be disjoint; they may share access rights. For example, in below figure, we have three domains: D_1 D_2, and D_3. The access right < O_4, (print}> is shared by D_2 and D_3,it implies that a process executing in either of these two domains can print object O4.
- A domain can be realized in different ways, it can be a user, process or a procedure. ie. each user as a domain, each process as a domain or each procedure as a domain.