Subject Title: Operating Systems Common to CSE&IT Branches					
Course Code: 19ITT222	Year and Semester: II Year II semester	L	T	P	C
Prerequisites: Knowledge on basic components of Computer System Organization and Data structures		3	0	0	3

Course Objectives:

1. Study the basic concepts and functions of operating system

2. Learn about Processes, Threads and Scheduling algorithms

3. Understand the principles of concurrency and Deadlocks

4. Learn various memory management schemes

5. Study I/O management and File systems

UNIT–I: Introduction to Operating System Concepts
10Hours

What Operating Systems do, Computer System Organization, Functions of Operating systems, Types of Operating Systems, Operating Systems services, System calls, Types of System calls, Operating System Structures, Distributed Systems, Special purpose systems.

UNIT– II: Process Management **12Hours**

Process concept, Process State Diagram, Process control block, Process Scheduling- Scheduling Queues, Schedulers, Scheduling Criteria, Scheduling algorithm's and their evaluation, Operations on Processes, Interprocess Communication.

Threads - Overview, User and Kernel threads, Multi-threading Models

UNIT – III: Concurrency **12Hours**

Process Synchronization, The Critical- Section Problem, Peterson's Solution, Synchronization Hardware, Semaphores, Monitors, Classic Problems of Synchronization.

Principles of deadlock - System Model, Deadlock Characterization, Methods for Handling Deadlocks: Deadlock Prevention, Detection and Avoidance, Recovery form Deadlock

UNIT– IV: Memory Management **12Hours**

Logical vs physical address space, Swapping, Contiguous Memory Allocation, Paging, Structure of the Page Table, Segmentation.

Virtual Memory Management: Virtual memory overview, Demand Paging, Page-Replacement & its algorithms, Allocation of Frames, Thrashing

UNIT – V: File system Interface **10Hours**

The concept of a file, Access Methods, Directory structure, file sharing, protection.

File System implementation- File system structure, Allocation methods, Free-space management.

Mass-storage structure- Overview of Mass-storage structure, Disk scheduling, Swap space management

Text Books:

1. Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin and Greg Gagne 9th Edition, John Wiley and Sons Inc., 2012

Operating Systems – Internals and Design Principles, William Stallings, 7th Edition, Prentice Hall, 2011

Reference Books:

1. Modern Operating Systems, Andrew S. Tanenbaum, Second Edition, Addison Wesley.
2. Operating Systems: A Design-Oriented Approach, Charles Crowley, Tata Mc Graw Hill Education.
3. Operating Systems: A Concept-Based Approach, D M Dhamdhere, Second Edition, Tata Mc Graw-Hill Education

e- Resources & other digital material

https://en.wikipedia.org/wiki/Operating_system

https://www.tutorialspoint.com/operating_system/

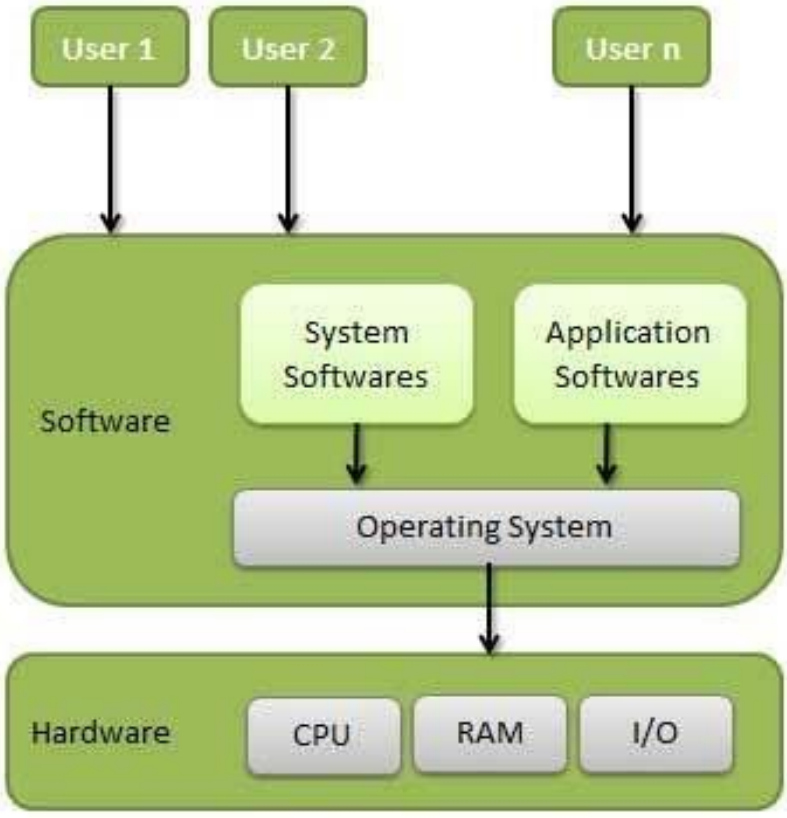
UNIT-1

Syllabus: Introduction to Operating System Concepts:

What Operating Systems do, Computer System Organization, Functions of Operating systems, Types of Operating Systems, Operating Systems services, System calls, Types of System calls, Operating System Structures, Distributed Systems, Special purpose systems.

Operating Systems-Concepts: An Operating System (OS) is an interface between computer user and computer hardware. An operating system is software which performs all the basic tasks like file management, memory management, process management, handling input and output, and controlling peripheral devices such as disk drives and printers. Some popular Operating Systems include Linux, Windows, OS X, VMS, OS/400, AIX, z/OS, etc.

Definition: An operating system is a program that acts as an interface between the user and the computer hardware and controls the execution of all kinds of programs.



What Operating Systems Do (Functions of an Operating System): Computer system can be divided roughly into four components: the ***hardware,*** the ***operating system,*** the

application programs, and the ***users*** as shown above. Following are some of important functions of an operating System from User and System view.

- Memory Management
- Processor Management
- Device Management
- File Management
- Security
- Control over system performance
- Job accounting
- Error detecting aids
- Coordination between other software and users

Memory Management: Memory management refers to management of Primary Memory or Main Memory. Main memory is a large array of words or bytes where each word or byte has its own address.

Main memory provides a fast storage that can be accessed directly by the CPU. For a program to be executed, it must in the main memory. An Operating System does the following activities for memory management −

- Keeps tracks of primary memory, i.e., what part of it are in use by whom, what part are not in use.
- In multiprogramming, the OS decides which process will get memory when and how much.
- Allocates the memory when a process requests it to do so.
- De-allocates the memory when a process no longer needs it or has been terminated.

Processor Management: In multiprogramming environment, the OS decides which process gets the processor when and for how much time. This function is called process scheduling. An Operating System does the following activities for processor management -

- Keeps tracks of processor and status of process. The program responsible for this task is known as **traffic controller**.
- Allocates the processor (CPU) to a process.
- De-allocates processor when a process is no longer required.

Device Management: An Operating System manages device communication via their respective drivers. It does the following activities for device management −

- Keeps tracks of all devices. Program responsible for this task is known as the **I/O controller**.
- Decides which process gets the device when and for how much time.
- Allocates the device in the efficient way.
- De-allocates devices.

File Management: A file system is normally organized into directories for easy navigation and usage. These directories may contain files and other directions.

An Operating System does the following activities for file management −

- Keeps track of information, location, uses, status etc. The collective facilities are often known as **file system**.
- Decides who gets the resources.
- Allocates the resources.
- De-allocates the resources.

Other Important Activities: Following are some of the important activities that an Operating System performs −

1. **Security** − By means of password and similar other techniques, it prevents unauthorized access to programs and data.
2. **Control over system performance** − Recording delays between request for a service and response from the system.
3. **Job accounting** − Keeping track of time and resources used by various jobs and users.
4. **Error detecting aids** − Production of dumps, traces, error messages, and other debugging and error detecting aids.
5. **Coordination between other software's and users** − Coordination and assignment of compilers, interpreters, assemblers, and other software to the various users of the computer systems.

Computer System Organization: Computer System Organization provides general knowledge of the structure of a computer system.

Computer System Operation: A modern general-purpose computer system consists of one or more CPU's and several device controllers connected through a common bus that provides access to shared memory as shown below.

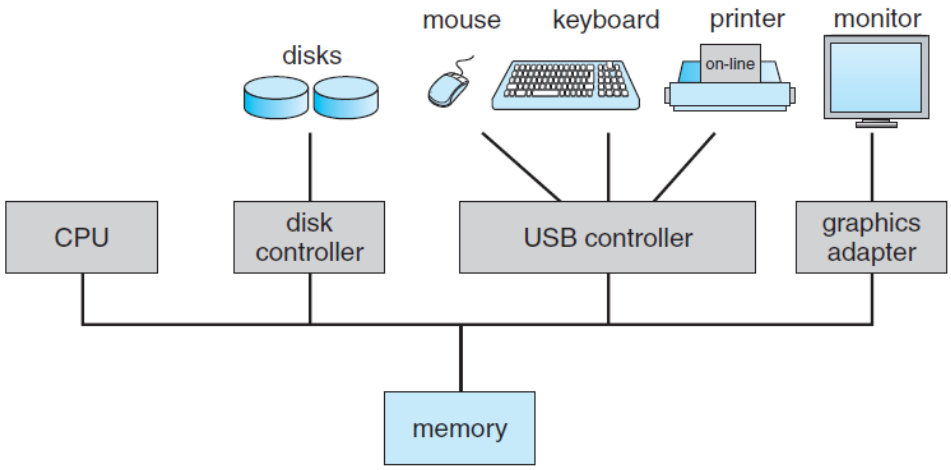


Figure 1.2 A modern computer system.

The CPU and the device controllers can execute in parallel, competing for memory cycles. To ensure orderly access to the shared memory, a memory controller synchronizes access to the memory.

For a computer to start running—for instance, when it is powered up or rebooted—it needs to have an initial program to run. This initial program, or **bootstrap program**, tends to be simple. Typically, it is stored within the computer hardware in read-only memory (**ROM**) or electrically erasable programmable read-only memory (**EEPROM**), known by the general term **firmware**. It initializes all aspects of the system, from CPU registers to device controllers to memory contents. The bootstrap program must know how to load the operating system and how to start executing that system. To accomplish this goal, the bootstrap program must locate the operating-system kernel and load it into memory.

Once the kernel is loaded and executing, it can start providing services to the system and its users. Some services are provided outside of the kernel, by system programs that are loaded into memory at boot time to become **system processes**. On UNIX, the first system process is "init," and it starts many other daemons. Once this phase is complete, the system is fully booted, and the system waits for some event to occur.

The occurrence of an event is usually signalled by an **interrupt** from either the hardware or the software. Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus. Software may trigger 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 usually contains the starting address where the service routine for the interrupt is located. The interrupt service routine executes; on completion, the CPU resumes the interrupted computation.

Storage Structure: The CPU can load instructions only from memory, so any programs to run must be stored there. General-purpose computers run most of their programs from rewritable memory, called main memory (also called **random-access memory**, or **RAM**).

Main memory commonly is implemented in a semiconductor technology called **dynamic random-access memory** (**DRAM**).

Computers use other forms of memory as well like ROM, PROM, EPROM, EEPROM. The immutability of ROM is of use in game cartridges. EEPROM can be changed but cannot be changed frequently and so contains mostly static programs. For example, smartphones have EEPROM to store their factory-installed programs.

All forms of memory provide an array of bytes. Each byte has its own address. Interaction is achieved through a sequence of load or store instructions to specific memory addresses. The load instruction moves a byte or word from main memory to an internal register within the CPU, whereas the store instruction moves the content of a register to main memory. Aside from explicit loads and stores, the CPU automatically loads instructions from main memory for execution.

Most computer systems provide **secondary storage** as an extension of main memory. The main requirement for secondary storage is that it be able to hold large quantities of data permanently. The most common secondary-storage device is a **magnetic disk**, which provides storage for both programs and data. Most programs (system and application) are stored on a disk until they are loaded into memory. Many programs then use the disk as both the source and the destination of their processing.

The storage structure contains cache memory, CD-ROM, magnetic tapes, and so on. Each storage system provides the basic functions of storing a data and holding that data until it is retrieved later. The main differences among the various storage systems lie in speed, cost, size, and volatility.

The wide variety of storage systems can be organized in a hierarchy according to speed and cost as shown below.

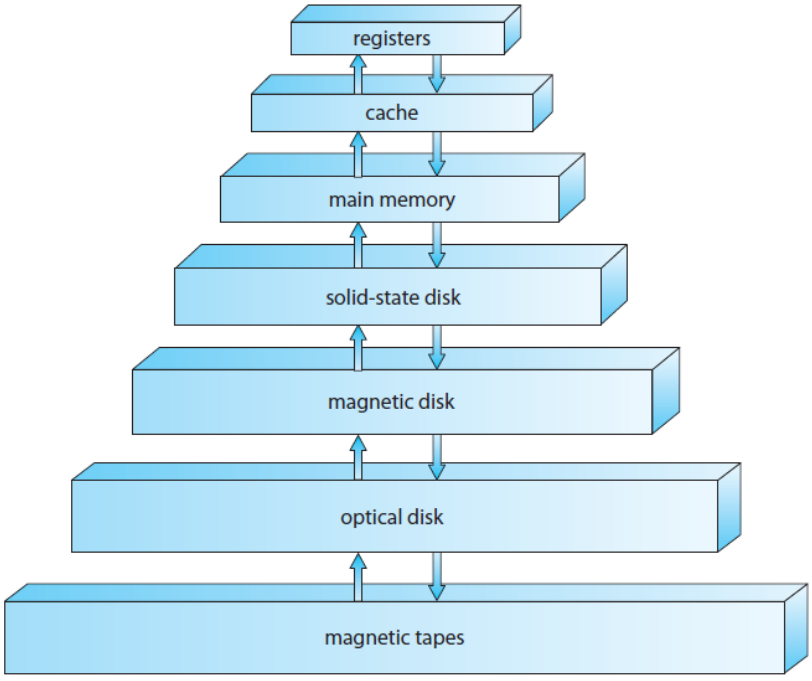


Figure 1.4 Storage-device hierarchy.

The higher levels are expensive, but they are fast. As we move down the hierarchy, the cost per bit generally decreases, whereas the access time generally increases.

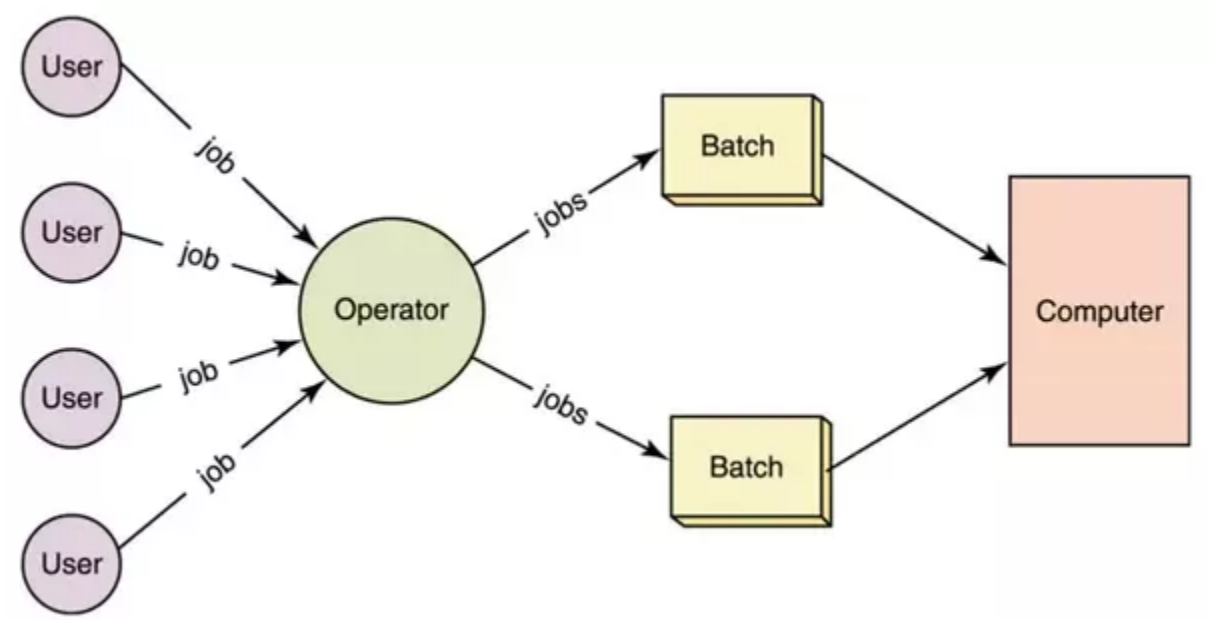
In addition to differing in speed and cost, the various storage systems are either volatile or non-volatile. As mentioned earlier, **volatile storage** loses its contents when the power to the device is removed. In the absence of expensive battery and generator backup systems, data must be written to **non-volatile storage** for safekeeping. In the hierarchy shown above, the storage systems above the solid-state disk are volatile, whereas those including the solid-state disk and below are non-volatile.

I/O Structure: A general-purpose computer system consists of CPUs and multiple device controllers that are connected through a common bus. Depending on the controller, more than one device may be attached. For instance, seven or more devices can be attached to the **small computer-systems interface (SCSI)** controller. A device controller maintains some local buffer storage and a set of special-purpose registers. The device controller is responsible for moving the data between the peripheral devices that it controls and its local buffer storage. Typically, operating systems have a **device driver** for each device controller. This device driver understands the device controller and provides the rest of the operating system with a uniform interface to the device.

To start an I/O operation, the device driver loads the appropriate registers within the device controller. The device controller, in turn, examines the contents of these registers to determine what action to take (such as "read a character from the keyboard"). The controller starts the transfer of data from the device to its local buffer. Once the transfer of data is complete, the device controller informs the device driver via an interrupt that it has finished its operation. The device driver then returns control to the operating system, possibly returning the data or a pointer to the data if the operation was a read. For other operations, the device driver returns status information.

Types of Operating Systems: Operating systems are there from the very first computer generation and they keep evolving with time. Some of the important types of operating systems which are most used are given below.

1. **Batch operating system:** The users of a batch operating system do not interact with the computer directly. Each user prepares his job on an off-line device like punch cards and submits it to the computer operator. To speed up processing, jobs with similar needs are batched together and run as a group. The programmers leave their programs with the operator and the operator then sorts the programs with similar requirements into batches. Examples are Payroll Systems, Bank statements etc.…
   The problems with Batch Systems are as follows:
   - Lack of interaction between the user and the job.
   - CPU is often idle, because the speed of the mechanical I/O devices is slower than the CPU.
   - Difficult to provide the desired priority.



2. **Time Shared Operating System:** Time-sharing is a technique which enables many people, located at various terminals, to use a particular computer system at the same time. Time-sharing or multitasking is a logical extension of multiprogramming. Processor's time which is shared among multiple users simultaneously is termed as time-sharing.

The main difference between Multi-programmed Batch Systems and Time-Sharing Systems is that in case of multi-programmed batch systems, the objective is to maximize processor use, whereas in Time-Sharing Systems, the objective is to minimize response time.

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

The operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time. Examples are Linux, Unix, Multics, Windows 2000 server, Windows NT Server etc.…

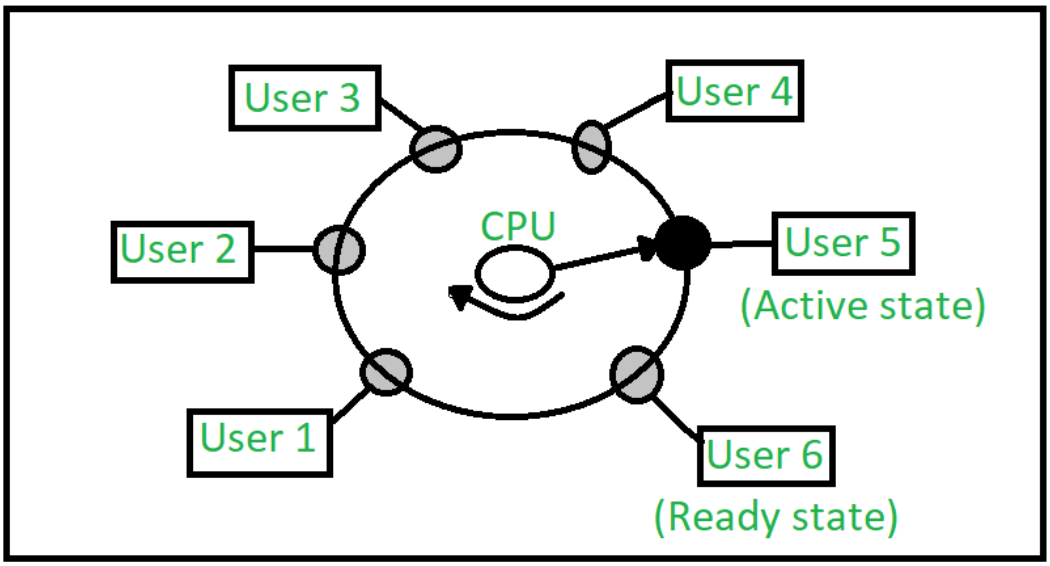
Advantages of Timesharing operating systems are as follows −

- Provides the advantage of quick response.
- Avoids duplication of software.
- Reduces CPU idle time.

Disadvantages of Time-sharing operating systems are as follows −

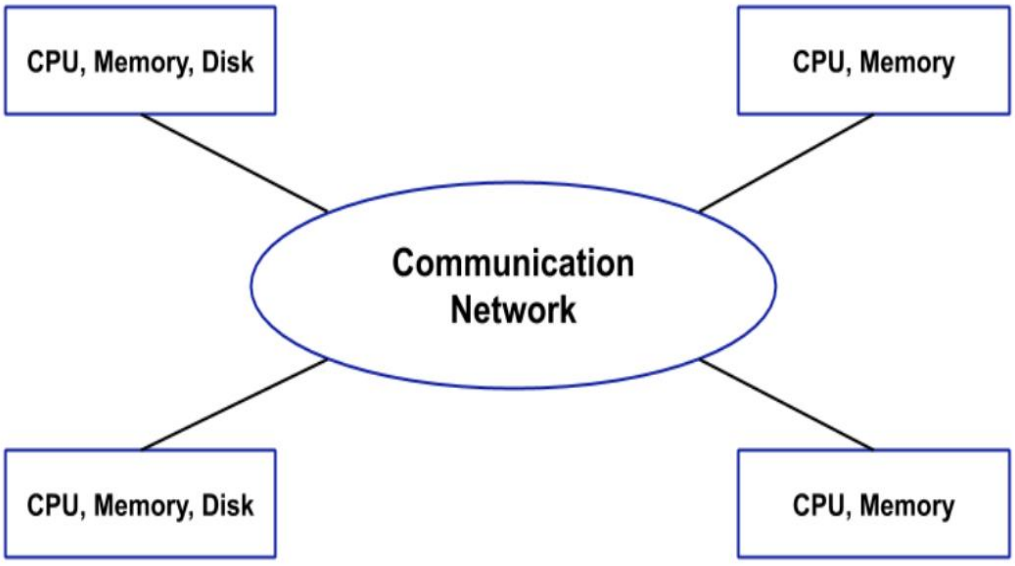
- Problem of reliability
- Question of security and integrity of user programs and data.

- Problem of data communication.



3. **Distributed operating System:** A distributed operating system is an extension of the network operating system that supports higher levels of communication and integration of the machines on the network. Distributed Operating System runs on multiple, independent CPU's. Distributed systems use multiple central processors to serve multiple real-time applications and multiple users. Data processing jobs are distributed among the processors accordingly.

The processors communicate with one another through various communication lines (such as high-speed buses or telephone lines). These are referred as loosely coupled systems or distributed systems where each processor has its own local memory and processors communicate with one another through various communication lines, such as high-speed buses or telephone lines.



The advantages of distributed systems are as follows –

- With resource sharing facility, a user at one site may be able to use the resources available at another.
- Speedup the exchange of data with one another via electronic mail.
- If one site fails in a distributed system, the remaining sites can potentially continue operating.
- Better service to the customers.
- Reduction of the load on the host computer.
- Reduction of delays in data processing.

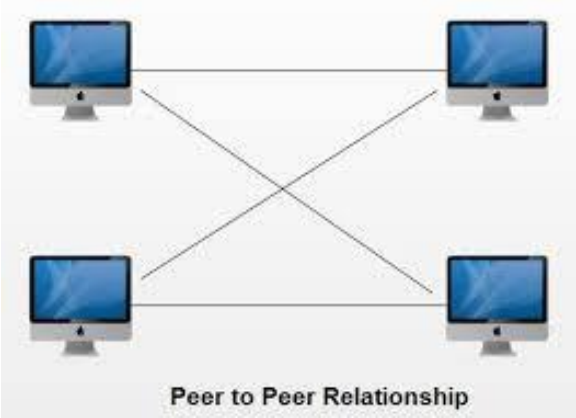
Disadvantages:

1. Since the data is shared among all the computers, so to make the data secure and accessible to few computers, you need to put some extra efforts.
2. If there is a problem in the communication network, then the whole communication will be broken.

4. **Network operating System:** An operating system that provides the connectivity among several autonomous computers is called a network operating system. A typical configuration for a network operating system is a collection of personal computers along with a common printer, server and file server for archival storage, all tied together by a local network.

Mainly there are two types of network operating systems named as peer-to-peer and client / server.

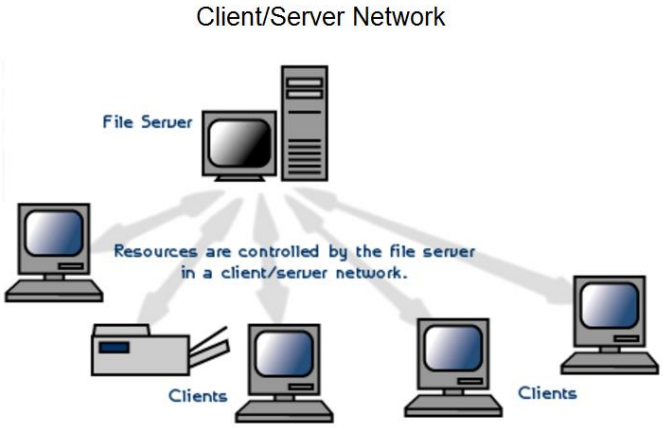
Peer-to-peer network operating systems allow users to share resources and files located on their computers and to access shared resources found on other computers. In a peer-to-peer network, all computers are considered equal; they all have the same privileges to use the resources available on the network. Peer-to-peer networks are designed primarily for small to medium local area networks. Examples are Ubuntu, Linux Mint, Manjaro etc…



Peer to Peer Relationship

Client/server network operating systems allow the network to centralize functions and applications in one or more dedicated file servers. The file servers become the heart of the system, providing access to resources, and providing security. The workstations (clients) have access to the resources available on the file

servers. The network operating system allows multiple users to simultaneously share the same resources irrespective of physical location. Novell Netware and Windows 2000 Server are examples of client/ server network operating systems.



Examples of network operating systems include Microsoft Windows Server 2003, Microsoft Windows Server 2008, UNIX, Linux, Mac OS X, Novell NetWare, and BSD.

The advantages of network operating systems are as follows −

- Centralized servers are highly stable.
- Security is server managed.
- Upgrades to new technologies and hardware can be easily integrated into the system.
- Remote access to servers is possible from different locations and types of systems.

The disadvantages of network operating systems are as follows −

- High cost of buying and running a server.
- Dependency on a central location for most operations.
- Regular maintenance and updates are required.

5. **Real Time Operating System:** A real-time system is defined as a data processing system in which the time interval required to process and respond to inputs is so small that it controls the environment. The time taken by the system to respond to an input and display of required updated information is termed as the response time. So, in this method, the response time is very less as compared to online processing.

Real-time systems are used when there are rigid time requirements on the operation of a processor, or the flow of data and real-time systems can be used as a control device in a dedicated application. A real-time operating system must have well-defined, fixed time constraints, otherwise the system will fail. For example, scientific experiments, medical image systems, industrial control systems, weapon systems, robots, air traffic control systems, etc.

There are two types of real-time operating systems.

- **Hard real-time systems:** Hard real-time systems guarantee that critical tasks complete on time. In hard real-time systems, secondary storage is limited or missing, and the data is stored in ROM. In these systems, virtual memory is almost never found.
- **Soft real-time systems:** Soft real-time systems are less restrictive. A critical real-time task gets priority over other tasks and retains the priority until it completes. Soft real-time systems have limited utility than hard real-time systems. For example, multimedia, virtual reality, Advanced Scientific Projects likes undersea exploration and planetary rovers, etc.

6. **Embedded Operating System:** An Embedded Operating System is designed to perform a specific task for a particular device which is not a computer. For example, the software used in elevators is dedicated to the working of elevators only and nothing else. So, this can be an example of Embedded Operating System. The Embedded Operating System allows the access of device hardware to the software that is running on the top of the Operating System.

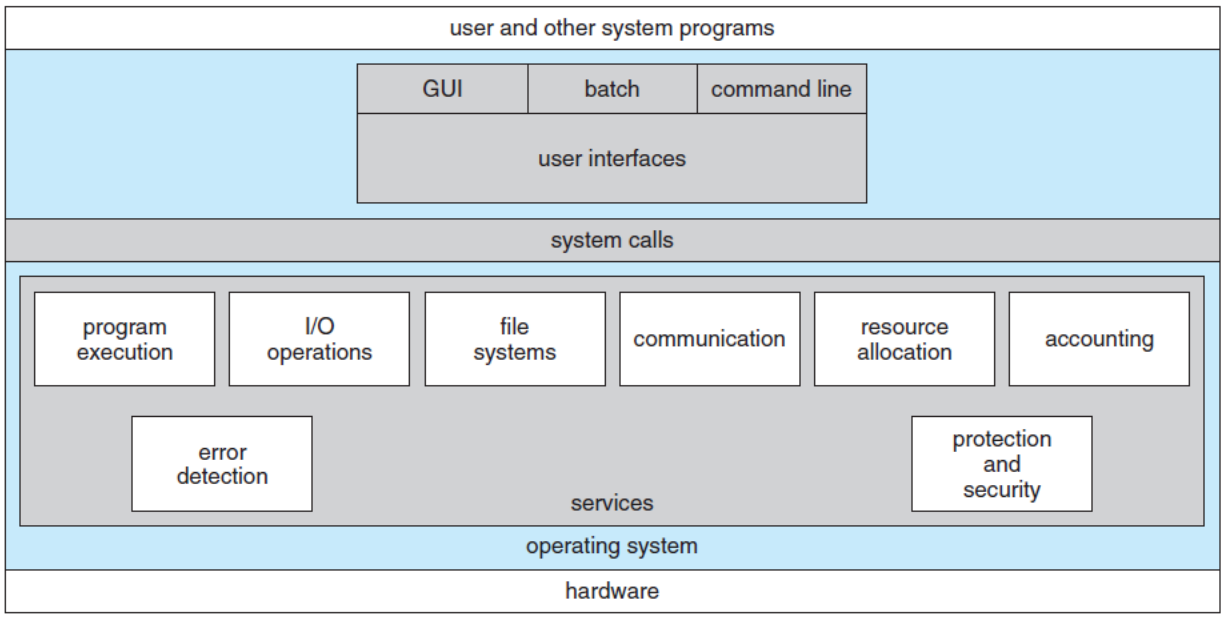
   Advantages:
   1. Since it is dedicated to a particular job, so it is fast.
   2. Low cost.
   3. These consume less memory and other resources.

   Disadvantages:
   1. Only one job can be performed.
   2. It is difficult to upgrade or is nearly scalable.

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

- **User interface:** Almost all operating systems have a **user interface** (UI). This interface can take several forms. One is a **command-line interface (CLI),** which uses text commands and a method for entering them. Another is a **batch interface,** in which commands and directives to control those commands are entered into files, and those files are executed. Most commonly a **graphical user interface (GUI)** is used. Here, the interface is a window system with a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text.
- **Program execution:** The system must be able to load a program into memory and to run that program. The program must be able to end its execution, either normally or abnormally (indicating error).
- **I/O Operations:** A running program may require I/O device. For specific devices, special functions may be desired (such as recording to a CD or DVD drive). For efficiency and protection, users usually cannot control I/O devices directly. Therefore, the operating system must provide a means to do I/O.
- **File-system manipulation:** Operating System is used to control operations on files like creating, opening, reading, and writing.

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



A View of Operating System Services

Operating System Structures: An operating system is a construct that allows the user application programs to interact with the system hardware. Since the operating system is such a complex structure, it should be created with utmost care so it can be used and modified easily. An easy way to do this is to create the operating system in parts. Each of these parts should be well defined with clear inputs, outputs, and functions.

- **Simple Structure:** Many operating systems do not have well-defined structures. Frequently, such systems started as small, simple, and limited systems and then grew beyond their original scope. MS-DOS is an example of such a system as shown below.

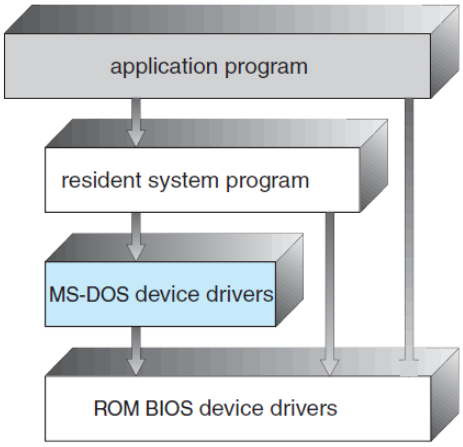


Figure 2.11 MS-DOS layer structure.

In MS-DOS, the interfaces and levels of functionality are not well separated. For instance, application programs can access the basic I/O routines to write directly to the display and disk drives. Such freedom leaves MS-DOS vulnerable to errant (or malicious) programs, causing entire system crashes when user programs fail.

Another example of limited structuring is the original UNIX operating system as shown below.

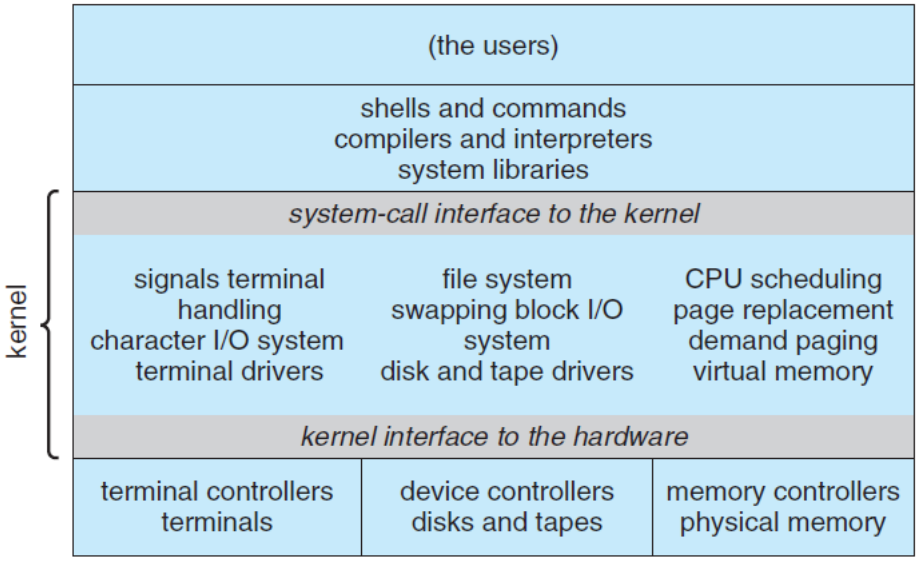


Figure 2.12 Traditional UNIX system structure.

Like MS-DOS, UNIX initially was limited by hardware functionality. It consists of two separable parts: the kernel and the system programs. The kernel is further separated into a series of interfaces and device drivers, which have been added and expanded over the years as UNIX has evolved.

- **Layered Approach:** One way to achieve modularity in the operating system is the layered approach. In this, the bottom layer is the hardware, and the topmost layer is the user interface. An image demonstrating the layered approach is as follows –

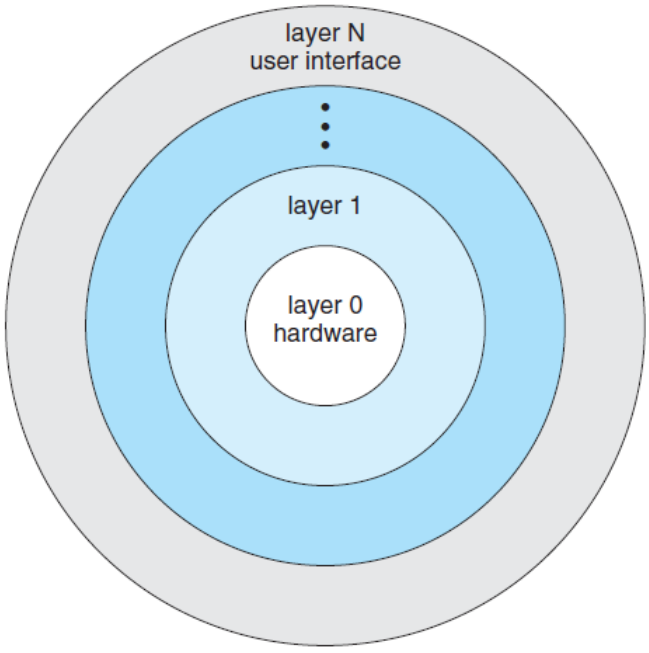


Figure 2.13 A layered operating system.

The main advantage of the layered approach is simplicity of construction and debugging. The layers are selected so that each uses functions (operations) and

services of only lower-level layers. This approach simplifies debugging and system verification. The first layer can be debugged without any concern for the rest of the system, because, by definition, it uses only the basic hardware (which is assumed correct) to implement its functions. Once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and so on. If an error is found during the debugging of a particular layer, the error must be on that layer, because the layers below it are already debugged. Thus, the design and implementation of the system are simplified.

The major difficulty with the layered approach involves appropriately defining the various layers. Because a layer can use only lower-level layers, careful planning is necessary. Another problem with layered implementations is that they tend to be less efficient than other types.

- **Microkernels:** In the mid-1980's, researchers at Carnegie Mellon University developed an operating system called **Mach** that modularized the kernel using the **microkernel** approach. This method structures the operating system by removing all nonessential components from the kernel and implementing them as system and user-level programs. The result is a smaller kernel as shown below.

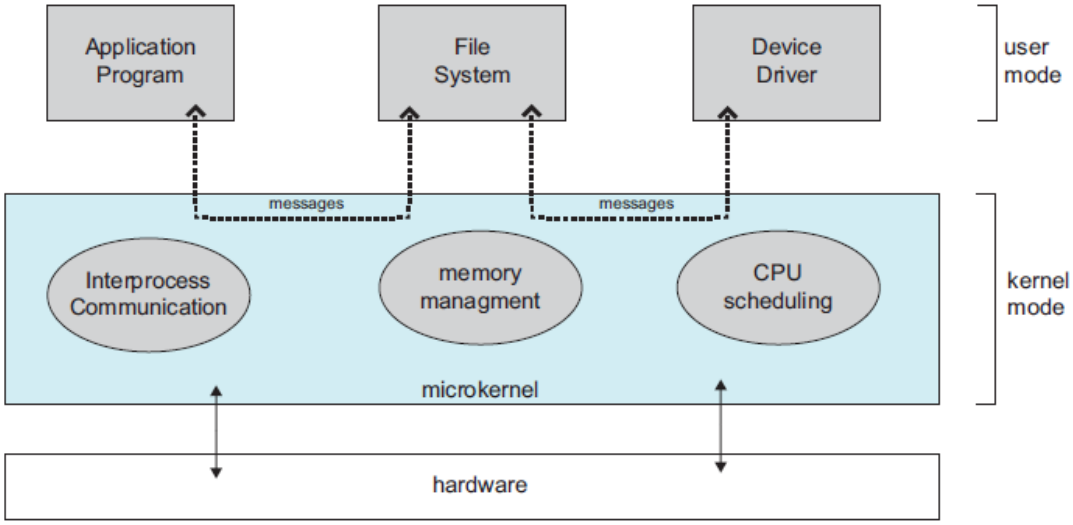


Figure 2.14 Architecture of a typical microkernel.

One benefit of the microkernel approach is that it makes extending the operating system easier. All new services are added to user space and consequently do not require modification of the kernel. When the kernel does have to be modified, the changes tend to be fewer, because the microkernel is a smaller kernel. The microkernel also provides more security and reliability since most services are running as user—rather than kernel— processes. If a service fails, the rest of the operating system remains untouched.

Unfortunately, the performance of microkernels can suffer due to increased system-function overhead.

- **Modules:** The best current methodology for operating-system design involves using **loadable kernel modules**. Here, the kernel has a set of core components and links

in additional services via modules, either at boot time or during run time. This type of design is common in modern implementations of UNIX, such as Solaris, Linux, and Mac OS X, as well as Windows.

The Solaris operating system structure, shown below, is organized around a core kernel with seven types of loadable kernel modules.

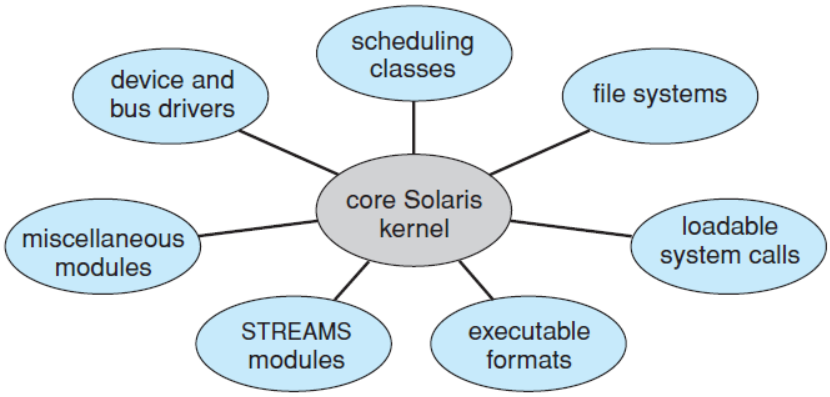


Figure 2.15 Solaris loadable modules.

- **Hybrid Systems:** Very few operating systems adopt a single, strictly defined structure. Instead, they combine different structures, resulting in hybrid systems that address performance, security, and usability issues. Examples for Hybrid systems are Apple Mac OS X, iOS, and Android.

  The Apple Mac OS X operating system uses a hybrid structure as shown below.

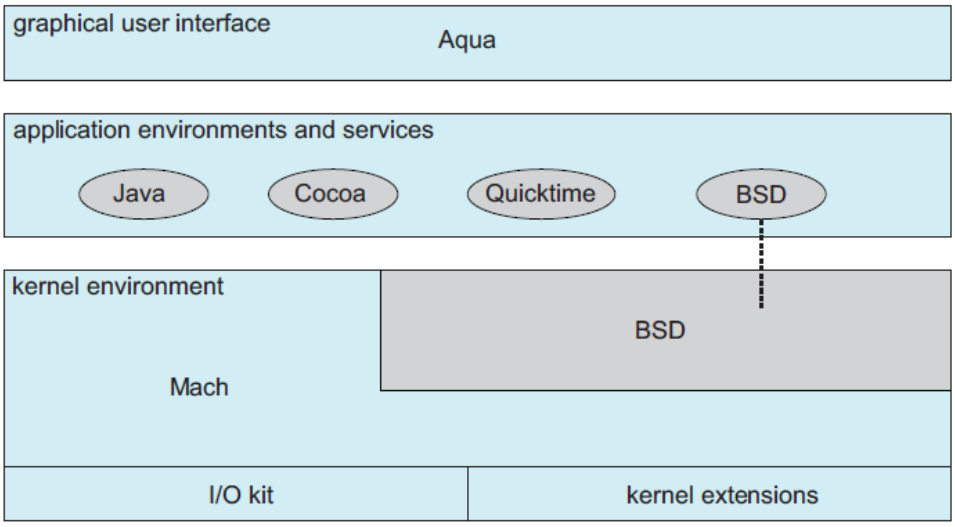


Figure 2.16 The Mac OS X structure.

The top layers include the ***Aqua*** user interface and a set of application environments and services. Notably, the **Cocoa** environment specifies an API for the Objective-C programming language, which is used for writing Mac OS X applications. Below these layers is the ***kernel environment***, which consists primarily of the Mach microkernel and the BSD UNIX kernel. Mach provides memory management; support for remote procedure calls (RPCs) and inter process

communication (IPC) facilities, including message passing; and thread scheduling. The BSD component provides a BSD command-line interface, support for networking and file systems, and an implementation of POSIX APIs, including Pthreads. In addition to Mach and BSD, the kernel environment provides an I/O kit for development of device drivers and dynamically loadable modules.

iOS is a mobile operating system designed by Apple to run its smartphone, the ***iPhone***, as well as its tablet computer, the ***iPad***. iOS is structured on the Mac OS X operating system, with added functionality pertinent to mobile devices, but does not directly run Mac OS X applications. The structure of iOS is shown below.

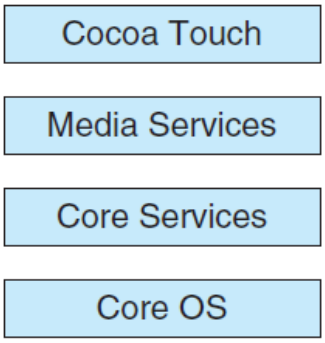


Figure 2.17 Architecture of Apple's iOS.

Cocoa Touch is an API for Objective-C that provides several frameworks for developing applications that run on iOS devices. The **media services** layer provides services for graphics, audio, and video. The **core services** layer provides a variety of features, including support for cloud computing and databases. The bottom layer represents the core operating system, which is based on the kernel environment.

The Android operating system was designed by the Open Handset Alliance (led primarily by Google) and was developed for Android smartphones and tablet computers. The structure of Android is shown below.

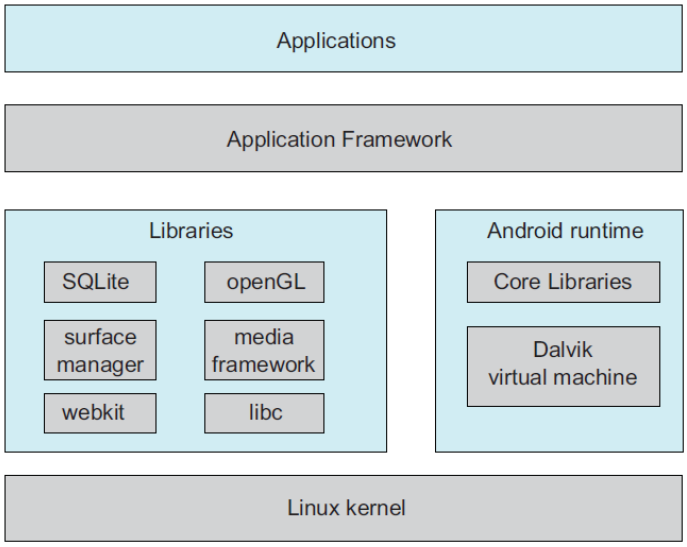


Figure 2.18 Architecture of Google's Android.

Android is like iOS in that it is a layered stack of software that provides a rich set of frameworks for developing mobile applications. The Android runtime environment includes a core set of libraries as well as the Dalvik virtual machine. Software designers for Android devices develop applications in the Java language. However, rather than using the standard Java API, Google has designed a separate Android API for Java development. The Java class files are first compiled to Java bytecode and then translated into an executable file that runs on the Dalvik virtual machine. The Dalvik virtual machine was designed for Android and is optimized for mobile devices with limited memory and CPU processing capabilities.

The set of libraries available for Android applications includes frameworks for developing web browsers (webkit), database support (SQLite), and multimedia. The libc library is like the standard C library but is much smaller and has been designed for the slower CPUs that characterize mobile devices.

Distributed Systems: A distributed system is a collection of physically separate, possibly heterogeneous computer systems that are networked to provide the users with access to the various resources that the system maintains. Access to a shared resource increases computation speed, functionality, data availability, and reliability.

A network, in the simplest terms, is a communication path between two or more systems. Distributed systems depend on networking for their functionality. Networks are characterized based on the distances between their nodes. A local-area network (LAN) connects computers within a room, a floor, or a building. A wide-area network (WAN) usually links buildings, cities, or countries. Distributed operating systems depend on networking for their operation. Distributed OS runs on and controls the resources of multiple machines. It provides resource sharing across the boundaries of a single computer system.

Advantages:

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

Special purpose systems: There are various classes of computer systems based upon their computational speed, usage, and hardware. The following are some special purpose systems according to specific applications.

- **Real time embedded systems**: Real Time embedded systems are specially designed for serving autonomous computers like satellites, robotics, hydroelectric dams etc.…A real time system uses priority scheduling algorithm to meet the responsive requirement of a real time application. Memory management in real time system is comparatively less demanding than in other types of multiprogramming systems.

Embedded systems are small computers having a limited set of hardware like a small processor capable of processing a limited set of instructions. A variety of embedded systems exists of which some are computers with standard operating systems, some have dedicated programs embedded in their limited memories and often some do not even have any software, hardware to do processing.

Nearly all embedded systems use real-time operating system because they are used as control devices and have rigid time requirements. Sensors are used to input data such as temperature, air pressure, etc., from the environment to embedded system where that data is analyzed, and several other controls are adjusted by embedded system itself to control the situation of system.

Few examples are home appliance controllers, weapon controllers, boiler temperature controllers, fuel injection system in automobile engines, etc. A real-time system needs that processing to be done in fixed time constraints.

- **Multimedia systems:** Multimedia refers to data of multiple types that includes audio and video including conventional data like text-files, word-processing documents, spreadsheets, etc. It requires that audio and video data must be processed based upon certain time restrictions. This is called as streaming. It is usually 30 frames per second for a video file.

  Applications such as video conferencing, movies and clips downloaded over internet, mp3, DVD, VCD playing, and recording are examples of various multimedia applications. A multimedia application is usually a combination of both audio and video. The multimedia is not limited to desktop operating system or computers, but it is also becoming popular in handheld.
- **Handheld systems:** Hand-held systems refers to small portable devices that can be carried along and can perform normal operations. They are usually battery powered. Examples include Personal Digital Assistants (PDAs), mobile phones, palm-top computers, pocket-PCs etc. As they are handheld devices, their weights and sizes have certain limitations as a result they are equipped with small memories, slow processors, and small display screens, etc.

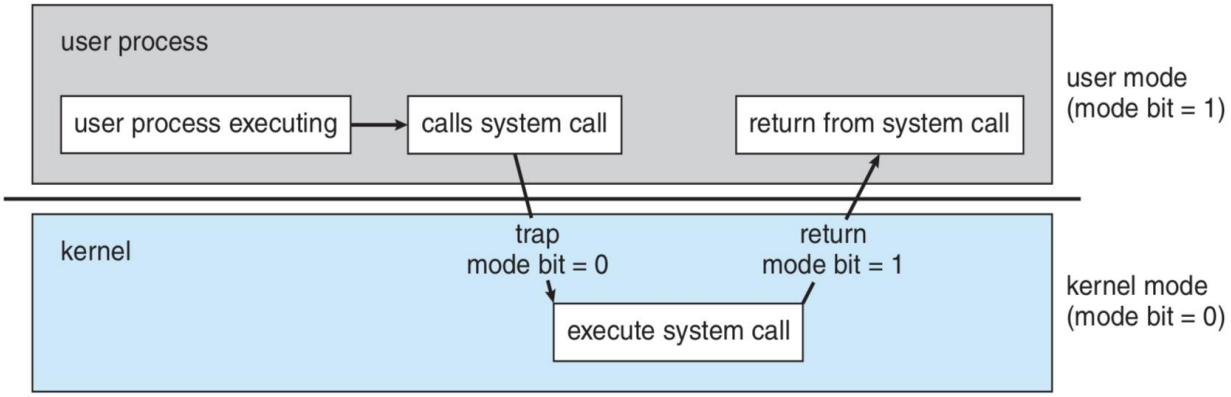
  The physical memory capacity is very less (512 KB to 128 MB) hence the operating systems of these devices must manage the memory efficiently. As the processors are slower due to battery problem, the operating system should not burden.

  Developers of handheld systems and applications face many challenges, most of which are due to the limited size of such devices. For example, a PDA is typically about 5 inches in height and 3 inches in width, and it weighs less than one-half pound. Because of their size, most handheld devices have a small amount of memory, slow processors, and small display screens.

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

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

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



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

The hardware allows privileged instructions to be executed only in kernel mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system.

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

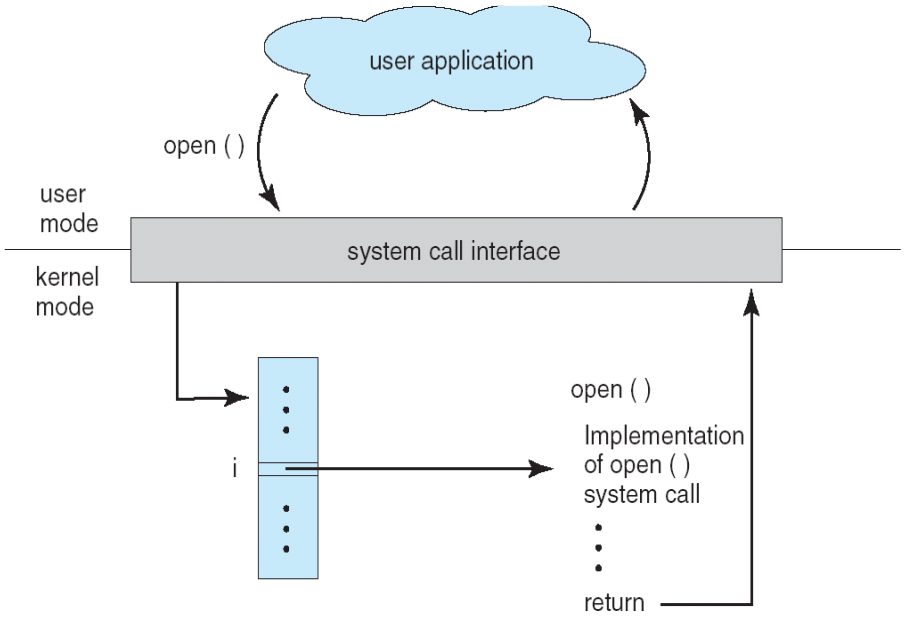
Introduction to System Calls: In computing, a **system call** is the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on. A system call is a way for programs to **interact with the operating system**. A computer program makes a system call when it makes a request to the operating system's kernel. System call **provides** the services of the operating system to the user programs via Application Program Interface (API). It provides an interface between a

process and operating system to allow user-level processes to request services of the operating system. System calls are the only entry points into the kernel system. All programs needing resources must use system calls.

Services Provided by System Calls:

1. Process creation and management
2. Main memory management
3. File Access, Directory and File system management
4. Device handling(I/O)
5. Protection
6. Networking, etc.

The following figure represents handling of open () system call.



Types of System Calls: System calls can be grouped roughly into six major categories: process control, file manipulation, device manipulation, information maintenance, communications, and protection.

1. Process Control:
   - fork ( ) : This system call is used to create a new process, which is called as child process.
   - wait ( ): A call to wait() blocks the calling process until one of its child processes exits or a signal is received.
   - exit ( ): This system call terminates a process normally.
2. File Manipulation:
   - open ( ): This is used for creating and opening a file.
   - read ( ): This is used to read content from specified file descriptor.
   - write ( ): This is used to wrote content to specified file descriptor.
   - close ( ): This is used to close opened file descriptors.
3. Device Manipulation:

- iocntl ( ): This system call is used to control specified I/O device.
- read ( ): This system call is used to read data from specified device.
- write ( ): This system call is used to write data to specified device.

4. Information Maintenance:
   - getpid() : returns the process ID (PID) of the calling process.
   - alarm() : The ***alarm()*** system call is used to generate a ***SIGALRM*** signal after a specified amount of time elapsed.
   - sleep() : sleep for the specified number of seconds.
5. Communications:
   - pipe ( ) : This system call sends output of one command as input of another command.
   - shmget ( ): This system call is used to create a shared memory.
   - shmat ( ): This system call is used to access shared memory.
6. Protection and Security :
   - chmod ( ): This system call is used to change file access permissions.
   - chown ( ): This system call is used to change ownership of a file.
   - umask ( ): This system call is used to change default umask value.

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

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

FREQUENTLY ASKED QUESTIONS

1. What is an Operating System? What are different types of Operating Systems?
2. Explain functions of an Operating System?
3. Explain the operating system structure and its functions.
4. Explain about services of Operating Systems? (Or) With a neat sketch, explain in detail about the interrelation between various services provided by the operating system.
5. Draw a modern computer system.
6. Define the essential properties of the following types of operating systems: i) Batch ii) Time sharing iii) Real time iv) Parallel v) Distributed vi) Handheld.
7. Explain the importance of Real-Time Embedded systems.
8. What is a System Call? Explain about different types of System Calls?
9. Explain the Time-shared operating system.
10. Draw and explain OS layered and modular architecture and its services.
11. Explain briefly Layered Operating system structure with neat sketch.
12. Distinguish between client-server and peer-to-peer models of distributed systems.
13. Explain the Dual-Mode operation of an operating system.
14. Explain how multiprogramming increases the utilization of CPU?
15. List and briefly describe types of operating systems.
16. Write about monolithic kernel, layered kernel, and micro kernel structures of operating systems.
17. What are the various components of operating system structure and explain the simple and layered approach of operating system in detail?

UNIT-2

Syllubus: **Process Management**: Process concept – Process State Diagram, Process Control Block, Process scheduling – Scheduling queues, Schedulers, Scheduling Criteria, Scheduling algorithms and their evolution, operations on Processes, Inter process communication.
Threads : Overview, User and Kernel Threads, Multi Threading models.

Process Concept: A process is an instance of a program that is running in a computer. A process is more than the program code, which is sometimes known as the **text section.** It also includes the current activity, as represented by the value of the **program counter** and the contents of the processor's registers. A process generally also includes the process **stack,** which contains temporary data (such as function parameters, return addresses, and local variables), and a **data section,** which contains global variables. A process may also include a **heap,** which is memory that is dynamically allocated during process run time. The structure of a process in memory is shown in the following figure.

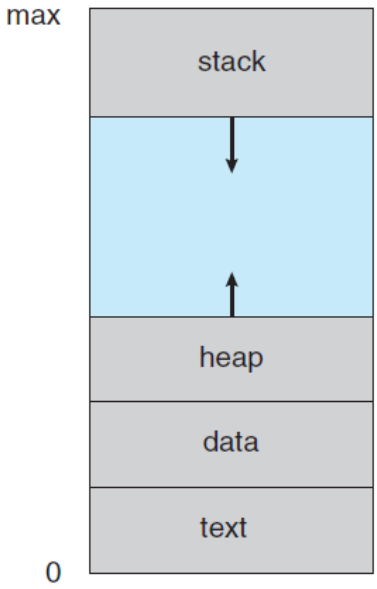


Figure 3.1 Process in memory.

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

by the current activity of that process. Each process may be in one of the following states:

- **New.** The process is being created.
- **Running.** Instructions are being executed.
- **Waiting.** The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- **Ready.** The process is waiting to be assigned to a processor.
- **Terminated.** The process has finished execution.

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

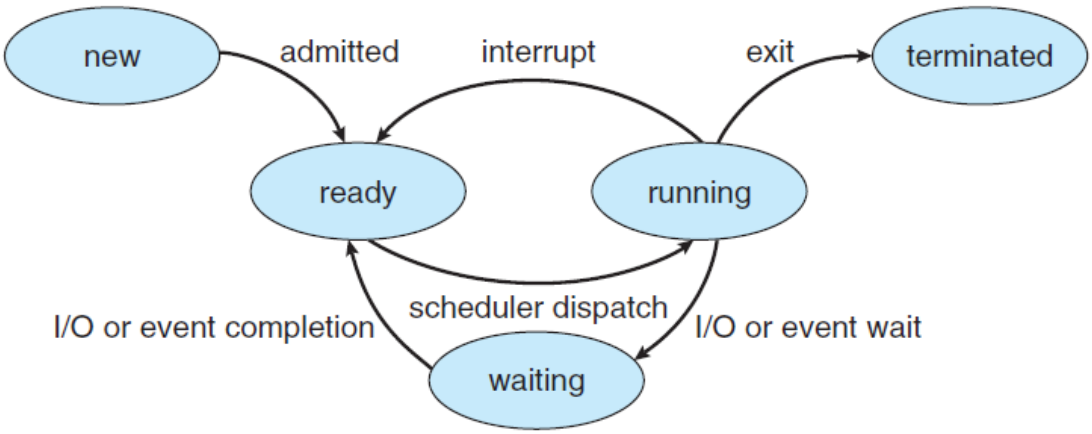
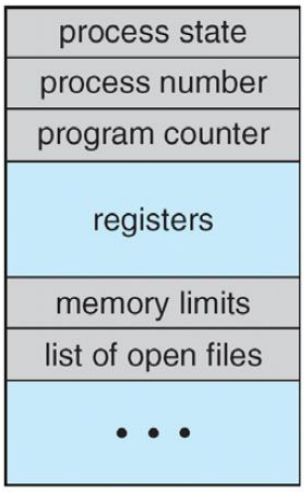


Figure 3.2 Diagram of process state.

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



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

- **Process state.** The state may be new, ready, running, waiting, halted, and so on.

- **Program counter:** The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers:** This block indicates type of registers used by the process. The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information.
- **CPU-scheduling information:** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Memory-management information:** This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.
- **Accounting information:** This information includes the amount of CPU and real time used, time limits, account 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.

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

- **Job Queue** - which consists of all processes in the system.
- **Ready Queue**:The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue.** This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.
- **Device Queue**: The list of processes waiting for a particular I/O device is called a **device queue.** When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request. Suppose the process makes an I/O request to a shared device, such as a disk. Since there are many processes in the system, the disk may be busy with the I/O request of some other process. The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O device is called a **device queue**.

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

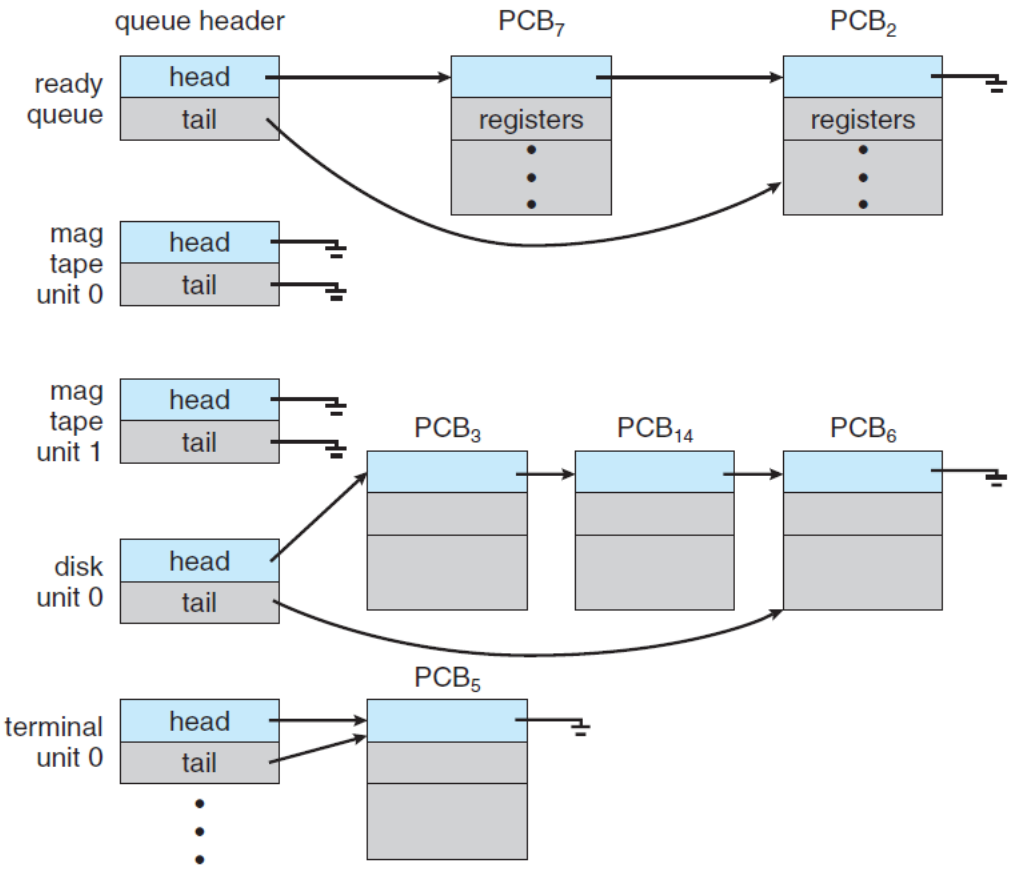


Figure 3.5 The ready queue and various I/O device queues.

A common representation for a discussion of process scheduling is a **queueing diagram** which is shown below. Two types of queues are present: the ready queue and a set of device queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system. A new process is initially put in the ready queue. It waits there until it is selected for execution, or is **dispatched.** Once the process is allocated the CPU and is executing, one of several events could occur:

- The process could issue an I/O request and then be placed in an I/O queue.
- The process could create a new subprocess and wait for the subprocess's termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

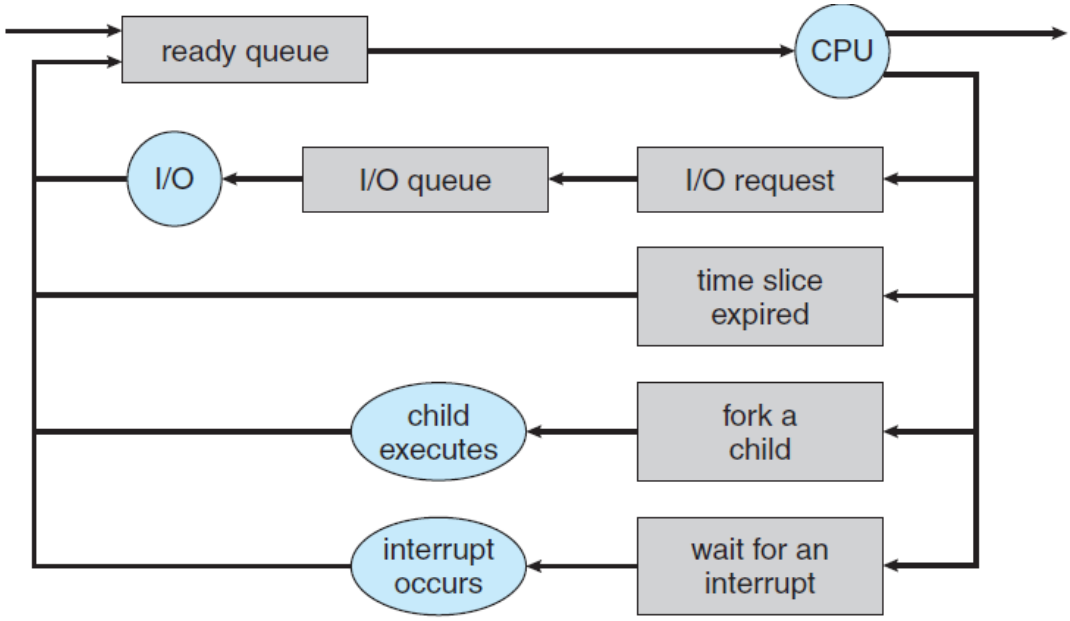


Figure 3.6 Queueing-diagram representation of process scheduling.

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

There are three different types of schedules:

- Long-Term scheduler or Job Scheduler
- Short-Term Scheduler or CPU Scheduler
- Medium-Term Scheduler.

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

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

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

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

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

The **short-term scheduler, or** CPU **scheduler,** selects from among the processes that are ready to execute and allocates the CPU to one of them.

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

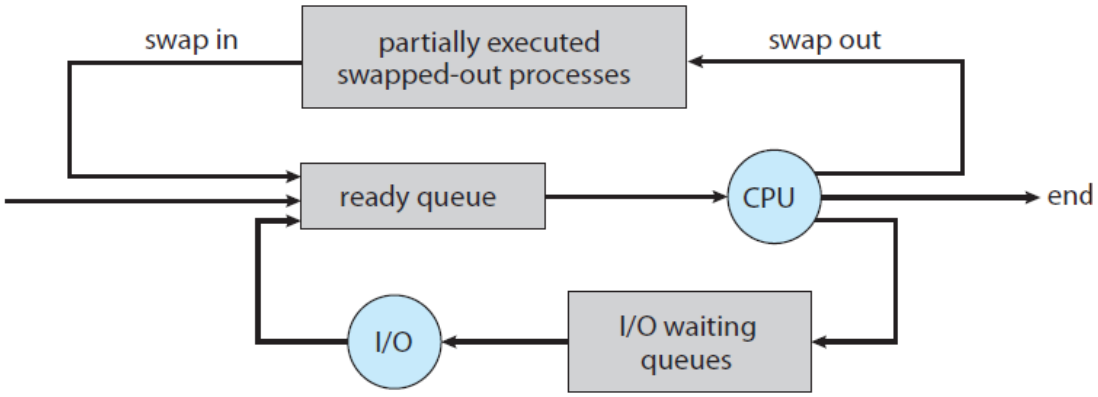


Figure 3.7 Addition of medium-term scheduling to the queueing diagram.

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

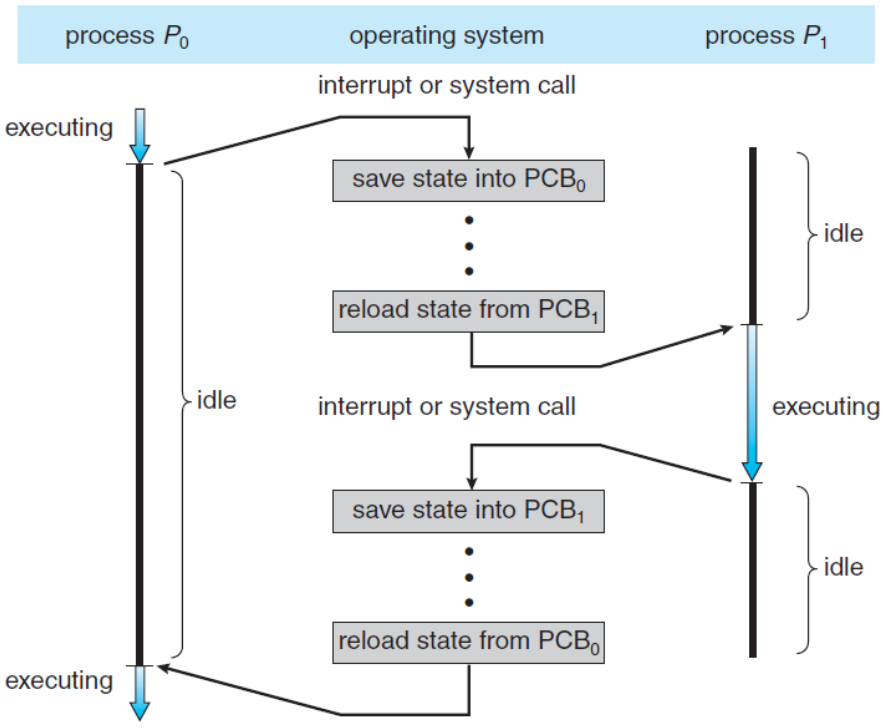


Figure 3.4 Diagram showing CPU switch from process to process.

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

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

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

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

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

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

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

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

A new process is created by the **fork()** system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process. Both processes (the

parent and the child) continue execution at the instruction after the f ork(), with one difference: The return code for the fork() is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

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

Process Termination: A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit() system call. At that point, the process may return a status value (typically an integer) to its parent process (via the wait() system call). All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system.

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

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

Interprocess Communication: Processes executing concurrently in the operating system may be either independent processes or cooperating processes. A process is **independent** if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent. A process is **cooperating** if **it** can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

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

- **Information sharing:** Since several users may be interested in the same piece of information, sharing provides an environment to allow concurrent access to such information.
- **Computation speedup:** To execute a particular task faster, break it into subtasks, each of which will be executing in parallel with the others.
- **Modularity:** To construct the system in a modular fashion, divide the system functions into separate processes or threads.
- **Convenience:** Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel.

Cooperating processes require an **interprocess communication (IPC)** mechanism that will allow them to exchange data and information. There are two

fundamental models of interprocess communication: **(1) shared memory** and (2) **message passing.** In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. In the messagepassing model, communication takes place by means of messages exchanged between the cooperating processes. The two communications models are contrasted in the following figure.

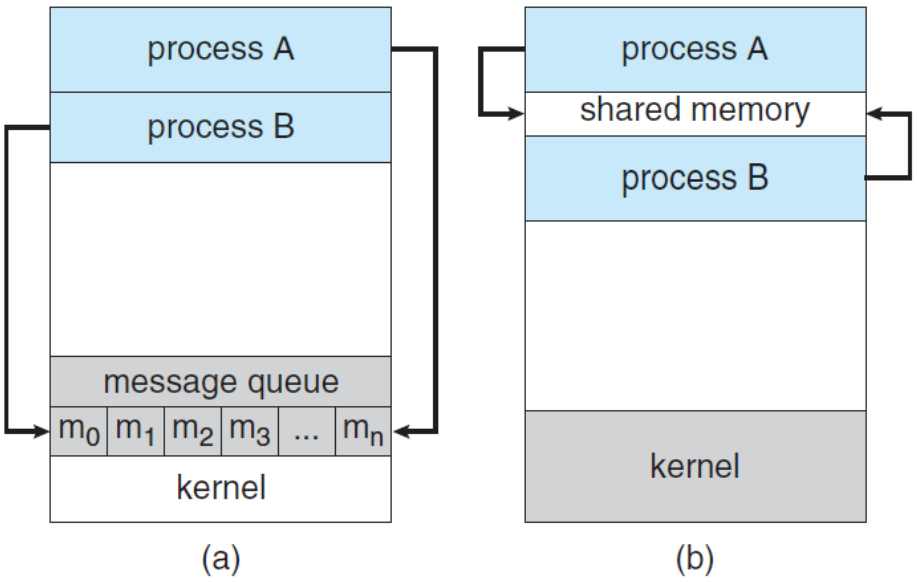


Figure 3.12 Communications models. (a) Message passing. (b) Shared memory.

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

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

One solution to the producer-consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, must have available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

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

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

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

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

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

The Producer process

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

The Consumer process

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

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

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

If processes P and Q want to communicate, they must send messages to and receive messages from each other; a **communication link** must exist between them. This link can be implemented in a variety of ways.

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

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

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

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

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

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

In contrast, a mailbox that is owned by the operating system has an existence of its own. The operating system then must provide a mechanism that allows a process to do the following:

- ❖ Create a new mailbox.
- ❖ Send and receive messages through the mailbox.
- ❖ Delete a mailbox.

- **Synchronous or asynchronous communication:** Communication between processes takes place through calls to sendO and receive () primitives. There are different design options for implementing each primitive. Message passing may be either **blocking** or **nonblocking**— also known as **synchronous** and **asynchronous.**
  - ❖ **Blocking send.** The sending process is blocked until the message is received by the receiving process or by the mailbox.
  - ❖ **Nonblocking send.** The sending process sends the message and resumes operation.
  - ❖ **Blocking receive.** The receiver blocks until a message is available.
  - ❖ **Nonblocking receive.** The receiver retrieves either a valid message or a null.

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

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

Thread: A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.Atraditional (or ***heavyweight***) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time. The following figure illustrates the difference between a traditional **single-threaded** process and a **multithreaded** process.

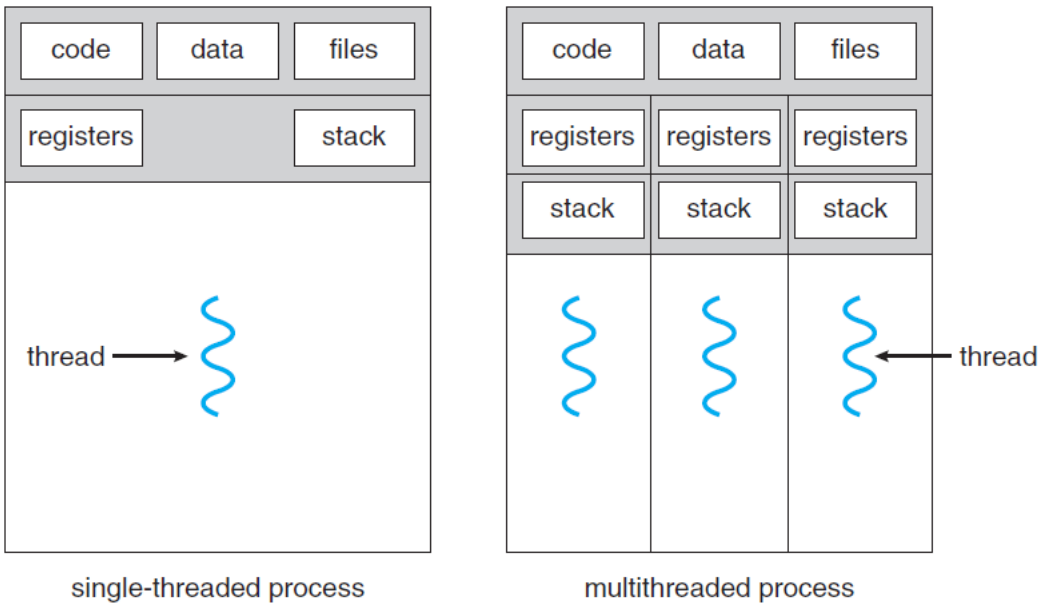


Figure 4.1 Single-threaded and multithreaded processes.

Difference between Process and Thread:

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

Most software applications that run on modern computers are typically is implemented as a separate process with several threads of control. A web browser might have one thread display images or text while another thread retrieves data from the network, for example. A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.

In certain situations, a single application may be required to perform several similar tasks. For example, a web server accepts client requests for web pages, images, sound, and so forth. A busy web server may have several (perhaps

thousands of) clients concurrently accessing it. If the web server ran as a traditional single-threaded process, it would be able to service only one client at a time, and a client might have to wait a very long time for its request to be serviced.

One solution is to have the server run as a single process that accepts requests. When the server receives a request, it creates a separate process to service that request. In fact, this process-creation method was in common use before threads became popular. Process creation is time consuming and resource intensive.

It is generally more efficient to use one process that contains multiple threads. If the web-server process is multithreaded, the server will create a separate thread that listens for client requests. When a request is made, rather than creating another process, the server creates a new thread to service the request and resume listening for additional requests. This is diagrammatically shown below.

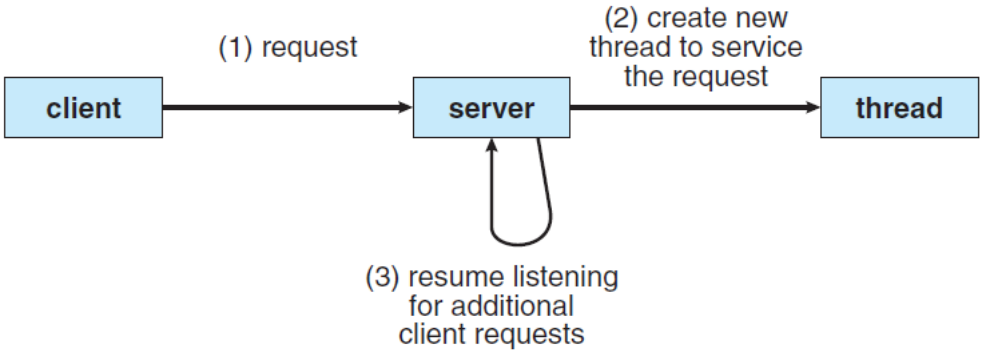


Figure 4.2 Multithreaded server architecture.

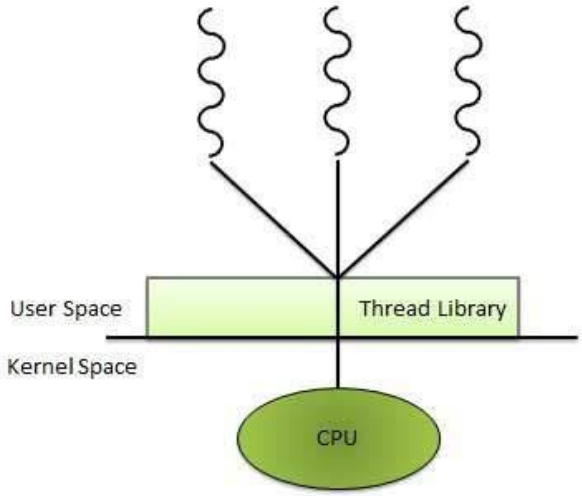
Benefits of Multithreded Programming: The benefits of multithreaded programming can be broken down into four major categories:

- **Responsiveness**. Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. This quality is especially useful in designing user interfaces. For instance, consider what happens when a user clicks a button that results in the performance of a time-consuming operation. A single-threaded application would be unresponsive to the user until the operation had completed. In contrast, if the time-consuming operation is performed in a separate thread, the application remains responsive to the user.
- **Resource sharing**. Processes can only share resources through techniques such as shared memory and message passing. Such techniques must be explicitly arranged by the programmer. However, threads share the memory and the resources of the process to which they belong by default. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.
- **Economy**.Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads. in general it is significantly more time consuming to create and manage processes than threads. In Solaris, for example, creating a process is about thirty times slower than is creating a thread, and context switching is about five times slower.

- **Scalability.** The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores. A single-threaded process can run on only one processor, regardless how many are available.

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

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



Advantages:

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

Disadvantages:

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

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

Advantages:

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

Disadvantages:

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

However, support for threads may be provided either at the user level, for **user threads**, or by the kernel, for **kernel threads**. User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system. Virtually all contemporary operating systems—including Windows, Linux, Mac OS X, and Solaris— support kernel threads.

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

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

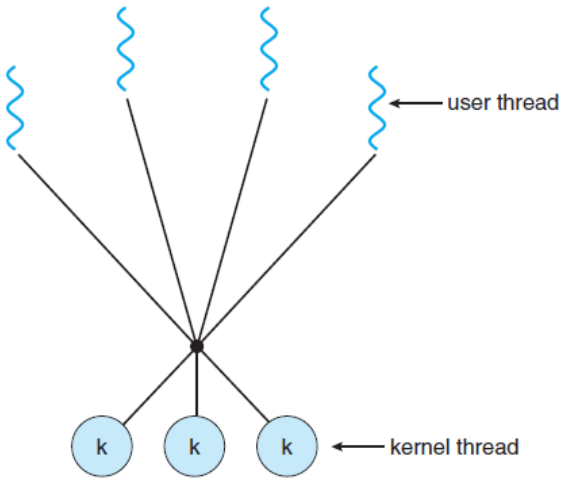


Figure 4.7 Many-to-many model.

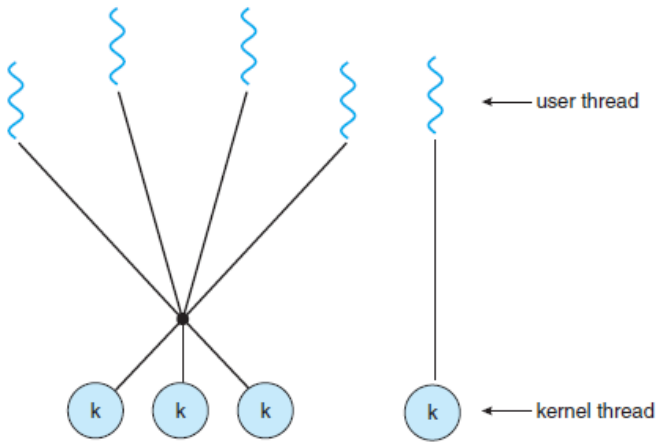


Figure 4.8 Two-level model.

One variation on the many-to-many model still multiplexes many userlevel threads to a smaller or equal number of kernel threads but also allows a user-level thread to be bound to a kernel thread. This variation is sometimes referred to as the **two-level model** as shown below. The Solaris operating system supported the two-level model in versions older than Solaris 9.

- **Many to One Model:** Many to one model maps many user level threads to one Kernel level thread. Thread management is done in user space. When thread makes a blocking system call, the entire process will be blocked. Only one thread can access the Kernel at a time,so multiple threads are unable to run in parallel on multiprocessors. If the user level thread libraries are implemented in the operating system in such a way that system does not support them then Kernel threads use the many to one relationship models.

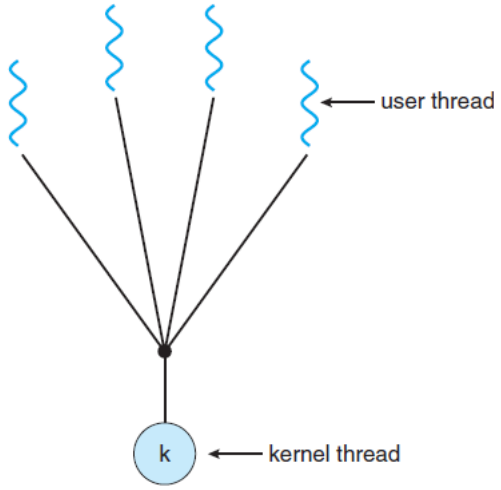


Figure 4.5 Many-to-one model.

- **One to One Model**: There is one to one relationship of user level thread to the kernel level thread.This model provides more concurrency than the many to one model. It also another thread to run when a thread makes a blocking system call. It support multiple thread to execute in parallel on microprocessors. Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, windows NT and windows 2000 use one to one relationship model.

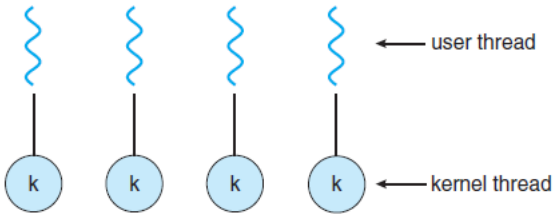


Figure 4.6 One-to-one model.

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

- **CPU Utilization:** In general, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).
- **Throughput:** One measure of work is the number of processes that are completed per time unit, called throughput. For long processes, this rate

may be one process per hour; for short transactions, it may be 10 processes per second.

- **Turnaround time:** The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
- **Waiting time:** Waiting time is the sum of the periods spent waiting in the ready queue.
- **Response time:** Response time, is the time it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device.

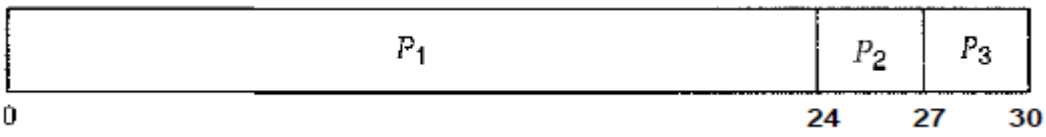
Process Scheduling Algorithms: CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. There are many different CPU scheduling algorithms as shown below.

1. **First Come First Serve (FCFS) Scheduling:** In this scheduling, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue.

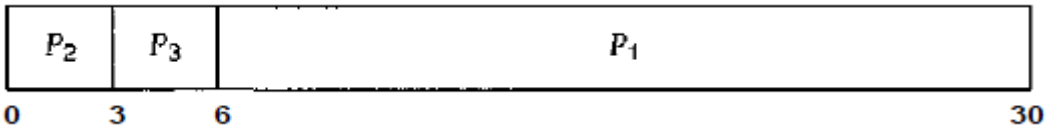
   Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

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



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



The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds. This reduction is substantial. Thus, the average waiting time under an FCFS policy

is generally not minimal and may vary substantially if the process's CPU burst times vary greatly.

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

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

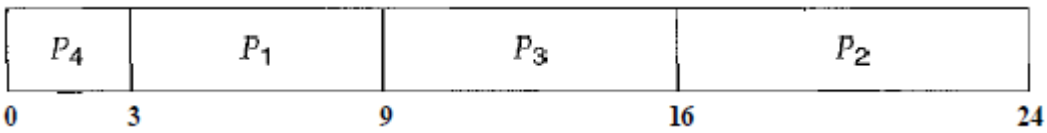
Disadvantages:

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

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

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

Using SJF scheduling, the following Gantt chart occurred.

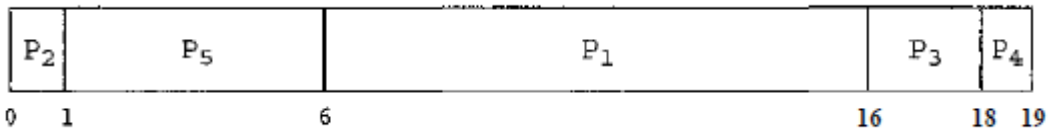


The waiting time is 3 milliseconds for process *P1,* 16 milliseconds for process *P2,* 9 milliseconds for process *P3,* and 0 milliseconds for process P4. Thus, the average waiting time is (3 + 16 + 9 + 0)/4 - *7* milliseconds.

3. **Priority Scheduling:** The SJF algorithm is a special case of the general **priority scheduling algorithm.** A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority. Consider the following set of processes, assumed to have arrived at time 0, in the order *P1, P2, P3, P4, P5* with the length of the CPU burst given in milliseconds:

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

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



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

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

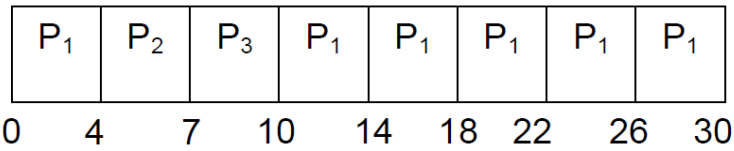
4. **Round Robin(RR) Scheduling:** The **round-robin (RR) scheduling algorithm** is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added to switch between processes. A small unit of time, called a **time quantum** or time slice, is defined. A time quantum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

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

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

Process	Burst Time
P_1	24
P_2	3
P_3	3

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



0 4 7 10 14 18 22 26 30

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

5. **Multilevel Queue Scheduling:** A **multilevel queue scheduling algorithm** partitions the ready queue into several separate queues as shown below. The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its own scheduling algorithm. In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. For example, the foreground queue may have absolute priority over the background queue. Let's look at an example of a multilevel queue scheduling algorithm with five queues, listed below in order of priority:

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

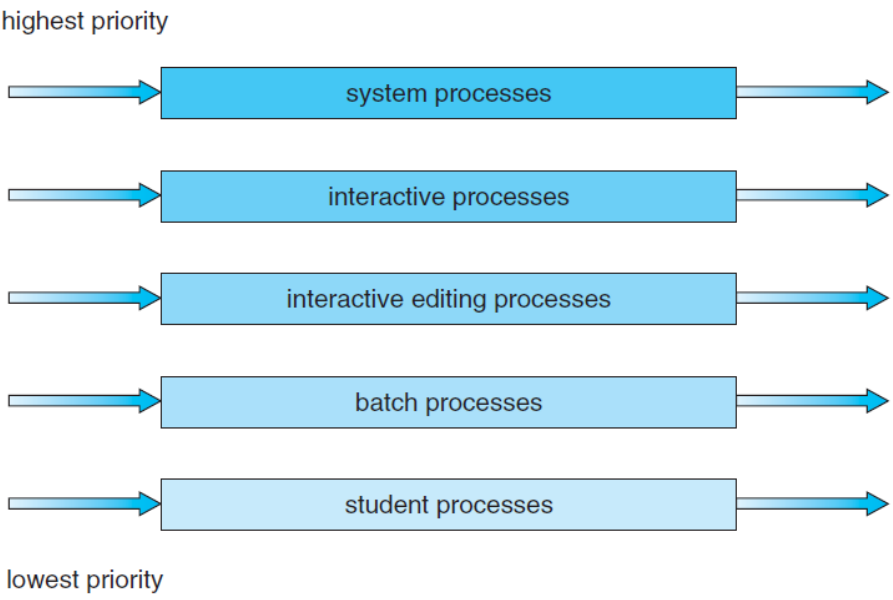


Figure 6.6 Multilevel queue scheduling.

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

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

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

   For example, consider a multilevel feedback-queue scheduler with three queues, numbered from 0 to 2 as shonw below. The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes in queue 2 will only be executed if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2. A process in queue 1 will in turn be preempted by a process arriving for queue 0.

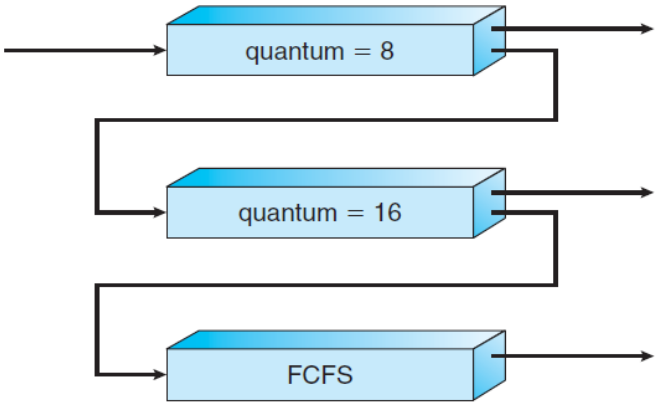


Figure 6.7 Multilevel feedback queues.

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

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

Algorithm Evaluation: How do we select a CPU-scheduling algorithm for a particular system? As we have many scheduling algorithms, each with its own parameters. As a result, selecting an algorithm can be difficult.

The first problem is defining the criteria to be used in selecting an algorithm. criteria are often defined in terms of CPU utilization, response time, or throughput. To select an algorithm, we must first define the relative importance of these elements. Our criteria may include several measures, such as these:

- Maximizing CPU utilization under the constraint that the maximum response time is 1 second
- Maximizing throughput such that turnaround time is (on average) linearly proportional to total execution time

Once the selection criteria have been defined, evaluate the algorithms under consideration. We next describe the various evaluation methods we can use. Evaluation methods are

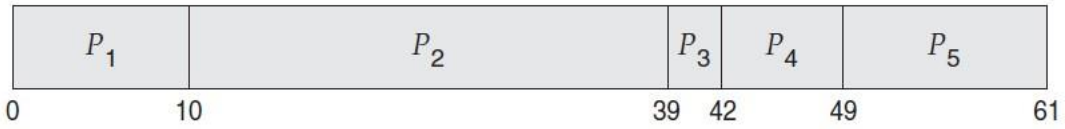
- **Deterministic Modeling:** One major class of evaluation methods is analytic evaluation. Analytic evaluation uses the given algorithm and the system workload to produce a formula or number that evaluates the performance of the algorithm for that workload.

  One type of analytic evaluation is deterministic modeling. This method takes a particular predetermined workload and defines the performance of

each algorithm for that workload. For example, assume that we have the workload shown below. All five processes arrive at time 0, in the order given, with the length of the CPU burst given in milliseconds:

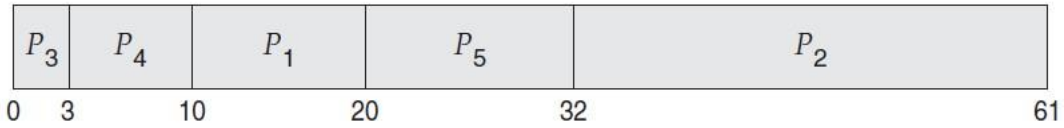
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

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



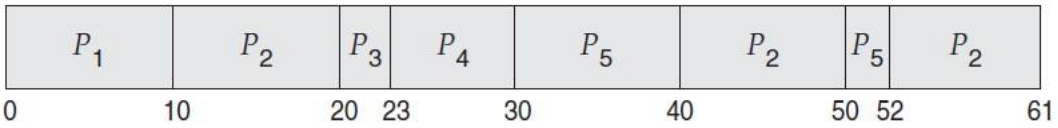
The waiting time is 0 milliseconds for process P1, 10 milliseconds for process P2, 39 milliseconds for process P3, 42 milliseconds for process P4, and 49 milliseconds for process P5. Thus, the average waiting time is (0 + 10 + 39 + 42 + 49)/5 = 28 milliseconds.

With nonpreemptive SJF scheduling, we execute the processes as



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

With the RR algorithm,



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

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

Deterministic modeling is simple and fast. It gives us exact numbers, allowing us to compare the algorithms. However, it requires exact numbers for input, and its answers apply only to those cases.

- **Queueing Models:** On many systems, the processes that are run vary from day to day, so there is no static set of processes (or times) to use for deterministic modeling. The computer system is described as a network of servers. Each server has a queue of waiting processes. The CPU is a server with its ready queue, as the I/O system with its device queues. By knowing arrival rates and service rates, we can compute utilization, average queue length, average wait time, and so on. This area of study is called queueing-network analysis.

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

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

  This equation, known as **Little's formula**, is particularly useful because it is valid for any scheduling algorithm and arrival distribution. We can use Little's formula to compute one of the three variables if we know the other two.

  For example, if we know that 7 processes arrive every second (on average) and that there are normally 14 processes in the queue, then we can compute the average waiting time per process as 2 seconds.
- **Simulation:** To get a more accurate evaluation of scheduling algorithms, use simulations. The simulator has a variable representing a clock; as this variable's value is increased, the simulator modifies the system state to reflect the activities of the devices, the processes, and the scheduler. As the simulation executes, statistics that indicate algorithm performance are gathered and printed.

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

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

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

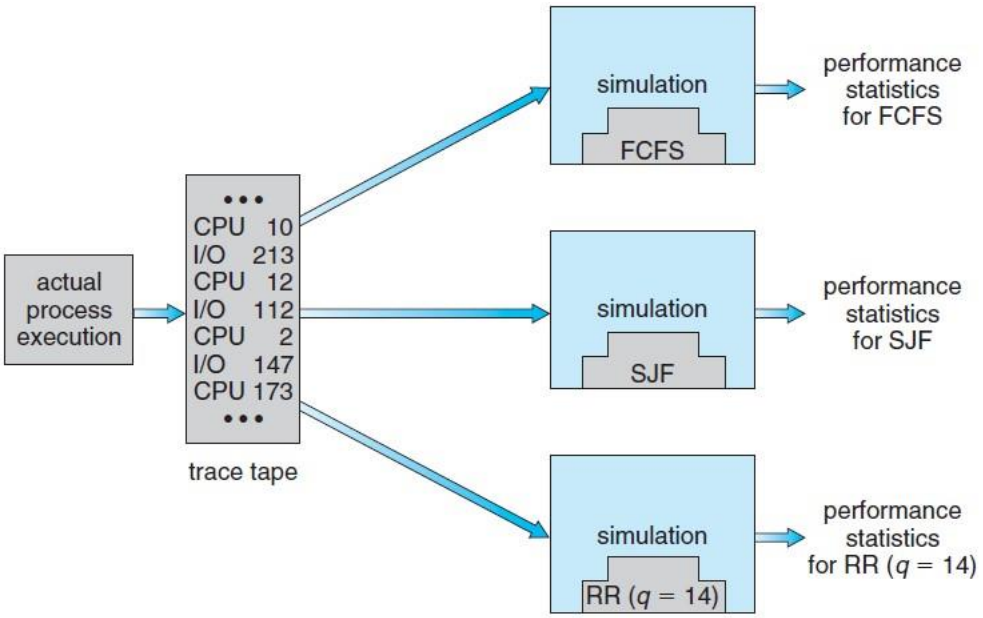


Figure 6.25 Evaluation of CPU schedulers by simulation.

FREQUENTLY ASKED QUESTIONS

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

UNIT-3

Syllabus: **Concurrency:** Process synchronization, the critical-section problem, Peterson's Solution, synchronization Hardware, semaphores, classic problems of synchronization, monitors.

Principles of deadlock – system model, deadlock characterization, Methods for Handling Deadlocks: deadlock prevention, detection and avoidance, recovery form deadlock.

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

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

Figure 5.1 General structure of a typical process P_i.

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

- **Mutual exclusion:** If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
- **Progress:** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can enter its critical section next.
- **Bounded waiting:** There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

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

int turn;

boolean flag [2];

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

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

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

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

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

Figure 5.2 The structure of process P_i in Peterson's solution.

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

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

will reset f lag[j] to false, allowing Pi to enter its critical section. If *Pj* resets flag [j] to true, it must also set turn to i.

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

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

Solution to the critical-section problem using locks.

The TestAndSet() instruction can be used to solve Critical Section problem. The important characteristic here is that this instruction is executed atomically. Thus, if two TestAndSet() instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order. If the machine supports the TestAndSet () instruction, then we can implement mutual exclusion by declaring a Boolean variable lock, initialized to false. The structure of process P_i is shown below.

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

Figure 5.3 The definition of the `test_and_set()` instruction.

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

Figure 5.4 Mutual-exclusion implementation with `test_and_set()`.

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

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

The definition of the Swap () instruction.

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

      // critical section

   lock = FALSE;

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

Mutual-exclusion implementation with the Swap() instruction.

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

boolean waiting[n];

boolean lock;

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

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&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 5.7 Bounded-waiting mutual exclusion with `test_and_set()`.

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

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

The definition of signal () is as follows:

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

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

There are two types of semaphores: binary semaphore and counting semaphore. The value of a **counting semaphore** can range over an unrestricted domain. The value of a **binary semaphore** can range only between 0 and 1. On some systems, binary semaphores are known as **mutex locks,** as they are locks that provide *mutualexclusion.*

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

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

Mutual-exclusion implementation with semaphores.

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a wait( ) operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a signal( ) operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

Semaphores are also used to solve various synchronization problems. For example, consider two concurrently running processes: *P1* with a statement S1 and *P2* with a statement *S2*. Suppose we require that *S2* be executed only after S1 has completed. We can implement this scheme readily by letting P1 and P*2* share a common semaphore **synch**, initialized to 0, and by inserting the statements

S1;

signal(synch);

in process *P1,* and the statements

wait(synch);

S2;

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

Implementation:

The main disadvantage of the semaphore definition given here is that it requires **busy waiting.** While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a **spinlock** because the process "spins" while waiting for the lock.

When a process executes the wait( ) operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can *block* itself. The block operation places a process into a waiting

queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal( ) operation. The process is restarted by a wakeup( ) operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue.

The following C code implements this.

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

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

The wait( ) semaphore operation can now be defined as

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

The signal( ) semaphore operation can now be defined as

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

The block( ) operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process P.

Deadlocks and Starvation: The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. When such a state is reached, these processes are said to be **deadlocked.**

For example, consider a system consisting of two processes, P0 and P1, each accessing two semaphores, S and Q, set to the value 1: Suppose that *P0* executes wait (S) and then P1 executes wait (Q). When P0 executes wait(Q), it must wait until P1 executes signal(Q). Similarly, when P1 executes wait(S), it must wait until P0 executes signal(S). Since these signal () operations cannot be executed, P0 and P1 are deadlocked.

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

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

Classic Problems of Synchronization:

The Bounded-Buffer Problem: In this problem the pool consists of *n* buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0. The code for the producer and consumer processes is shown below.

In our problem, the producer and consumer processes share the following data structures:

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

The code for the producer process is shown below

```
do {
    . . .
   /* produce an item in next_produced */
    . . .
   wait(empty);
   wait(mutex);
    . . .
   /* add next_produced to the buffer */
    . . .
   signal(mutex);
   signal(full);
} while (true);
```

Figure 5.9 The structure of the producer process.

The code for the consumer process is shown below

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

Figure 5.10 The structure of the consumer process.

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

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

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

semaphore mutex, wrt;

int readcount;

The semaphores mutex and wrt are initialized to 1; readcount is initialized to 0. The semaphore wrt is common to both reader and writer processes. The mutex semaphore is used to ensure mutual exclusion when the variable readcount is updated. The readcount variable keeps track of how many processes are currently reading the object. The semaphore wrt functions as a mutual-exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical sections. The structure of reader and writer processes are shown below.

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

The structure of a writer process.

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

The structure of a reader process.

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

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

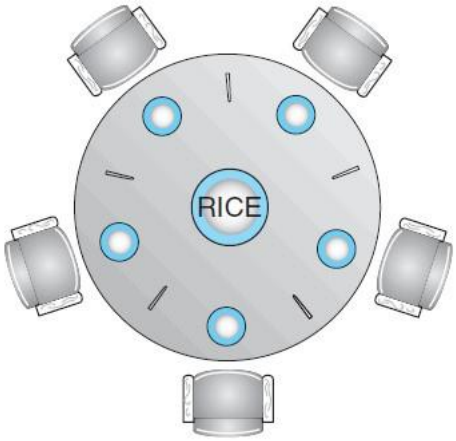


Figure 5.13 The situation of the dining philosophers.

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

semaphore chopstick[5];

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

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for awhile */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
} while (true);
```

Figure 5.14 The structure of philosopher *i*.

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

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

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

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

  wait(mutex);
  //critical section

wait(mutex);

In this case, a deadlock will occur.

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

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

A Monitor is a type, or abstract data type presents a set of programmer-defined operations that are provided mutual exclusion within the monitor. The monitor type also contains the declaration of variables, along with the bodies of procedures or functions. The syntax of a monitor is shown below.

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

   function P1 ( . . . ) {
      . . .
   }

   function P2 ( . . . ) {
      . . .
   }

         .
         .
         .
   function Pn ( . . . ) {
      . . .
   }

   initialization_code ( . . . ) {
      . . .
   }
}
```

Figure 5.15 Syntax of a monitor.

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

condition x, y;

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

x.wait() ;

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

x.signal() ;

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

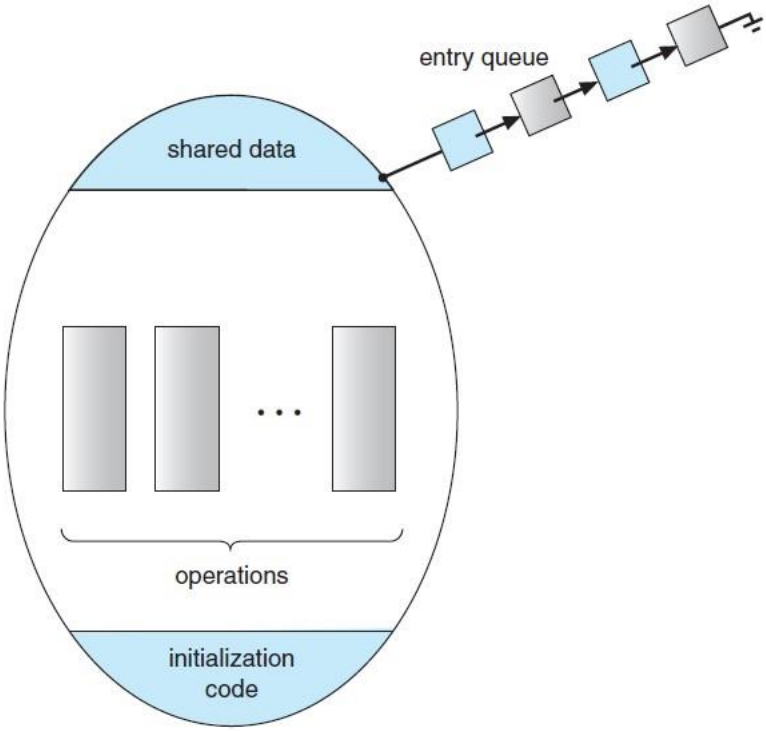


Figure 5.16 Schematic view of a monitor.

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

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

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

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

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

Here, Philosopher *i* can set the variable state[i] = eating only if her two neighbors are not eating: (state [(i+4) % 5] != eating) and (state [(i+1) % 5] != eating).

condition self [5];

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

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

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

```
monitor DiningPhilosophers
{
   enum {THINKING, HUNGRY, EATING} state[5];
   condition self[5];

   void pickup(int i) {
      state[i] = HUNGRY;
      test(i);
      if (state[i] != EATING)
         self[i].wait();
   }

   void putdown(int i) {
      state[i] = THINKING;
      test((i + 4) % 5);
      test((i + 1) % 5);
   }

   void test(int i) {
      if ((state[(i + 4) % 5] != EATING) &&
       (state[i] == HUNGRY) &&
       (state[(i + 1) % 5] != EATING)) {
         state[i] = EATING;
         self[i].signal();
      }
   }

   initialization_code() {
      for (int i = 0; i < 5; i++)
         state[i] = THINKING;
    }
}
```

Figure 5.18 A monitor solution to the dining-philosopher problem.

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

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

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

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

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

The operation x. signal () can be implemented as

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

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

x.wait(c);

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

R.acquire(t);

//access the resource;

R. release() ;

where R is an instance of type ResourceAllocator.

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

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

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

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

Figure 5.19 A monitor to allocate a single resource.

FREQUENTLY ASKED QUESTIONS

1. What is a Critical-section Problem? Give the conditions that a solution to the critical section problem must satisfy? Or Discuss solution to the critical section problem?
2. What is a Semaphore? Also give the operations for accessing semaphores?
3. Distinguish between counting and binary semaphore?
4. Define semaphore? Explain the usage and implementation of semaphores.
5. Describe disadvantages of the semaphore.
6. What is a semaphore? List the types of semaphores and show that if the wait() and signal() semaphore operations are not executed atomically, then mutual exclusion may be violated?
7. Give the Peterson's solution to the Critical-Section Problem?
8. What is Dining Philosophers Problem? Discuss the solution to Dining Philosophers problem using Monitors?
9. Discuss the Bounded-Buffer Problem?
10. Briefly explain the Readers-Writers Problem? (Or) Explain in detail about readers-writers problem of synchronization?
11. Define busy waiting? How to overcome it? (Or) Explain wait and signal semaphore operations without busy waiting.
12. State the Critical Section Problem. Illustrate the software based solution to the Critical Section Problem?

13. How does the signal() operation associated with monitors differ from the corresponding operation defined for semaphores?
14. Define monitor? Explain how it overcomes the drawback of semaphores?
15. What is Synchronization? Explain how semaphores can be used to deal with n-process crictical section problem?
16. Discuss mutual exclusion implementation with test() and set() instruction?
17. Explain in detail Synchronization implementation in Linux.
18. What is a Monitor? Give the schematic view of the basic monitor.

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

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

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources are partitioned into several types, each consisting of some number of identical instances. Under the normal mode of operation, a process may utilize a resource in only the following sequence:

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

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

Necessary Conditions: A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. **Mutual exclusion.** At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. **Hold and wait.** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
3. **No preemption.** Resources cannot be preempted.; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
4. **Circular wait:** A set *{P0, P0, …, Pn}* of waiting processes must exist such that *P0* is waiting for a resource held by *P1*, *P1* is waiting for a resource held by P2, •••, P_{n-1} is waiting for a resource held by *Pn,* and *Pn,* is waiting for a resource held by P0.

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

A directed edge from process P_i to resource type *Rj* is denoted by $P_i \rightarrow Rj$; it signifies that process P_i has requested an instance of resource type Rj and is

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

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

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

Consider the following resource allocation graph, containing

➔ The sets P, *R,* and E:

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

➔ Resource instances:

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

➔ Process states:

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

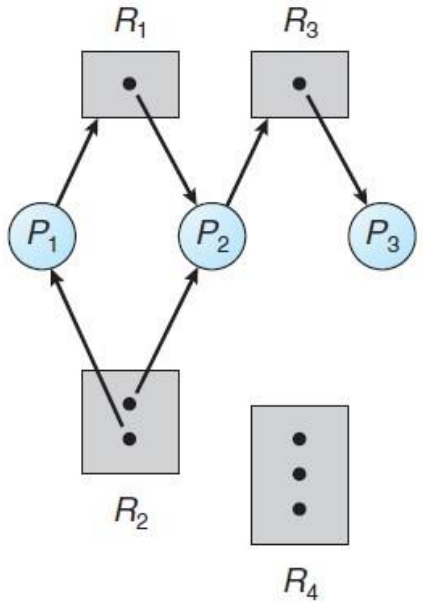


Figure 7.1 Resource-allocation graph.

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

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

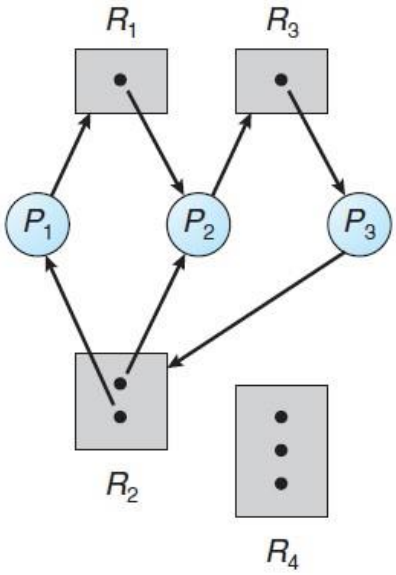


Figure 7.2 Resource-allocation graph with a deadlock.

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

P1→R1→P2→R3→P3→R2→P1

P2→R3→P3→R2→P2

Here, Processes P1, P2, and P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by process P3. Process P3 is waiting for either process P1 or P2 to release resource R2. In addition, process P1 is waiting for process P2 to release resource R1.

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

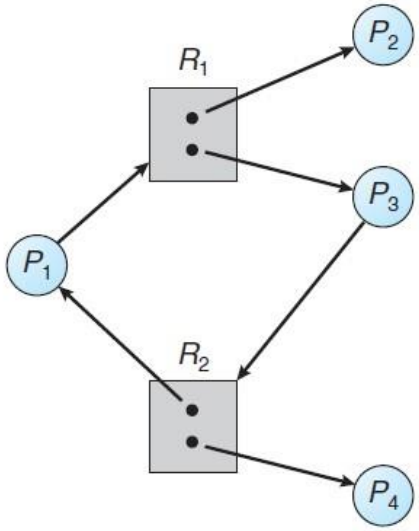


Figure 7.3 Resource-allocation graph with a cycle but no deadlock.

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

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

Advantages of Deadlock Prevention:

- No preemption necessary
- Needs no runtime computation
- Feasible to enforce via compile time checks.

- Works well processes that perform a single burst of activity.

Disadvantages of Deadlock Prevention:

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

Deadlock Avoidance: A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circularwait condition can never exist. The resource-allocation *state* is defined by the number of available and allocated resources and the maximum demands of the processes.

Safe state: A state is *safe* if the system can allocate resources to each process in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence. A sequence of processes $<P_1, P_2, ..., P_n>$ is a safe sequence for the current allocation state if, for each P_i, the resource requests that P_i can be satisfied by the currently available resources plus the resources held by all P_j, with $j < i$. In this situation, if the resources that *Pi* needs are not immediately available, then P_i can wait until all P_j have finished. When they have finished, P_j can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When P_i terminates, P_{i+1} can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be *unsafe.*

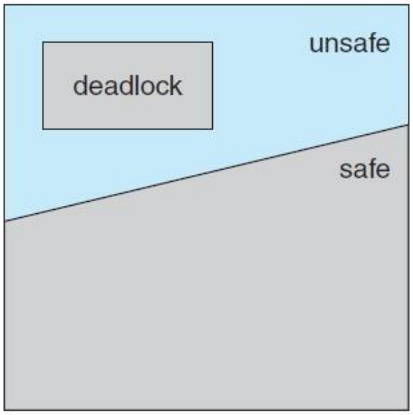


Figure 7.6 Safe, unsafe, and deadlocked state spaces.

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

Consider a system with 12 magnetic tape drives and three processes: P_0, P_1, and P_2. Process P_0 requires 10 tape drives, process P_1 may need as many as 4 tape drives, and process P_2 may need up to 9 tape drives. Suppose that, at time t_o, process P_0 is holding 5 tape drives, process P_1 is holding *2* tape drives, and process P_2 is holding 2 tape drives. (Thus, there are 3 free tape drives.)

	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

At time T_0, the system is in a safe state. The sequence < P_1, P_0, P_2> satisfies the safety condition. Process P_1 can immediately be allocated all its tape drives and then return them (the system will then have 5 available tape drives); then process P_0 can get all its tape drives and return them (the system will then have 10 available tape drives); and finally process P_2 can get all its tape drives and return them (the system will then have all 12 tape drives available).

A system can go from a safe state to an unsafe state. Suppose that, at time T_1, process P_2 requests and is allocated one more tape drive. The system is no longer in a safe state. At this point, only process P_1 can be allocated all its tape drives. When it returns them, the system will have only 4 available tape drives. Since process P_0 is allocated 5 tape drives but has a maximum of 10, it may request 5 more tape drives. Since they are unavailable, process P_0 must wait. Similarly, process P_2 may request an additional 6 tape drives and have to wait, resulting in a deadlock. Here, mistake was in granting the request from process P_2 for one more tape drive. If we had made P_2 wait until either of the other processes had finished and released its resources, then we could have avoided the deadlock. The following are two different types of deadlock avoidance algorithms.

1. **Resource-Allocation-Graph Algorithm:** A claim edge $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j 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 P_i requests resource R_j, the claim edge $P_i \rightarrow R_j$ is converted to a request edge. Similarly, when a resource R_j is released by P_j, the assignment edge $R_j \rightarrow P_i$ is reconverted to a claim edge $P_i \rightarrow R_j$.

   Suppose that process P_i requests resource R_j. The request can be granted only if converting the request edge $P_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow P_i$ does not result in the formation of a cycle in the resource-allocation graph. An algorithm for detecting a cycle in this graph requires an order of n^2 operations, where n is the number of processes in the system.

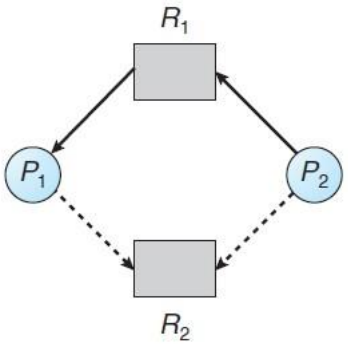


Figure 7.7 Resource-allocation graph for deadlock avoidance.

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

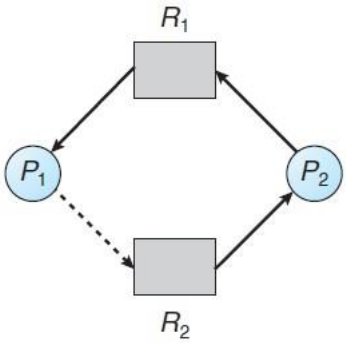


Figure 7.8 An unsafe state in a resource-allocation graph.

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

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

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

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

Safety Algorithm:

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

Resource-Request Algorithm: Let $Request_i$ be the request vector for process P_i. If *Request[j]* = *k*, then
process *Pi* wants *k* instances of resource type *Rj*. When a request for resources is made by process *Pi*, the following actions are taken:

1. If $Request_i$ < *Need*, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If $Request_i$ < *Available*, go to step 3. Otherwise, P_i must wait, since the resources are not available.
3. *Available* = *Available* – $Reques_i$;
   $Allocation_i = Allocation_i + Request_i$
   $Need_i = Necd_i - Request_i$

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

	Allocation	*Max*	*Available*
	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* and is as follows:

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

Suppose now that process P_0 requests 7 additional instance of resource type *A* , 4 instances of resource type B and 3 instances of resource type C. Now check for Need <= Work(Avaialable) for all the values of i.

- ➔ For process P_0, Since (7, 4, 3) <= (3, 3, 2) is false. Make process P_0 to wait.
- ➔ For process P_1, Since (1, 2, 2) <= (3, 3, 2) is true. Process it. After completion of P_1, it releases all the releases it held. So, available is updated as follows:

  Aaialable = Available + $Allocation_i$

  Available = (3, 3, 2) + (2, 0, 0) = (5, 3, 2)
- ➔ For process P2, Since (6, 0 ,0) <= (5, 3, 2) is False, Make P2 to wait.
- ➔ For process P3, Since (0, 1 ,1) <= (5, 3, 2) is True, Process it. After completion of P3, it releases all the resources it held. So, Available is updated as follows.

  Available = (5, 3, 2) + (2, 1, 1) = (7, 4, 3)
- ➔ For process P4, Since (4, 3 ,1) <= (7, 4, 3) is True, Process it. After completion of P4, it releases all the resources it held. So, Available is updated as follows.

  Available = (7, 4, 3) + (0, 0, 2) = (7, 4, 5)
- ➔ For process P0, Since (7, 4 ,3) <= (7, 4, 5) is True, Process it. After completion of P0, it releases all the resources it held. So, Available is updated as follows.

  Available = (7, 4, 5) + (0, 1, 0) = (7, 5, 5)
- ➔ For process P2, Since (6, 0 ,0) <= (7, 5, 5) is True, Process it. After completion of P2, it releases all the resources it held. So, Available is updated as follows.

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

Deadlock detection: If a system does not empliy either a deadlock prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system must provide:

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

The following are different types of deadlock detection algorithms:

1. **Single Instance of Each Resource Type (Wait-For Graph):** Wait-For graph is an sibling of Resource-Allocation Graph algorithm. More precisely, an edge from P_i to P_j in a Wait-For graph implies that process P_i is waiting for

process P_j to release a resource that P_i needs. An edge $P_i \rightarrow P_j$ exists in a wait-fpr graph if and only the corresponding resource allocation graph contains two edges $P_i \rightarrow R_j$ and $R_j \rightarrow P_j$ for more resource. For example,

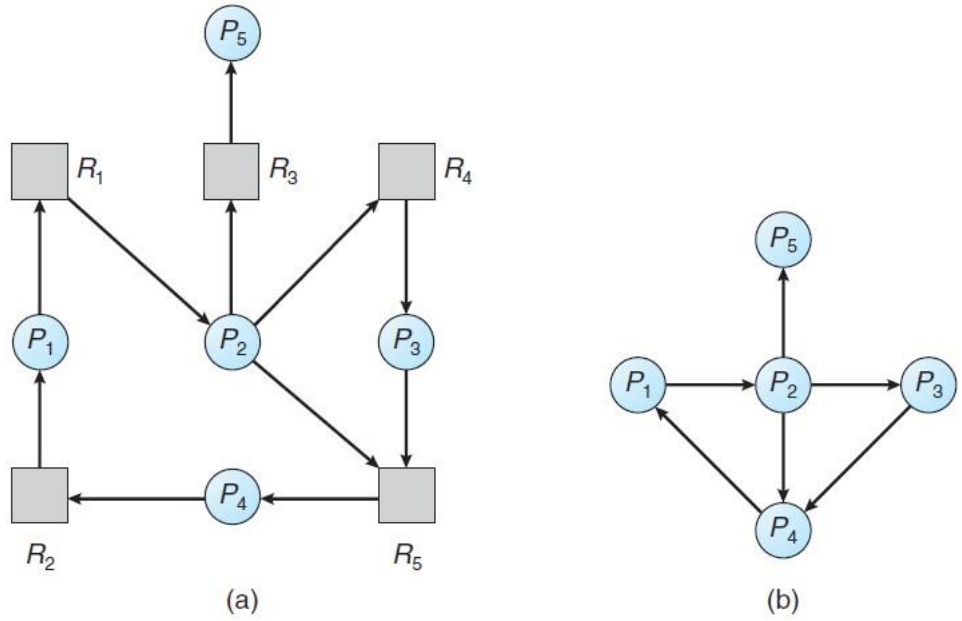


Figure 7.9 (a) Resource-allocation graph. (b) Corresponding wait-for graph.

A deadlock exists in the system if and only if the wait-for graph contains a cycle. To detect deadlocks, the system needs to maintain the wai-for graph and periodically invoke an algorithm that's earches for a cycle in the graph. An algorithm to detect a cyckle in a graph requires an order of n^2 operstions, where n is the number of vertices in the graph.

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

Algorithm:

1. Let *Work* and *Finish* be vectors of length *m* and *n,* respectively. Initialize *Work= Available. Fori*= 0, 1, … , *n-1,* if $Allocation_i$ # 0, then *Finish[i] =false;* otherwise, *Finish[i] = true.*
2. Find an index *i* such that both
   Finish[i] ==false
   $Request_i$ < *Work*
   If no such *i* exists, go to step 4.
3. *Work= Work+* $Allocation_i$
   Finish[i] = true
   Go to step 2.

4. If *Finish[i] ==false* for some *i,* 0 < *i* < *n,* then the system is in a deadlocked state. Moreover, if *Finish[i] ==false,* then process P_i is deadlocked.

This algorithm requires an order of m x n^2 operations to detect whether the system is in a deadlocked state. Consider a system with 5 processes P0 through P4 and three resource types A, B and C. Resource type A has 7 instances, B has two instances, and C has 6 instances. Suppose that, at time T_0, we have the following resoutce 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	

For above request algorithm leaves system in safe state with safe sequence <P0, P2, P3, P4, P1>. Suppose now that process P_2 makes one additional request for an instance of type C. The Request matrix is modified as follows, which leaves the system in unsafe state.

	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

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

1. Process Termination:

- Abort all deadlocked processes: This method clearly will break the deadlock cycle, but at great expense; the deadlocked processes may have computed for long time, and result of these partial computations must be discarded and probably will have to be recomputed later.
- Abort one process at a time until the deadlock cycle is eliminated: This method incurs considerable overhead, since, after each process is aborted, a deadlock detection algorithm must be invoked to determine whether any processes are still deadlocked.

2. **Resource Preemption:** To eliminate deadlocks using resource preemption, successively preempt some resources from processes and give these resources to othee processes until the deadlock cycle is broken.

FREQUENTLY ASKED QUESTIONS

1. What are the necessary conditions for the occurance of deadlock?
2. Write about characterization of deadlock by resource allocation graph?
3. Explain methods used for ensuring atleast one of the necessary conditions cannot hold?
4. What is a Deadlock? How to detect deadlock?
5. Explain Resouce Allocation Graph algorithm for Deadlock avoidance?
6. What are the various methods for deadlock avoidance?
7. Discuss various techniques to recover from deadlock?
8. How deadlock avoidance differ from deadlock prevention?
9. Is it possible to have a deadlock involving only a single process? Explain.
10. Explain about Deadlock avoidance algorithms in detail?
11. Is it possible to have a deadlock involving only a single process? Explain?
12. Write about Resource Allocation Graph?
13. State and explain two ways used for handling deadlocks?
14. Discuss various methods for the prevention of deadlocks?
15. Consider the snapshot of the system processes P1, P2, P3, P4, P5, resources A, B, C, D. Allocation {0 0 1 2, 1 0 0 0, 1 3 5 4, 0 6 3 2, 0 0 1 4}
    Max {0 0 1 2, 1 7 5 0, 2 3 5 6, 0 6 5 2, 0 6 5 6}
    Available {1 5 2 0}
    What is the content of Need matrix? And find out is the system in safe state?
16. Write about characterization of deadlock by resource allocation graph?
17. Consider the snapshot of the system processes P0, P1, P2, P3, P4, resources A, B, C.
    Allocation {0 1 0, 2 0 0, 3 0 3, 2 1 1, 0 0 2}
    Max {0 0 0, 2 0 2, 0 0 0, 1 0 0, 0 0 2}
    Available {0 0 0}
    What is the content of Need matrix? And find out is the system in safe state?
18. Explain various methods used to recover from deadlock after detection algorithm detects the deadlock? (Or) Explain various ways of aborting a process in order to eliminate a deadlock?

UNIT – 4

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

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

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

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

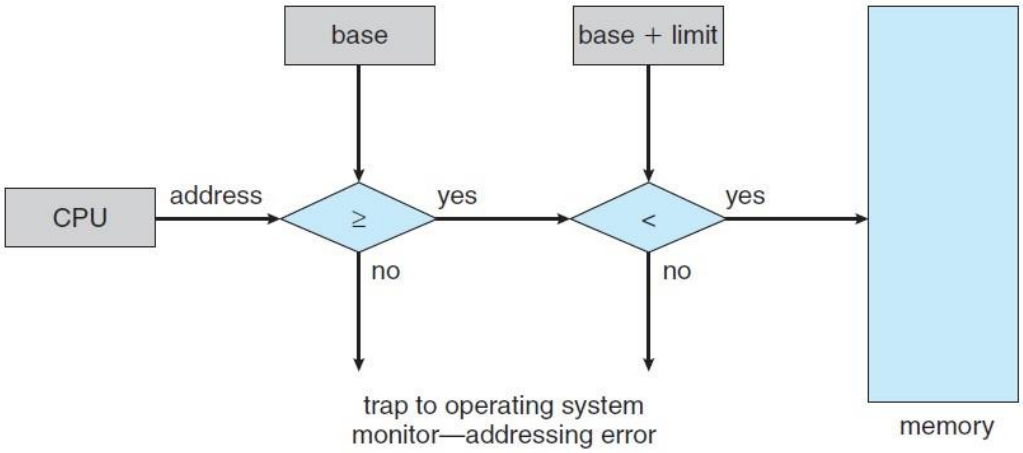


Figure 8.2 Hardware address protection with base and limit registers.

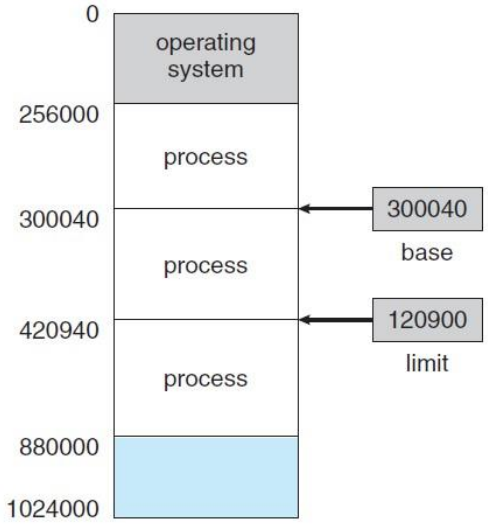


Figure 8.1 A base and a limit register define a logical address space.

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

- **Compile Time:** If you know at compile time where the process will reside in memory, then **absolute code** can be generated. For example, if you know that a user process will reside starting at location *R,* then the generated compiler code will start at that location and extend up from there.
- **Load time:** If it is not known at compile time where the process will reside in memory, then the compiler must generate **relocatable code. In** this case, final binding is delayed until load time.
- **Execution time.** If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.

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

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

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

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

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

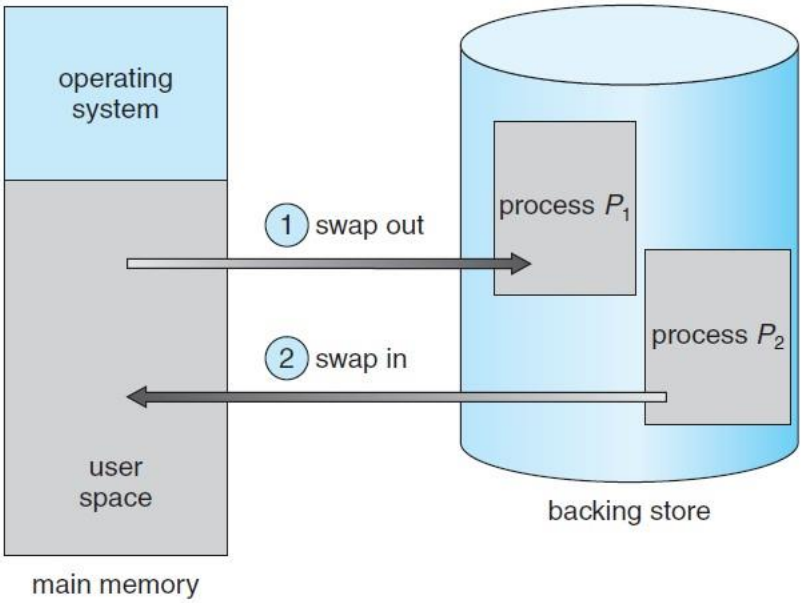


Figure 8.5 Swapping of two processes using a disk as a backing store.

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

Swapping requires a backing store. The backing store is commonly a fast disk. Whenever the CPU scheduler decides to execute a process, it calls the dispatcher. The dispatcher checks to see whether the next process in the queue is in memory. If it is not, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process. It then reloads registers and transfers control to the selected process.

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

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

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

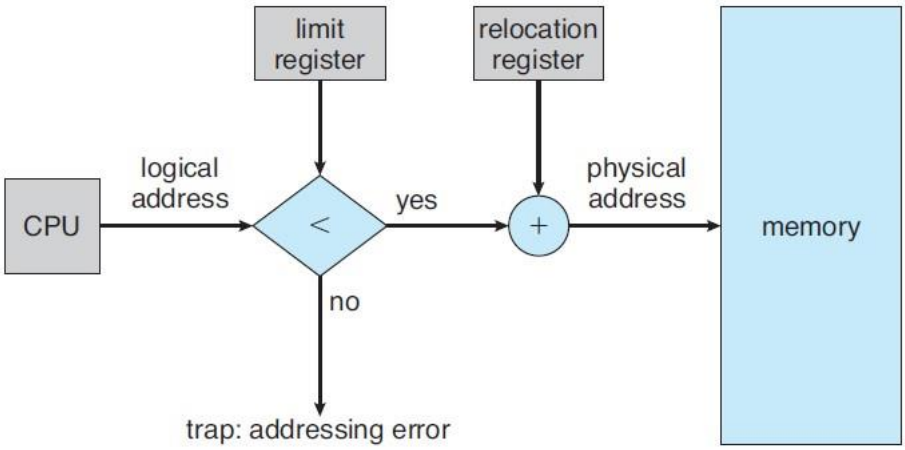


Figure 8.6 Hardware support for relocation and limit registers.

When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch.

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

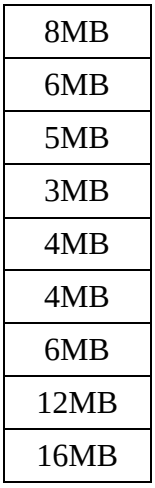
1. **Fixed partitioning**
2. **Dynamic sized partitioning (or) Variable sized Partitioning**

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

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

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

Equal Size Partitions



Unequal size partitions

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

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

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

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

→ Main memory utilization is extremely inefficient.

Advantages and disadvantages of Fixed Partition size:

Advantages:

→ Simple to implement

→ Less overhead

Disadvantages:

→ Memory is not efficiently used

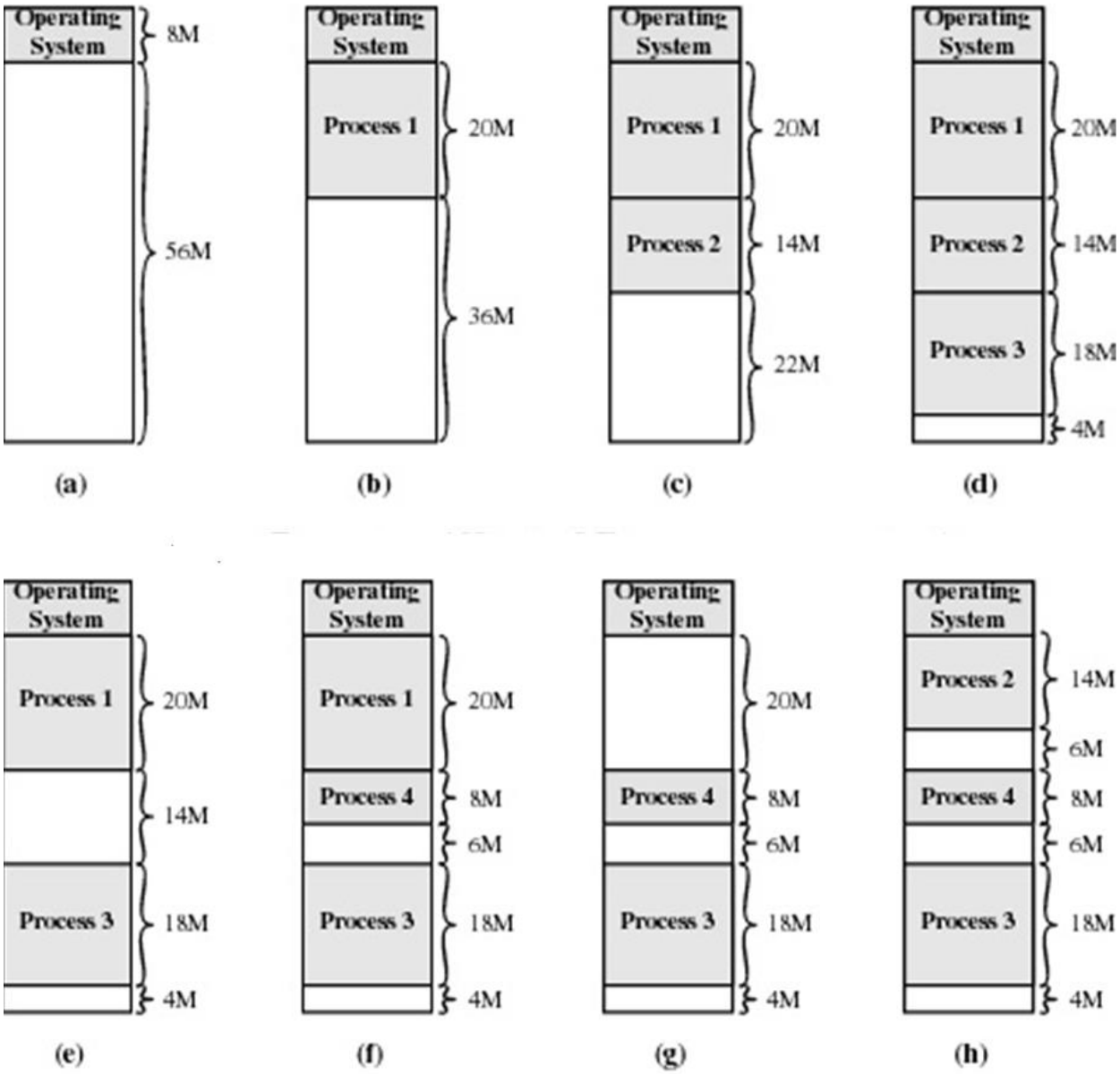
→ Poor utilization of Processors

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

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

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

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



The Effect of Dynamic Partitioning

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

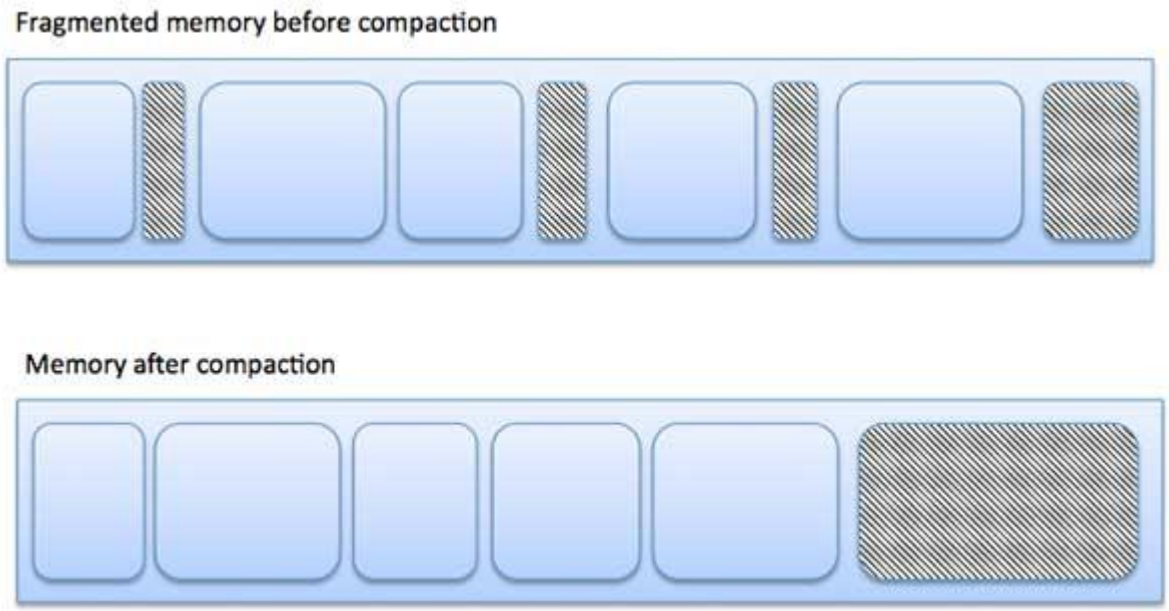
- **First fit:** Allocate the *first* hole that is big enough. Searching can start either at the beginning of the set of holes or where the previous first-fit search ended.
- **Best fit:** Allocate the *smallest* hole that is big enough. Search the entire list, unless the list is ordered by size. This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

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

- **Internal Fragmentation:** Unused memory which is internal to a partition is called as Internal Fragmentation. The internal fragmentation can be reduced by effectively assigning the smallest partition but large enough for the process.
- **External Fragmentation:** External fragmentation exists when there is enough total memory space to satisfy a request, but the available spaces are not contiguous. One solution to the problem of external fragmentation is **Compaction**.

  Compaction is the process of shuffling the memory contents so as to place all free memory together in one large block. To make compaction feasible, relocation should be dynamic.

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



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

Differences between Internal and External Fragmentation:

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

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

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

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

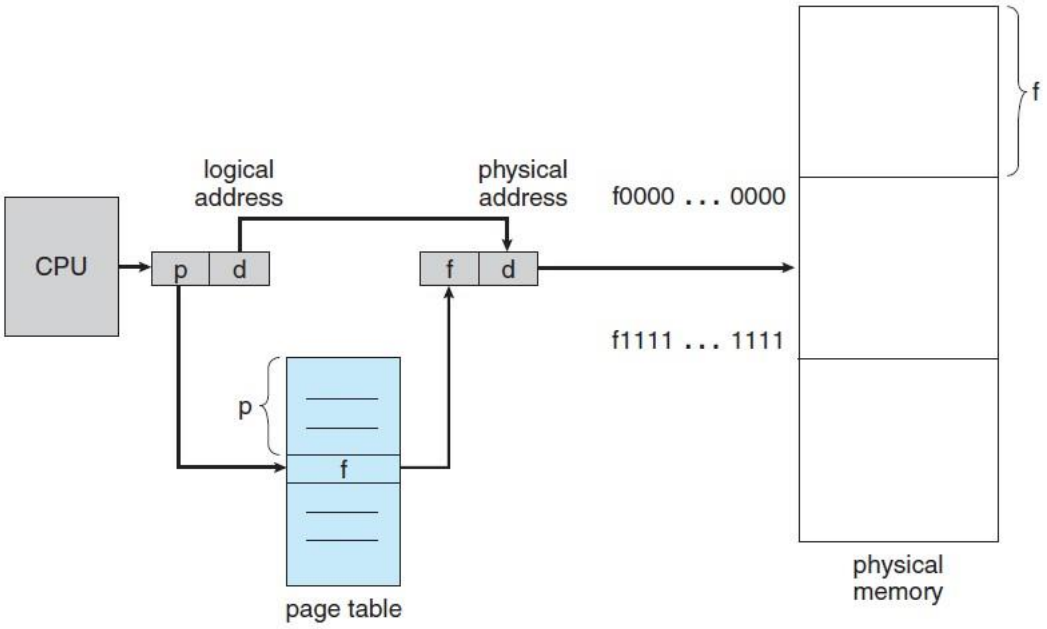


Figure 8.10 Paging hardware.

Every address generated by the CPU is divided into two parts: a **page number (p)** and a **page offset (d).** The page number is used as an index into a **page table.** The page table contains the base address of each page in physical memory. This

base address is combined with the page offset to define the physical memory address that is sent to the memory unit. The paging model of memory is shown in below figure.

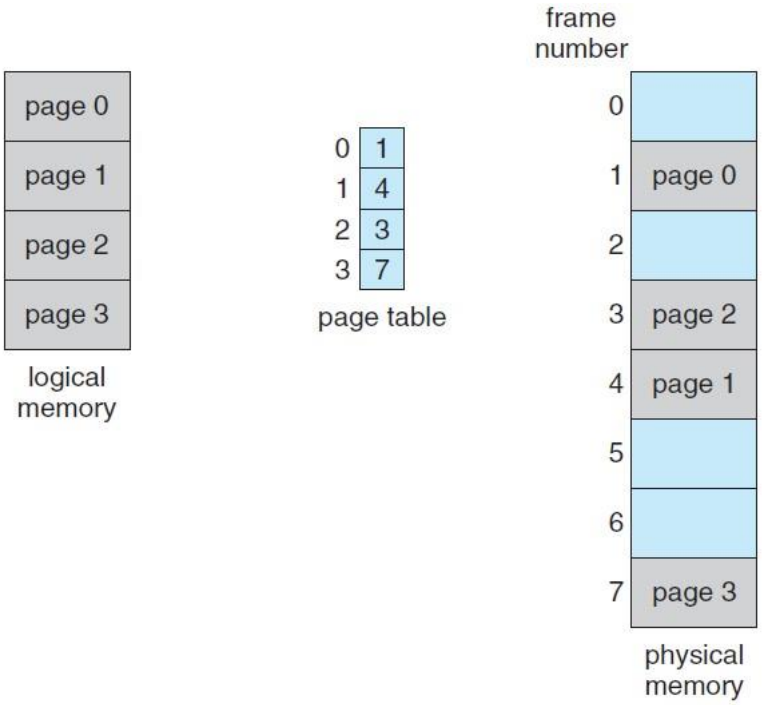
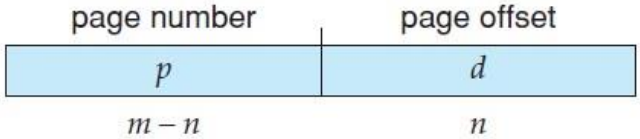


Figure 8.11 Paging model of logical and physical memory.

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 16MB per page, depending on the computer architecture. 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:



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

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

Paging example for a 32-byte memory with 4-byte pages. Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 (= (5 x 4) + 0). Logical address 3 (page 0, offset 3) maps to physical address 23 (= (5x4) + 3). Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 (= (6x4) + 0). Logical address 13 maps to physical address 9.

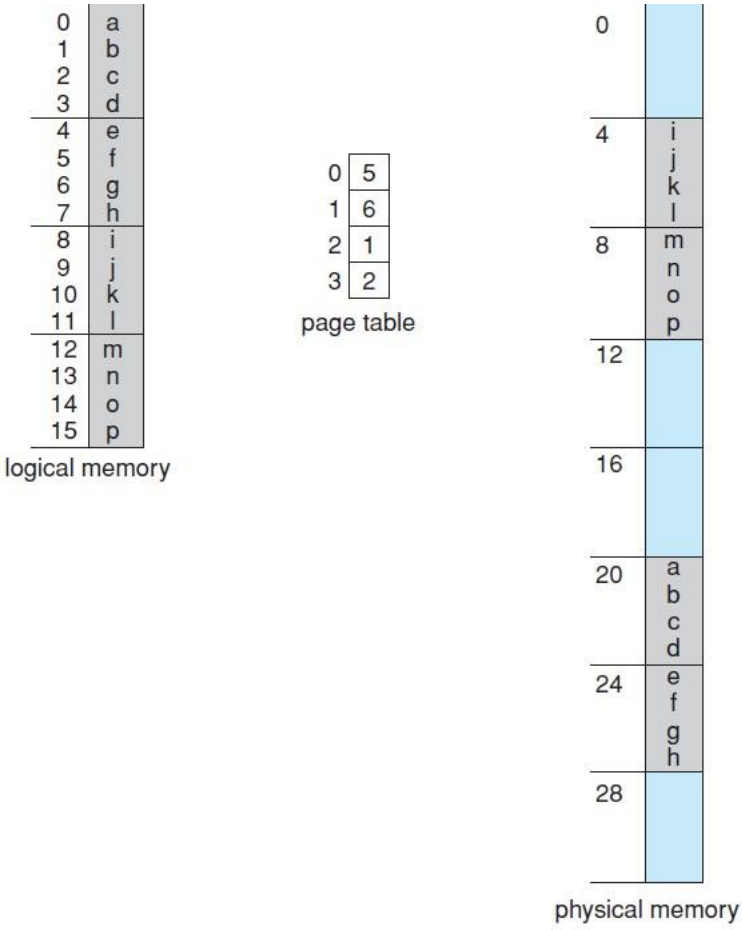


Figure 8.12 Paging example for a 32-byte memory with 4-byte pages.

Hardware Support: As size of the page table increases, memory access time increases. Such that performance of a computer decreases. To overcome this problem, use a special, small, fastlookup hardware cache, called a translation look-aside buffer (TLB). The TLB is associative, high-speed memory. Each entry in the TLB consists of two parts: **a key (or tag) and a value**. When the associative memory is presented with an item, the item is compared with all keys simultaneously. If the item is found, the corresponding value field is returned.

The TLB is used with page tables in the following way. The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, its page number is presented to the TLB. If the page number is found(TLB Hit), its frame number is immediately available and is used to access memory.

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

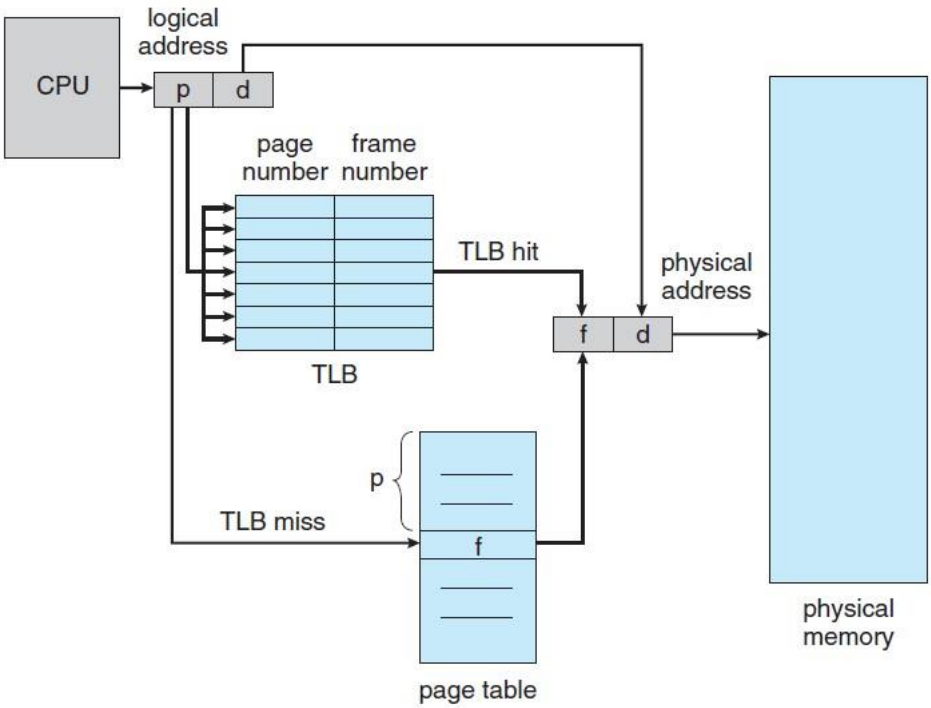


Figure 8.14 Paging hardware with TLB.

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

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

Protection: Memory protection in a paged environment is accomplished by **protection bits** associated with each frame. Normally, these bits are kept in the page table. One additional bit is generally attached to each entry in the page table: a **valid-invalid** bit. When this bit is set to "valid," the associated page is in the process's logical address space and is thus a legal (or valid) page. When the bit is set to"invalid,'" the page is not in the process's logical address space. The operating system sets this bit for each page to allow or disallow access to the page.

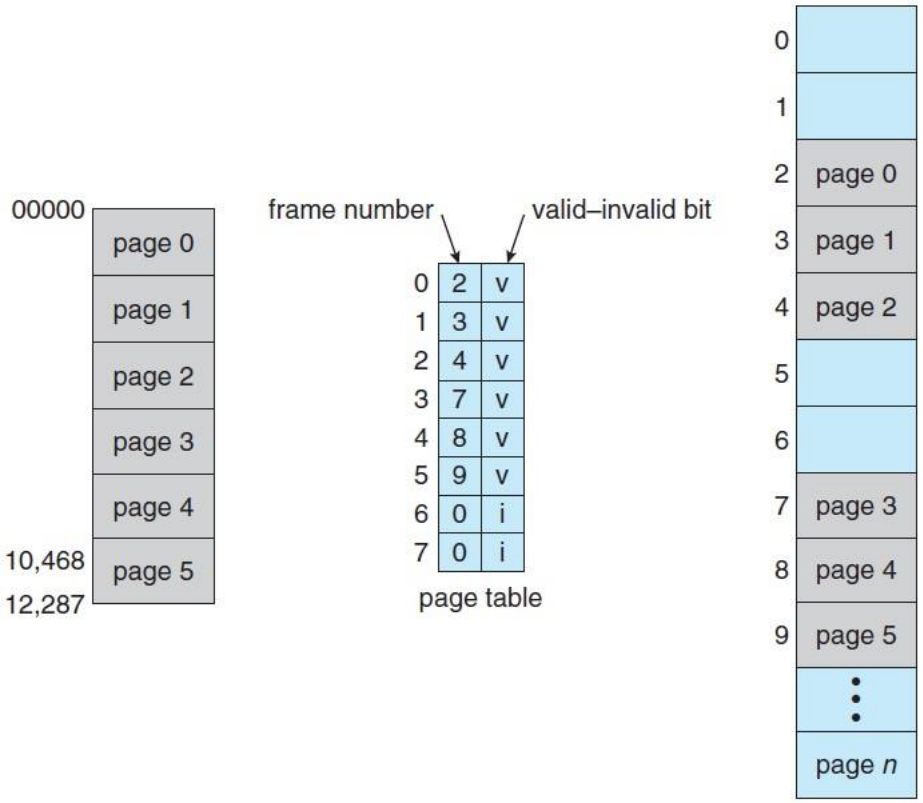


Figure 8.15 Valid (v) or invalid (i) bit in a page table.

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

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

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

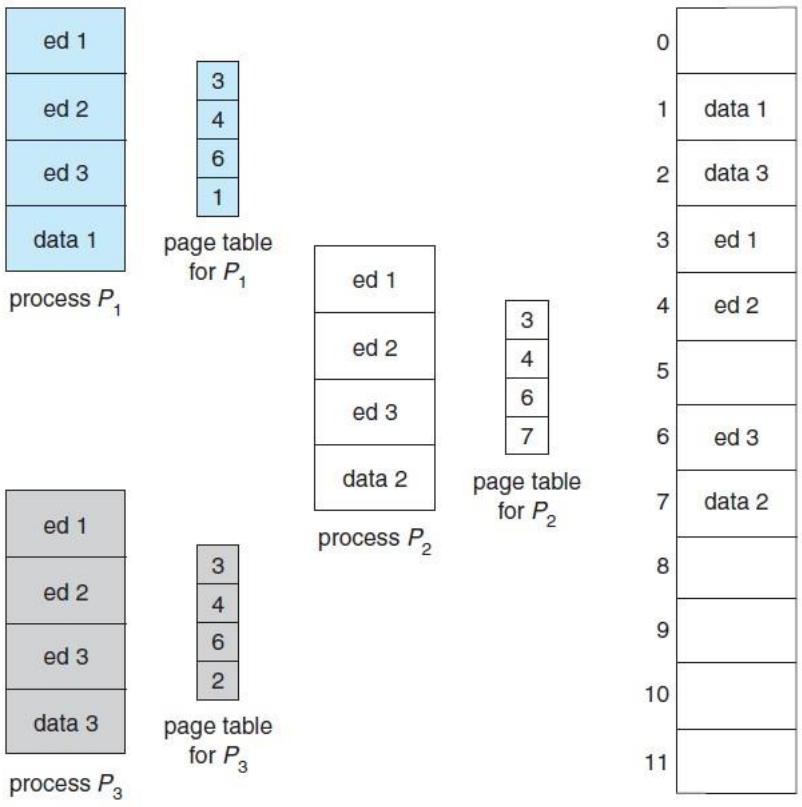
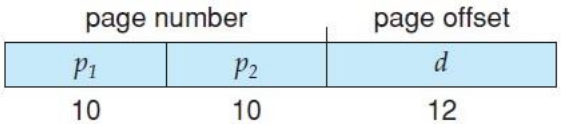


Figure 8.16 Sharing of code in a paging environment.

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

- **Hierarchical Paging:** In this, page table will be divided into smaller pieces. Example for Hierarchical Paging is two-level paging algorithm, in which the page table itself is also paged. Consider a 32 bit machine with a page size of 4 KB. A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits. Because we page the page table, the page number is further divided into a 10-bit page number and a 10-bit page offset. Thus, a logical address is as follows:



Where P_1 is an index into the outer page table and P_2 is the displacement within the page of the outer page table.

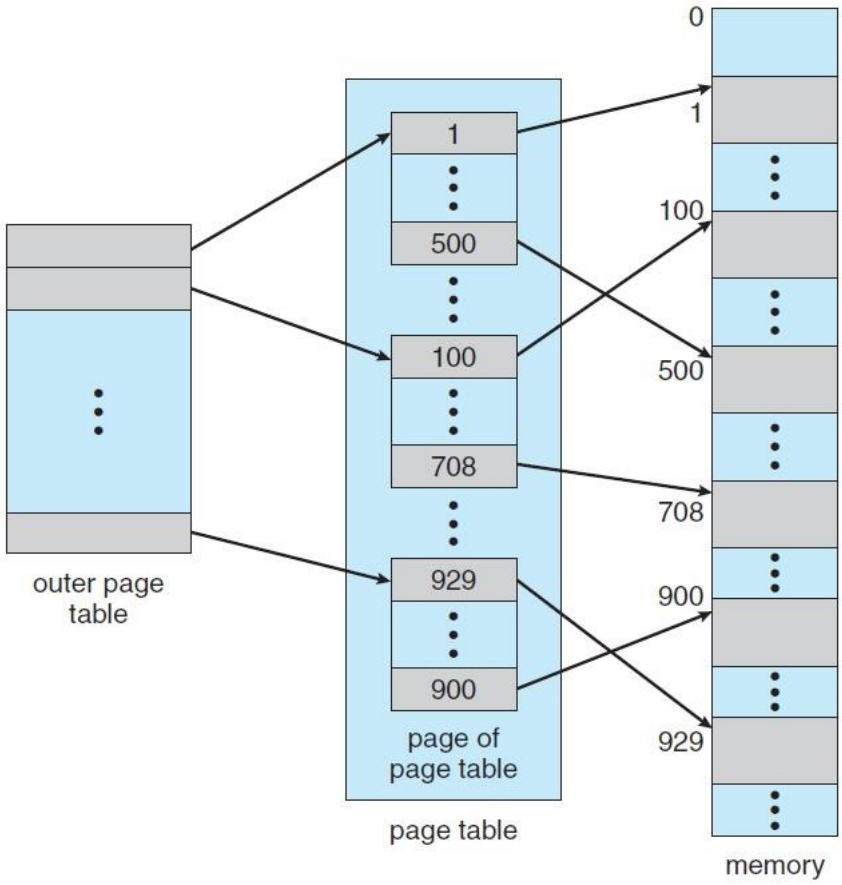


Figure 8.17 A two-level page-table scheme.

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

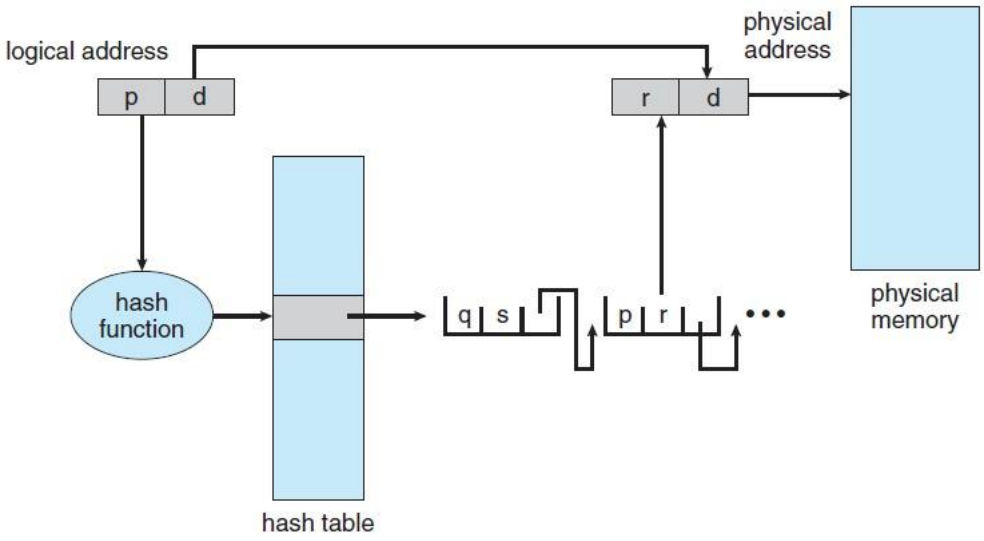


Figure 8.19 Hashed page table.

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

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

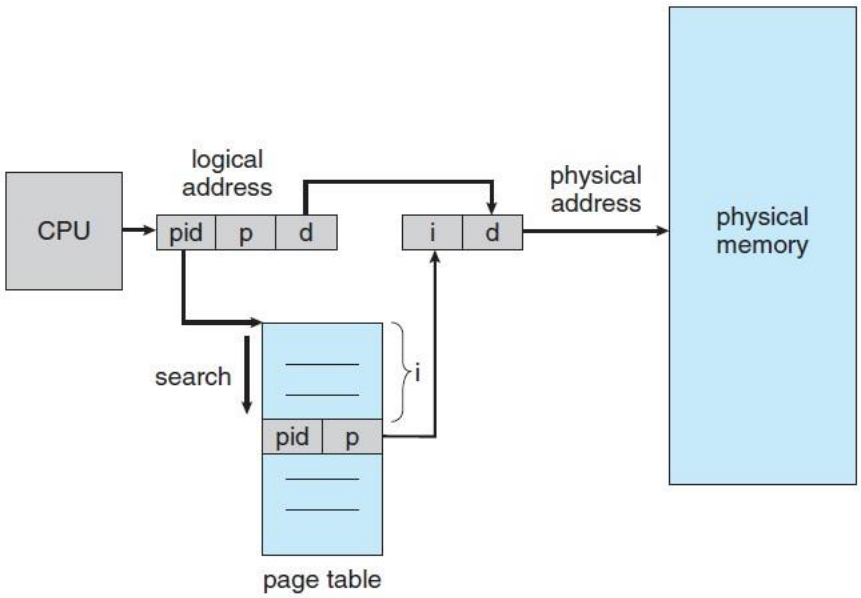


Figure 8.20 Inverted page table.

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

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

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

Basic Method: A logical address space is a collection of segments. Each segment has a name and a length. The addresses specify both the segment name and the

offset within the segment. Segments are numbered and are referred to by a segment number, rather than by a segment name. Thus, a logical address consists of a *two tuple:*

< segment-number, offset >.

Hardware: the following diagram represents segmentation hardware.

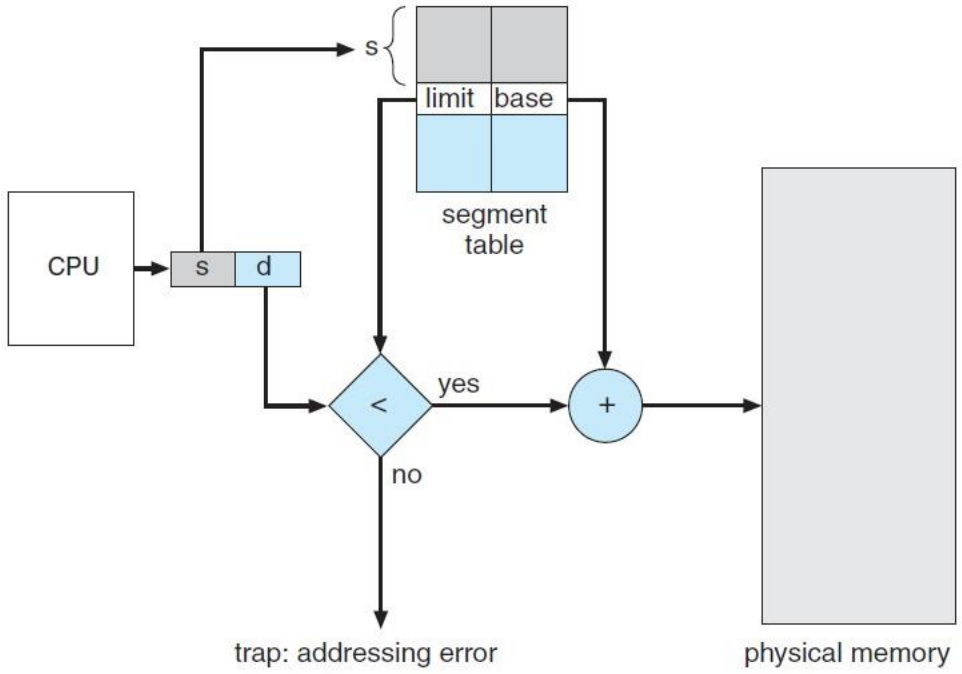


Figure 8.8 Segmentation hardware.

A logical address consists of two parts: a segment number, *s,* and an offset into that segment, *d.* The segment number is used as an index to the segment table. Each entry in the segment table has a *segment base* and a *segment limit.* The segment base contains the starting physical address where the segment resides in memory, whereas the segment limit specifies the length of the segment. The offset *d* of the logical address must be between 0 and the segment limit. If it is not, a trap message will be sent to the operating system. When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte.

For example, We have five segments numbered from 0 through 4. The segments are stored in physical memory as shown below.

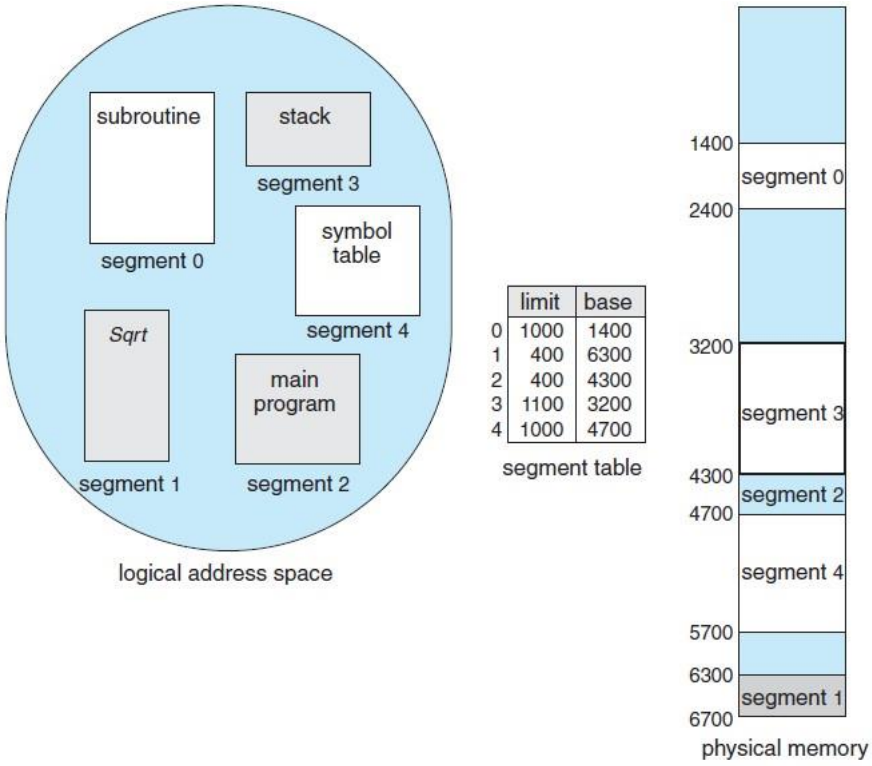


Figure 8.9 Example of segmentation.

The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit). For example, segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353. A reference to segment 3r byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052. A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1,000 bytes long.

Differences between Segmentation and Paging:

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

Virtual Memory: Virtual Memory allows execution of partially loaded processes. One major advantage of this scheme is that programs can be larger than physical memory. **Virtual memory** involves the separation of logical memory as perceived by users from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available as shown below.

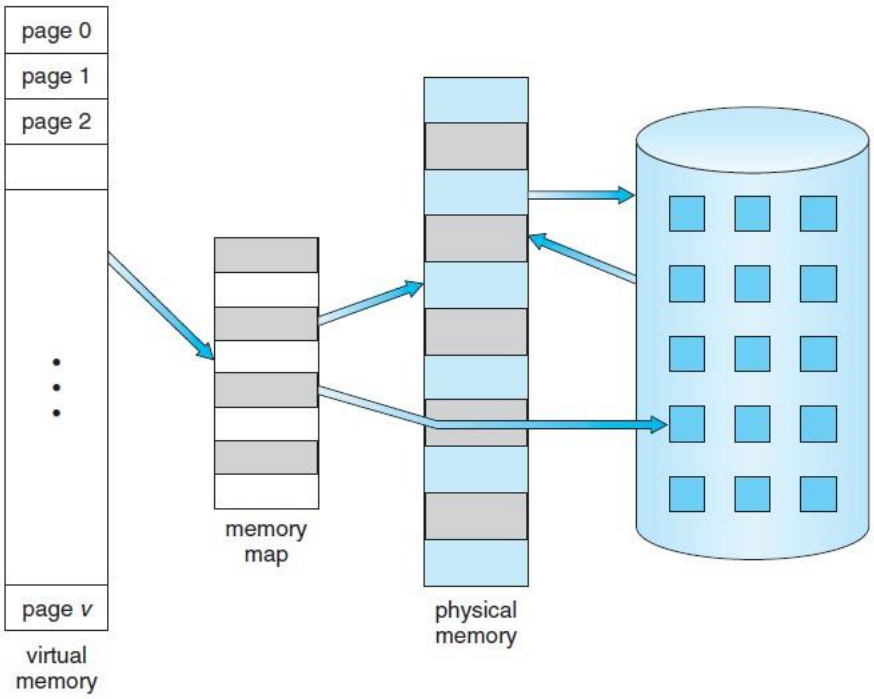


Figure 9.1 Diagram showing virtual memory that is larger than physical memory.

Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available; she can concentrate instead on the problem to be programmed.

In addition to separating logical memory from physical memory, virtual memory allows files and memory to be shared by two or more processes through **page sharing**. Virtual memory can be implemented by using **Demand Paging**.

Demand Paging: In Demand Paging, pages are only loaded when they are demanded during program execution; pages that are never accessed are thus never loaded into physical memory. A demand-paging system is similar to a paging system with swapping as shown below.

Here processes reside in secondary memory (usually a disk). To execute a process, swap it into memory. Rather than swapping the entire process into memory, however, use a **lazy swapper.** A lazy swapper never swaps a page into memory unless that page will be needed.

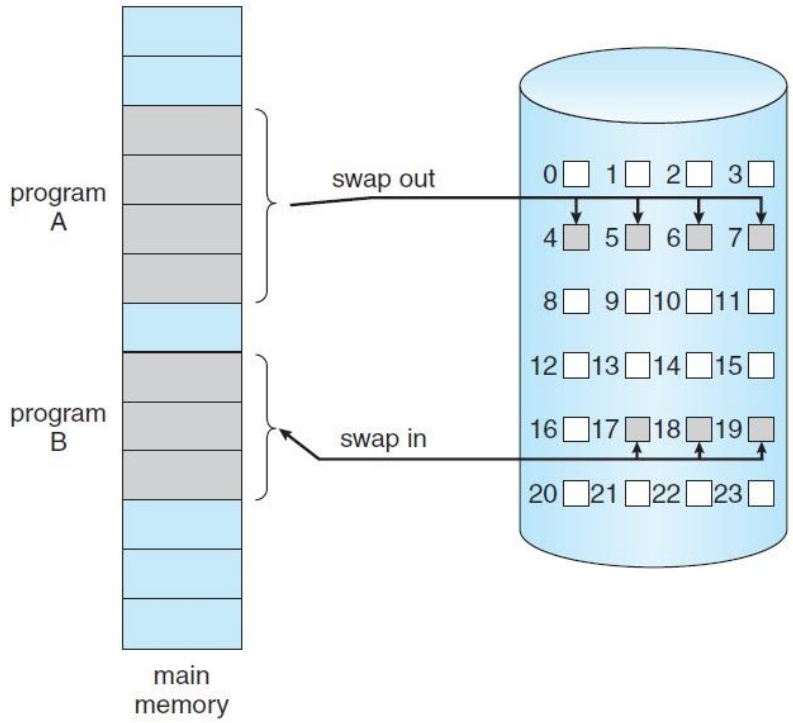


Figure 9.4 Transfer of a paged memory to contiguous disk space.

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

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

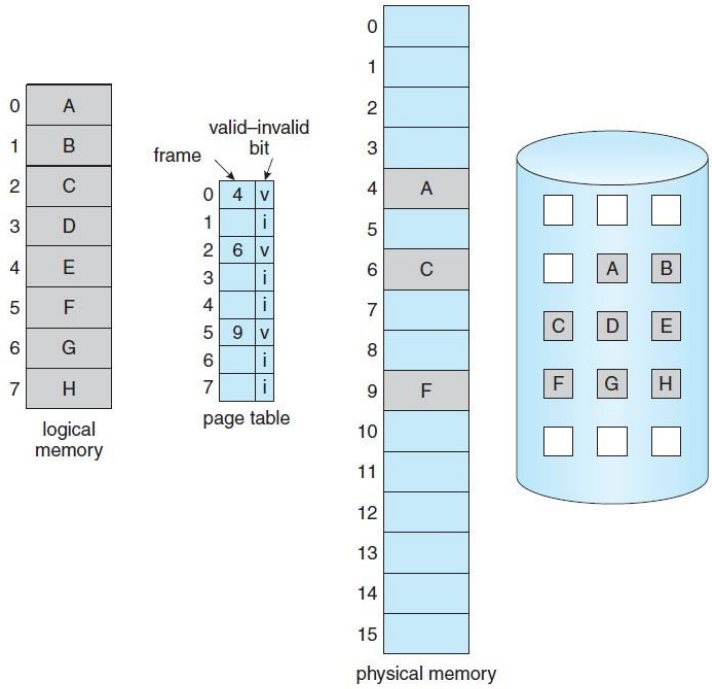


Figure 9.5 Page table when some pages are not in main memory.

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

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

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

The following diagram shows steps for handling Page fault.

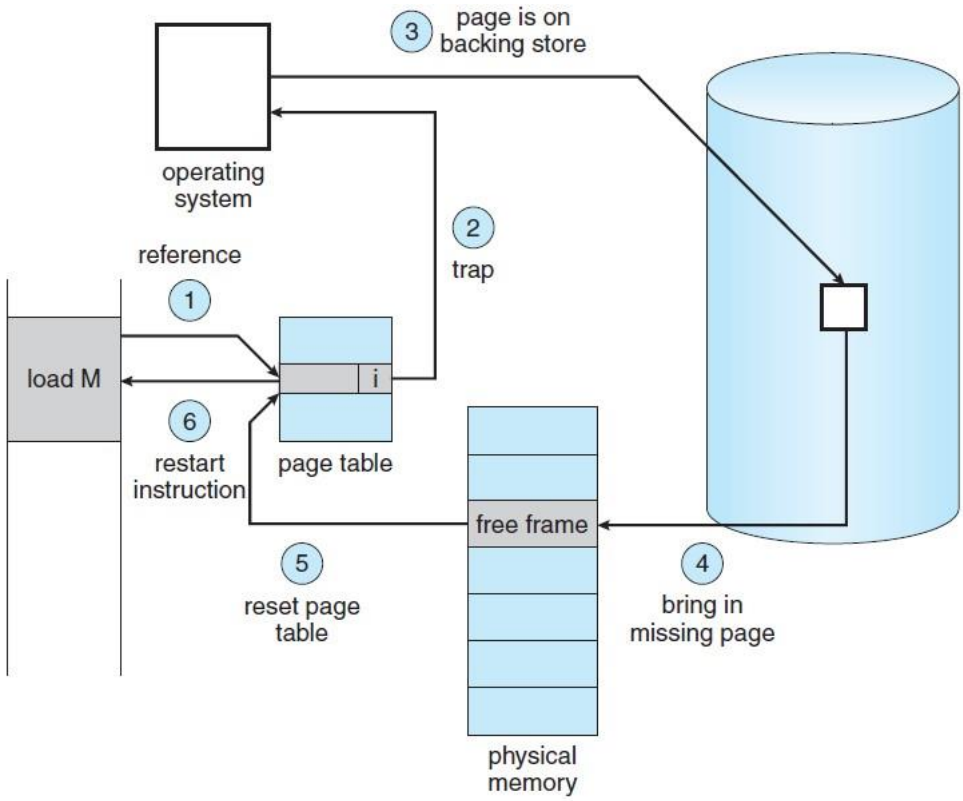


Figure 9.6 Steps in handling a page fault.

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

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

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

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

Where, *ma* is memory access time, Page fault time is the time waited by CPU due to page fault.

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

Advantages of Demand Paging:

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

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

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

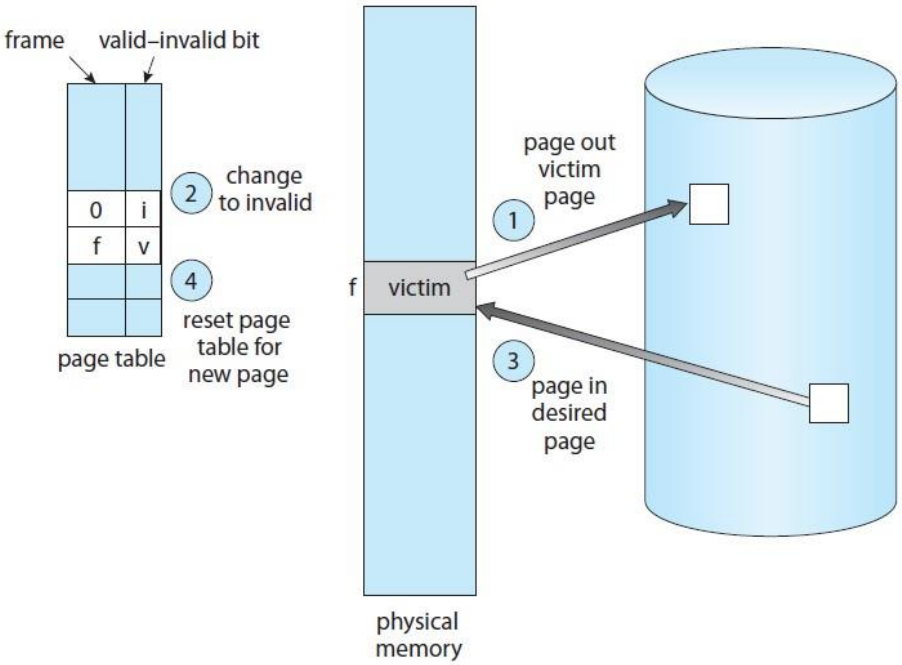


Figure 9.10 Page replacement.

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

1. **FIFO Page Replacement:** The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. This technique, creates a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue. For example, consider the following reference string with 3 frames.
   7, 0,1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2,1, 2, 0, 1, 7, 0,1

String	**7**	**0**	**1**	**2**	**0**	**3**	**0**	**4**	**2**	**3**	**0**	**3**	**2**	**1**	**2**	**0**	**1**	**7**	**0**	**1**
Frame 1	7	7	7	**2**	2	2	2	**4**	4	4	**0**	0	0	0	0	0	0	**7**	7	7
Frame 2		0	0	0	0	**3**	3	3	**2**	2	2	2	2	**1**	1	1	1	1	**0**	0
Frame 3			1	1	1	1	**0**	0	0	**3**	3	3	3	3	**2**	2	2	2	2	**1**
P . Fault	**Y**	**Y**	**Y**	**Y**	**N**	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**	**N**	**N**	**Y**	**Y**	**N**	**N**	**Y**	**Y**	**Y**

Total number of page faults: 15

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

2. **LRU Page Replacement**: The key distinction between the FIFO and LRU algorithms is that the FIFO algorithm uses the time when a page was brought into memory, whereas the LRU algorithm uses the time when a page is to be

used. LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time. Use past knowledge rather than future. For example, consider the following reference string with 3 frames: 7, 0,1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2,1, 2, 0, 1, 7, 0,1

String	**7**	**0**	**1**	**2**	**0**	**3**	**0**	**4**	**2**	**3**	**0**	**3**	**2**	**1**	**2**	**0**	**1**	**7**	**0**	**1**
Frame 1	7	7	7	**2**	2	2	2	**4**	4	4	**0**	0	0	**1**	1	1	1	1	1	1
Frame 2		0	0	0	0	0	0	0	0	**3**	3	3	3	3	3	**0**	0	0	0	0
Frame 3			1	1	1	**3**	3	3	**2**	2	2	2	2	2	2	2	2	**7**	7	7
P . Fault	**Y**	**Y**	**Y**	**Y**	**N**	**Y**	**N**	**Y**	**Y**	**Y**	**Y**	**N**	**N**	**Y**	**N**	**Y**	**N**	**Y**	**N**	**N**

Total number of page faults: 12

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

3. **Optimal Page Replacement:**

Counter - Based: An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms. In this technique, Replace the page that will not be used for the longest period of time. Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string. For example, consider the following reference string with 3 frames:7, 0,1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2,1, 2, 0, 1, 7, 0,1

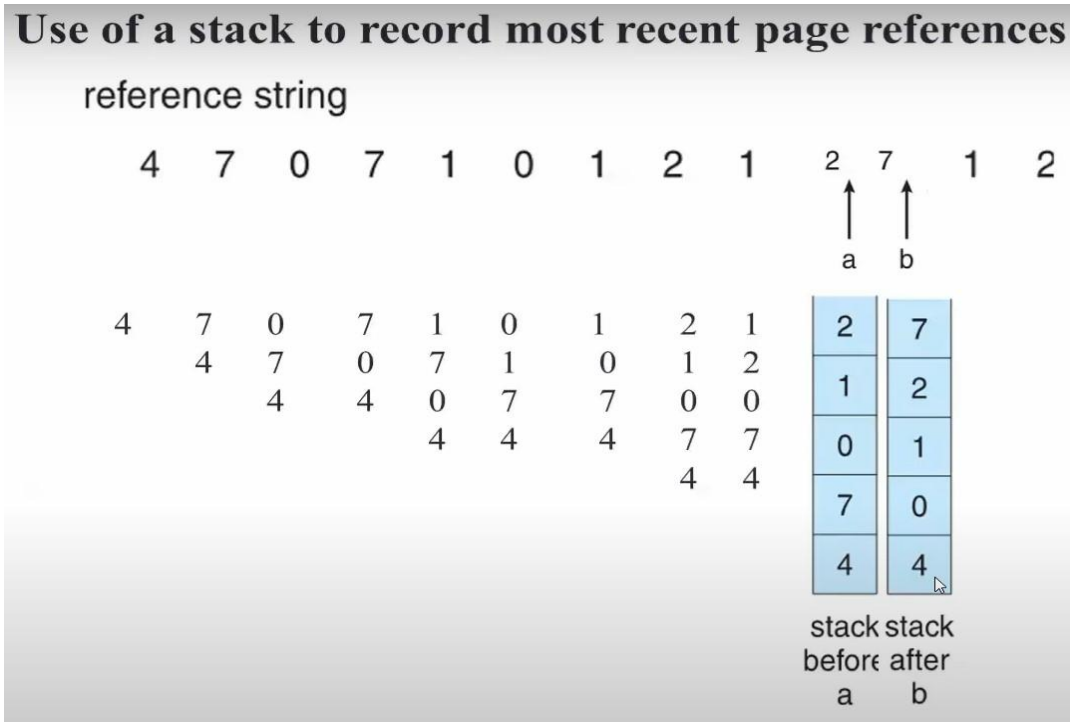
String	**7**	**0**	**1**	**2**	**0**	**3**	**0**	**4**	**2**	**3**	**0**	**3**	**2**	**1**	**2**	**0**	**1**	**7**	**0**	**1**
Frame 1	7	7	7	**2**	2	2	2	2	2	2	2	2	2	2	2	2	2	**7**	7	7
Frame 2		0	0	0	0	0	0	**4**	4	4	**0**	0	0	0	0	0	0	0	0	0
Frame 3			1	1	1	**3**	3	3	3	3	3	3	3	**1**	1	1	1	1	1	1
P . Fault	**Y**	**Y**	**Y**	**Y**	**N**	**Y**	**N**	**Y**	**N**	**N**	**Y**	**N**	**N**	**Y**	**N**	**N**	**N**	**Y**	**N**	**N**

Total number of page faults: 09

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

Stack – Based: Another approach to implementing LRU replacement is to keep a stack of page numbers. Whenever a page is referenced, it is removed

from the stack and put on the top. In this way, the most recently used page is always at the top of the stack and the least recently used page is always at the bottom as shown below.



4. **LRU Approximation Algorithm:** Here, reference bit for a page is set by the hardware whenever that page is referenced (either a read or a write to any byte in the page). Reference bits are associated with each entry in the page table. Initially, all bits are cleared (to 0) by the operating system. As a user process executes, the bit associated with each page referenced is set (to 1) by the hardware.

   After some time, we can determine which pages have been used and which have not been used by examining the reference bits. This information is the basis for many page-replacement algorithms that approximate LRU replacement.

   There are two types of LRU approximation algorithms : Second Chance Algorithm and Enhanced Second Chance Algorithm.

   Second Chance Algorithm: The basic algorithm of second-chance replacement is a FIFO replacement algorithm. When a page has been selected, inspect its reference bit. If the value is 0, proceed to replace this page; but if the reference bit is set to 1, give the page a second chance and move on to select the next FIFO page.

   When a page gets a second chance, its reference bit is cleared, and its arrival time is reset to the current time. Thus, a page that is given a second chance will not be replaced until all other pages have been replaced.

   One way to implement the second-chance algorithm (sometimes referred to as the **clock** algorithm) is as a circular queue. A pointer (that is, a hand on the clock) indicates which page is to be replaced next. When a frame is needed, the pointer advances until it finds a page with a 0 reference bit. As it advances, it clears the reference bits.

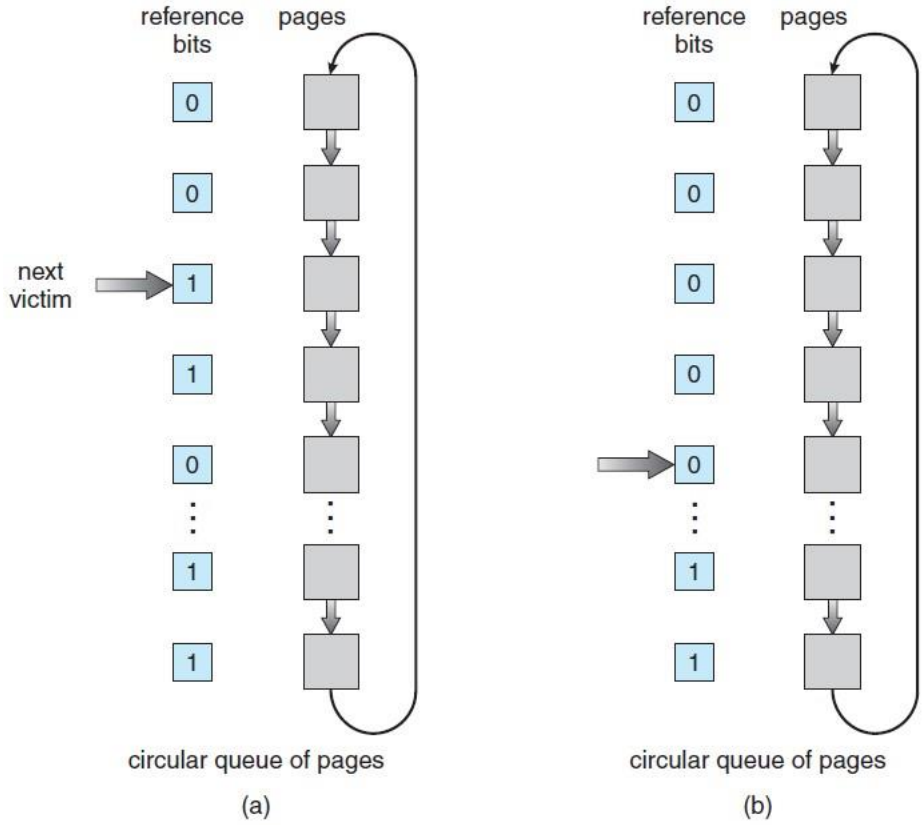


Figure 9.17 Second-chance (clock) page-replacement algorithm.

Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position.

Enhanced Second Chance Algorithm: We can enhance the second-chance algorithm by considering the reference bit and the modify bit as an ordered pair. With these two bits, we have the following four possible classes:

1) (0, 0) neither recently used nor modified—best page to replace
2) (0, 1) not recently used but modified—not quite as good, because the page will need to be written out before replacement.
3) (1, 0) recently used but clean—probably will be used again soon.
4) (1, 1) recently used and modified—probablywill be used again soon, and the page will be need to be written out to disk before it can be replaced.

5. **Counting-Based Page Replacement:** There are many other algorithms that can be used for page replacement. For example, we can keep a counter of the number of references that have been made to each page and develop the following two schemes :

- **Least Frequently Used**: The **least frequently used (LFU) page-replacement algorithm** requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count. For example, consider the following reference string with 3 frames: 1, 3, 3, 2, 5, 2, 1, 4, 2, 2, 5

String	1	3	3	2	5	2	1	4	2	2	5
Frame 1	1	1	1	1	5	5	1	4	4	4	5
Frame 2		3	3	3	3	3	3	3	3	3	3
Frame 3				2	2	2	2	2	2	2	2
P . Fault	Y	Y	N	Y	Y	N	Y	Y	N	N	Y

Frequencies							
1	0	1	0	1	0		
2	0	1	2	3	4		
3	0	1	2				
4	0	1	0				
5	0	1	0	1			

Total No: of Page Faults: 07

- **Most Frequently Used:** The **most frequently used (MFU)** page-replacement algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used. For example, consider the following reference string with 3 frames: 1, 3, 3, 2, 5, 2, 1, 4, 2, 2, 5

String	1	3	3	2	5	2	1	4	2	2	5
Frame 1	1	1	1	1	1	1	1	4	4	4	4
Frame 2		3	3	3	5	5	5	5	5	5	5
Frame 3				2	2	2	2	2	2	2	2
P . Fault	Y	Y	N	Y	Y	N	N	Y	N	N	N

Frequencies							
1	0	1	2				
2	0	1	2	3	4		
3	0	1	2	0			
4	0	1					
5	0	1	2				

Total No: of Page Faults: 05

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

memory pages to be replaced with the 94th, and so on. When the process terminated, the 93 frames would once again be placed on the free-frame list.
There are many variations on this simple strategy.

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

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

  $$\mathbf{S = \sum S_i}$$

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

  $$\mathbf{a_i = S_i / S * m}$$

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

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

  In this way, both processes share the available frames according to their "needs," rather than equally.
- **Global versus Local Allocation:** With multiple processes competing for frames, classify page-replacement algorithms into two broad categories: **global replacement** and **local replacement.**

  Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process; that is, one process can take a frame from another. Local replacement requires that each process select from only its own set of allocated frames.

Thrashing: The phenomenon of excessively moving pages between memory and external disk is called as Thrashing. A process is thrashing if it is spending more time paging than executing.
Cause of Thrashing: The operating system monitors CPU utilization. If CPU utilization is too low, we increase the degree of multiprogramming by introducing a

new process to the system. A global page-replacement algorithm is used; it replaces pages without regard to the process to which they belong. Now suppose that a process enters a new phase in its execution and needs more frames. It starts faulting and taking frames away from other processes.

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

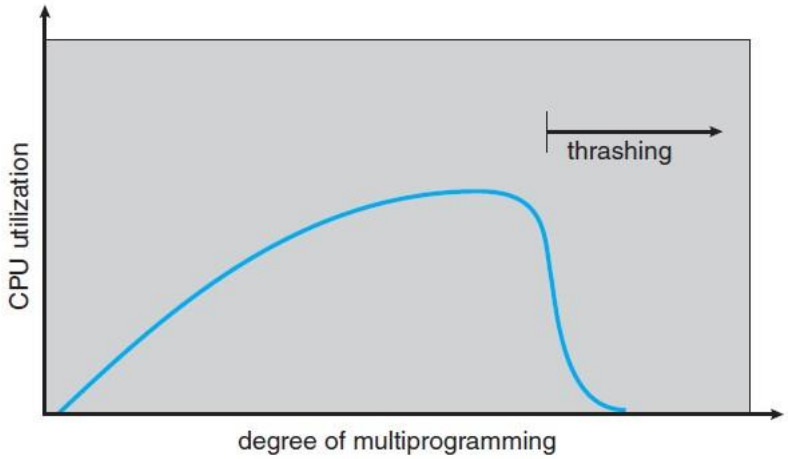


Figure 9.18 Thrashing.

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

Working-Set Model: The idea is to examine the most recent ▲ page references. The set of pages in the most recent ▲ page references is the working set. If a page is in active use, it will be in the working set. If it is no longer being used, it will drop from the working set ▲ time units after its last reference.

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

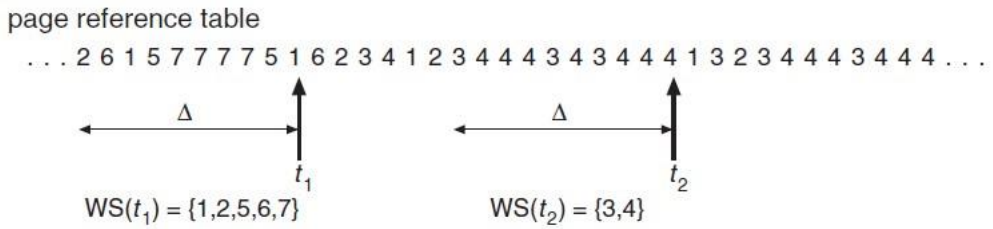


Figure 9.20 Working-set model.

If ▲= 10 memory references, then the working set at time *t1* is {1, *2,* 5, 6, *7}.* By time *h,* the working set has changed to {3, 4}. The accuracy of the working set depends on the selection of ▲. If ▲ is too small, it will not encompass the entire

locality; if ▲ is too large, it may overlap several localities. In the extreme, if ▲ is infinite, the working set is the set of pages touched during the process execution.

The most important property of the working set, then, is its size. If we compute the working-set size, WSS_j, for each process in the system, we can then consider that

$$D = \sum WSS_i$$

where D is the total demand for frames. Each process is actively using the pages in its working set. Thus, process i needs WSS_i frames. If the total demand is greater than the total number of available frames ($D > m$), thrashing will occur, because some processes will not have enough frames.

Page Fault Frequency: If the actual page-fault rate exceeds the upper limit, we allocate the process another frame. If the page-fault rate falls below the lower limit, we remove a frame from the process. Thus, we can directly measure and control the page-fault rate to prevent thrashing.

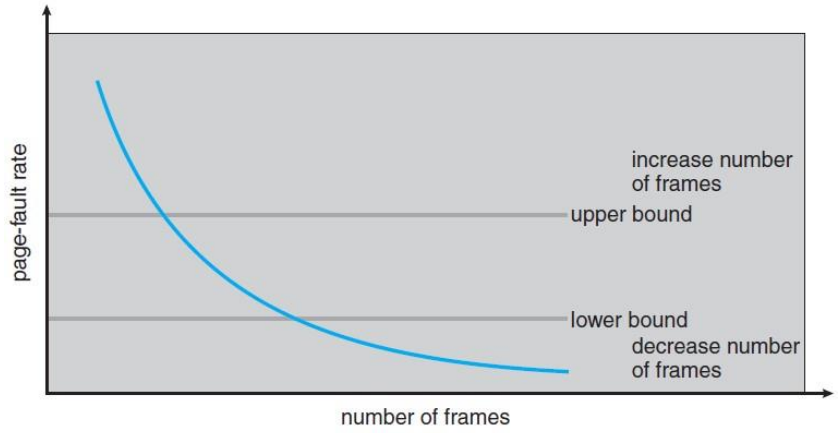


Figure 9.21 Page-fault frequency.

FREQUENTLY ASKED QUESTIONS

1. Distinguish between Logical and Physical address space?
2. What is the purspose of Paging and page tables?
3. What is Paging? Discuss the Paging model of logical and physical memory?
4. What is a Virtual Memory? Discuss the benefits of virtual memory technique?
5. What is Thrashing? What can the system do to eliminate this problem?
6. What is a Pagefault? Explain the steps involved in handling a pagefault with a neat sketch?
   OR
   Discuss the procedure for handling page fault in Demand Paging?
7. Explain the need of page replacement and explain with an example?
8. Suggest the best algorithm for the given reference string.
   7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 with three page frames.
9. Compute the number of page faults for optimal page replacement strategy for the given reference string 1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 7, 8, 9, 5, 4, 5, 4, 2 with 4 page frames.
10. Consider the following reference string:
    1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6

How many page faults would occur for the optimal page replacement algorithm, assuming three frames and all frames are initially empty.

11. Discuss the hardware support required to support demand paging?
12. What factors effecting the performance of demand paging?
13. Explain how demand paging effects the performance of a computer system?
14. Explain the difference between internal and external fragmentation?
15. Discuss various issues related to the allocation of frames to processes?
16. What is the cause of Thrashing? How does the system detect Thrashing? How to eliminate this problem?

UNIT - 5

Syllabus: File System Interface: The concept of a File, Access Methods, Directory Structure, File Sharing, Protection.

File System Implementation: File System Structure, Allocation methods, Free space management.

Mass storage structure: Overview of Mass storage structre, Disk scheduling, Swap space management.

File System Interface

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

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

File attributes: File attributes are settings associated with **computer files** that grant or deny certain rights to how a **user** or the operating system can access that file. A file is named, for the convenience of its human users, and is referred to by its name. A name is usually a string of characters, such as *example.c.* Following are some of the attributes of a file :

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

File Operations: The OS provides systems calls to create, write, read, reset, and delete files. The following are the specific duties an OS must do for each of the five basic file operations.

- **Creating a file:** Two steps are necessary to create a file. First, space in the file system must be found for the file. Second, an entry for the new file must be made in the directory.
- **Writing a file:** To write a file, make a system call specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the file's location. The system must keep a *write* pointer to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.
- **Reading a file:** To read from a file, use a system call that specifies the name of the file and where (in memory) the next block of the file should be put. Again, the directory is searched for the associated entry, and the system needs to keep a *read* pointer to the location in the file where the next read is to take place. Once the read has taken place, the read pointer is updated. Both the read and write operations use this same pointer, saving space and reducing system complexity.
- **Repositioning within** a **file:** The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value. This file operation is also known as a file *seek.*
- **Deleting a file:** To delete a file, search the directory for the named file. Having found the associated directory entry, release all file space, so that it can be reused by other files, and erase the directory entry.
- **Truncating a file**: The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged—except for file length—but lets the tile be reset to length zero and its file space released.

File types: A common technique for implementing file types is to include the type as part of the file name. The name is split inro two parts: a name and an extension, usually separated by a period. The system uses the extension to indicate the type of operations that can be done on that file. The following table indicates different types of files.

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

Figure 11.3 Common file types.

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

- Sequential access
- Direct/Random access
- Other access methods
- **Sequential Access:** The simplest access method is sequential access. Information in the file is processed in order, one record after the other. For example, editors and compilers usually access files in this fashion.

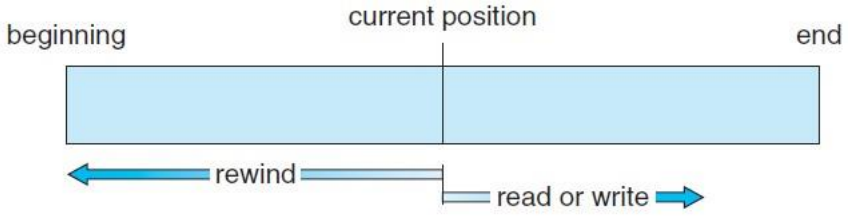


Figure 11.4 Sequential-access file.

A read operation—*read next*—reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location. Similarly, the write operation—*write next*—appends to the end of the file and advances to the end of the newly written material (the new end of file).

- **Direct Access:** The direct-access method is based on a disk model of a file, since disks allow random access to any file block. For direct access, the file is viewed as a numbered sequence of blocks or records. Thus, we may read block 14, then read block 53, and then write block 7. There are no

restrictions on the order of reading or writing for a direct-access file. Direct-access files are of great use for immediate access to large amounts of information.

For the direct-access method, use read n, where n is the block number and write n, where n is the block number. The block number provided by the user to the operating system is normally a **relative block number.** A relative block number is an index relative to the beginning of the file. Thus, the first relative block of the file is 0, the next is 1, and so on.

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

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

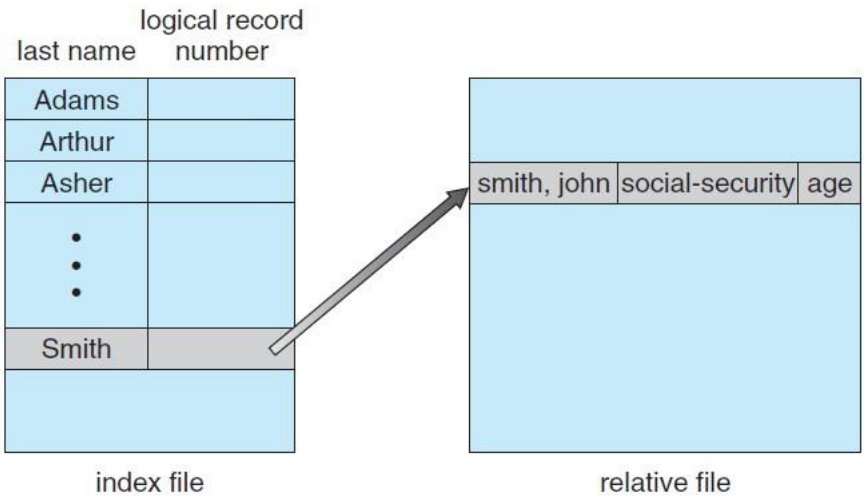


Figure 11.6 Example of index and relative files.

Directory Structure:

Directory Overview: The directory can be viewed as a symbol table that translates file names into their directory entries. The following are different types of operations that can be performed on directories.

- Search for a file: This operation allows to search a directory structure to find the entry for a particular file.
- Create a file: New files need to be created and added to the directory
- Delete a file: When a file is no longer used, erase from the directory
- List a directory: This operation list the files in adircetory and the connetnts of the directory entry for each file in the list.

- Rename a file: This operation allows to change name of a file.
- Traverse the file system: This operation allows to navigate through entire file system.

There are many types of directory structure in Operating System. They are as follows:

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 in the same directory, which is easy to support and understand as shown below.

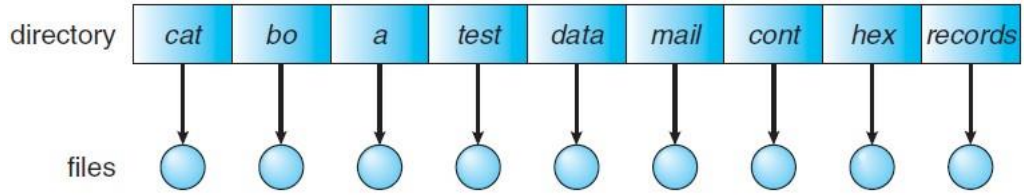


Figure 11.9 Single-level directory.

Limitations:

- Since, all files are in the same directory, they must have unique names.
- If two user call their data file ***test***, then the unique name rule is violated.
- Files are limited in length.
- Even a single user may find it difficult to remember the names of all files as the number of file increases.
- Keeping track of so many file is daunting task.

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

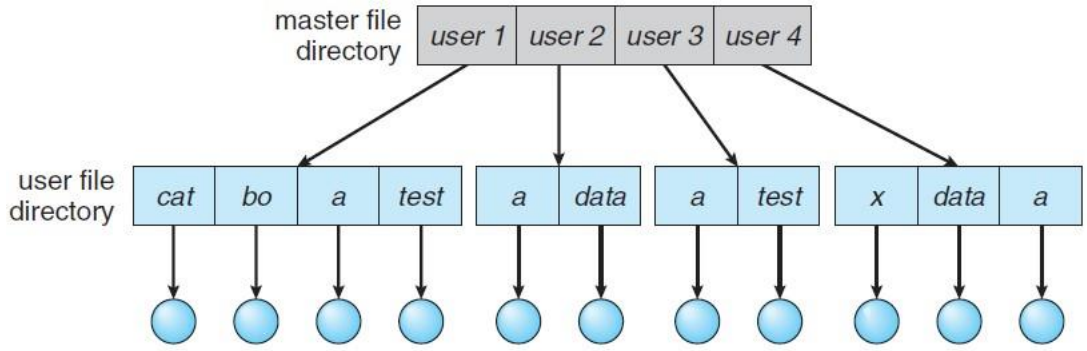


Figure 11.10 Two-level directory structure.

A two-level directory can be thought of as a tree, or an inverted tree, of height 2. The root of the tree is the MFD. Its direct descendants are the UFDs. The descendants of the UFDs are the files themselves. The files are the leaves of the tree. Specifying a user name and a file name defines a path in the tree from the root (the MFD) to a leaf (the specified file).

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

This structure effectively isolates one user from another. Isolation is an advantage when the users are completely independent but is a disadvantage when the users *want* to cooperate on some task and to access one another's files.

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

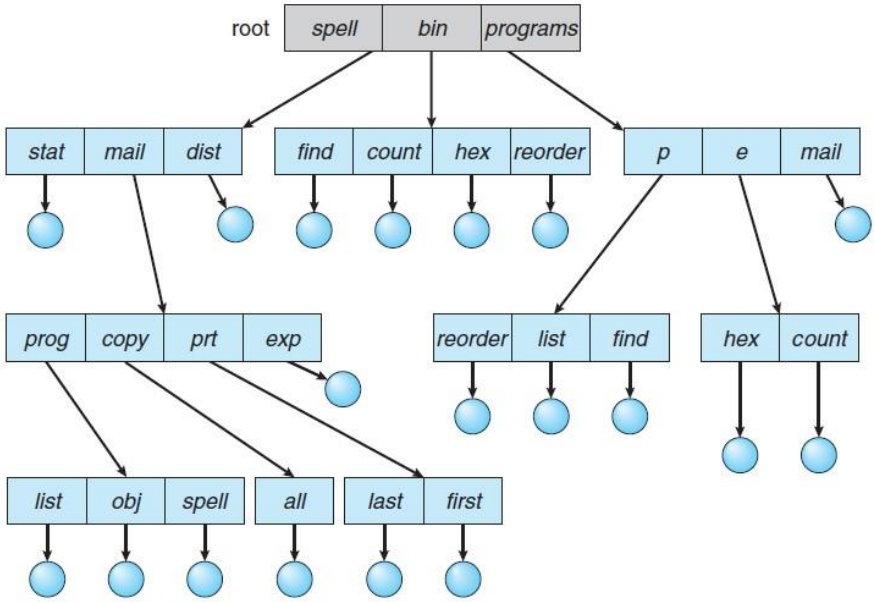


Figure 11.11 Tree-structured directory structure.

Each process has a current directory. Current directory should contain most of the files that are of current interest to the process. When a reference is made to a file, the current directory is searched. The user can change his current directory whenever he desires. If a file is not needed in the current directory then the user usually must either specify a path name or change the current directory. Path names can be of two types: *absolute* and *relative.* An absolute path **name** begins at the root and follows a path down to the specified file, giving the directory names on the path. A relative path **name** defines a path from the current directory. For example, if the

current directory is *root/spell'/mail,* then the relative path name *prt/first* refers to the same file as does the absolute path name r*oot/spcll/mail/prt/first.*

4. **Acyclic Graph Directories:** Acyclic Graph is the graph with no cycles. It allows directories to share sub directories and files. With a shared file, only one actual file exists, so any changes made by one person are immediately visible to the another.

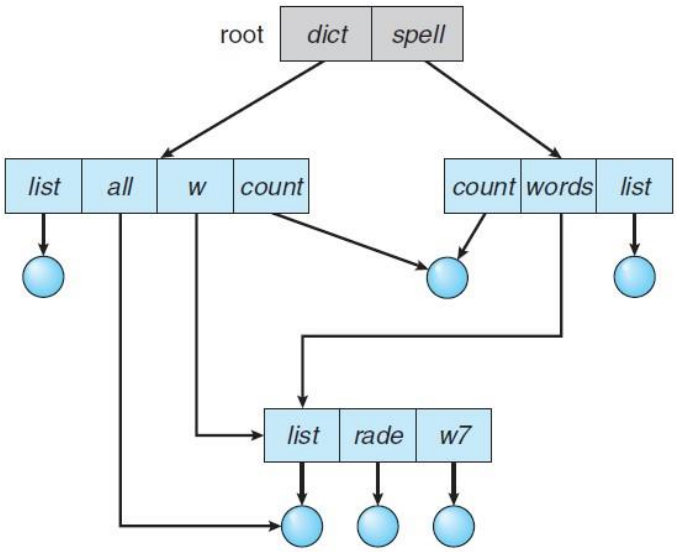


Figure 11.12 Acyclic-graph directory structure.

It is important to note that a shared file (or directory) is not the same as two copies of the file. With two copies, each programmer can view the copy rather than the original, but if one programmer changes the file, the changes will not appear in the other's copy. With a shared file, only *one* actual file exists, so any changes made by one person are immediately visible to the other.

Allow directories to link to one another, allow multiple directories to contain same file i.e., only one copy of the file exists and any change in the file can be viewed by all directories in which it is contained.

5. **General Graph Directory:** General graph directory allow cycles, more flexible, more costly. This directory needs garbage collection. If cycles are allowed in the graphs, then several problems can arise:
    - Search algorithms can go into infinite loops.
    - Sub-trees can become disconnected from the rest of the tree

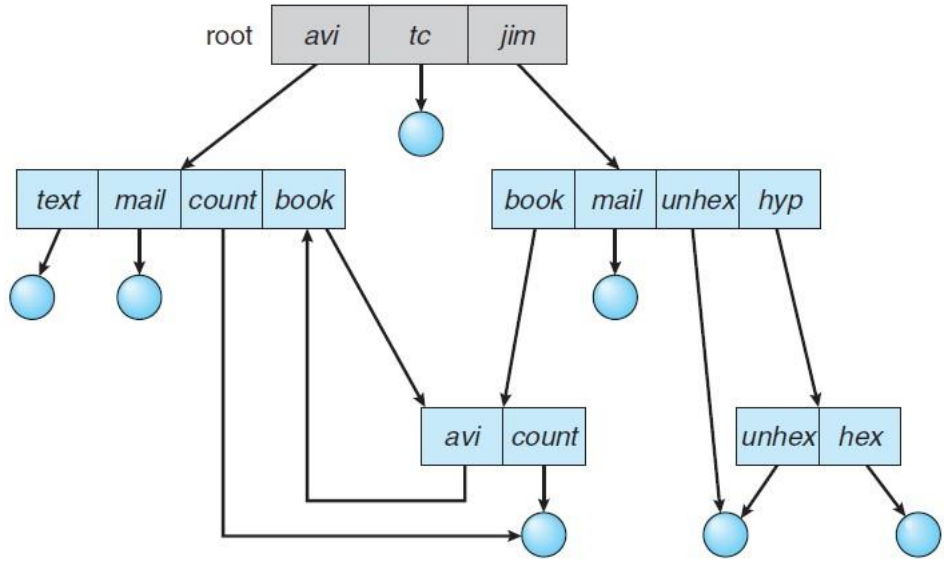


Figure 11.13 General graph directory.

File Sharing: File sharing is the practice of sharing or offering access to digital information or resources, including documents, multimedia (audio/video), graphics, computer programs, images and e-books. It is the private or public distribution of data or resources in a network with different levels of sharing privileges.

→ **Multiple Users:** Most systems have evolved to use the concepts of file (or directory) *owner* (or *user)* and *group.* The owner is the user who can change attributes and grant access and who has the most control over the file. The group attribute defines a subset of users who can share access to the file. For example, the owner of a file on a UNIX system can issue all operations on a file, while members of the file's group can execute one subset of those operations, and all other users can execute another subset of operations. Exactly which operations can be executed by group members and other users is definable by the file's owner.

The owner and group IDs of a given file (or directory) are stored with the other file attributes. When a user requests an operation on a file, the user ID can be compared with the owner attribute to determine if the requesting user is the owner of the file. Likewise, the group IDs can be compared. The result indicates which permissions are applicable. The system then applies those permissions to the requested operation and allows or denies it.

→**Remote File Systems:** Networking allows the sharing of resources spread across a campus or even around the world. One obvious resource to share is data in the form of files. The first implemented method involves manually transferring files between machines via programs like ftp. The second major method uses a **distributed file system (DFS)** in which remote directories are visible from a local machine. the third method, the **World Wide Web,** is a reversion to the first. A browser is needed to gain access to the remote files, and separate operations (essentially a wrapper for ftp) are used to transfer files.

FTP is used for both anonymous and authenticated access. **Anonymous access** allows a user to transfer files without having an account on the remote

system. The World Wide Web uses anonymous file exchange almost exclusively. DFS involves a much tighter integration between the machine that is accessing the remote files and the machine providing the files.

- **The Client-Server Model:** Remote file systems allow a computer to mount one or more file systems from one or more remote machines. In this case, the machine containing the files is the *server,* and the machine seeking access to the files is the *client.* The client-server relationship is common with networked machines. Generally, the server declares that a resource is available to clients and specifies exactly which resource (in this case, which files) and exactly which clients. A server can serve multiple clients, and a client can use multiple servers, depending on the implementation details of a given client-server facility.
- **Distributed information systems:** To make client-server systems easier to manage, **distributed information** systems, also known as **distributed naming services,** provide unified access to the information needed for remote computing. The **domain name system (DNS)** provides host-name-to-network-address translations for the entire Internet (including the World Wide Web).

  A newer approach is the ***Lightweight Directory-Access Protocol, LDAP,*** which provides a ***secure single sign-on*** for all users to access all resources on a network. This is a secure system which is gaining in popularity, and which has the maintenance advantage of combining authorization information in one central location.
- **Failure Modes:** Local file systems can fail for a variety of reasons, including failure of the disk containing the file system, corruption of the directory structure or other disk-management information (collectively called metadata), disk-controller failure, cable failure, and host-adapter failure. User or systems-administrator failure can also cause files to be lost or entire directories or volumes to be deleted.

  Many of these failures will cause a host to crash and an error condition to be displayed, and human intervention will be required to repair the damage.

  Remote file systems may fail due to complexity of network systems, hardware failure, poor hardware configuration, or networking implementation issues.

→**Consistancy Semantics:** ***Consistency Semantics*** deals with the consistency between the views of shared files on a networked system. The following are examples for different consistence semantics.

- **UNIX Semantics:** The UNIX file system uses the following semantics.
  - ✓ Writes to an open file by a user are visible immediately to other users that have this file open.
  - ✓ One mode of sharing allows users to share the pointer of current location into the file. Thus, the advancing of the pointer by one user affects all sharing users.
  - ✓ The file is associated with a single exclusive physical resource, which may delay some accesses.

- **Session Semantics:** The Andrew File System, AFS uses the following semantics:
  - ✓ Writes to an open file by a user are not visible immediately to other users that have the same file open.
  - ✓ Once a file is closed, the changes made to it are visible only in sessions starting later. Already open instances of the file do not reflect these changes.

  According to these semantics, a file can be associated with multiple (possibly different) views. Almost no constraints are imposed on scheduling accesses. No user is delayed in reading or writing their personal copy of the file.
- **Immutable Shared Files Semantics:** A unique approach is that of **immutable shared files.** Once a file is declared as *shared* by its creator, it cannot be modified. An immutable file has two key properties: Its name may not be reused, and its contents may not be altered. Thus, the name of an immutable file signifies that the contents of the file are fixed.

Protection: Files must be kept safe for reliability and protection. Reliability is generally provided by duplicate copies of files. Many computers have systems programs that automatically (or through computer-operator intervention) copy disk files to tape at regular intervals (once per day or week or month) to maintain a copy should a file system be accidentally destroyed. File systems can be damaged by hardware problems (such as errors in reading or writing), power surges or failures, head crashes, dirt, temperature extremes, and vandalism. Files may be deleted accidentally. Bugs in the file-system software can also cause file contents to be lost.

Protection can be provided in many ways as follows:

- **Types of Access:** Protection mechanisms provide controlled access by limiting the types of file access that can be made. Access is permitted or denied depending on several factors, one of which is the type of access requested. Several different types of low-level operations may be controlled:
  - ✓ Read: Read from the file.
  - ✓ Write: Write or rewrite the file.
  - ✓ Execute: Load the file into memory and execute it.
  - ✓ Append: Write new information at the end of the file.
  - ✓ Delete: Delete the file and free its space for possible reuse.
  - ✓ List: List the name and attributes of the file.
  - ✓ Higher-level operations, such as copy, can generally be performed through combinations of the above.
- **Access Control:** The most common approach to the protection problem is to make access dependent on the identity of the user. Different users may need different types of access to a file or directory. The most general scheme to implement identity dependent access is to associate with each file and directory an access-control list (ACL) specifying user names and the types of access allowed for each user. When a user requests access to a particular file, the operating system checks the access list associated with that file. If

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

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

Bit	Files	Directories
R	Read	Read directory contents. Required to get a listing of the directory.
W	Write	Change directory contents. Required to create or delete files.
X	Execute	Execute the file

- **Other Protection Approaches and Issues :** Another approach to the protection problem is to associate a password with each file. Just as access to the computer system is often controlled by a password, access to each file can be controlled in the same way. If the passwords are chosen randomly and changed often, this scheme may be effective in limiting access to a file. The use of passwords has a few disadvantages, however. First, the number of passwords that a user needs to remember may become large, making the scheme impractical. Second, if only one password is used for all the files, then once it is discovered, all files are accessible. Protection is on an all-or-none basis. Some systems allow a user to associate a password with a subdirectory, rather than with an individual file, to deal with this problem.

File System Implementation

File system Structure: To provide efficient and convenient access to the disk, the operating system imposes one or more file systems to allow the data to be stored, located, and retrieved easily. A file system poses two quite different design problems. The first problem is defining how the file system should look to the user. This task involves defining a file and its attributes, the operations allowed on a file, and the directory structure for organizing files. The second problem is creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices.

The file system itself is generally composed of many different levels as shown below.

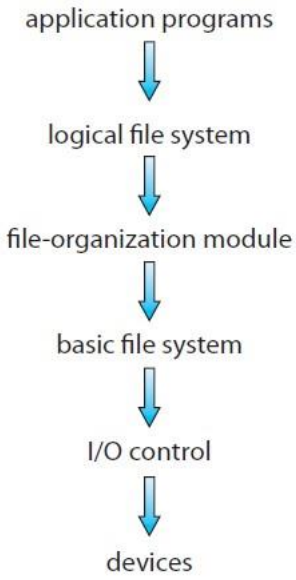


Figure 12.1 Layered file system.

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 interrupt 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 lowlevel, hardware-specific instructions that are used by the hardware controller, which interfaces the I/O device to the rest of the system. The device driver usually writes specific bit patterns to special locations in the I/O controller's memory to tell the controller which device location to act on and what actions to take.

The **basic file system** needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk.

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

Finally, the 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. A **file-control block** (FCB) contains information about the file, including ownership, permissions, and location of the file contents. The logical file system is also responsible for protection and security.

UNIX uses the **UNIX file system** (UFS), which is based on the Berkeley Fast File System (FFS). Windows NT, 2000, and XP support disk file-system formats of FAT, FAT32, and KTFS (or Windows NT File System), as well as CD-ROM, DVD, and floppy-disk file-system formats. Although Linux supports over forty different

file systems, the standard Linux file system is known as the **extended file system,** with the most common version being ext2 and ext3.

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

1. **Contiguous Allocation:** A single continuous set of blocks is allocated to a file at the time of file creation. Thus, this is a pre-allocation strategy, using variable size portions. The file allocation table needs just a single entry for each file, showing the starting block and the length of the file. This method is best from the point of view of the individual sequential file. Multiple blocks can be read in at a time to improve I/O performance for sequential processing. It is also easy to retrieve a single block. For example, if a file starts at block b, and the i^{th} block of the file is wanted, its location on secondary storage is simply b + i - 1.

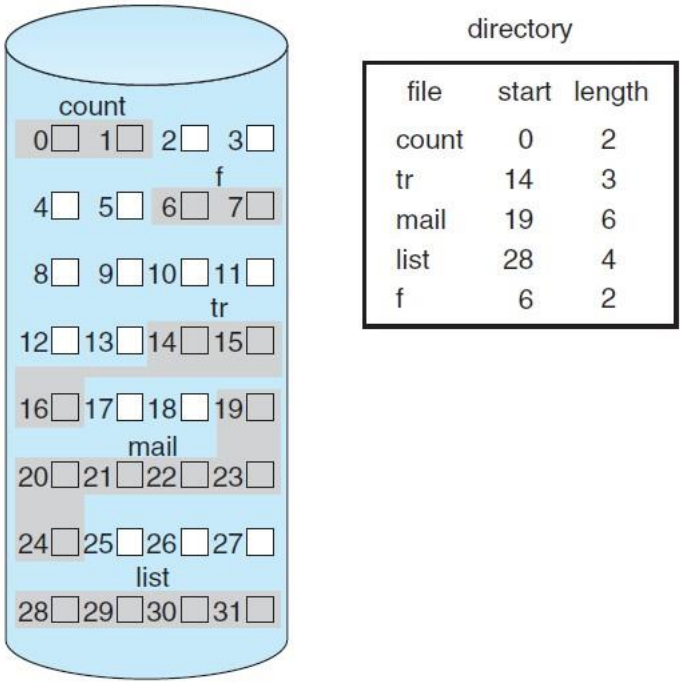


Figure 12.5 Contiguous allocation of disk space.

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

Advantages:

- Both the Sequential and Direct Accesses are supported by this. For direct access, the address of the k^{th} block of the file which starts at block b can easily be obtained as (b+k).
- This is extremely fast since the number of seeks are minimal because of contiguous allocation of file blocks.

Disadvantages:

- This method suffers from both internal and external fragmentation. This makes it inefficient in terms of memory utilization.
- Increasing file size is difficult because it depends on the availability of contiguous memory at a particular instance.

2. **Linked Allocation:** In this scheme, each file is a linked list of disk blocks which **need not be** contiguous. The disk blocks can be scattered any where on the disk.
The directory entry contains a pointer to the starting and the ending file block. Each block contains a pointer to the next block occupied by the file. The file 'jeep' in following image shows how the blocks are randomly distributed. The last block (25) contains -1 indicating a null pointer and does not point to any other block.

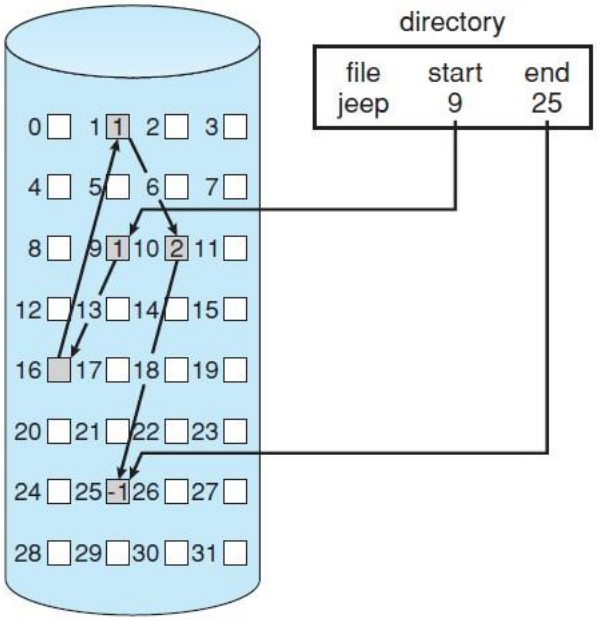


Figure 12.6 Linked allocation of disk space.

Advantages:

- This is very flexible in terms of file size. File size can be increased easily since the system does not have to look for a contiguous chunk of memory.
- This method does not suffer from external fragmentation. This makes it relatively better in terms of memory utilization.

Disadvantages:

- Because the file blocks are distributed randomly on the disk, a large number of seeks are needed to access every block individually. This makes linked allocation slower.
- It does not support random or direct access. We can not directly access the blocks of a file. A block k of a file can be accessed by traversing k blocks sequentially (sequential access) from the starting block of the file via block pointers.
- Pointers required in the linked allocation incur some extra overhead.

3. **Indexed Allocation:** In this scheme, a special block known as the **Index block** contains the pointers to all the blocks occupied by a file. Each file has its own index block. The ith entry in the index block contains the disk address of the ith file block. The directory entry contains the address of the index block as shown in the image:

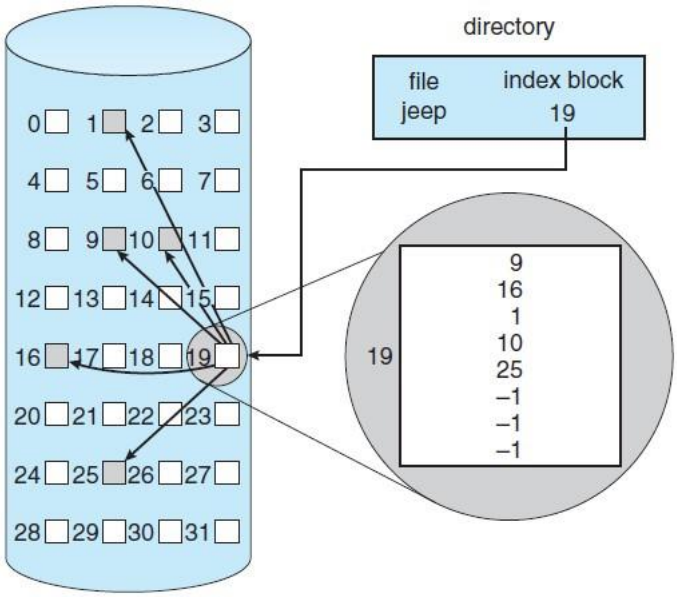


Figure 12.8 Indexed allocation of disk space.

Advantages:

- This supports direct access to the blocks occupied by the file and therefore provides fast access to the file blocks.
- It overcomes the problem of external fragmentation.

Disadvantages:

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

Free Space Management: Since disk space is limited, we need to reuse the space from deleted files for new files, if possible. To keep track of free disk space, the system maintains a **free-space list.** The free-space list records *all free* disk blocks—those not allocated to some file or directory. To create a file, we search the free-space list for the required amount of space and allocate that space to the new file. This space is then removed from the free-space list. When a file is deleted, its disk space is added to the free-space list.

Bit Vector: The free-space list is implemented as a bit **map** or bit vector. Each block is represented by 1 bit. If the block is free, the bit is 1; if the block is allocated, the bit is 0. For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25,26, and 27 are free and the rest of the blocks are allocated. The free-space bit map would be

0011110011111100011000000111000000 …

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

Linked List: Another approach to free-space management is to link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory.

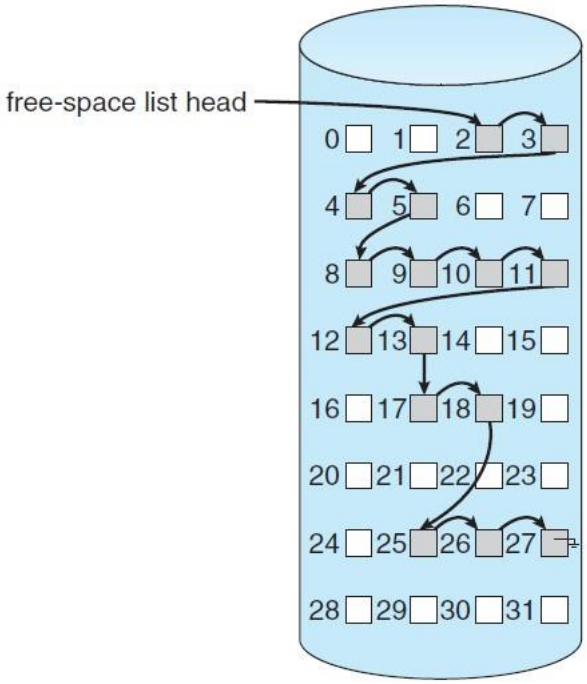


Figure 12.10 Linked free-space list on disk.

This first block contains a pointer to the next free disk block, and so on. In the above example, keep a pointer to block 2 as the first free block. Block *2* would contain a pointer to block 3, which would point to block 4, which would point to block 5, which would point to block 8, and so on.

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

Grouping: A modification of the free-list approach is to store the addresses of *n* free blocks in the first free block. The first n-1 of these blocks are actually free. The last block contains the addresses of another *n* free blocks, and so on. The addresses of a large number of free blocks can now be found quickly, unlike the situation when the standard linked-list approach is used.

Counting: Rather than keeping a list of *n* free disk addresses, we can keep the address of the first free block and the number *n* of free contiguous blocks that follow the first block. Each entry in the free-space list then consists of a disk address and a count.

Mass storage structure

Overview of Mass Storage Structure:

Magnetic Disks:

→Traditional magnetic disks have the following basic structure:

- One or more platters in the form of disks covered with magnetic media. Hard disk platters are made of rigid metal, while "floppy" disks are made of more flexible plastic.
- Each platter has two working surfaces.
- Each working surface is divided into a number of concentric rings called tracks. The collection of all tracks that are the same distance from the edge of the platter, (i.e. all tracks immediately above one another in the following diagram) is called a cylinder.
- Each track is further divided into sectors, traditionally containing 512 bytes of data each, although some modern disks occasionally use larger sector sizes.
- The data on a hard drive is read by read-write heads. The standard configuration (shown below) uses one head per surface, each on a separate arm, and controlled by a common arm assembly which moves all heads simultaneously from one cylinder to another.

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

- The positioning time, or the seek time or random access time is the time required to move the heads from one cylinder to another.
- The rotational latency is the amount of time required for the desired sector to rotate around and come under the read-write head.This can range any where from zero to one full revolution, and on the average will equal one-half revolution.
- The transfer rate, which is the time required to move the data electronically from the disk to the computer.

→Disk heads "fly" over the surface on a very thin cushion of air. If they should accidentally contact the disk, then a head crash occurs, which may or may not permanently damage the disk or even destroy it completely. For this reason it is normal to park the disk heads when turning a computer off, which means to move the heads off the disk or to an area of the disk where there is no data stored.

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

→Disk drives are connected to the computer via a cable known as the I/O Bus. Some of the common interface formats include Enhanced Integrated Drive Electronics, EIDE; Advanced Technology Attachment, ATA; Serial ATA, SATA, Universal Serial Bus, USB; Fiber Channel, FC, and Small Computer Systems Interface, SCSI.

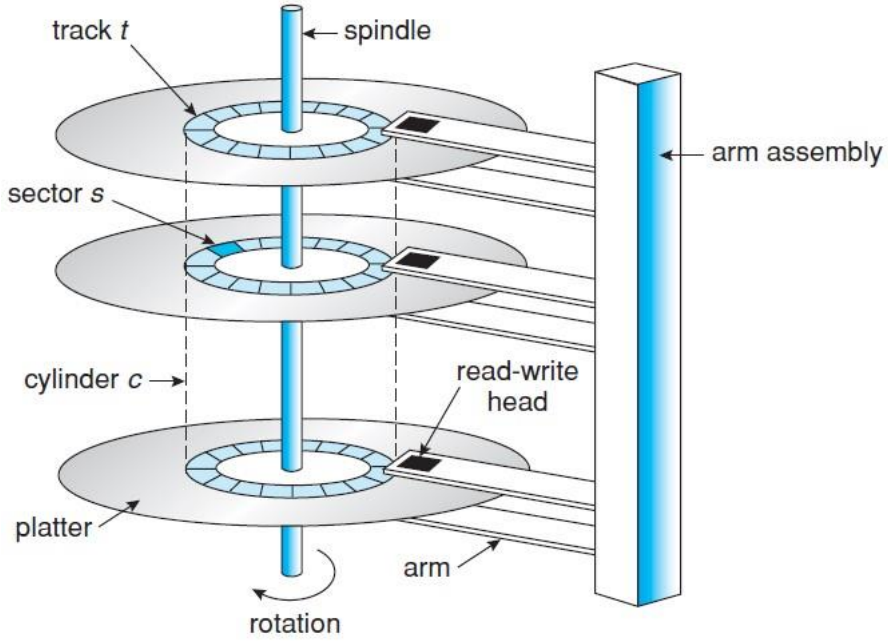


Figure 10.1 Moving-head disk mechanism.

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

Tape capacities vary greatly, depending on the particular kind of tape drive. Typically, they store from 300 GB to 10 TB.

Tapes and their drivers are usually categorized by width, including 4, 8, and 19 millimeters and 1/4 and 1/2 inch. Some are named according to technology, such as LTO-2 and SDLT.

Solid-State Disks: An SSD is nonvolatile memory that is used like a hard drive. SSDs have the same characteristics as traditional hard disks but can be more reliable because they have no moving parts and faster because they have no seek time or latency.

In addition, they consume less power. However, they are more expensive per megabyte than traditional hard disks, have less capacity than the larger hard disks, and may have shorter life spans than hard disks, so their uses are somewhat limited.

Disk Scheduling: Disk Scheduling is done by operating systems to schedule I/O requests arriving for disk. Disk scheduling is also known as I/O scheduling. The following factors effects disk scheduling when multiple requests are made.

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

- The **rotational latency** is the additional time for the disk to rotate the desired sector to the disk head.
- The disk **bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.

By servicing I/O requests in good order we can improve both access time and bandwidth. Whenever a process needs I/O to or from the disk, it issues a system call to the operating system.

If the desired disk drive and controller are available, the request can be serviced immediately. If the drive or controller is busy, any new requests for service will be placed in the queue of pending requests for that drive.

Thus, when one request is completed, the operating system chooses which pending request to service next. The following are different types of disk scheduling algorithms.

1. **FCFS Scheduling:** FCFS is the simplest of all the Disk Scheduling Algorithms. In FCFS, the requests are addressed in the order they arrive in the disk queue. Consider, for example, a disk queue with requests for I/O to blocks on cylinders**:** 98, 183, 37,122, 14, 124, 65, 67

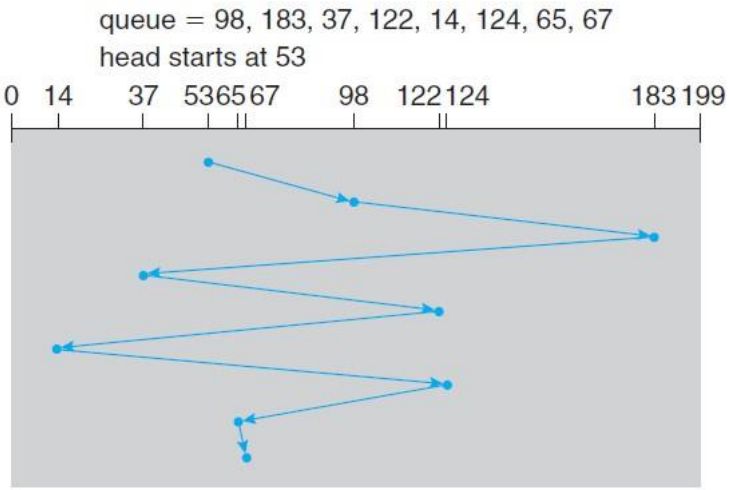


Figure 10.4 FCFS disk scheduling.

If the disk head is initially at cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 124/65, and finally to 67, for a total head movement of 640 cylinders.

Advantages:

- Every request gets a fair chance
- No indefinite postponement

Disadvantages:

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

2. **SSTF Scheduling:** In SSTF (Shortest Seek Time First), requests having shortest seek time are executed first. So, the seek time of every request is calculated in advance in queue and then they are scheduled according to their calculated seek time. As a result, the request near the disk arm will get

executed first. SSTF is certainly an improvement over FCFS as it decreases the average response time and increases the throughput of system.

For our example request queue, the closest request to the initial head position (53) is at cylinder 65. Once we are at cylinder 65, the next closest request is at cylinder 67. From there, the request at cylinder 37 is closer than the one at 98, so 37 is served next. Continuing, service the request at cylinder 14, then 98,122, 124, and finally 183.

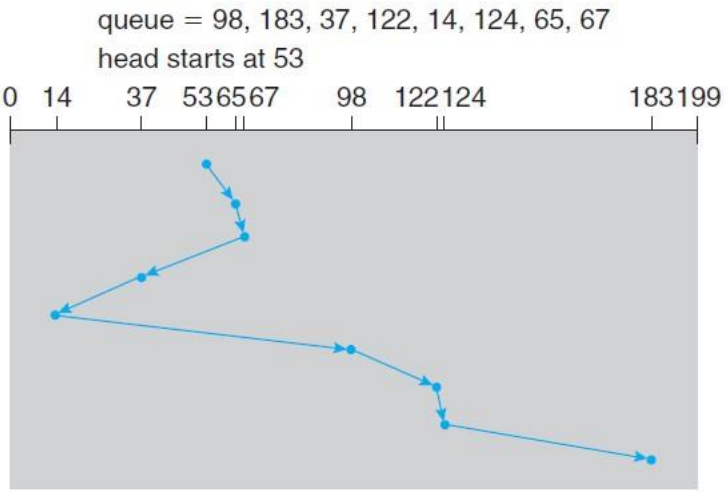


Figure 10.5 SSTF disk scheduling.

This scheduling method results in a total head movement of only 236 cylinders—little more than one-third of the distance needed for FCFS scheduling of this request queue. This algorithm gives a substantial improvement in performance.

Advantages:

- Average Response Time decreases
- Throughput increases

Disadvantages:

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

3. **SCAN Scheduling:** In the **SCAN algorithm,** the disk arm starts at one end of the disk and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk. The SCAN algorithm is sometimes called the **elevator algorithm,** since the disk arm behaves just like an elevator in a building, first servicing all the requests going up and then reversing to service requests the other way.

   Before applying SCAN to schedule the requests on cylinders 98,183, 37,122,14, 124, 65, and 67, we need to know the direction of head movement in addition to the head's current position (53). If the disk arm is moving toward 0, the head will service 37 and then 14. At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the requests at 65, 67, 98, 122, 124, and 183 as shown below.

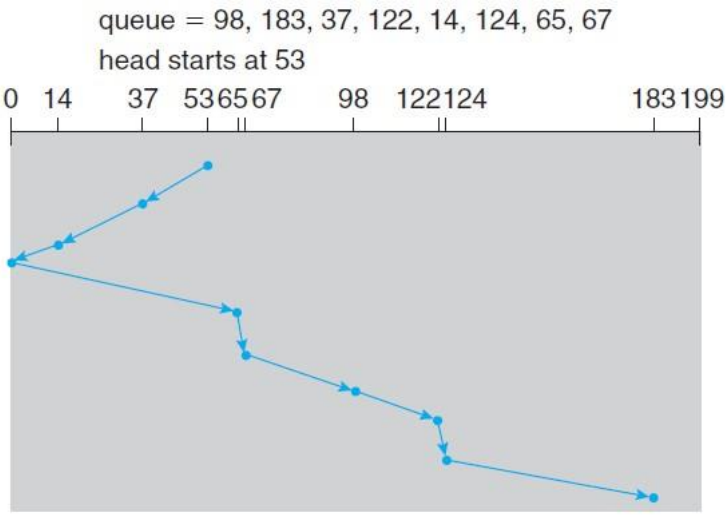


Figure 10.6 SCAN disk scheduling.

Advantages:

- High throughput
- Low variance of response time
- Average response time

Disadvantages:

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

4. **C-SCAN Scheduling:** Circular SCAN (C-SCAN) **scheduling** is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way.

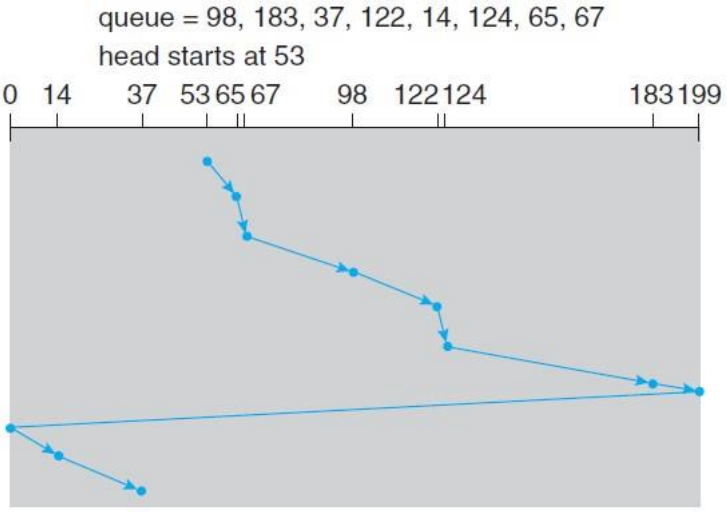


Figure 10.7 C-SCAN disk scheduling.

When the head reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip as shown below. The C-SCAN scheduling algorithm essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one.

Advantages:

- Provides more uniform wait time compared to SCAN

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

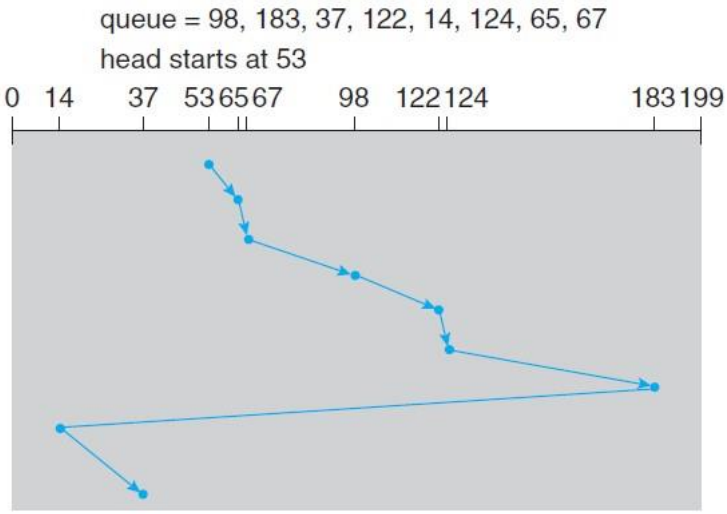


Figure 10.8 C-LOOK disk scheduling.

Swap Space Management: **Swap-space management** is another low-level task of the operating system. The area on the disk where the swapped out processes are stored is called **swap space**.

Virtual memory uses disk space as an extension of main memory. Since disk access is much slower than memory access, using swap space significantly decreases system performance. The main goal for the design and implementation of swap space is to provide the best throughput for the virtual memory system.

Swap Space Use: Swap space is used in various ways by different operating systems, depending on the memory-management algorithms in use.

- For instance, systems that implement swapping may use swap space to hold an entire process image, including the code and data segments.
- Paging systems may simply store pages that have been pushed out of main memory.

It is safer to overestimate than to underestimate the amount of swap space required, because if a system runs out of swap space it may be forced to abort the processes or may crash entirely.

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.

Alternatively, swap space can be created in a separate **raw partition**. No file system or directory structure is placed in this space. Rather, a separate swap-space storage manager is used to allocate and deallocate the blocks from the raw partition.

Some operating systems are flexible and can swap both in raw partitions and in the file system space, example: **Linux**.

Example: 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 dedicated 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 greater than 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 of 3 indicates that the swapped page is mapped to three different processes.

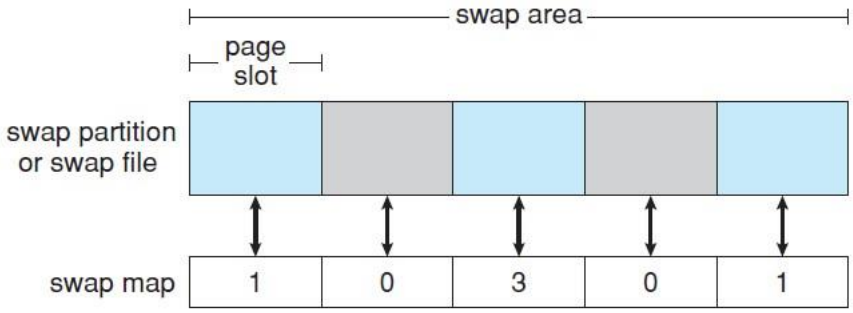


Figure 10.10 The data structures for swapping on Linux systems.

FREQUENTLY ASKED QUESTIONS

1. Write in detail about file attributes, operations and types and structures.
2. What is a File? Describe the attributes of a File?
3. Explain different operations on File.
4. How to provide protection to a file system? Explain.
5. Explain various file access methods with suitable examples.
6. Discuss the Indexed File allocation method with an example.
7. Discuss the Schematic view of a virtual file system with neat sketch.
8. Explain the bit vector representation of free space management.
9. Briefly explain about single-level, two-level and Tree-Structured directories.
10. Write short notes on :i) Contiguous and ii) Linked File allocation methods.
11. Define the terms seek time & rotational latency.
12. Explain File Free Space management approaches. OR Explain the various methods for free-space management

13. Differentiate SCAN, C-SCAN and LOOK, C-LOOK disk scheduling algorithms with an example? Or Explain in detail about various ways of accessing disk storage.

14. Describe in detail about tertiary storage structure or Mass storage structure?

15. Explain and compare the FCFS and SSTF disk scheduling algorithms.

16. Why disk scheduling is needed? Schedule the given requests with all available algorithms. 98, 183, 37, 122, 14, 124, 65, 67, 10, 150.

17. Suppose the read-write head is at track 90, moving towards track 299 (the highest numbered track on the disk) and disk request queue contains read/write requests for sectors on tracks: 86, 242, 171, 26, 281, 92, 13, and 150 respectively. What is the total number of head movements to satisfy the requests in the queue using: (i) FCFS (ii) SSTF (iii) SCAN (iv) C-SCAN