Ambo University Woliso Campus
School of Technology and Informatics
Department of Computer Science

Course title: - Operating System

Module Name: - Computer Architecture and operating system	Program: - Extension
Module code: - CoSc-M2041	Target Group: -CS 3rd extension
Course Name: - Operating System	Semester: - II
Course code: - CoSc2042	Content: - Course Handout
Academic year: - 2020/2012	ECTS credits: - 5
Status of the course: - Core	Instructor Name: - Husen Adem
Instructor Email: - igguumuude@gmail.com	Phone Number: - +251925100878

Course Objectives

At the end of this course you will be able to:

- *Explain the objectives and functions of modern operating systems*
- *Describe the functions of a contemporary operating system with respect to convenience, efficiency, and the ability to evolve.*
- *Explain the different states that a task may pass through and the data structures needed to support the management of many tasks.*
- *Explain conditions that lead to deadlock.*
- *Compare and contrast the common algorithms used for both preemptive and non-preemptive scheduling of tasks in operating systems, such as priority, performance comparison, and fair-share schemes.*
- *Explain the concept of virtual memory and how it is realized in hardware and software*

Table of Contents

Operating System
Chapter One (1)
History and Overview of an Operating System

1. Definition of Operating System

The operating system is an essential part of a computer system; it is an intermediary component between the application programs and the hardware. The ultimate purpose of an operating system is twofold: (1) **to provide various services to users' programs** and (2) **to control the functioning of the computer system hardware in an efficient and effective manner**.

In the simplest scenario, the operating system is the first piece of software to run on a computer when it is booted. Its job is to coordinate the execution of all other software, mainly user applications. It also provides various common services that are needed by users and applications.

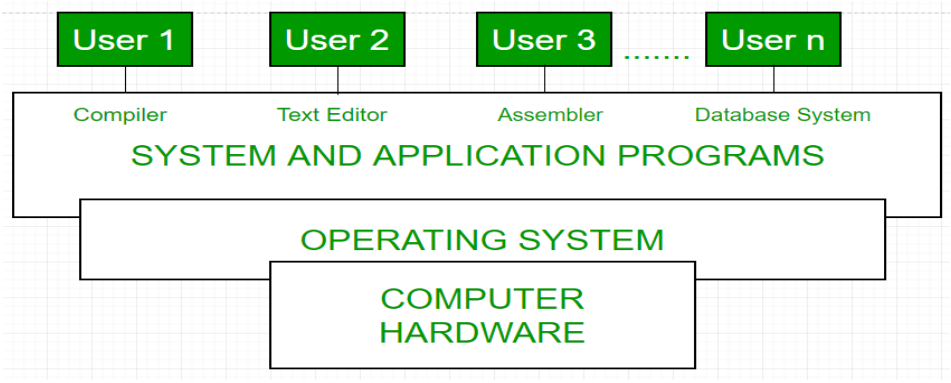


Figure 1-1 Conceptual view of a computer system

Software Components

A program is a sequence of instructions that enables a computer to execute a specific task. Before a program executes, it has to be translated from its original text form (source program) into a machine language program. The program then needs to be linked and loaded into memory. The software components are the collection of programs that execute in the computer. These programs perform computations and control, manage, and carry out other important tasks. There are two general types of software components: - System ***software***, ***Application software.***

System software is the set of programs that control the activities and functions of the various hardware components, programming tools and abstractions, and other utilities to monitor the state of the computer system. The most important part of system software is the operating system (OS) that directly controls and manages the hardware resources of the computer. The OS also provides

a set of services to the user programs. The most common examples of operating systems are *Linux, Unix, Windows, MacOS, and OS/2.*

Application software are the user programs and consists of those programs that solve specific problems for the users and execute under the control of the operating system. Application programs are developed by individuals and organizations for solving specific problems.

The purpose of an operating system involves two key goals:

- ✓ Availability of a convenient, easy-to-use, and powerful set of services that are provided to the users and the application programs in the computer system.
- ✓ Management of the computer resources in the most efficient manner.

Application and system programmers directly or indirectly communicate with the operating system in order to request some of these services.

The services provided by an operating system are implemented as a large set of system functions that include: -

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

The operating system is also considered a huge **resource manager** and performs a variety of services as efficiently as possible to ensure the desired performance of the system. It illustrates the programmers and end users communicating with the operating system and other system software. The most important active resource in the computer system is the CPU. Other important resources are memory and I/O devices. Allocation and deallocation of all resources in the system are handled by various resource managers of the operating system.

General-purpose operating systems support two general modes of operation: (1) **user mode** and (2) **kernel mode**, which is sometimes called the supervisor, protected, or privilege mode. A user process will normally execute in user mode. Some instructions, such as low-level I/O functions and memory access to special areas where the OS maintains its data structures, can execute only in kernel mode. As such, the OS in kernel mode has direct control of the computer system.

Operating System Interfaces

Users and application programmers can communicate with an operating system through its interfaces. There are **three general** levels of interfaces provided by an operating system: -

- ✓ Graphical user interface (**GUI**),
- ✓ Command line interpreter (also called the shell)
- ✓ System-call interface

The highest level is the graphical user interface (GUI), which allows I/O interaction with a user through intuitive icons, menus, and other graphical objects. With these objects, the user interacts with the operating system in a relatively easy and convenient manner for example, using the click of a mouse to select an option or command. The most common GUI examples are the Windows desktop and the X-Window in Unix. The user at this level is completely separated from any intrinsic detail about the underlying system. This level of operating system interface is not considered an essential part of the operating system; it is rather an add-on system software component.

The **second level** of interface, the command line interpreter, or shell, is a text-oriented interface. Advanced users and application programmers will normally directly communicate with the operating system at this level. In general, the GUI and the shell are at the same level in the structure of an operating system. The **third level**, the system-call interface, is used by the application programs to request the various services provided by the operating system by invoking or calling system functions.

2. Definition of interrupt

A signal that gets the attention of the CPU and is usually generated when I/O is required. For example, hardware interrupts are generated when a key is pressed or when the mouse is moved. Software interrupts are generated by a program requiring disk input or output. An internal timer may continually interrupt the computer several times per second to keep the time of day current or for timesharing purposes. When an interrupt occurs, control is transferred to the operating system, which determines the action to be taken. Interrupts are prioritized; the higher the priority, the faster the interrupt will be serviced.

3. Meaning of concurrency and the reasons for its importance

Concurrency is the execution of several instruction sequences at the same time**.** In an operating system, this happens when there are several process threads running in parallel. These threads may communicate with each other through either shared memory or message passing. Concurrency results in sharing of resources result in problems like: - deadlocks and resources starvation. It helps in techniques like coordinating execution of processes, memory allocation and execution scheduling for maximizing throughput. Problems in Concurrency: -

- ✓ sharing global resources safely is difficult;
- ✓ optimal allocation of resources is difficult;
- ✓ locating programming errors can be difficult, because the contexts in which errors occur cannot always be reproduced easily

4. History of Operating system

In order to find the first operative systems, we must travel to the decade of the 50's of the 20th Century. Previously, during the 1940s, programs were introduced directly onto the machine hardware through a series of micro switches. In the 1950s some technologies emerged that allowed a "simpler" interaction between the user and the computer.

Resident monitor: - this is a system that loads the program into the computer, reading it from a **tape** or **punched cards**. This technology gave rise to the first operating system in history, created in 1956 for an IBM 704 computer, which was responsible for loading programs successively (starting with the next one when the previous one had finished loading), reducing the work time required.

Temporary storage: this is a system that also tried to increase speed by simultaneously loading programs and executing tasks. In the 1960s, the rise of the integrated circuit launched the power of computers, and operating systems responded by becoming increasingly complex and offering new techniques.

Multiprogramming: - In this technique, the main memory already holds more than one program, and the operating system is responsible for allocating the machine's resources to execute tasks based on existing needs.

Timeshare: - This is a system that assigns the execution of applications within a group of users working online.

Real time: - it is used specially in the area of telecommunications, it is responsible for processing events external to the computer, so that, once a certain time has passed without success, it considers them as failed.

Multiprocessor: - these are systems that try to manage the readings and writings made in memory by two programs that are running simultaneously, in order to avoid errors. As their name suggests, they are designed for use in computers that use more than one processor.

In the **1970s**, IT continued to become increasingly complex, resulting in the first versions of some of the operating systems that have served as the basis for many of the ones we use today, such as **UNIX**.

The operating systems of this decade are still available only to highly qualified users, and their complexity means that they consume a large amount of resources. Among the most outstanding, in addition to UNIX, we find MULTICS, BDOS and CP/M, widely used in computers with Intel microprocessor.

The **1980s** gave rise to the boom in commercial computing. The arrival of computers in thousands of offices and homes changes the focus of operating systems, forcing the development of more user-friendly systems that introduced graphic elements such as menus.

In this decade the development is such that it gives rise to some operating systems already legendary, and that contribute to the rise of computing in later decades, such as C++, SunOS (developed by Sun Microsystems and derived from UNIX), AmigaOS (developed for the Commodore Amiga) and some classics such as these:-

MS-DOS: - developed by Microsoft for IBM PCs, which contributed enormously to the popularization of computing and gave rise to Windows systems.

Mac OS: - a system of Macintosh computers developed by Apple Inc, launched in 1984, and which included a novel graphic interface and the use of the mouse (a rarity at that time for users that were used to typing commands).

The decade of the **90's** continues with the explosive line marked in the 80's, giving rise to many of the operating systems that, in more modern versions, we use today:

GNU/Linux: it was developed based on UNIX, and which is one of the greatest exponents of free software. Today, GNU/Linux is widely used all over the world, having a pre-eminence close to 100% in fields as striking as supercomputers.

Solaris: - also developed on UNIX basis by Sun Microsystems for servers and workstations.

Microsoft Windows: - which has resulted in a popular family of commercially successful operating systems used by millions of users around the world.

In the first decade of the present century, new operating systems continue to succeed each other, perhaps with less impact than those that emerged in the previous decade, but have their own place. Highlights include SymbOS, MorphOS, Darwin, Mac OS, Haiku and OpenSolaris.

So now we see the current decade, in which the rise of phones gives rise to some popular operating systems, including **Android**, developed by Google or **iOS,** created by Apple.

Exercise

I. Choose appropriate answer from the given letter option.

1. A (an)_____ a sequence of instructions that enables a computer to execute a specific task
   A. Operating System
   B. Software component
   C. Program
   D. Component of Software
2. Users and application programmers can communicate with an _____ through its interfaces
   A. Operating System
   B. GUI (Graphical user interface)

C. CLI (Command Line Interpreter)
D. System Interface

3. First operating system is created for an (a)_____ in 1956
   A. Apple Computer
   B. Timesharing computer
   C. Microsoft company
   D. IBM 704 computer
4. _____is a system that loads the program into the computer, reading it from a tape or punched cards.
   A) Resident Monitor
   B) Temporary storage
   C) Real time
   D) Time Sharing
5. General-purpose operating systems support two general modes of operation; those are: -
   A) User and system Mode
   B) Kernel and system mode
   C) User and kernel mode
   D) System mode

II. Write short Answer for the following question

1) There are two general types computer software component, list them and describe each component clearly. (2)
2) The services provided by an operating system are implemented as a large set of system functions. List those functions!
3) Briefly write operating system history!
4) Accept you are working at AUWC as technician lab assistance, then in a lab 204 four computer system call interface is not working, what will be occur?

Chapter two (2)
Process and Threads

1.1. Operating system type and their function

Batch operating system

- In this type of system, there is no direct interaction between user and the computer.
- The user has to submit a job (written on cards or tape) to a computer operator.
- Then computer operator places a batch of several jobs on an input device.
- Jobs are batched together by type of languages and requirement.
- Then a special program, the monitor, manages the execution of each program in the batch.
- The monitor is always in the main memory and available for execution.

Advantages: -

- No interaction between user and computer.
- No mechanism to prioritize the processes.

Disadvantages: -

- Large Turnaround time.
- More difficult to debug program.
- Due to lack of protection scheme one batch job can affect pending jobs.

Time sharing operating system

Time Sharing is a logical extension of multiprogramming. Multiple jobs are executed simultaneously by switching the CPU back and forth among them. The switching occurs so frequently (speedy) that the users cannot identify the presence of other users or programs. Users can interact with his program while it is running in timesharing mode. Processor's time is shared among multiple users. An interactive or hands on computer system provides online communication between the user and the system. A time-shared operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer. Each user has at least one separate program in memory. A time-shared operating system allows many users to share computer simultaneously. Since each action or command in a time-share system tends to be short, only a little CPU time is needed for each user.

Advantages: -

- Easy to use
- User friendly
- Quick response time

Disadvantages: -

- If any problem affects the OS, you may lose all the contents which have stored already.
- Unwanted user can use your own system in case if proper security options are not available.

Real Time Operating System

A real time operating system is used, when there are rigid (strict) time requirements on the operation of a processor or the flow of data. It is often used as a control device in a dedicated application. Systems that control scientific experiments, medical imaging systems, and industrial control system are real time systems. These applications also include some home appliance system, weapon systems, and automobile engine fuel injection systems. Real time Operating System has well defined, fixed time constraints. Processing must be done within defined constraints or the system will fail. Since meeting strict deadlines is crucial in real time systems, sometimes an operating is simply a library linked in with the application programs. There are two types of real time operating system,

Hard real system:

This system guarantees that critical tasks complete on time. Many of these are found in industrial process control, avionics, and military and similar application areas. This goal says that all delays in the system must be restricted.

Soft real system:

In soft real-time system, missing an occasional deadline, while not desirable, is acceptable and does not cause any permanent damage. Digital audio or multimedia systems fall in this category. An example of real time system is e-Cos.

Distributed operating system

The motivation behind developing distributed operating systems is the availability of powerful and inexpensive microprocessors and advances in communication technology. These advancements in technology have made it possible to design and develop distributed systems comprising of many computers that are inter connected by communication networks. The main benefit of distributed systems is its low price/performance ratio.

Advantages Distributed Operating System

- ✓ As there are multiple systems involved, user at one site can utilize the resources of systems at other sites for resource-intensive tasks.
- ✓ Fast processing.
- ✓ Less load on the Host Machine.
- ✓ Types of Distributed Operating Systems

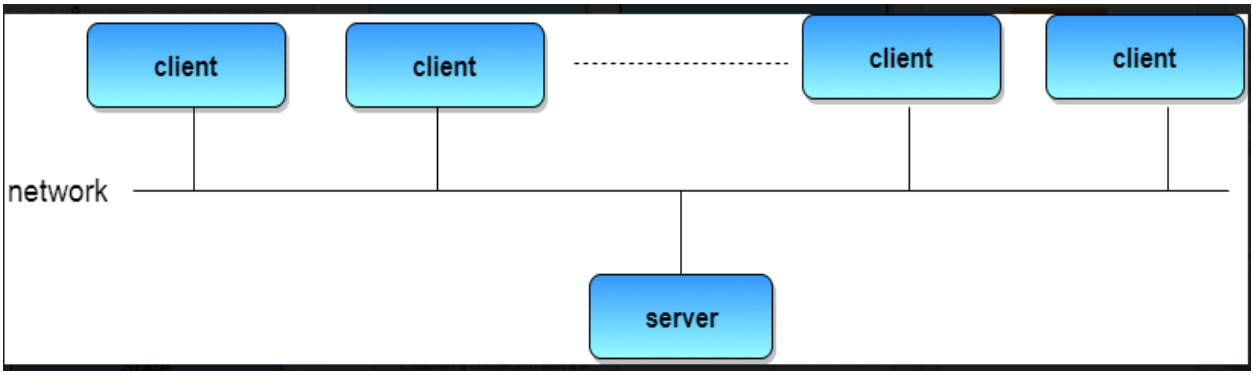
Following are the two types of distributed operating systems used:

- Client-Server Systems
- Peer-to-Peer Systems

Client-Server Systems

Centralized systems today act as **server systems** to satisfy requests generated by **client systems**.

The general structure of a client-server system is depicted in the figure below:



Server Systems can be broadly categorized as: **Compute Servers** and **File Servers.**

- **Compute Server systems**, provide an interface to which clients can send requests to perform an action, in response to which they execute the action and send back results to the client.
- **File Server systems**, provide a file-system interface where clients can create, update, read, and delete files.

Peer-to-Peer Systems

The growth of computer networks - especially the Internet and World Wide Web (WWW) – has had a profound influence on the recent development of operating systems. When PCs were introduced in the 1970s, they were designed for **personal** use and were generally considered standalone computers. With the beginning of widespread public use of the Internet in the 1990s for electronic mail and FTP, many PCs became connected to computer networks.

In contrast to the **Tightly Coupled** systems, the computer networks used in these applications consist of a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory. The processors communicate with one another through various communication lines, such as high-speed buses or telephone lines. These systems are usually referred to as loosely coupled systems (or distributed systems). The general structure of a client-server system is depicted in the figure below:

Personal Computer Operating Systems

The next category is the personal computer operating system. All Modern computers support multiprogramming, often with more than one programs started up at boot time. Their job is to provide good support to a single user. They are widely used for word processing, spreadsheets and Internet access. Common examples of personal computer operating system are Linux, FreeBSD, Windows Vista, and Macintosh operating system.

Handhelds Computer Operating Systems

Continuing on down to smaller and smaller systems, we come to handheld computers. A handheld computer or PDA (personal digital assistant) is a small computer that fits in a pocket and performs a small number of functions, such as electronics address book and memo pad. The OS that runs on handhelds are increasingly sophisticated with the ability to handle telephony, photography and other functions. One major difference between handhelds and personal computer OS is that the former does not have multi gigabyte hard disks. Two of the most popular operating systems for handhelds are Symbian OS and Palm OS.

Embedded Operating Systems

Embedded systems run on the computers that control devices that are not generally thought of as computers and which do not accept user installed software. The main property which distinguishes embedded systems from handhelds is the certainty that no untrusted software will ever run on it. So, there is no need for protections between applications, leading to some simplifications. Systems such as QNX and VxWorks are popular embedded operating system.

Mainframe Operating Systems

The operating system found in those room sized computers which are still found in major corporate data centers. These computers differ from personal computers in terms of their I/O capacity. They typically offer **three** kinds of services: batch, transaction processing, and timesharing. Batch operating system is one that processes routine jobs without any interactive user presents, such as claim processing in an insurance and sales reporting etc. **Transaction processing system** handles large numbers of small requests, for example check processing at a bank and airline reservation. **Time sharing** allows multiple remote users to run jobs on the computer at once, such as querying a database. An example mainframe operating system is OS/390 and a descendant of OS/360.

Server Operating Systems

They run on servers, which are very large personal computers, workstations, or even mainframes. They serve multiple users at once over a network and allow the users to share hardware and software resources. Servers can provide print service, file service or web service. Typically, server operating systems are Solaris, FreeBSD, and Linux and Windows Server 200x.

Operating system architecture

1. **Monolithic system**

In this approach the entire operating system runs as a single program in kernel mode. The operating system is written as a collection of procedures, linked together into a single large executable binary program. When this technique is used, each procedure in the system has a well-defined interface in terms of parameters and results, and each one is free to call any other one, if the latter provides some useful computation that the former needs. To construct the actual object program of the operating system, when this approach is used, one

first compiles all the individual procedure and then binds (group) them all together into a single executable file using the system linker.

The services (system calls) provided by the operating system are requested by putting the parameters in a well-defined place (e.g., on the stack) and then executing a trap instruction. This instruction switches the machine from user mode to kernel mode and transfers control to the operating system. The operating system then fetches the parameters and determines which system call is to be carried out. This organization suggests a basic structure for the operating system. A main program that invoke (call up) the requested service procedure. A set of service procedures that carry out the system calls. A set of utility procedures that help the service procedure. In this model, for each system call there is one service procedure that takes care of it and executes it. The utility procedures do things that are needed by several services' procedure, such as fetching data from user programs.

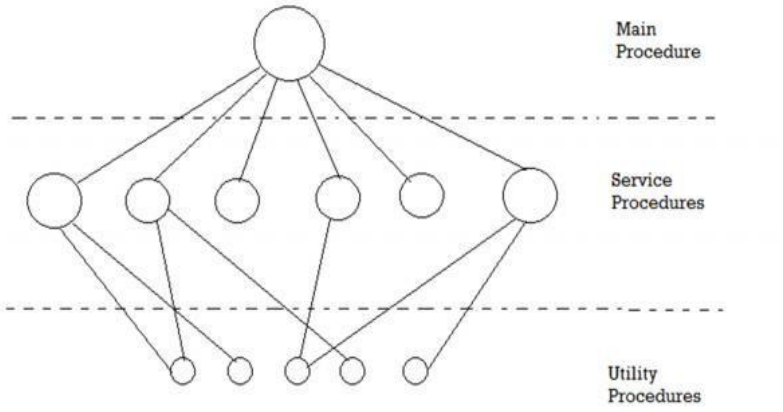


Figure 1-3. A simple structuring model for a monolithic system.

2. Layered system

In this system, operating system is organized as a hierarchy of layers

Layer	Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

The system had six layers. Layer 0 dealt with allocation of the processor, switching between processes when interrupts occurred or timers expired. Layer 0 provided the basic multiprogramming of the CPU. Layer 1 did the memory management. It allocated space for process in main memory and on a 512K word drum used for holding parts of processes for which there was no room in main memory.

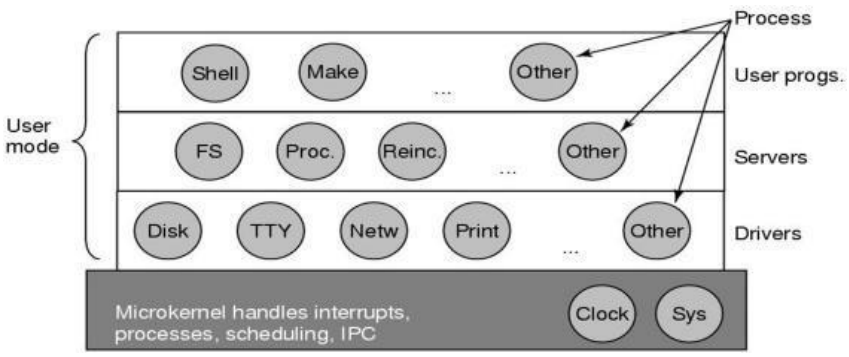
Layer 2 handled communication between each process and the operator console (i.e. user). Layer 3 takes care of managing the I/O devices and buffering the information streams to and from them. Layer 4 was where the user programs were found. The system operator process was located in layer 5.A further generalization of the layering concept was present in the MULTICS system. Instead of layers, MULTICS was described as having a series of concentric rings, with the inner ones being more privileged than the outer ones.

When a procedure in an outer ring wanted to call a procedure in an inner ring, it had to make the equivalent of a system call, that is, a TRAP instruction whose parameters were carefully checked for validity before allowing the call to proceed. Although the entire operating system was part of the address space of each user process in MULTICS, the hardware made it possible to designate individual procedures (memory segments, actually) as protected against reading, writing, or executing.

3. Microkernel

With the layered approach, the designers have a choice where to draw the kernel user boundary. Traditionally, all the layers went in the kernel, but that is not necessary. In fact, a strong case can be made for putting as little as possible in kernel mode because bugs in the kernel can bring down the system instantly. In contrast, user processes can be set up to have less power so that a bug may not be fatal. The basic idea behind the microkernel design is to achieve high reliability by splitting the operating system up to into small, well defined modules, only one of which the microkernel runs in kernels mode and the rest of all are powerless user processes which would run in user mode. By running each device driver and file system as separate user processes, a bug in one of these can crash that component but cannot crash the entire system. Examples of microkernel are Integrity, K42, L4, PikeOS, QNX, Symbian, and MINIX 3.

MINIX 3 microkernel is only 3200 lines of C code and 800 lines of assembler for low level functions such as catching interrupts and switching processes. The C code manages and schedules processes, handles inter-process communication and offer a set of about 35 systems calls to the rest of OS to do its work.



Structure of MINIX 3 system.

The process structure of MINIX 3 is shown in figure above with kernel call handler labeled as *Sys*. The device driver for the clock is also in the kernel because the scheduler interacts closely with it. All the other device drivers run as separate user processes. Outside the kernel, the system is structured as three layers of processes all running in user mode. The lowest layer contains the device driver. Since they run in user mode, they do not have access to the I/O port space and cannot issue I/O commands directly.

Above driver is another user mode layer containing servers, which do most of the work of an operating system. One interesting server is the reincarnation server, whose job is to check if the other servers and drivers are functioning correctly. In the event that a faulty one is detected; it is automatically replaced without any user intervention. All the user programs lie on the top layer.

Process

A process is a program in execution, ready to execute, or one that has executed partially and is waiting for other services in the computer system. In simpler terms, a process is an instantiation of a program, or an execution instance of a program. A process is a dynamic entity in the system because it exhibits behavior (state changes), and is capable of carrying out some (computational) activity, whereas a program is a static entity. This is similar to the difference that exists between a class (a static entity) and an object (a dynamic entity) in object-oriented programming. Two basic types of processes are system processes and user processes. System processes execute operating system services, user processes execute application programs. Process management is one of the major functions of the operating system; it involves creating processes and controlling their execution. In most operating systems, several processes are stored in memory at the same time and the OS manages the sharing of one or more processors (CPUs) and other resources among the various processes. This technique implemented in the operating system is called multiprogramming.

One of the requirements of multiprogramming is that the OS must allocate the CPUs and other system devices to the various processes in such a way that the CPUs and other devices are maintained busy for the longest total time interval possible, thus minimizing the idle time of these devices. If there is only one CPU in the computer system, then only one process can be in execution at any given time. Some other processes are ready and waiting for CPU service while other processes are waiting or receiving I/ 0 service. The CPU and the I/ 0 devices are considered active resources of the system because these resources provide some service to the various processes during some finite interval of time. Processes also request access to passive resources, such as

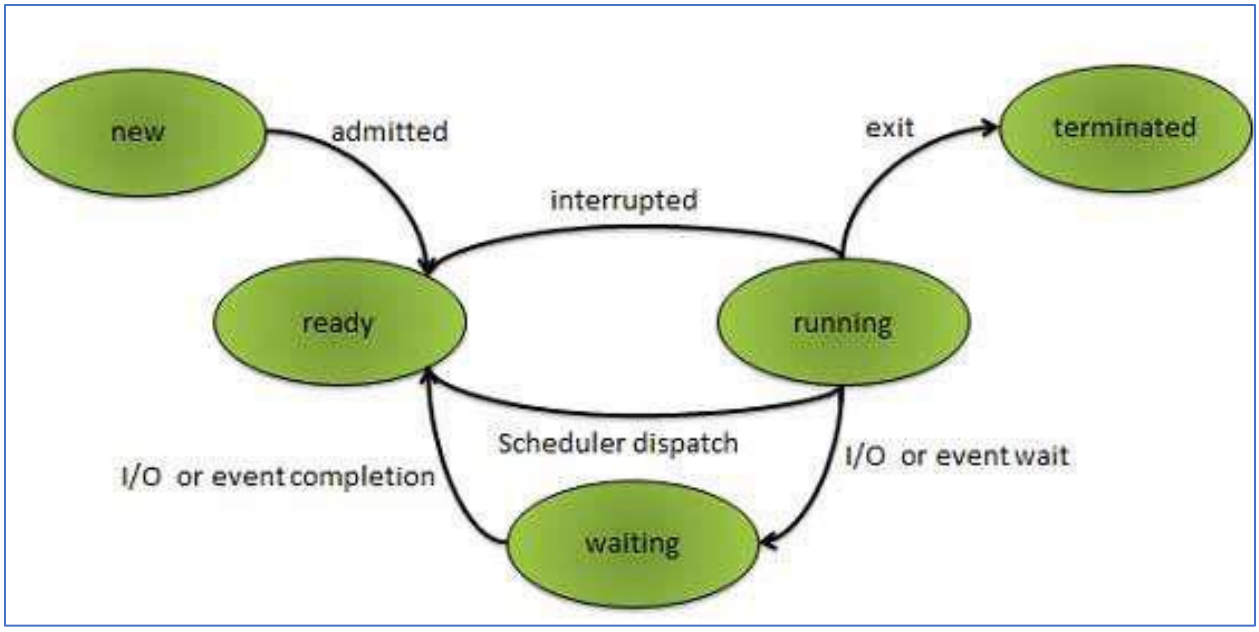
memory. The operating system can be represented by an abstract machine view, in which the processes have the illusion that each one executes on its own machine. Using this abstraction, a process is defined as a computational unit with its own environment, and components that include a process identifier, address space, program code, data, and resources required. For every program that needs execution in the system, a process is created and is allocated resources (including some amount of memory). The address space of a process is the set of memory addresses that it can reference.

Process States

The process manager controls execution of the various processes in the system. Processes change from one state to another during their lifetime in the system. From a high level of abstraction, processes exhibit their behavior by changing from one state to the next. The state changes are really controlled by the OS. For example, when the process manager blocks a process because it needs a resource that is not yet available, the process changes from the running state to the waiting for resource state. When a process is waiting for service from the CPU, it is placed in the ready queue and is in the ready state. In a similar manner, when a process is waiting for I/O service, it is placed in one of several I/O queues and it changes to a wait state. The OS interrupts a process when it requests a resource that is not available. If a process is interrupted while executing, the OS selects and starts the next process in the ready queue. During its lifetime, a process will be in one of the various states mentioned previously:

- Created
- Waiting for CPU (i.e., the process is ready)
- Executing (i.e., receiving service from the CPU)
- Waiting for I/O service (the process is blocked)
- Receiving I/O service
- Waiting for one or more passive resources
- Interrupted by the OS
- Terminated

A state diagram represents the various states of a process and the possible transitions in the behavior of a process. Figure below shows the state diagram of a typical



Threading

In addition to processes, modern operating systems support computational units called threads. A process can have multiple threads or sequences of executions. A thread is often called a lightweight process and is a (dynamic) component of a process. Several threads are usually created within a single process. These threads share part of the program code and the resources that have been allocated to the process. Most modern operating systems support multithreading feature that allows multiple threads to execute within a single process. Multithreading enables a programmer to define several tasks in a single process; each task is assigned to a thread.

1. Multithreading

The operating system manages processes and threads in a multiprogramming environment. From a computational point of view, the execution of threads is handled much more efficiently than the execution of processes. According to this view, threads are the active elements of computation within a process. All threads that belong to a process share the code and resources of the process. The thread identifier uniquely identifies the thread. The process that owns the thread represents the environment in which the thread executes. Threads have their own attributes, such as

- Execution state
- Context (the program counter within the process)
- Execution stack
- Local memory block (for local variables)
- Reference to the parent process to access the shared resources allocated to the process

A thread descriptor-a data structure used by the OS to store all the relevant data of a thread-contains the following fields: -

- Thread identifier
- Execution state of the thread
- Process owner of the thread
- List of related threads
- Execution stack
- Thread priority
- Thread-specific resources

2. User-Level Threads

With the user-level threads (ULT), the thread management is carried out at the level of the application without kernel intervention. The application carries out thread's management using a thread library, such as the POSIX P threads Library. Using this standard library, the application invokes the appropriate functions for the various thread management tasks such as creating, suspending, and terminating threads. As mentioned previously, an application starts executing as a process with the base thread. The process can then create new threads and pass control to one of them to execute. Thread switching and other thread management tasks are carried out by the process using other functions of the thread library. All the necessary data structures are within the address space of a process. The scheduling of threads is normally independent of the scheduling of processes, which is carried out at the kernel level.

3. Kernel level thread

With kernel-level threads (KLT), the thread management tasks are carried out by the kernel. A process that needs these thread-handling services has to use the system call interface of the kernel thread facility. The kernel maintains all the information for the process and its threads in the descriptors previously described. One of the advantages of kernel-level threads is that the process will not be blocked if one of its threads becomes blocked. Another advantage is the possibility of scheduling multiple threads on multiple CPUs. Linux, Unix (several implementations), Windows, and OS/2 are examples of operating systems that provide kernel-level threads. Sun Solaris (Sun Microsystems) provides a facility with combined user-level and kernel-level threads.

Exercise

I. Choose the best answer from the given alternative option

1) Server Systems can be broadly categorized as: _____and_____
   A. Compute servers and follow servers
   B. File Servers and Data Servers
   C. Compute Servers and File Servers
   D. Data Servers and Datum servers
2) A (an)_____ is a program in execution, ready to execute
   A. Program
   B. Threads
   C. Operating system
   D. Process
3) Among the following which one is the advantages of time-sharing operating system?
   A) No interaction between user and computer
   B) Easy to use
   C) Fast Processing
   D) Less Load on the Host Machine
4) In _____ system, missing an occasional deadline, while not desirable, is acceptable and does not cause any permanent damage.
   A) Hard real system
   B) Soft real system
   C) Real time operating system
   D) Batch operating system
5) The _____ of a process is the set of memory addresses that it can reference.
   A) Address space
   B) Process states
   C) Process manager
   D) Process ID

II. Matching parts. Based on their relation match B to A

A	B
1. Client-Server system	A. Decentralized system
2. Per-to-peer system	B. Centralized system
3. Monolithic system	C. organized as a hierarchy of layers
4. Layered System	D. Single program in kernel mode
5. Microkernel	E. MINIX

III. Give the brief answer for the following question.

1. Briefly explain peer to peer system with example!
2. Compare and contrast types of operating system!
3. Write the difference between client server system and peer to peer system!
4. How you define real time operating system? And write example for types of it!

Chapter Three (3)
Design Principles

3.1 Process Control Block (PCB)

For every new process in the system, a data structure called the *process descriptor*, also known as a *process control block* (PCB), is created. It is on this data structure that the OS stores all data about a process. The various queues that the OS manipulates thus contain process descriptors or pointers to the process descriptors, not actual processes. A process descriptor represents a process in the system and when the process terminates, its descriptor is normally destroyed.

For a user process, its *owner is the user*, for a system process, the owner is the *system administrator* or the *system service that created the process*. If a process creates another process, this second process is referred to as a *child process*. The most important of these data fields are as follows: -

- Process name or process identifier (ID)
- Process owner
- Process state
- List of threads
- List of resources
- List of child processes
- Address space
- Process privileges or permissions
- Current content of the various hardware registers in one of the CPUs before the context switch

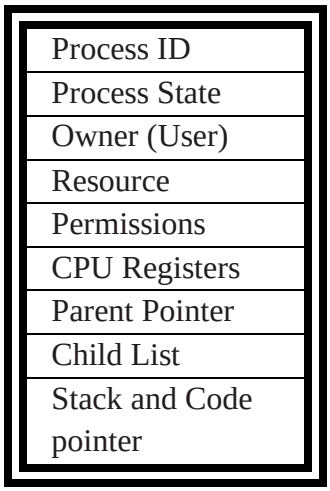


Figure Simplified structure of a process descriptor or process control block (PCB)

Process control block includes CPU scheduling, I/O resource management, file management information etc. The PCB serves as the repository for any information which can vary from process to process. Loader/linker sets flags and registers when a process is created. If that process gets suspended, the contents of the registers are saved on a stack and the pointer to the particular stack frame is stored in the PCB. By this technique, the hardware state can be restored so that the process can be scheduled to run again.

3.2. Context Switching

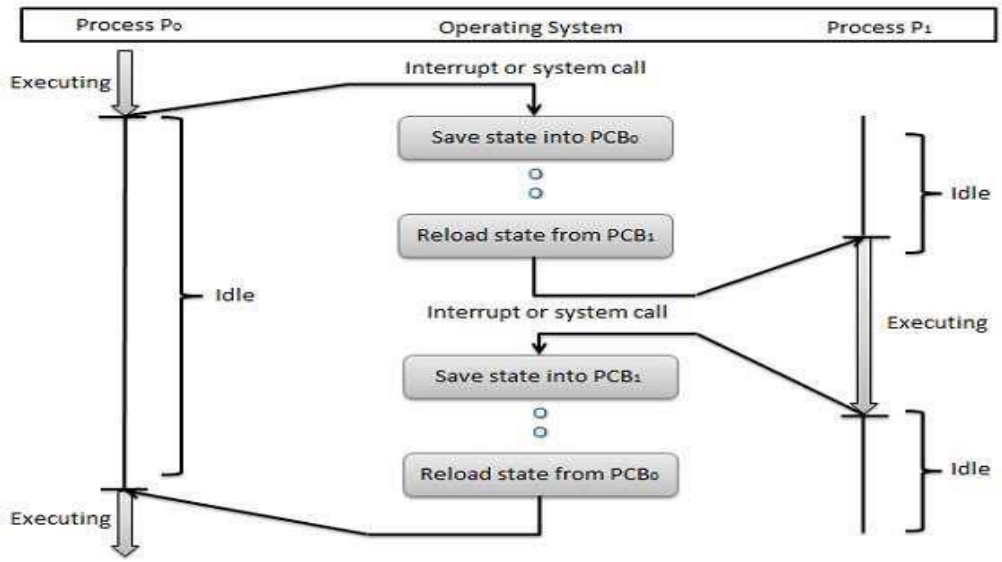
When the OS interrupts an executing process, it carries out a *context switch* the *changing of the CPU from one process to another*. This involves deallocating the CPU from the current process and allocating the CPU to the next process selected by the scheduler. This is a simplistic explanation of a context switch. The context of the current process includes the complete information of the process and it is stored in its PCB. For the other process, its complete context is loaded from its PCB. Context switching is carried out with hardware support. The time it takes for this changeover is called the context switch time. This time interval should be very short because it is considered overhead or nonproductive time during which the CPU is not servicing any (user) processes.

3.3. Interrupting Processes

When a process is executing, it is in the running state. At any time, instant, the OS may interrupt the process and switch to the next process that is waiting for CPU service. There are three basic reasons for interrupting this process:

- The running process needs an I/ 0 operation, such as reading data from a file. In this case, the process invokes a system function of the OS and suspends itself. When the I/O operation is complete, the process is reactivated and placed again in the ready queue to wait for CPU service. In the meantime, the OS has switched to the next process and starts to execute it.
- The running process is interrupted by a timer under OS control. The process returns to its ready state in the ready queue to wait for CPU service. The OS removes the next process from the ready queue and switches to this process.
- A high priority process arrives or is made to transition to the ready state, and the OS interrupts the running process if it has a lower priority. The interrupted process is normally returned to the ready queue to wait for CPU service.

Some hardware systems employ two or more sets of *processor registers* to reduce the amount of context switching time. When the process is switched, the following information is stored.



The selection of the next process to select from the ready queue is carried out by the *scheduler*. The allocation of the CPU to this new process is carried out by the *dispatcher*, another component of the OS. Assume that there is one CPU and several I/O devices. Only one process is receiving CPU service-that is, only one process is actually executing. Another process is receiving I/O service from a specific I/O device. Other processes in memory are waiting either for CPU or for I/O service. Processes in memory could also be waiting for an event (such as the unlocking of a list) to occur. As mentioned earlier, computer systems have the capabilities to overlap CPU and I/O service-that is, they provide these various services simultaneously to different processes. For example, the CPU is servicing process Pl while the I/O device 1 is servicing process P2, while the I/O device 2 is providing service to process P3. These three activities are occurring at the same time; processes Pl, P2, and P3 are all receiving different services at the same time. The computer system controls this overlapping of CPU and I/O devices via its interrupt mechanisms.

Dispatcher

The **dispatcher** is a *kernel module* that takes control of the CPU from the current process and gives it to the process selected by the short-term scheduler. This function involves:

- ***Switching the context*** (i.e., saving the context of the current process and restoring the context of the newly selected process)
- *Switching to user mode*
- ***Jumping to the proper*** location in the user program to restart that program

The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**.

DEADLOCK

Deadlock is the state of indefinite waiting that processes may reach when competing for system resources or when attempting to communicate. When a process requests a resource to the operating system, it normally waits for the resource to become available. A process acquires a resource when the operating system allocates the resource to the process. After acquiring a resource, the process holds the resource. The resources of concern are those that will normally not be shared, such as a printer unit, a magnetic tape unit, and a CD unit. These kinds of resources can only be acquired in a mutually exclusive manner. During its normal execution, a process usually needs access to one or more resources or instances of a resource type. The following sequence of steps describes how a process acquires and uses resources:

a. Request one or more resource instances.
b. Acquire the resource instances if they are available. The operating system allocates an available resource when it is requested by a process. If the resource instances are not available, the process has to wait until the resource instances become available.
c. Use the resource instance for a finite time interval.
d. Release the resource instances. The operating system deallocates the resource instances and makes them available for other requesting processes.

Suppose there are two processes, Pl and P2, and two resource types, RI and R2. The processes are attempting to complete the following sequence of steps in their executions:

1. Process Pl requests resource R1.
2. Process P2 requests resource R2.
3. Process Pl acquires resource R1.
4. Process P2 acquires resource R2.
5. Process Pl requests resource R2.
6. Process Pl is suspended because resource R2 is not available.
7. Process P2 requests resource R1.
8. Process P2 is suspended because resource R1 is not available.
9. The executions of both processes have been suspended.

If the operating system does not *avoid*, prevent, or detect deadlock and recover, the deadlocked processes will be blocked forever and the resources that they hold will not be available for other processes.

Necessary Conditions

There are four conditions that are necessary for existence of deadlock: -

1. **Mutual Exclusion** - At least one resource must be held in a non-sharable mode; If any other process requests this resource, then that process must wait for the resource to be released.
2. **Hold and Wait** - A process must be simultaneously holding at least one resource and waiting for at least one resource that is currently being held by some other process.

3. **No preemption** - Once a process is holding a resource (i.e. once its request has been granted), then that resource cannot be taken away from that process until the process voluntarily releases it.
4. **Circular Wait** - A set of processes {P0, P1, P2, . . ., PN} must exist such that every P [I] is waiting for P [(I + 1) % (N + 1)]. (Note that this condition implies the ***hold-and-wait condition***, but it is easier to deal with the conditions if the four are considered separately.)

Resource Allocation Graph

In some cases, deadlocks can be understood more clearly through the use of **Resource-Allocation Graphs**, having the following properties:

- A set of **resource categories**, {R1, R2, R3, . . ., RN}, which appear as **square nodes** on the graph. Dots inside the resource nodes indicate specific instances of the resource. (E.g. *two dots might represent two laser printers.)*
- A set of processes, {P1, P2, P3, . . ., PN}
- **Request Edges** - A set of directed arcs from Pi to Rj, indicating that process Pi has requested Rj, and is currently waiting for that resource to become available.
- **Assignment Edges** - A set of directed arcs from Rj to Pi indicating that resource Rj has been allocated to process Pi, and that Pi is currently holding resource Rj.
- Note that a **request edge** can be converted into an **assignment edge** by reversing the direction of the arc when the request is granted. (However, note also that request edges point to the category box, whereas assignment edges emanate from a particular instance dot within the box.)

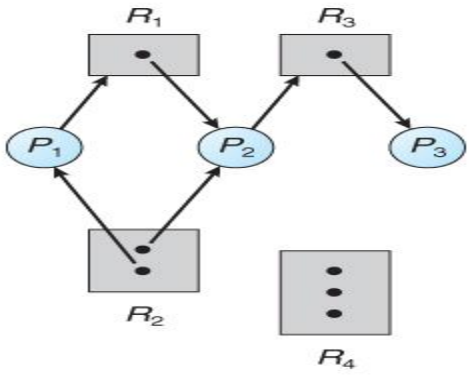
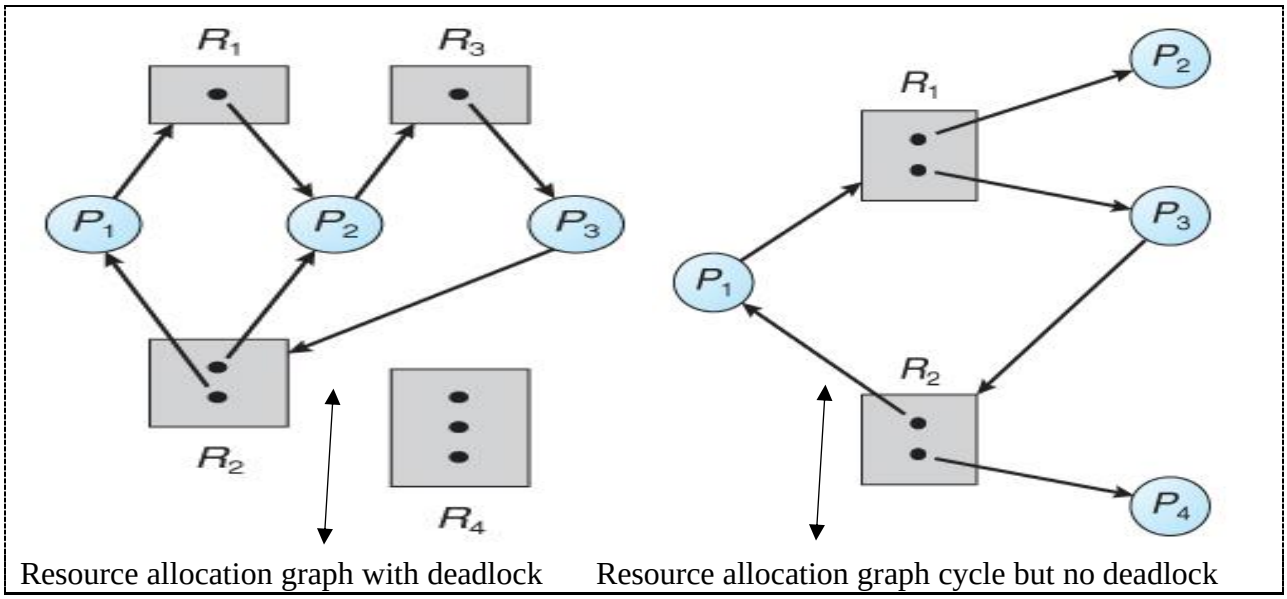


Figure 3.1. Example of resource allocation graph

If a resource-allocation graph contains no cycles, then the system is not deadlocked. (When looking for cycles, remember that these are directed graphs.) See the example in Figure 3.1 above. If a resource-allocation graph does contain cycles AND each resource category contains only a single instance, then a deadlock exists. If a resource category contains more than one instance,

then the presence of a cycle in the resource-allocation graph indicates the possibility of a deadlock, but does not guarantee one.



Resource allocation graph with deadlock Resource allocation graph cycle but no deadlock

Methods for Handling Deadlocks

There are three general methods to handle deadlock:

- Prevention
- Avoidance
- Detection and recovery

The first two methods for dealing with deadlock guarantee that deadlock *will not occur*. The prevention methods are simpler and more *straightforward* to understand and learn. The main rationale used is to disallow *at least one of the four conditions for deadlock*. The deadlock avoidance methods use the current resource-allocation state of the system and the maximum resource claim of the processes. With this information, the operating system can decide to delay the allocation of resources to one or more processes. The third method, detection and recovery, allows a system to reach a deadlock state. A detection algorithm periodically checks if the system has entered a deadlock state. Another algorithm is used to allow the system to recover from a deadlock state.

Deadlock Prevention

As mentioned before, there are four necessary conditions for deadlock to occur:

1. ***Mutual exclusion***
2. ***Hold and wait***.
3. ***Circular wait:***
4. ***No preemption:***

Deadlock prevention methods ensure that at least one of the four conditions are never met (always false). To simplify the following discussion on deadlock prevention, assume that all resources must be accessed in a mutually exclusive manner, and that once a process has acquired a resource, it will not be forced to release the resource. This assumption implies that conditions 1 and 4 will be considered true. The discussion of deadlock prevention will focus on the other two conditions: hold and wait, and circular wait.

1. Disallowing Hold and Wait

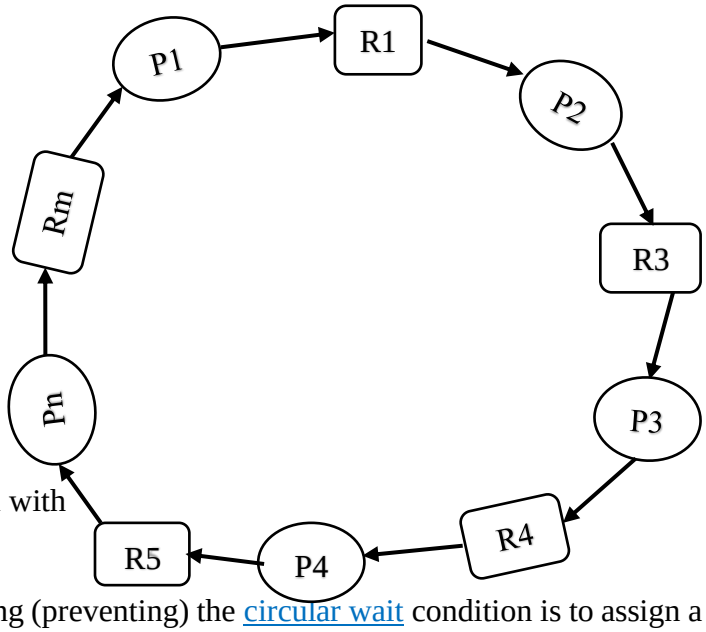
The strategy applied in these techniques for deadlock handling is to preclude a process from holding one or more resources and at the same time requesting other resources. There are two techniques to accomplish this:

- A process must acquire all the resources it needs before starting to use acquired resources.
- A process must release all resources it has acquired before requesting any new resources.

The first technique is more appropriate in batch systems, and the second in interactive systems. In both techniques, the performance of the operating system is decreased. The system throughput, resource utilization, and average wait time of processes are affected. Starvation is also possible because a low priority process may have to wait forever.

2. Disallowing Circular Wait

The circular wait condition exists in a collection of processes Pl to Pn, if process Pl is waiting for a resource held by P2, process P2 is waiting for a resource held by P3, and process Pn is waiting for a resource held by Pl.



Resource allocation graph with Circular waiting

A technique for disallowing (preventing) the circular wait condition is to assign a total ordering to all the resource types in the system. Each resource type can be assigned a unique integer number, or an index number. A simple application of this ordering is to only allow a process to acquire a resource, Rk, if Rk > Rj for some resource Rj that the process already holds. If process Pi requests

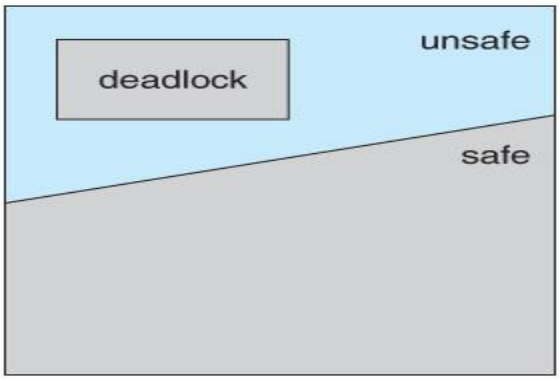
a resource, Rj, and the process already holds resource Rk such that Rk > Rj, the system would not allow the allocation of resources in this order. To follow the total order imposed by the system for resource allocation, process Pi must release resource Rk so that the process can acquire resource Rj and then acquire resource Rk.

Deadlock Avoidance

One method for avoiding deadlocks is to require additional information about how resources may be requested. Each request for resources by a process requires that the system consider the resources currently available, the resources currently allocated to the process, and the future requests and releases of each process, to decide whether the current request can be satisfied or must wait to avoid a possible future deadlock. The simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need. Given a priori information about the maximum number of resources of each type that may be requested by each process, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state. A deadlock avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular wait condition can never exist.

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 $<P_1, P_2 \ldots P_n>$ is a safe sequence for the current allocation state if, for each P_i, the resources that P_i can still request can be satisfied by the currently available resources plus all the resources held by all the P_j with $j < i$. In this situation, if the resources that P_i needs are not immediately available, then P_i can wait until all P_j have finished. When they have finished, P_i 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 terminate. If no such sequence exists, then the system is said to be unsafe. If a system is in a safe state, there can be no deadlocks. An unsafe state is not a deadlocked state; a deadlocked state is conversely an unsafe state. Not all unsafe states are deadlocks, however an unsafe state may lead to a deadlock state. Deadlock avoidance makes sure that a system never enters an unsafe state. The following diagram shows the relationship between safe, unsafe, and deadlock states.

Deadlock, Safe and Unsafe state 

diagram

Let' consider the following example to explain how a deadlock avoidance algorithm works. There is a system with 12 tape drives and three processes.

Process	Max Need	Allocated	Available
P0	*10*	*5*	*3*
P1	*4*	*2*	*5*
P2	*9*	*2*	*10*

The available column shows that initially there are three tapes drives available and when process P1 finishes, the two rape drives allocated to it are returned, making the total number of tape drives 5. With 5 available tape drives, the maximum remaining future needs of P0 (of 5 tape drives) can be met. Once this happens, the 5 tape drives that P0 currently holds will go back to the available pool of drives, making the grand total of available tape drives 10. With 10 available drives, the maximum future need of P2 of 7 drives can be met. Therefore, system is currently in a safe state, with the safe sequence <P1, P0, P2>.

Deadlock Detection and Recovery

With the deadlock detection method, the operating system allocates resources to the requesting processes whenever sufficient resources are available. Since this may lead to deadlock, the operating system must periodically execute a deadlock detection algorithm.

Deadlock Detection

Deadlock detection involves maintaining information of the current resource allocation state and invoking an algorithm that uses the information of the resource allocation state to detect deadlock, if it has occurred.

The operating system monitors the allocation and deallocation of resources to and from processes and updates the resource allocation state. A deadlock detection algorithm basically checks for cycles in the resource allocation graph. There are several algorithms developed for discovering cycles in the resource graph, but the simple algorithms can be applied only when there is one

resource instance of every resource type. An important issue is the frequency of invoking the deadlock detection algorithm. There are two approaches to consider:

- Deadlock can occur only when some process issues a request that cannot be immediately granted by the operating system. Thus, the deadlock detection algorithm can be invoked every time a process requests a resource that cannot be immediately granted (allocated). In this case, the operating system can directly identify the specific process that caused deadlock, in addition to the set of processes that are in deadlock. This approach involves considerable overhead.
- The detection algorithm is invoked periodically using a period not too long or too short. Since deadlock reduces the CPU utilization and the system throughput, the detection algorithm may be invoked when the CPU utilization drops below 40%.

Deadlock Recovery

Once deadlock is detected, the operating system must attempt to recover. There are several approaches for deadlock recovery:

1.Aborting Processes

The simplest approach to deadlock recovery is to terminate one or more processes. An obvious process to kill is one in the circular wait cycle. A more complete strategy can be defined by the following:

- Terminate all processes that have been identified to be in deadlock.
- Preempt resources one by one and run the detection algorithm until deadlock ceases to exist. Roll back these processes to a state previous to their resource allocation.
- Terminate every process and run the detection algorithm to check if deadlock still exists, and continue until deadlock ceases to exist.

2.Rolback

With recovery via rollback, the system periodically stores the state of the system. A checkpoint is a data structure that stores at least the state of a process, each resource state, and a time stamp. The checkpoints are stored on a file and are used to restart a process in a previous state before deadlock occurred. The recovery procedure involves a rollback of every process in a deadlock state to a previous state that has been defined as a checkpoint and stored in a special file.

Synchronization

Synchronization is the coordination of the activities of two or more processes that usually need to carry out the following activities:

- Compete for resources in a mutually exclusive manner.
- Cooperate in sequencing or ordering specific events in their individual activities.

i. No synchronization

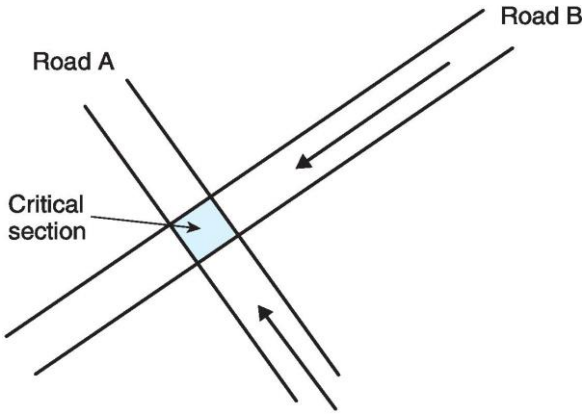
When two or more processes execute and independently attempt to simultaneously share the same resources, their executions generally will not produce correct results. Since synchronization is

absent, the results of these process executions depend on their relative speeds and on the order of use of the resources.

ii. Mutual Exclusion

To solve the race condition problem when a group of processes are competing to use a resource, only one process must be allowed to access the shared resource at a time. In other words, two or more processes are prohibited from simultaneously or concurrently accessing a shared resource. When one process is using a shared resource, any other process that needs access to that resource is excluded from using it at the same time. This condition is called mutual exclusion. At any given time, only one process is allowed to access a shared resource; all other processes must wait.

iii. Critical section



If the two vehicles reach the intersection at the same time, there will be a collision, which is an undesirable event. The road intersection is critical for both roads because it is part of Road A and also part of Road B, but only one vehicle should enter the intersection at any given time. Therefore, mutual exclusion should be applied on the road intersection. The part or segment of code in a process that accesses and uses a shared resource is called the *critical section.* Every process that needs to access a shared resource has its own critical section.

Semaphores

The solution to the *road intersection* analogy shown in above is the installation of a traffic light to coordinate the use of the shared area in the intersection. A semaphore is the equivalent of a traffic light. It is an abstract data type that functions as a software synchronization tool that can be used to implement a solution to the critical section problem. A semaphore is an object whose methods are invoked by the processes that need to share a resource.

Producer-consumer Problems

The *bounded-buffer problem*, also known as the *producer-consumer problem*, involves two processes: (1) the **producer process**, which produces data items and inserts them in the buffer,

one by one; and (2) **the consumer process**, which removes the data items from the buffer and consumes them, one by one. The producer and consumer processes continuously need access to the shared buffer, and both processes operate at their own individual speeds. The problem is to synchronize the activities of both of them. The shared buffer has N slots, each one capable of storing a data item. The producer-consumer problem has the following restrictions:

- The producer cannot deposit a data item into the buffer when the buffer is full.
- The consumer cannot remove a data item from the buffer when the buffer is empty.

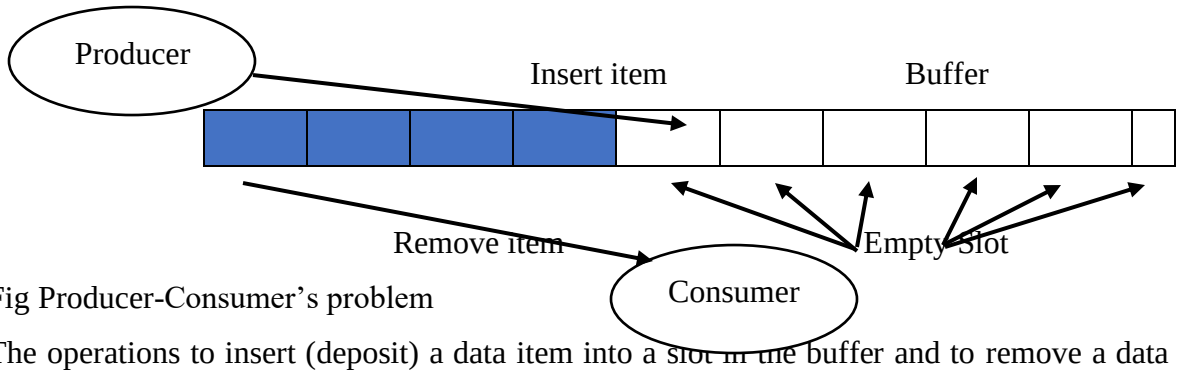


Fig Producer-Consumer's problem

The operations to insert (deposit) a data item into a slot in the buffer and to remove a data item from a slot in the buffer are mutually exclusive. The general solution to the producer-consumer problem requires three semaphores:

- A counting semaphore, full, for counting the number of full slots.
- A counting semaphore, empty, for counting the number of empty slots.
- A binary semaphore, mutex, for mutual exclusion.

Monitors

A monitor is a mechanism that implements mutual exclusion. It is a synchronization tool that operates on a higher level than a semaphore in the synchronization of processes. Monitors are abstract data types implemented as a class. Therefore, they use the encapsulation principle to integrate data and operations and to protect the private members; they are also normally implemented in object-oriented programming languages.

Only a single process can execute an operation of the monitor at any given time. The monitor object provides mutual exclusion, and its member function execution is treated as a critical section. An arriving process must wait in the entry queue of the monitor before entering. After entering, a process can be suspended and later reactivated. Several queues are maintained by the monitor to hold waiting processes that have entered the monitor but have been suspended. In addition to the entry queue, there are one or more condition queues. Each one corresponds to a condition variable, which is similar to a semaphore and is defined with two operations: wait and signal. These operations can be described as follows:

- A process that invokes the wait operation on condition variable x, releases mutual exclusion, places itself on the condition queue for x, and suspends itself.

- A process that invokes the signal operation on condition variable x, and reactivates one waiting process from the condition queue of x; the current process eventually releases mutual exclusion and exits the monitor. The reactivated process reevaluates the condition to continue execution.

A process that completes execution inside the monitor releases mutual exclusion and exits. A new process is then allowed to enter the monitor from the entry queue.

Readers-Writers problems

The readers-writers problem is another classic synchronization problem, slightly more complex than the consumer-producer problem. The readers-writers problem includes two types of processes: readers and writers. There are several processes of each type, and all of these processes share a data resource. The readers access the data resource to read only; the writers need access to the data resource to update the data. The problem must satisfy the following conditions:

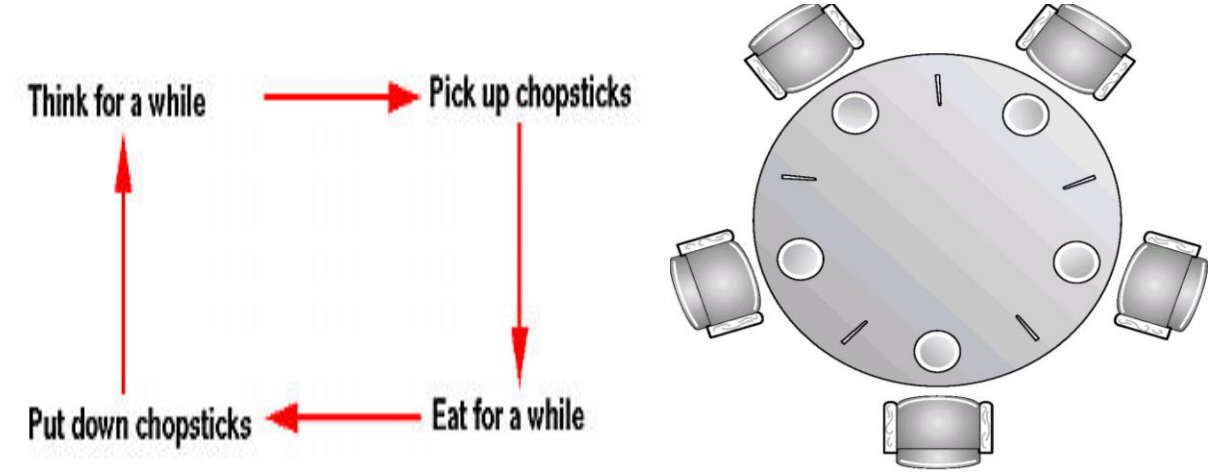
- Any number of reader processes can access the shared data resource simultaneously.
- If a writer process has gained access to the shared data resource, the process has exclusive access to the shared data.

In this problem, there is a need to define two levels of mutual exclusion:

- Individual mutually exclusive access to the shared data resource
- Group exclusive access to the shared data resource

The Dining Philosophers

The dining philosophers is an example of a classical synchronization problem and includes more detailed allocation and deallocation of resources for a set of five processes. Five philosophers spend their lives thinking and eating. When a philosopher is hungry, he or she sits at a round table that has five plates and five forks (or chopsticks), with a fork placed between each pair of plates. In the center of the table, there is a big bowl of spaghetti that represents an endless supply of food. Figure below shows the layout of the resources and the five philosophers at the table. The philosophers are represented by the position each takes at the table, labeled as PO, Pl, P2, P3, and P4. The forks are represented by lines on both sides of every plate, JO, fl, f2, f3, and f 4.



Bankers Algorithms **Reading Assignment**

Exercise

I. Choose the best among the given chooses

1. A Process Control Block (PCB) does not contain which of the following?
   A) Code
   B) Stack
   C) Data
   D) Bootstrap program
2. What is a Process Control Block?
   A) Process type variable
   B) Data Structure
   C) A secondary storage
   D) A Block in memory
3. The entry of all the PCBs of the current processes is in _____
   A) Process Register
   B) Program Counter
   C) Process Table
   D) Process unit
4. Which of the following condition is required for a deadlock to be possible?
   A) Mutual exclusion
   B) A process may hold allocated resource while waiting assignment for other
   C) No resource can be forcibly removed from a process holding it
   D) All can be Answer
5. Which one of the following is a visual (mathematical) way to determine the deadlock occurrence?
   A) Inversion graph
   B) Starvation graph
   C) Resource allocation graph
   D) None of the mentioned
6. A system is in the safe state if _____
   A) The system can allocate resources to each process in some order and still avoid a deadlock
   B) There exists a safe sequence
   C) All of the mentioned above
   D) None of the above
7. To avoid deadlock _____
   A) All deadlock processes must be aborted
   B) Resource allocation must be done only once
   C) There must be a fixed number of resources to allocate
   D) Inversion technique can be used

II. Write answer for the following question, try each with example!

2. Write and discus about bankers' algorithms by your own words according to your understanding!
3. What mean critical section and how to solve critical section?
4. Write the difference between dispatcher and interrupting processes
5. Briefly explain about methods to handle deadlocks

Chapter 4
Scheduling Algorithms

General Types Scheduling

Three general types of scheduling are often considered in studying operating systems:

- **Long-term scheduling**: The operating system decides to create a new process from the jobs waiting in the input queue. This decision controls the degree of multiprogramming. The operating system may decide to create a new process when a currently executing process terminates or when it needs to increase or limit the degree of multiprogramming. The selection of which job to select from the waiting list is based on several criteria.
- **Medium-term scheduling**: The operating system decides when and which process to swap out or swap in from or to memory. This also controls the degree of multiprogramming.
- **Short-term scheduling**: The operating system decides which process to execute next. In other words, the operating system decides when and to which process the CPU will be allocated next. This type of scheduling is often called CPU scheduling.

Process scheduling Concepts

In a system with a multiprogramming operating system, there are usually several processes in the ready queue waiting to receive service from the CPU. The degree of multiprogramming represents the number of processes in memory. CPU scheduling focuses on selecting the next process from the ready queue and allocating the CPU to that process for the duration of its current CPU burst.

Every process that requests CPU service carries out the following sequence of actions:

1. Join the ready queue and wait for CPU processing.
2. Execute (receive CPU service) for the duration of the current CPU burst or for the duration of the time slice (timeout).
3. Join the I/O queue to wait for I/O service or return to the ready queue to wait for more CPU service.
4. Terminate and exit if service is completed-Le., if there are no more CPU or I/O bursts. If more service is required, return to the ready queue to wait for more CPU service.

The CPU scheduler

The CPU scheduler is the part of the operating system that selects the next process to which the CPU will be allocated, deallocates the CPU from the process currently executing, and allocates the CPU to the newly selected process. The basic mechanism used by the scheduler defines three basic functions:

- *Insertion of processes* that request CPU service into the ready queue. This queue is normally implemented as a linked list of process control blocks (PCBs) belonging to the processes waiting for CPU service. This queue is usually a data structure that represents a

simple first-in-first-out (FIFO) list, a set of simple lists, or a priority list. This function is carried out by the enqueuer, a component of the scheduler.

- *The occurrence of a context switch*, carried out by the context switcher that saves the context of the current process and deallocates the CPU from that process.
- *The selection of the next process* from the ready queue and loading its context. This can be carried out by the dispatcher, which then allocates the CPU to the newly selected process.

CPU scheduling policies

There are two general categories of CPU scheduling policies:

- Non preemptive scheduling
- Preemptive scheduling

In *non-preemptive scheduling*, a process that is executing will continue until completion of its CPU burst. The process will then change to its wait state for I/O service, or terminate (change to the terminate state) and exit the system. In *preemptive scheduling*, the process that is executing may be interrupted before completion of its current CPU burst and moved back to the ready queue. A process can be interrupted for one of the following reasons:

- The allocated service interval (time slice) expires.
- Another process with a higher priority has arrived into the ready queue.

Priorities can be used with either preemptive or non-preemptive scheduling. Depending on the goals of an operating system, one or more of various scheduling policies can be used; each will result in a different system performance. The criteria are based on relevant performance measures and the various scheduling policies are evaluated based on the criteria. There are several relevant performance measures to consider:

- **CPU utilization**: The proportion of time that the CPU spends executing processes.
- **Throughput**: The total number of processes that are executed and completed during some observation periods.
- **Process average waiting time**: The average of the waiting intervals of all processes that are completed.
- **Average turnaround time**: The average of the intervals from arrival until completion, for all processes.
- **Average response time:** The average of the intervals from the time a process sends a command or request to the operating system until a response is received, for all processes. This metric is used mainly in interactive systems.
- **Fairness:** A metric that indicates if all processes are treated in a similar manner. The normalized turnaround time is often used for this purpose.

The most relevant scheduling policies in general-purpose operating systems are as follows:

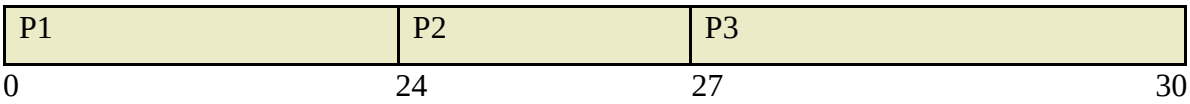
- **First-come-first-served (FCFS):** The order of process arrival to the ready queue determines the order of selection for CPU service. This policy is normally single class and non-preemptive.
- **Shortest job first (SJF):** The process with the shortest CPU burst is the one selected next from the ready queue. Also called shortest process next (SPN), it is typically considered a multiclass and a non-preemptive scheduling policy.
- **Longest job first (L JF):** The process with the longest CPU burst is selected next from the ready queue. Also called longest job next (LJN), it is considered a multiclass policy and typically a non-preemptive scheduling policy.
- **Priority scheduling**: A priority is assigned to each type of process. The process with the highest priority is the one selected next. These scheduling policies are multiclass and can be preemptive or non-preemptive.
- **Round robin (RR):** Processes are basically selected in the same order of arrival to the ready queue but can only execute until the time slice expires. The interrupted process is placed at the back of the ready queue. This scheduling policy can be single-class or multiclass, and it is the most common preemptive scheduling policy used in time-sharing systems.
- **Shortest remaining time (SRT),** also known as shortest remaining time first (SRTF): A new process that arrives will cause the scheduler to interrupt the currently executing process if the CPU service period of the newly arrived process is less than the remaining service period of the process currently executing (receiving CPU service). There is then a context switch and the new process is started immediately.

First-Come First-Served

- Jobs are executed on first come, first served basis.
- It is a non-preemptive scheduling algorithm.
- Easy to understand and implement.
- Its implementation is based on FIFO queue.
- Poor in performance, as average wait time is high.

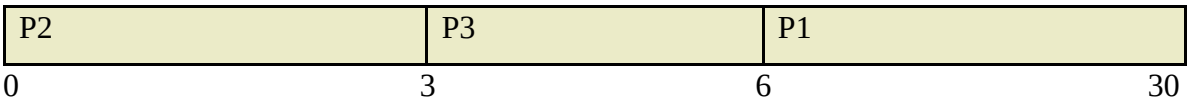
Process	Burst Time
P1	24
P2	3
P3	3

Suppose that the processes arrive in the order: *P1, P2, P3* The Gantt Chart for the schedule is:



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

Suppose that the processes arrive in the order ***P2, P3, P1***. The Gantt chart for the schedule is:



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

➔ Much better than previous case. *Convoy effect* short process behind long process

Process	Arrival Time	Execute Time	Service Time
P0	0	5	0
P1	1	3	5
P2	2	8	8
P3	3	6	16

Example 1

Wait time of each process is as follows:

Process	Wait time = Service time -Arrival time
P0	0 – 0 = 0
P1	5-1 = 4
P2	8 – 2 = 6
P3	16 – 3 = 13

Average waiting time (AWT) = (0 + 4 + 6 + 13)/4 = 23/4 = 5.75

Shortest Job First

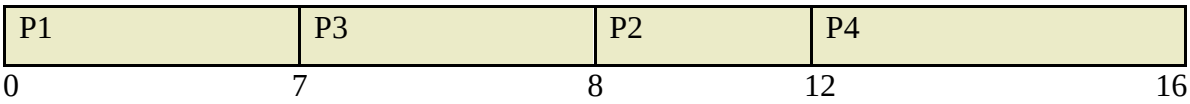
Shortest process next (SPN), also known as **shortest job first** (SJF), is a scheduling policy in which the scheduler selects from the ready queue the process with the shortest CPU service time interval (burst). This scheduling policy can be considered multiclass because the scheduler gives preference to the group of processes with the shortest CPU burst. It is also a non-preemptive scheduling policy. An internal priority is used for each group or class of processes. The operating system assigns a higher priority to the group of processes that has the shortest CPU service time interval (or CPU burst). In other words, the scheduler gives preference to the groups of processes with shorter CPU bursts over other groups of processes. This scheduling policy is not fair compared to FCFS scheduling. Shortest process next scheduling is provably optimal because it results in the minimum wait time for processes. However, when processes with shorter service

periods continue arriving into the ready queue, the processes with longer service demand periods may be left waiting indefinitely. This situation is known as starvation.

SJF is optimal – gives minimum average waiting time for a given set of processes.

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

SJF (non-preemptive)



Average waiting time = [0 +(8-2) +(7-4) +(12-5)] /4 =4

Process	Arrival Time	Execute Time	Service Time
P0	0	5	3
P1	1	3	0
P2	2	8	16
P3	3	6	8

Example of SJF or SPF

Wait time of each process is as follows:

Process	**Wait Time:** Service Time - Arrival Time
P0	3 – 0 = 3
P1	0 – 0 = 0
P2	16 – 2 = 14
P3	8 – 3 = 5

Average Wait Time: (3+0+14+5) / 4 = 5.50

Round Robing Scheduling

Round robin (RR) scheduling is used in time-sharing systems. It is the most common of the preemptive scheduling policies. Every process is allocated the CPU for a short-fixed interval called the time quantum, or time slice. After this short interval expires, the process that is executing (receiving CPU service) is interrupted by the operating system. The time slice is usually much shorter than the average CPU burst of the processes. When the time slice expires, the scheduler

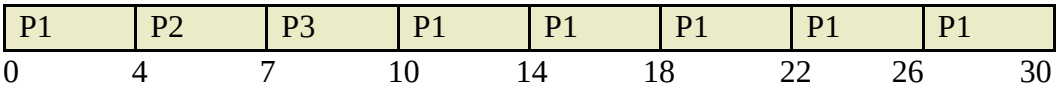
carries out a context switch to the next process selected from the ready queue. After a process executes for the duration of the time slice, it is interrupted and cycled back to the ready queue. In this manner, the ready queue is treated as a circular queue. A process will continue to cycle through the CPU and ready queue until it completes its current CPU burst.

The operating system using this scheduling scheme attempts to allocate the CPU in a uniform manner to all processes in the ready queue for a fixed short interval (the time slice). Thus, all processes in the ready queue are given an equal chance to receive service from the CPU for a short-fixed period. The main advantage that this policy provides to users is interactive computing. The time quantum (or time slice) is considered a system parameter. Its value is usually less than the CPU burst for most processes, and it is much longer than the context switch time. H the time quantum is too long or too short, performance will be affected significantly.

Example 1 of RR with Time Quantum = 4

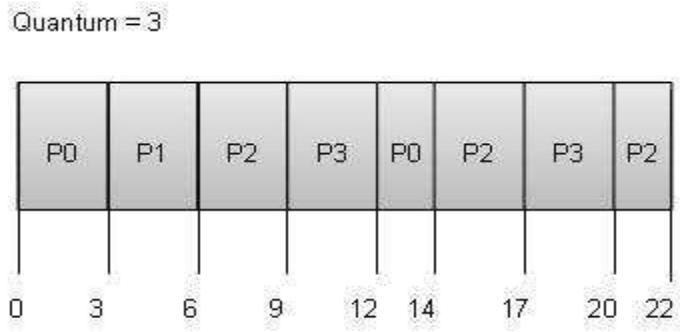
Process	**Burst Time**
P1	24
P2	*3*
P3	*3*

The Gantt chart is:



Average waiting time = [(30-24) +4+7]/3 = 17/3 =5.66

Example 2 of Round Robing with time Quantum = 3



Round Robing example two with quantum time 3. Try by yourself

Wait time of each process is as follows:

Process	**Wait time** = *Service time – Arrival time*
P0	(0 - 0) + (12-3) = 9
P1	(3 - 2) = 4
P2	(6-2) + (14-9) + (20-17) = 12
P3	(9-3) + (17-12) = 11

Average Wait Time: (9+2+12+11) / 4 = 8.5

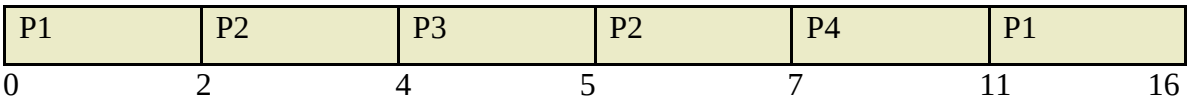
Shortest Remaining Time

Shortest remaining time (SRT), also known as shortest remaining time first (SRTF), is a preemptive version of SPN scheduling. With this scheduling policy, a new process that arrives will cause the scheduler to interrupt the currently executing process if the CPU service time interval of the newly arrived process is less than the remaining service time interval of the process currently executing (receiving CPU service). A context switch occurs and the new process is started immediately.

When a process completes CPU service, the next process selected from the ready queue is the one with the shortest remaining service time. The scheduler selects from the ready queue the process with the shortest CPU service period (burst). As with SPN, this scheduling policy can be considered multiclass because the scheduler gives preference to the group of processes with the shortest remaining service time and the processes with the shortest CPU burst. An internal priority is used for each group or class of processes. The operating system assigns the highest priority to the groups of processes that have the shortest CPU service period (or CPU burst). In other words, the scheduler gives preference to the groups of processes with shorter CPU bursts over other groups of processes. This scheduling policy is not fair compared to FCFS and RR scheduling. When processes with shorter service time continue arriving into the ready queue, the processes with longer service demand times will always be interrupted and may be left waiting indefinitely. As mentioned before, this situation is known as starvation. This is Example

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

SRT (preemptive)



Average waiting time = ***(9 + 1 + 0 +2)/4 =3***

Dynamic Priority Scheduling

In dynamic priority scheduling, the CPU scheduler dynamically adjusts the priority of a process as it is executing. The typical approach is to adjust the priority based on the level of expectation that the process will carry out a system call (typically an I/O request). However, this requires the

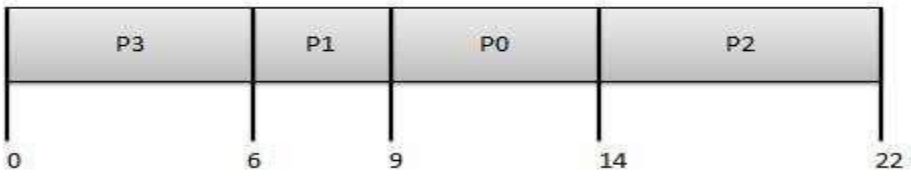
CPU scheduler to predict future process requests. Although we cannot precisely predict the future, we can use an approximation (also known as a heuristic) that is based on observed program behavior: A process will tend to carry out in the near future what it has done in the recent past. Thus, a process that has just carried out an I/O operation will tend to request another I/ 0 operation. This leads to the following algorithm:

1. Allocate the CPU to the highest priority process.
2. When a process is selected for execution, assign it a time slice.
3. If the process requests an I/ 0 operation before the time slice expires, raise its\priority (i.e., assume it will carry out another I/O request soon).
4. If the time slice expires, lower its priority (i.e., assume it is now in a CPU burst) and allocate the CPU to the highest priority ready process.

Some operating systems that implement dynamic priority scheduling will use a fixed time slice value. Other operating systems will make the time slice a function of the priority (giving a shorter time slice to processes that are expected to perform I/O, thus allowing a relatively quick decision that the process is now computing). See Example given below and try it by your-self then check your answer with done answer

Process	Arrival Time	Execute Time	Priority	Service Time
P0	0	5	1	9
P1	1	3	2	6
P2	2	8	1	14
P3	3	6	3	0

Example of priority based scheduling



Wait time of each process is as follows:

Process	**Wait Time**: Service time - Arrival time
P0	9 – 0 = 9
P1	6 – 1 = 5
P2	14 – 2 = 12
P3	0 – 0 = 0

Average Wait Time: (9+5+12+0) / 4 = 6.5

Other scheduling Policies

longest job first

Longest process next (LPN) scheduling, also known as longest job first (LJF), is not a very common scheduling policy. Similar to SPN scheduling, it is a multiclass scheduling policy that can be preemptive or non-preemptive. The only difference with SPN is that higher priorities are assigned to the group of processes with longer CPU bursts. LPN scheduling does not exhibit the fairness shown by FCFS scheduling as processes are given different priorities. Processes with low average CPU service demand may starve or may have to wait too long compared to the processes in the other groups.

Real-Time Scheduling Policies

Real-time systems are ones that continuously interact with an external environment. In these systems, the behavior is defined by the specified *timing constraints*. Real-time systems are sometimes known as *reactive systems*. One of the goals of real-time scheduling is to guarantee fast response of the high priority real-time processes. The second general goal of real-time scheduling is to guarantee that the processes can be scheduled in some manner in order to meet their individual deadlines. The performance of the system is based on this guarantee. A real-time process has a deadline requirement. This process will normally have relatively high priority and must complete its service before the deadline expires. A real-time process can be periodic or sporadic. A *periodic process* is started every **p** time units. This specific time interval is known as the period. The other two relevant timing properties of a periodic process are its computation time requirement, **c**, and its deadline, **d**. The process must complete execution before its deadline expires. A **sporadic process** is normally started by an external random event. After the occurrence of the specified event, the process must start and complete before its deadline expires. Real-time scheduling normally includes priorities and preemption. There are two widely known real-time scheduling policies: rate monotonic and the earliest deadline first. With **rate monotonic scheduling** (RMS), priorities of the processes are statically assigned in reverse order of period length. Higher priorities are assigned to processes with shorter periods, which implies that more frequently executing processes are given higher priority. With earliest deadline first scheduling (EDFS), the priorities of the processes are assigned statically or dynamically. Processes with earlier deadlines are given higher priorities.

Multiprocessor Systems

A multiprocessor computer system has two or more processors. The main goal of these systems is to improve the overall performance of the system. There are two general categories of multiprocessor computer systems: *tightly coupled* and *loosely coupled*. Tightly coupled computer systems have two or more processors that share the system main memory or a common block of memory, and are controlled by the operating system of the computer system. Loosely coupled computer systems are composed of several semi-autonomous units, each with a processor, memory, and communication facilities.

The actual operation and performance of multiprocessor systems depend on the granularity of the configuration. This relates to the synchronization needed among the various processes that execute concurrently or simultaneously. On one extreme (an ideal situation), there are several processes executing in the system; each process is allocated a processor and the execution is completely independent of the other processes. The other extreme is a very fine granularity of parallelism, in which a task needs parallel computing to perform its complex algorithm. Between these extreme levels of granularity, we can identify course, medium, and fine levels of parallelism.

Coarse granularity is used with concurrent processes that need some level of synchronization to share resources and/or to communicate. Medium granularity is used with threads that are executed concurrently and that need synchronization. A multicore processor is a processing unit composed of two or more cores. Each core is capable of executing instructions. Computer systems may be designed using cores configured tightly or loosely. Multicore processors are used in many applications. As with general multiprocessor systems, performance gained by the use of a multicore processor depends very much on the software algorithms and implementation. Many typical applications, however, do not realize significant speedup factors. The parallelization of software is a significant ongoing topic of research.

One of the goals of conventional processor scheduling is to keep execution units busy by assigning each processor a thread to run. Recent research on scheduling multicore systems focus on high utilization of on-chip memory, rather than of execution cores, to reduce the impact of expensive DRAM and remote cache accesses. A simplified model of a multiprocessor system consists of a single ready queue and follows a FCFS scheduling policy. This model is an example of a model of a multiprocessor system with coarse granularity.

Exercise

I. Chose the best answer from the given letter of option

1. Which of the following scheduling algorithm comes under **preemptive** scheduling?
   A) FCFS (first come first service)
   B) Round Robing
   C) Multilevel Queue scheduling
   D) Largest Job First (LJF)
2. Which scheduler **doesn't** maintain the **Degree of Multiprogramming**?
   A) Short-Term Scheduler
   B) Mid-Term Scheduler
   C) Long-Term Scheduler
   D) None of the Above
3. In Preemptive Priority Scheduling, if a high priority process arrives in the ready queue and a low priority process is executing then what will happen?
   A) The currently executing process will be preempted and the new process will be assigned to the CPU.
   B) The CPU will keep on executing the current process and the new process has to wait in the ready queue.
   C) The new process will be shifted to I/O queue.
   D) The system will crash
4. Which module gives control of the CPU to the process selected by the short-term scheduler?
   a. Scheduler
   b. Interrupt
   c. Dispatcher
   d. None of the mentioned
5. The interval from the time of submission of a process to the time of completion is termed as _____
   i. Waiting time
   ii. Turnaround time
   iii. Response time
   iv. Throughput
6. Which algorithm is defined in Time quantum?
   A) Shortest job scheduling algorithm
   B) Priority scheduling algorithm
   C) Round robin scheduling algorithm
   D) Multilevel queue scheduling algorithm
7. Process are classified into different groups in _____
   a. Shortest job scheduling algorithm
   b. Round robin scheduling algorithm
   c. Priority scheduling algorithm
   d. Multilevel queue scheduling algorithm
8. In priority scheduling algorithm, when a process arrives at the ready queue, its priority is compared with the priority of _____
   a. Current running process
   b. All existing process
   c. Parent process
   d. Other process

II. Write correct answer for the following given and try to show necessary step

1. Suppose that the process arrives in the order of: P0, P1, P2, P3 and *burst time* 25, 2, 3 and 5 respectively, draw Gantt chart by using FCFS scheduling algorithm and calculate average waiting time? (2 mark)
2. Suppose that the process arrives in order of: P0, P1, P2 and burst time for each process respectively 6, 20, 4, show Gantt chart and calculate average waiting time by using Round robin scheduling algorithm where quantum is **3.** (2 mark)

Chapter 5
Memory Management

The major tasks of the memory manager are the allocation and deallocation of main memory. Because main memory is one of the most important resources in a computer system, the management of memory can significantly affect the performance of the computer system. Memory management is an important part of the functions of the operating system.

In simple operating systems without multiprogramming, memory management is extremely primitive; memory is allocated to only one program (or job) at a time. In early operating systems with multiprogramming, memory was divided into a number of partitions-blocks of contiguous memory that could be allocated to a process. The degree of multiprogramming determines the number of partitions in memory (i.e., the maximum number of processes that can reside in memory). When a process completes and terminates, memory is deallocated and the partition becomes available for allocation to another process.

One of the problems present in memory management with partitions is memory fragmentation the existence of some amount of allocated memory that is not used by the process, or of relatively small memory blocks that cannot be allocated to a process. This problem reduces the memory utilization and can affect other system performance metrics.

Process Address Space

A *logical address* is a reference to some location of a process. The process address space is the set of *logical addresses* that a process references in its code. The operating system provides a mechanism that maps the logical addresses to physical addresses. When memory is allocated to the process, its set of logical addresses will be bound to physical addresses. Three types of addresses are used in a program before and after memory is allocated:

1. **Symbolic addresses**: The addresses used in a source program. The variable names, symbolic constants, and instruction labels are the basic elements of the symbolic address space.
2. **Relative addresses**: A compiler converts symbolic addresses into relative addresses.
3. **Physical addresses**: The final address generated when a program is loaded and ready to execute in physical memory; the loader generates these addresses.

Binding

The memory manager allocates a block of memory locations to the absolute program and the loader moves the program into the allocated memory block. At load time, the absolute program is loaded starting at a particular physical address in main memory. The relocatable addresses are mapped to

physical addresses; each logical address is bound to a physical address in memory. The absolute program becomes an executable program right after its logical addresses are translated (mapped) to the corresponding physical addresses.

The general address translation procedure is called *address binding*. If the mapping (or conversion) of logical addresses to physical addresses is carried out before execution time, it is known as early or static binding. A more advanced binding technique delays the mapping from logical to physical addresses until the process starts to execute. This second type of binding is called late or dynamic binding.

Static and Dynamic Loading

With static loading, the absolute program (and data) is loaded into memory in order for execution to start. With dynamic loading, the modules of an external library needed by a program are not loaded with the program. The routines of the library are stored on a disk in relocatable form and are loaded into memory only when they are needed by the program. The main advantage of dynamic loading is the improved memory utilization.

Static and Dynamic Linking

When static linking is used, the linker combines all other modules needed by a program into a single absolute load module before execution of the program starts. When dynamic linking is used, the building of the absolute form of a program is delayed until execution time. For every call to a routine of a library, a stub is executed to find the appropriate library in memory. This type of linking is commonly used with shared libraries such as Dynamic Linked Libraries (DLL). Only a single copy of a shared library is needed in memory.

Contiguous Memory Allocation

Operating systems with multiprogramming and simple memory management divide the system memory into *partitions-blocks* of contiguous memory, each one allocated to an active process. The degree of multiprogramming is determined by the number of *partitions in memory*. As mentioned before, when a process completes and terminates, its memory space is deallocated and that amount of memory becomes available. Known as *memory partitioning*, this type of memory management was used in the early multiprogramming operating systems. In addition to the allocation and deallocation of partitions to and from processes, the memory manager also provides two additional basic functions. The first is the *protection of the memory space* in the partition allocated to a

process from the memory references generated by a process in a different partition. The second function is the *management of shared memory in a partition* by two or more processes.

Partitioned memory allocation can be *fixed* or *dynamic*, depending on whether the partitions are fixed-sized or variable-sized blocks of memory. With fixed partitions, the number of partitions is fixed; with variable partitions, the number and size of partitions vary because these are dynamically created when memory is allocated to a process.

Fixed Partitions

In this memory management scheme, memory is divided into fixed-sized partitions that are not normally of the same size. One partition is allocated to each active process in the multiprogramming set. The number and the size of the partitions are fixed. There is one special partition, the system partition, in which the memory-resident portion of the operating system is always stored. The rest of the partitions are allocated to user processes.

An important problem in memory allocation with fixed partitions is fragmentation, the portion of memory allocated but not used. The unused portion of memory inside a partition is called *internal fragmentation.* The selection of a process from the input queue allocating a partition to it is an important issue in memory allocation. There are two general techniques for this:

- A technique using a queue for every partition. A process will be assigned the smallest partition large enough for the process. This technique minimizes the internal fragmentation.
- A technique using a single queue for the processes waiting for memory. The next process is selected from the queue and the system assigns the smallest available partition to the process.

Dynamic Partitions

Dynamic partitioning is a memory management scheme that uses variable size partitions; the system allocates a block of memory sufficiently large according to the requirements of a process. The partition is created dynamically, when there is sufficient memory available. The number of partitions is also variable. The memory manager allocates memory to requesting processes until there is no more memory available or until there are no more processes waiting for memory. Assume that memory was allocated to processes P6, P5, P2, P3, and P4, in that order. The five partitions were created dynamically, and the amount of memory left is located at the top of memory. Contiguous blocks of available (unallocated) memory are called *holes*. If a hole is sufficiently large, it can be allocated to a process of the same or smaller memory size.

In dynamic partitioning, the holes represent the memory that is available, and if they are too small, they cannot be allocated. They represent external fragmentation. The total fragmentation in this case is 75K + 200K. The problem is that this total amount of memory is not contiguous memory; it is fragmented.

The operating system can use several techniques to allocate holes to processes requesting memory. The first technique is called **best-fit**: It selects the hole that is closest in size to the process. The second technique is called **first-fit**: It selects the first available hole that is large enough for the process. The third technique is called **next-fit**: It selects the next available hole that is large enough for the process, starting at the location of the last allocation. The most appropriate allocation technique is not easy to determine. It depends on the arrival sequence of processes into the input queue and their corresponding sizes.

Dynamic partitioning requires the system to use dynamic relocation, a facility to relocate processes in memory, even after execution has begun. This is considered late binding. With hardware support, the relocation of the relative addresses can be performed each time the CPU makes a reference to memory, during the execution of a process.

Swapping

Dynamic relocation is also important in swapping. When a process is blocked (suspended) while waiting for I/O service, the system assumes that the process will not become ready for a relatively long-time interval. The system can swap out (or move) the blocked process to secondary storage (disk) and make that memory available to other processes. At some later time, the system swaps back the process from the secondary storage to main memory. The locations in memory into which the process is swapped back are not normally the same locations where the process was originally stored in main memory.

Performance is usually affected by swapping. The total overhead time includes the time it takes to move the entire process to a disk and to copy the process back to memory, as well as the time the process takes competing to regain main memory (memory allocation). In order for swapping to be effective, this total time must be less than the time the process is to spend blocked (waiting for I/O service). Another important problem is the accumulated wait time of a process when there is not sufficient memory available for the process to be loaded and executed. These two quantities can be used as performance measures of the memory management subsystem.

Noncontiguous Memory Allocation

Fragmentation (internal and external) is the main problem in contiguous memory allocation. Modern operating systems use more advanced memory allocation schemes. This section discusses two common techniques for *noncontiguous memory* allocation: paging and segmentation.

Paging

With noncontiguous memory allocation, the process address space is divided into small fixed-sized blocks of logical memory called *pages*. The size of a process is consequently measured in the number of pages. In a similar manner, physical memory is divided into small fixed-sized blocks of (physical) memory called frames. If a 15-page process is waiting for memory, the system needs to find any 15 frames to allocate to this process. The size of a page is a power of two-for example, a size of 1K = 1024 bytes. The size of a frame is the same as that of a page because the system allocates any available frame to a page of a process. The frames allocated to the pages of a process need not be contiguous; in general, the system can allocate any empty frame to a page of a particular process. With paging, there is no external fragmentation, but there is potential for a small amount of internal fragmentation that would occur on the last page of a process.

Logical Addresses

A logical address of a process consists of a page number and an offset. Any address referenced in a process is defined by the page that the address belongs to and the relative address within that page. When the system allocates a frame to this page, it translates this logical address into a physical address that consists of a frame number and the offset. For memory referencing, the system needs to know the correspondence of a page of a process to a frame in physical memory, and for this, it uses a page table. A page table is a data structure (array or linked list) used by the OS with data about the pages of a process. There is one table entry for every page. Since the logical address of a process consists of the page number and the offset, the least significant bits of the address correspond to the offset, and the most significant bits to the page number.

Address Translation

The operating system maintains a table or list of the currently empty (and available) frames. In addition to this table, for every process the system maintains a table with the frame allocation to each page of the process.

Segmentation

Segments are variable-length modules of a program that correspond to logical units of the program; that is, segments represent the modular structure of how a program is organized. Each segment is actually a different logical address space of the program. Examples of segments are the program's main function, additional functions, data structures, and so on. Before a program can execute, all its segments need to be loaded non-contiguously into memory (these segments do not need to be contiguous in memory). For every segment, the operating system needs to find a contiguous block of available memory to allocate to the segment. There are three main differences between segmentation and paging:

- Not all segments of a process are of the same size.
- The sizes of segments are relatively large compared to the size of a page.
- The number of segments of a process is relatively small.

The operating system maintains a segment table for every process and a list of free memory blocks. The segment table consists of an entry for every segment in a process. For each segment, the table stores the starting address of the segment and the length of the segment. When the system allocates memory to each segment of a process, a segment table is set up for the process. A logical address of a process *consists of two parts*: the *segment number* and *an offset*. For example, suppose a 20-bit address is used with 8 bits for the segment number and 12 bits for the segment offset. The maximum segment size is 4096 (212) and the maximum number of segments that can be referenced is 256 (28). The translation of a logical address to a physical address with segmentation is carried out using the segment table. With the segment number in the left 8 bits of the logical address, the system looks up the segment number from the segment table. Using the length of the segment from the table, the system checks if the address is valid by comparing the segment length with the offset. The starting physical address of the segment is retrieved from the table and added to the offset.

Virtual Memory

The memory space of a process is normally divided into blocks that are either pages or segments. Virtual memory management takes advantage of the typical behavior of a process: Not all blocks of the process are needed simultaneously during the execution of a process. Therefore, not all the blocks of a process need separate main memory allocation. Thus, the physical address space of a process is smaller than its logical address space.

Basic Concepts

The virtual address space of a process is the entire set of all its addresses in the absolute program. After linkage, the absolute version of the program is stored on disk. The disk area that stores all the processes in absolute form is called the *virtual memory*. The physical address space of a process is much smaller than its virtual address because only a portion of the process will ever be loaded into main memory. Assuming that virtual memory is implemented with paging, not all the pages of a process are stored in physical memory. A page reference is the page that has the address being referenced. The virtual memory manager swaps in a page of an executing process whenever the execution of a process references a page that is not in physical memory. Any unused page will normally be swapped out to a disk. The operating system should provide efficient means to translate virtual addresses to physical addresses. The size of the virtual address space is greater than the size of the physical address space. Thus, the operating system must also provide effective and efficient techniques to load the needed blocks of a program as it continues executing. Operating systems implement virtual memory management using segments or pages.

Process Locality

A process in execution only references a subset of its addresses during a specific interval of time. This behavior is called *reference locality*. A process executes in a series of phases and spends a finite amount of time in each phase, referencing a subset of its pages in each phase. This subset of pages is called a *process locality*. The process starts execution in the first phase of execution referencing a subset of its pages (its virtual address space) and is spending a small amount of time in this phase. The process then moves to its next phase of execution and uses another subset of its pages for some other amount of time, and so on until the process terminates. Each subset of its pages is called a locality. The executing process changes from locality to locality.

Memory Protection

Modern operating systems have memory protection that has two goals:

- Processes will not adversely affect other processes.
- Programming errors will be caught before large amounts of damage can be done.

The first goal is achieved by ensuring that a process can access memory only within its own address space. Thus, the operating system will block memory accesses for addresses that are beyond the bounds of the process's memory. The second goal is achieved by portions of the process's memory being marked as

follows:

1. Read enabled: The memory can be read as data.
2. Write enabled: Variables in this area can be changed.
3. Execute enabled: The area contains instructions.

Shared Memory

It is often useful for multiple processes to have access to shared code in memory, which is most often implemented for shared libraries. In this scenario, some of the code needed by a process will be in a dynamic linked library (DLL). When a process first attempts to use this library module, the OS will ensure that it is loaded into memory. If additional processes wish to use this library module, the OS will recognize that it is already loaded into memory and will arrange for the additional processes to have access to the module. Dynamic linked libraries are used for common libraries that many applications use. Using shared libraries saves memory because only one copy of the library module needs to be loaded. Execution time is also saved for the additional processes that wish to use the library module.

Paging with Virtual Memory

As mentioned previously, the operating system translates a virtual address after a physical address after allocating a frame to a page when necessary. The system needs hardware support to carry out the translation from a virtual address to a physical address. As a process proceeds in its execution, it references only a subset of its pages during any given time interval. These are the pages that the process needs to proceed in its current phase of execution; these pages are the current locality of the process.

Paging Policies

Several important issues have to be resolved to completely implement virtual memory:

- When to swap in a page-the fetch policy.
- The selection of the page in memory to replace when there are no empty frames the replacement policy.
- The selection of the frame in which to place the page that was fetched-the placement policy
- The number of frames to allocate to a process.

Fetch Policy

For resolving the fetch policy, there are two approaches to consider:

- Demand paging, in which a page is not loaded until it is referenced by a process.
- Preparing, in which a page is loaded before it is referenced by a process.

In demand paging, a process generates a page fault when it references a page that is not in memory.

This can occur in either of the following conditions:

- The process is fetching an instruction.
- The process is fetching an operand of an instruction.

With preparing, other pages are loaded into memory. It is not very useful if these additional pages loaded are not referenced soon by the process. Preparing can be used initially when the process starts. These pages are loaded into memory from the secondary storage device (virtual memory).

Replacement Policy

The locality of a process consists of the subset of the pages that are used together at a particular time. As a process proceeds in its execution, its locality changes and one or more pages not in memory will be needed by the process in order to continue execution. The following steps are carried out by the operating system when a page fault occurs:

1. The process that generated the page fault is suspended.
2. The operating system locates the referenced page in the secondary storage device, using the information in the page tables.
3. If there are no free frames, a page is selected to be replaced and this page is transferred back to the secondary storage device if necessary.
4. The referenced page is loaded into the selected frame, and the page and frame tables are updated.
5. The interrupted program is scheduled to resume execution.

If there are no free frames in memory, a page in memory is replaced to make available an empty frame for the new page. The replaced page may have to be transferred back into the secondary storage device, if it has been modified. There are several replacement policies that are discussed in the sections that follow. The placement policy determines in which frame to store the fetched page. This is not a real issue in paged virtual memory management. In a more detailed view, when a page fault occurs, the MMU (memory management unit) hardware will cause a page fault interrupt to the CPU. The operating system responds to the interrupt by taking the following steps to handle the page fault:

1. Verify that the page reference address is a valid (or potentially valid) address for the process that caused the page fault. If it is not, then the process will be terminated with a protection violation.
2. Locate a frame into which the desired page can be loaded. Clearly, a frame that is not currently being used is desired. If all frames are currently in use, choose a frame using one of the paging policies discussed next. This step is a resource allocation procedure and therefore must be performed under lock (since multiple processes may simultaneously need to have additional frames allocated).
3. Swap out the page that is currently occupying the frame that was selected in Step 2 above. This procedure can be optimized by noting that pages that have not changed do not need to be swapped out (a copy of the page already exists on the disk). Clearly, pages that do not have Write enabled cannot have changed, so they never need to be swapped out. Even

a page that does have Write enabled may not have actually changed since it was loaded from a disk. In many systems, the MMU will mark a page entry whenever a write occurs to that page. The OS can examine this "modified" bit (also known as a "dirty" bit) to determine if it needs to swap out the page.

4. Load the desired page. The OS can locate a copy of the page on disk: either in the original program file (if it has never been loaded before) or in the swap file.

Frame Allocation

There are *two general* groups of paging algorithms: those that use *static allocation* and those that use *dynamic allocation*. In **static allocation**, the system allocates a fixed number of frames to a process. In **dynamic allocation**, the system dynamically changes the number of frames it allocates to a process during its execution. The number of frames to allocate to a process is important for the performance of the system. In general, the more frames that are allocated to a process, the better the performance will be because there will be a reduced number of total page faults during the execution of a process. Too many frames allocated to a process would have the overall effect of reducing the degree of multiprogramming, which in turn reduces performance.

If the number of frames is too small, there will be too many page faults during execution of a process. An excessive number of page faults could lead to thrashing, a situation that can be worse than deadlock because one or more processes will be making no progress in their executions and it can completely bring the system down. Two general schemes are used by the operating system to allocate frames to the various processes. The simplest one is called *equal allocation*, and it divides the available frames equally among the active processes. The second scheme is called *proportional allocation*, and in it, the number of frames is proportional to its size (in pages) and also depends on the priority of the process.

Page Faults and Performance Issues

A page fault requires the operating system to handle the *page fault*, as discussed previously. The total time it takes to service a page fault includes several time components. The following are most relevant:

- The time interval to service the page fault interrupt.
- The time interval to store back (swap out) the replaced page to the secondary storage device.
- The time interval to load (swap in) the referenced page from the secondary storage device (disk unit).
- Delay in queuing for the secondary storage device.
- Delay in scheduling the process with the referenced page.

The most significant time component is the disk I/O time to swap out the replaced page and to swap in the referenced page. These I/O operations take several orders of magnitude more time than the access time to physical memory.

The Working Set Algorithm

The working set algorithm estimates the number of frames needed by a process in its next execution phase based on its current memory requirements. Given the last WSW page references, the working set is the set of pages referenced in that window. The quantity WSW is called the working set window. A good value of WSW results when the working set of a process equals its current locality. Based on the size of the working set, more frames can be allocated or deallocated to or from the process. The total number of pages that should be allocated to a process is the size of its working set.

Thrashing

Thrashing is a *condition or state* in the system into which all the time a process spends is dedicated for swapping pages; thus, no computation is carried out by the process. In principle, each active process should be allocated a sufficient number of frames for its current locality. If the number of frames allocated is too low, the execution of the process will generate an excessive number of page faults, and eventually the process will make no progress in its execution because it will spend practically all of its time paging-that is, swapping pages in and out. This condition is called *thrashing*. The operating system should manage the allocation of frames in such a manner that, when it deallocates frames to one or more processes, it can increase the degree of multiprogramming, if necessary. When one or more processes need more frames, the system must suspend some other process to increase the number of frames available and to allocate these to the processes that need them.

One approach used to prevent thrashing is to determine or estimate, for every active process, the sizes of the localities of the process in order to decide on the appropriate number of frames to allocate to that process. The performance of the system is affected by thrashing. The CPU utilization and throughput decrease at a very fast rate when thrashing occurs. The operating system increases the degree of multiprogramming in an attempt to improve the CPU utilization, and this might cause other processes to thrash, causing lower throughput and lower CPU utilization.

Thrashing of one process may cause other processes to thrash, if the operating system uses a global allocation strategy. This allocation strategy provides a global pool of frames for processes to select

a replacement frame when needed. With local allocation, a process can select a replacement frame only from the set of frames already allocated to it.

Caching

The most common technique used to reduce disk accesses is the **block cache** or **buffer cache**. (Cache is pronounced ‘‘cash’’ and is derived from the French *cacher*, meaning to hide.) In this context, a cache is a collection of blocks that logically belong on the disk but are being kept in memory for performance reasons. Various algorithms can be used to manage the cache, but a common one is to check all read requests to see if the needed block is in the cache. If it is, the read request can be satisfied without a disk access. If the block is not in the cache, it is first read into the cache, and then copied to wherever it is needed. Subsequent requests for the same block can be satisfied from the cache. Since there are many (often thousands of) blocks in the cache, some way is needed to determine quickly if a given block is present. The usual way is to hash the device and disk address and look up the result in a hash table. All the blocks with the same hash value are chained together on a linked list so the collision chain can be followed.

Exercise

I. Say true if statement is correct else false if statement is incorrect

1. A physical address is a reference to some location of a process
2. Contiguous blocks of available (unallocated) memory are called holes.
3. In contiguous memory allocation can cause paging problems.
4. In paging the process address space is divided into small fixed size blocks of logical memory.
5. Fragmentations are variable-length modules of a program that correspond to logical units of the program

II.Choose alternative answer among the given choose

1. which one of the following is correct paging algorithm?
   A) Equal allocation and proportional allocation C) Static allocation and equal allocation
   B) Static allocation and dynamic allocation D) Dynamic allocation and static allocation
2. The major tasks of the memory manager are the allocation and deallocation of main memory.
   B) True D) Both
   C) False E) None of the above
3. The general address translation procedure is called _____
   B) Address translation D) Address binding
   C) Address storing E) Address matching
5. In a fixed partition the unused portion of memory inside a partition is called_____
   A) External paging C) External fragmentation
   B) Internal paging D) Internal fragmentation
6. The disk area that stores all the processes in absolute form is called the_____.

A) Virtual memory
B) Cash memory
C) Main memory
D) Trash memory

III. Write clear answer for the following question

1. Write two goals of modern operating system to protect memory and describe each goal! (2 mark)
2. Write the paging policy briefly! (2 mark)
3. Write fetching policy briefly! (1 mark

Chapter 6
Device management

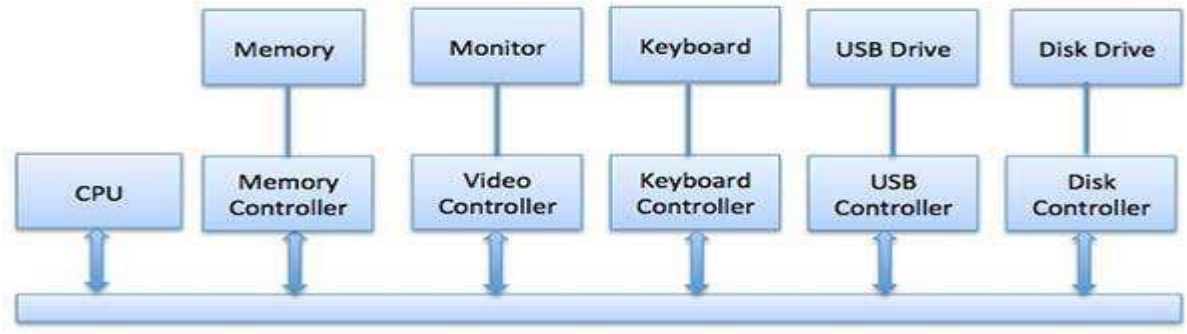
One of the important jobs of an Operating System is to manage various I/O devices including mouse, keyboards, touch pad, disk drives, display adapters, USB devices, Bitmapped screen, LED, Analog-to-digital converter, On/off switch, network connections, audio I/O, printers etc. An I/O system is required to take an application I/O request and send it to the physical device, then take whatever response comes back from the device and send it to the application. I/O devices can be divided into two categories:

- **Block devices**: A block device is one with which the driver communicates by sending entire blocks of data. For example, Hard disks, USB cameras, Disk-On-Key etc.
- **Character devices:** A character device is one with which the driver communicates by sending and receiving single characters (bytes, octets). For example, serial ports, parallel ports, sounds cards etc.

Device Controllers

Device drivers are software modules that can be plugged into an OS to handle a particular device. Operating System takes help from device drivers to handle all I/O devices. The *Device Controller* works like an interface between a device and a device driver. I/O units (Keyboard, mouse, printer, etc.) typically consist of a mechanical component and an electronic component where electronic component is called the device controller. There is always a device controller and a device driver for each device to communicate with the Operating Systems. A device controller may be able to handle multiple devices. As an interface its main task is to convert serial bit stream to block of bytes, perform error correction as necessary.

Any device connected to the computer is connected by a *plug* and *socket*, and the socket is connected to a *device controller*. Following is a model for connecting the CPU, memory, controllers, and I/O devices where CPU and device controllers all use a common bus for communication.



Synchronous vs Asynchronous I/O

- **Synchronous I/O** In this scheme CPU execution waits while I/O proceeds
- **Asynchronous I/O** proceeds concurrently with CPU execution

Communication to I/O Devices

The CPU must have a way to pass information to and from an I/O device. There are three approaches available to communicate with the CPU and Device.

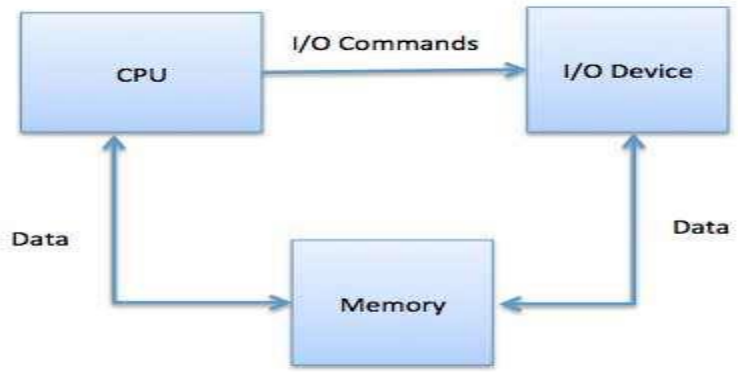
- Special Instruction I/O
- Memory-mapped I/O
- Direct memory access (DMA)

Special Instruction I/O

This uses CPU instructions that are specifically made for controlling I/O devices. These instructions typically allow data to be sent to an I/O device or read from an I/O device.

Memory-mapped I/O

When using memory-mapped I/O, the same address space is shared by memory and I/O devices. The device is connected directly to certain main memory locations so that I/O device can transfer block of data to/from memory without going through CPU.

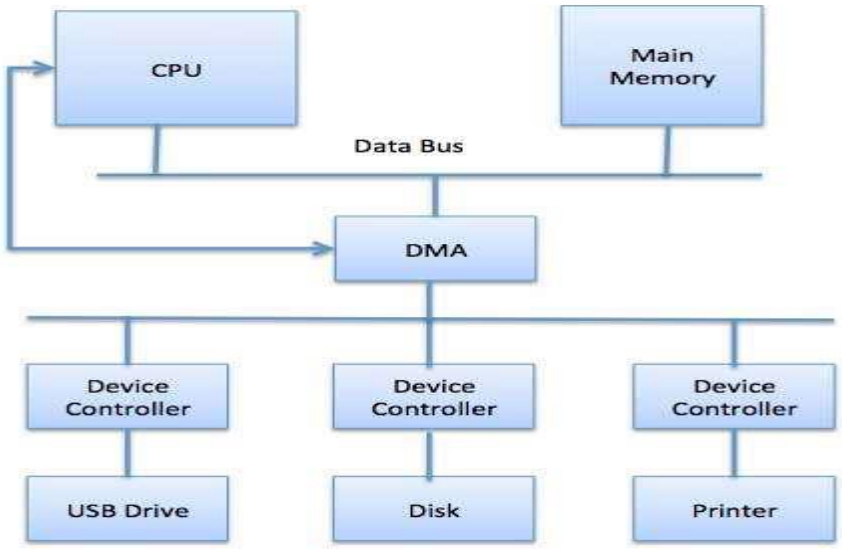


While using memory mapped IO, OS allocates buffer in memory and informs I/O device to use that buffer to send data to the CPU. I/O device operates asynchronously with CPU, interrupts CPU when finished. The advantage to this method is that every instruction which can access memory can be used to manipulate an I/O device. Memory mapped IO is used for most high-speed I/O devices like disks, communication interfaces.

Direct Memory Access (DMA)

Slow devices like keyboards will generate an interrupt to the main CPU after each byte is transferred. If a fast device such as a disk generated an interrupt for each byte, the operating system would spend most of its time handling these interrupts. So, a typical computer uses direct memory access (DMA) hardware to reduce this overhead.

Direct Memory Access (DMA) means CPU grants I/O module authority to read from or write to memory without involvement. DMA module itself controls exchange of data between main memory and the I/O device. CPU is only involved at the beginning and end of the transfer and interrupted only after entire block has been transferred. Direct Memory Access needs a special hardware called *DMA controller* (DMAC) that manages the data transfers and arbitrates access to the system bus. The controllers are programmed with source and destination pointers (where to read/write the data), counters to track the number of transferred bytes, and settings, which includes I/O and memory types, interrupts and states for the CPU cycles.



The operating system uses the DMA hardware as follows:

Step	Description
1	Device driver is instructed to transfer disk data to a buffer address X.
2	Device driver then instruct disk controller to transfer data to buffer.
3	Disk controller starts DMA transfer.
4	Disk controller sends each byte to DMA controller.
5	DMA controller transfers bytes to buffer, increases the memory address, decreases the counter C until C becomes zero.
6	When C becomes zero, DMA interrupts CPU to signal transfer completion.

Polling vs Interrupts I/O

A computer must have a way of detecting the arrival of any type of input. There are two ways that this can happen, known as **polling** and **interrupts**. Both of these techniques allow the processor

to deal with events that can happen at any time and that are not related to the process it is currently running.

Polling I/O

Polling is the simplest way for an I/O device to communicate with the processor to the processor. The process of periodically checking status of the device to see if it is time for the next I/O operation, is called *polling*. The I/O device simply puts the information in a Status register, and the processor must come and get the information. Most of the time, devices will not require attention and when one does it will have to wait until it is next interrogated by the polling program. This is an inefficient method and much of the processors time is wasted on unnecessary polls. Compare this method to a teacher continually asking every student in a class, one after another, if they need help. Obviously the more efficient method would be for a student to inform the teacher whenever they require assistance.

Interrupts I/O

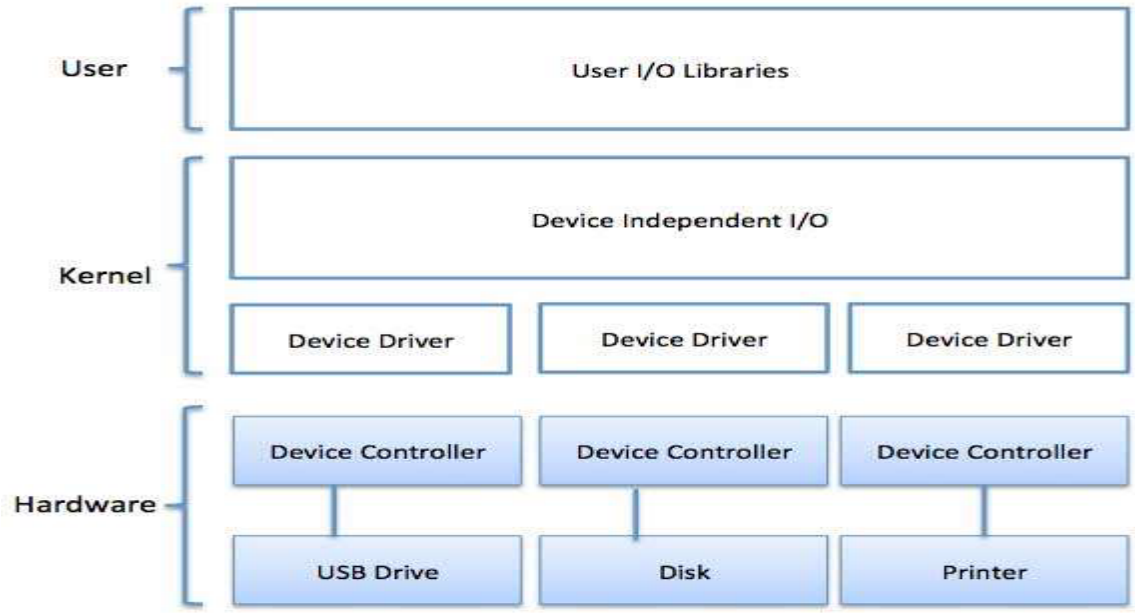
An alternative scheme for dealing with I/O is the interrupt-driven method. An interrupt is a signal to the microprocessor from a device that requires attention. A device controller puts an interrupt signal on the bus when it needs CPU's attention when CPU receives an interrupt, it saves its current state and invokes the appropriate interrupt handler using the interrupt vector (addresses of OS routines to handle various events). When the interrupting device has been dealt with, the CPU continues with its original task as if it had never been interrupted.

Software input and output management

I/O software is often organized in the following layers:

- **User Level Libraries:** This provides simple interface to the user program to perform input and output. For example, **stdio** is a library provided by C and C++ programming languages.
- **Kernel Level Modules:** This provides device driver to interact with the device controller and device independent I/O modules used by the device drivers.
- **Hardware**: This layer includes actual hardware and hardware controller which interact with the device drivers and makes hardware alive.

A key concept in the design of I/O software is that it should be device independent where it should be possible to write programs that can access any I/O device without having to specify the device in advance. For example, a program that reads a file as input should be able to read a file on a floppy disk, on a hard disk, or on a CD-ROM, without having to modify the program for each different device.



Device drivers are software modules that can be plugged into an OS to handle a particular device. Operating System takes help from device drivers to handle all I/O devices. Device drivers encapsulate device-dependent code and implement a standard interface in such a way that code contains device-specific register reads/writes. Device driver, is generally written by the device's manufacturer and delivered along with the device on a CD-ROM. A device driver performs the following jobs:

- To accept request from the device independent software above to it.
- Interact with the device controller to take and give I/O and perform required error handling
- Making sure that the request is executed successfully

How a device driver handles a request is as follows: Suppose a request comes to read a block N. If the driver is idle at the time a request arrives, it starts carrying out the request immediately. Otherwise, if the driver is already busy with some other request, it places the new request in the queue of pending requests.

Interrupt handlers

An interrupt handler, also known as an *interrupt service routine* or ISR, is a piece of software or more specifically a callback function in an operating system or more specifically in a device driver, whose execution is triggered by the reception of an interrupt. When the interrupt happens, the interrupt procedure does whatever it has to in order to handle the interrupt, updates data structures and wakes up process that was waiting for an interrupt to happen. The interrupt mechanism accepts an address a number that selects a specific interrupt handling routine/function from a small set. In

most architectures, this address is an offset stored in a table called the interrupt vector table. This vector contains the memory addresses of specialized interrupt handlers.

Device-Independent I/O Software

The basic function of the device-independent software is to perform the I/O functions that are common to all devices and to provide a uniform interface to the user-level software. Though it is difficult to write completely device independent software but we can write some modules which are common among all the devices. Following is a list of functions of device-independent I/O Software:

- Uniform interfacing for device drivers
- Device naming Mnemonic names mapped to Major and Minor device numbers
- Device protection
- Providing a device-independent block size
- Buffering because data coming off a device cannot be stored in final destination.
- Storage allocation on block devices
- Allocation and releasing dedicated devices
- Error Reporting

User-Space I/O Software

These are the libraries which provide richer and simplified interface to access the functionality of the kernel or ultimately interactive with the device drivers. Most of the user-level I/O software consists of library procedures with some exception like spooling system which is a way of dealing with dedicated I/O devices in a multiprogramming system. I/O Libraries (e.g., stdio) are in user-space to provide an interface to the OS resident device-independent I/O SW. For example putchar(), getchar(), printf() and scanf() are example of user level I/O library stdio available in C programming.

Kernel I/O Subsystem

Kernel I/O Subsystem is responsible to provide many services related to I/O. Following are some of the services provided:

- ***Scheduling*** - Kernel schedules a set of I/O requests to determine a good order in which to execute them. When an application issues a blocking I/O system call, the request is placed on the queue for that device. The Kernel I/O scheduler rearranges the order of the queue to improve the overall system efficiency and the average response time experienced by the applications.
- ***Buffering*** - Kernel I/O Subsystem maintains a memory area known as **buffer** that stores data while they are transferred between two devices or between a device with an application operation. Buffering is done to cope with a speed mismatch between the producer and consumer of a data stream or to adapt between devices that have different data transfer sizes.

- ***Caching*** - Kernel maintains cache memory which is region of fast memory that holds copies of data. Access to the cached copy is more efficient than access to the original.
- ***Spooling and Device Reservation*** - A spool is a buffer that holds output for a device, such as a printer, that cannot accept interleaved data streams. The spooling system copies the queued spool files to the printer one at a time. In some operating systems, spooling is managed by a system daemon process. In other operating systems, it is handled by an in-kernel thread.
- ***Error Handling*** - An operating system that uses protected memory can guard against many kinds of hardware and application errors.

System Recovery

The recovery process is designed to recover a server to a previous operating state, in the event of a hardware or operating system failure. The recovery process will begin by starting your computer using a Bootable Recover Assist Media or a Bootable Backup Media. The process will then load a recovery environment, which you can use to select the location of the backup to be used, and to start the recovery.

Exercise

I. Among the given chose select alternative option based on question.

1. _____is the simplest way for an I/O device to communicate with the processor to the processor.
   A) Polling C) Paging
   B) Fragmentation D) Interrupts
2. Among the following which one is not approaching available to communicate with the CPU and Device
   A) Special interaction C) Intercommunication
   B) Memory mapped D) Direct memory access
3. Which media used for system recovery?
   A) Flash C) Bootable backup media
   B) CD/DVD D) External hard disk
4. Direct Memory Access needs a special hardware called_____
   A) DMA controller C) Polling
   B) DMA D) Interrupts
5. Which one is not some of the services provided by kernel I/O subsystem?
   A) Handle error C) Buffering
   B) Scheduling D) Pulling

II. Write answer for the following question

1. List and discus about interrupt handlers?
2. What is polling?
3. What the difference between Direct memory access, memory mapped and special instruction?
4. Show clear idea about direct memory access and illustrate clearly?
5. List and discus about user space I/O software and kernel I/O subsystem?

Chapter 7
Security and protection

Overview of system security

Computer security is based on the answers to two questions:

- Who do you trust?
- How much do you trust them?

The specific answers to these questions are the basis for a security policy. A typical policy will define roles and access privileges for company employees, contractors, vendors, customers, and others. We often think of security as protecting a system from outsiders, but it must also protect it from insiders who are attempting (either accidentally or deliberately) to do things not allowed by the security policy.

Like the objects of an executing object-oriented program, resources are accessed by processes through operations defined by a capabilities list controlled by the appropriate operating system manager. Protection refers to the set of policies and mechanisms that define how resources and the operations defined on them are protected from processes. In order to avoid haphazard and indiscriminate use of the operations of a resource object, the memory manager maintains an access matrix that specifies which operations on each object are accessible to the various domains that can be assigned to processes.

External security refers to the way user programs access networks, servers, other hosts, and their resources and services. In contrast to protection, where a process hopes to gain straightforward access to an operation defined on a specific resource, security is about understanding how processes, by accident or by design, attempt to carry out some action to an operating system resource that could damage it or alter it in an unacceptable way.

Another important issue in security is access-the ability to access the resources of another computer across the *Internet* by using a *protocol* (SMTP, HTTP, FTP, TCP, UDP, etc.). *Any Internet user has access to public domain resources* (usually files and lightweight applications, most of which are on the Web) while more extended access is available across the Internet, on local area networks, and directly for those who have user accounts. Requests for user account access to networks and servers are controlled by *login names and passwords* and (for remote access) by IP addresses and port numbers. Users and processes are controlled in this way because untrusted users (which the system attempts to filter out by a firewall) are likely to cause deliberate damage by inserting viruses, worms, or spyware into the system. Such users may deliberately or

accidentally exploit an application or operating system vulnerability (such as the potential of a text reader to overflow its buffer if too much data is sent or the capability of circumventing the login procedure with a trap door) that has not been patched.

Problem of security

Security must defend a system and its computers against mistakes and malicious attacks by users and by automated or intelligent programs. A system vulnerability, such as the susceptibility of a buffer or a stack to data overflow into a protected memory space, may be exposed by a mistake (usually a programming error) or exploited on purpose. A malicious attack usually involves the deliberate installation of software on a computer, which can compromise either privacy or security. For instance, such an attack could install spyware, which could extract enough information from files to lead to identity theft, credit card fraud, or other financial loss. Such an attack could install a virus that alters the system (attributes or functions) in some way (usually the operating system or a popular application). Such an attack could install a worm that uses the system as a springboard to propagate itself to other machines and to launch attacks on other machines that it may not be able to access directly. The worm may reduce the performance of the system through a denial of service attack, a disruption that floods it with useless Internet packets.

Security and Protection Components

The security of a system depends on the following components:

- ✓ Physical security
- ✓ User authentication
- ✓ Protection
- ✓ Secure communications
- ✓ People

As we shall discuss throughout the rest of this chapter, it is often desirable to use several of these security components so that if one component is breached, the others will still protect the system.

Physical Security

A computer must be protected against being *lost or stolen*; otherwise, it is highly vulnerable. Additionally, when a computer or disk drive is discarded, the disk drive should be erased to ensure that others do not have access to important information. the contents of deleted files remain on the disk. Consequently, this erasure procedure should physically write to the disk, not just delete files. If a portable computer contains highly sensitive information (e.g., credit card numbers), then it is advisable that those files and/or the entire disk be encrypted. This will reduce the possibility of this information being maliciously accessed if the computer is lost or stolen. It is common for

modern computers to have a wireless communications capability such as Wi-Fi or Bluetooth. These wireless capabilities add useful features to the computer, but they also open up holes in the physical security of the system. Thus, it is important that other security mechanisms such as user authentication and encryption be used with all wireless communications.

User Authentication

User authentication is the action of the system verifying (to the best of its knowledge) that users are who they say they are and that those users are permitted access to the system. User authentication can be in the form of Something *You Know*. This usually consists of a login name, a password, and a (network) domain name. The security of such basic login schemes can be enhanced by requiring periodic password updates, requiring a minimum password length and/ or requiring that a minimum number of different types of characters (e.g., alpha, numeric) be used in a password so as to render it harder to guess. Usually, after the nth failed login attempt, further attempts by that user are disabled.

Newer systems will combine Something You Have with Something You Know. An example of this is using a Smart-Card, a card that the user plugs into a port on the computer-for login purposes. The user will then be asked for a pin code (Something You Know), which will then be encrypted by the Smart-Card. The encrypted value is sent to the network host, which will then verify it. There are several forms of biometrics that are now starting to be used for user authentication:

- ***Fingerprint***: Laptop computers are available that verify the user's fingerprint for logon.
- ***Voice print***: User's voice is checked and verified.
- ***Retina scan:*** While very accurate, this is cumbersome to do and is currently restricted to very limited applications.
- ***Facial imaging***: Currently, this technique is not accurate enough to uniquely identify an individual.

Protection

A resource is an object controlled by the operating system. It may be a software object such as a file or an application, or it may be a hardware object such as main memory, the CPU, or a peripheral device. An operation is a function defined on that resource. In the case of a file, for instance, operations that a process may execute on a file are create, open, copy, read, write, execute, close, and so forth. A domain specifies the resources that a particular process may access. Protection on the system consists of giving resource access to domains that should have it and denying it to domains that should not have it.

Memory protection

Memory protection is a special case of protection in which the operating system will set up the appropriate access tables, but the enforcement is handled by the hardware. Memory protection is

used to help ensure that a program bug does not have a dangerous effect. Typically, permission bits for read, write, and execute will be associated with each entry in the page table. The operating system will set these bits based on information that the compiler places in the program file for the program being executed. Table below shows the meaning and usage of the various combinations of permission bits.

Read	Write	Execute	Meaning	Usage
0	0	1	Execute-only	Instructions
0	1	0	Write-only	Not-Used
0	1	1	Write-Execute	Not-Used
1	0	0	Read-only	Constants
1	0	1	Read-Execute	Instructions-Constants
1	1	0	Read-Write	Data
1	1	1	Read-Write-Execute	No protection

Although the concept of having separate read, write, and execute permissions was developed about 40 years ago, not all CPUs support this concept. Some have no memory protection support and others only support read and write permissions. In fact, the CPUs used in PCs did not support a separate execute permission until 2006. The lack of execute permission provides an opening for hackers to penetrate a system. Without a separate execute permission, any data area of a program is effectively executable. Thus, if a hacker can trick a program into executing something in its data, the program can be led to execute instructions provided by the hacker (in what the program believes is data). This is the basis for the Buffer Overflow and Stack Overflow security attacks. Note that since the OS sets the permission bits based on the information provided by the compiler, it is necessary for the application developer to take advantage of this security feature in order for it to be fully enabled.

Secure Communications

A system must ensure that its communications with other systems are secure. In today's world, any communications over the Internet are vulnerable to eavesdropping. Similarly, any communications over a Wi-Fi wireless link can be intercepted. Thus, encryption should be used for any data transmitted where the physical security of the communication lines cannot be guaranteed.

- Ciphertext = Encrypt (plaintext, key1)
- Plaintext = Decrypt (ciphertext, key2)

A symmetric encryption system is one in which key1 = key2. The sender and receiver use the same key to encrypt and decrypt the data. This type of system requires that the key be kept secret. Consequently, it also requires a separate secure communications channel for the sender and receiver to communicate the key. Examples of this type of system are Data Encryption Standard (DES) and the Advanced Encryption Standard (AES).

People

The people who use a system must be properly trained in basic security concepts and the protection of information. No matter how secure the system is technically, if the users do not practice good security procedures, the system will be vulnerable.

System vulnerability

Software vulnerabilities are inherent in the operating system and most application programs. In some cases, they were deliberately designed by the programmer in order to facilitate his or her use of the program. In other cases, they are fundamental software defects that were not anticipated at design time.

Social Engineering

Social engineering is an attempt made by an attacker who pretends to be a trusted individual or institution in order to convince someone to divulge confidential information such as passwords or account numbers. There are two common examples:

- ***Phishing*** is the act of sending out an email that appears to be from a trusted institution, such as the receiver's bank. The user is asked to click on a web link to enter the desired confidential information. The link, of course, takes the user to an attacker's website rather than the trusted institution.
- ***Pretexting*** involves an attacker who makes a phone call to someone in an organization and poses as someone else and asks for confidential information.

Trojan Horse Programs

A *Trojan horse* is a program that masquerades as something beneficial but actually causes damage or invades privacy. It can also be a beneficial program that is accidentally or deliberately misused by a programmer. A prime example of a Trojan horse program is a terminal login emulator that hijacks/overrides an ATM or other password access-controlled system. In the first attempt to access a secure system, the user unwittingly communicates with the Trojan horse terminal emulator that intervenes between the user and the legitimate login software. The Trojan horse program logs the keystrokes that the user makes and then returns a message to the user that the password has been entered incorrectly and to try to login again. Having acquired the user's authentication data, the Trojan horse then permits the user to access the true terminal login software the next time. The user is none the wiser. The transaction is successfully completed and the user imagines that he or she must have keyed in the wrong PIN number on the first attempt. The Trojan horse then communicates the vital login information across the Internet to its control program.

Spyware

Spyware includes relatively innocuous software cookies that monitor the user's browser habits and report them back to an application when the computer is online. Browsers can be set so that a pop-up dialog box requests the browser's permission to deposit a cookie on the disk drive in exchange for access to the services provided by the website. A common form of spyware will *hijack* the user's web browser so that web searches are diverted to an unexpected website. Other spyware can search the hard drive for personal information, including social security numbers, credit card numbers, driver's license numbers, and other data that could be used to profile individuals, steal identities, or perpetrate credit card or other financial fraud (such as providing account numbers for bogus e-commerce and e-banking transactions).

Trap Doors

A trap door is a program fragment that issues a sequence of commands (sometimes just a special password) that the program writers have inserted to circumvent the security system that they designed to control access to their programs. Since these trap doors are not registered as normal logins, programmers who design software with trap doors have used them to carry out illegal activities undetected. Such activities include changing data in a database or tampering with accounting procedures and diverting funds from a group of accounts to a private account.

Invasive and Malicious Software

Invasive and malicious software includes viruses and worms and any other programs that have the property of being self-replicating and/or self-propagating (they can spread themselves from one computer to another). In addition to the ability to replicate themselves (often in hard-to-find places such as the boot sector of another file), viruses typically have harmful side effects. Some viruses can do something relatively innocuous such as changing wallpaper, posting flamboyant messages on the desktop, or otherwise changing the appearance of the graphical interface to the operating system. Other viruses can destabilize the operating system by deleting files, disabling applications, and causing other problems.

Viruses can be inadvertently transferred when files from a disk inserted into an external drive are copied to the internal hard disk. Viruses can also be transferred to the (internal) hard disk when email attachments are opened or downloaded. Viruses are fragments of operating system script code found in parts of files, sometimes hidden in the boot sector. Opening such a file can cause the virus script to execute. The first thing the virus code usually does is make a copy of itself to one or more files in the local file system. The adverse side effects caused when the virus changes the attributes of an object (e.g., by changing the registry in Windows) often appear after rebooting the machine. **Worms** are complete programs or sets of collaborating programs that have the ability to transfer themselves from one machine to another and to communicate with each other or to receive commands from a master program on a remote machine. They can transfer themselves to any computer logged on to the Internet that they are able to access.

In addition to the ability to migrate from one machine to another, worms often have additional capabilities. An important one is that most worms have to report back to a master process the exact machine and the conduit they used (IP address, port number, and domain) to install themselves successfully. Using this information, the master process can send command signals across the Internet to launch a distributed denial of service attack on one or more servers.

Defending the System and the User

The system can be defended against viruses in two ways. The best method of protection against viruses is to practice safe computing. This entails common sense practices such as setting the web browser to detect/reject cookies and not opening or downloading email attachments from untrusted or unknown correspondents. But no matter how careful you might be, using the Internet and/ or inserting disks into external drives and copying files from them to the hard disk drive or executing applications on them, viruses will end up infesting your computer. The best way to eliminate these viruses is by installing antivirus software that will scan all files for viruses and remove or

quarantine those that it discovers. Antivirus software should be updated often since it is only good against viruses that the programmers deemed were prevalent at the time it was written.

It is more difficult to defend the system against worms. A worm often gains access to a system by exploiting known vulnerabilities of system applications, such as the buffer overflow vulnerability of some network applications. Any system that a human user can access can be accessed automatically by a worm. Worms are also harder to delete. Since most worms on a system will be executing, the worm file cannot be simply deleted, as is the case with a virus. However, worms in the process of execution can be deleted with the aid of the process manager and the file manager.

Intrusion Detection Management

There are *two* basic *patterns for intrusion detection*: signature-based detection and anomaly detection. Like antivirus software, signature detection looks for known patterns of behavior established by previous attacks. For instance, multiple logons to an account indicate that an intruder may be trying to guess the password to an account. An application that scans port numbers by sending premature FIN packets indicates that an attacker is looking for open ports, while a connection followed by the transmission of an inordinate amount of data can indicate that the attacker is looking to exploit a buffer overflow to obtain a variety of sensitive system information that may include passwords and IP addresses that could provide access to system accounts. Anomaly detection is a process that looks for unusual patterns in computer behavior. For instance, a worm program exploiting a port or network application daemon might be detected by the transfer of unusually large amounts of data that would not normally be transferred when the daemon is in that particular state. In addition to scanning for attack patterns or anomalous behavior on the current state of the machine, intruders who may no longer be active or are temporarily quiescent can be detected by processing the log and audit files. With the aid of these files, past attack patterns can be detected and anomalous events can be analyzed. Information from log and audit files can be cross referenced with the information obtained from the scan of the most recent state of the machine to provide further evidence of the presence of an intruder.

Security and Privacy

As we have said, security involves preventing the unauthorized access to information. This is different from privacy, which involves preventing the unauthorized disclosure of information. What is the difference? Employees who may be authorized to access certain information violate a company's privacy policy if they disclose that information to someone who is not authorized to see it. Current commercial operating systems do nothing to enforce privacy restrictions. Once authorized users obtain information, the system allows them to do whatever they like with that

information. That is because the operating system considers information to be just a set of bits, with no understanding of what those bits represent. Thus, in today's world, enforcing a privacy policy is something that is a training and administrative issue with no technical support.

Secure Systems Versus Systems Security

Software plugins are developed by industry or open source programmers to cover up additional vulnerabilities as they are discovered or exposed. When vulnerabilities for an application or an operating system are discovered, the information about the availability of patches and the patches themselves are posted on websites or emailed to users and systems managers.

As new viruses are discovered, antivirus software is updated to recognize their signatures so that the new releases can be removed or at least quarantined. As specific websites with unacceptable content and addresses that send junk email proliferate, application-level filters for browsers and email applications can be updated and adjusted as needed. Security requires overhead, but overhead becomes faster and cheaper as system hardware and software evolves.

What if worrying about systems security was not an afterthought and operating systems designers created an operating system with the forethought to build security in from the foundation up? How can this be done without compromising connectivity and ease of use? An attempt to design a secure system that is easy to use and can handle insecurity from the foundation up has already been attempted with the Java programming language and the Java Virtual Machine (JVM). The basic strategy is to protect the management of main memory from the programmer. Main memory is viewed as the essential source of program and operating system vulnerability. The Java programming language does not allow its objects to directly access main memory. Moreover, various processes are classified and labeled as trustworthy or not and then allowed appropriate access by the JVM by indelibly marking the stack frame where the object is loaded and which the Java programs executing the object are unable to access. The three main vulnerabilities related to viruses and worms of an operating system (for a single host machine, a server, or a network of machines) are memory for process storage, disk drives for file storage, and the TCP /IP network protocol and the login procedure for Internet communication. Any attempt to design a future operating system with security built into the architecture must try to extend what the designers of the Java programming language did to the broader scope of the entire operating system, the network, and the resources that they manage.

Encryptions

Encryption is the process of using an algorithm to transform information to make it unreadable for unauthorized users. This cryptographic method protects sensitive data such as credit card numbers

by encoding and transforming information into unreadable cipher text. This encoded data may only be decrypted or made readable with a key. Symmetric-key and asymmetric-key are the two primary types of encryption.

What does Decryption mean?

Decryption is the process of transforming data that has been rendered unreadable through encryption back to its unencrypted form. In decryption, the system extracts and converts the garbled data and transforms it to texts and images that are easily understandable not only by the reader but also by the system. Decryption may be accomplished manually or automatically. It may also be performed with a set of keys or passwords

System Protections

Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by a malfunctioning subsystem. Also, an unprotected resource cannot defend against use (or misuse) by an unauthorized or incompetent user. A protection-oriented system provides means to distinguish between authorized and unauthorized usage. The role of protection in a computer system is to provide a mechanism for the enforcement of the policies governing resource use. These policies can be established in a variety of ways. Some are fixed in the design of the system, while others are formulated by the management of a system. Still others are defined by the individual users to protect their own files and programs. A protection system must have the flexibility to enforce a variety of policies. Policies for resource use may vary by application, and they may change over time. For these reasons, protection is no longer the concern solely of the designer of an operating system. The application programmer needs to use protection mechanisms as well, to guard resources created and supported by an application subsystem against misuse. Note that *mechanisms* are distinct from *policies*. Mechanisms determine *how* something will be done; policies decide *what* will be done. The separation of policy and mechanism is important for flexibility. Policies are likely to change from place to place or time to time. In the worst case, every change in policy would require a change in the underlying mechanism. Using general mechanisms enables us to avoid such a situation.

Principle of protections

Frequently, a guiding principle can be used throughout a project, such as the design of an operating system. Following this principle simplifies design decisions and keeps the system consistent and easy to understand. A key, time-tested guiding principle for protection is the *principle of less privilege* It dictates that programs, users, and even systems be given just enough privileges to

perform their tasks. Consider the analogy of a security guard with a passkey. If this key allows the guard into just the public areas that she guards, then misuse of the key will result in minimal damage. If, however, the passkey allows access to all areas, then damage from its being lost, stolen, misused, copied, or otherwise compromised will be much greater. An operating system following the principle of least privilege implements its features, programs, system calls, and data structures so that failure or compromise of a component does the minimum damage and allows the n1inimum damage to be done. The overflow of a buffer in a system daemon might cause the daemon process to fail, for example, but should not allow the execution of code from the daemon process's stack that would enable a remote user to gain maximum privileges and access to the entire system (as happens too often today).

Domain of protections

A computer system is a collection of processes and objects. By *objects*, we mean both hardware objects (such as the CPU, memory segments, printers, disks, and tape drives) and software objects (such as files, programs, and semaphores). Each object has a unique name that differentiates it from all other objects in the system, and each can be accessed only through well-defined and meaningful operations. Objects are essentially abstract data types.

Exercise

I. Select alternative answer from the list of letters based on the question.

1. A system must ensure that its communications with other systems are called_____
   A) Protection C) Vulnerability
   B) Secure D) Decryption
2. _____is the process of using an algorithm to transform information to make it unreadable for unauthorized users.
   A) Decryption C) Protection
   B) Encryption D) Cypher text
3. Which one is not biometrics that are now starting to be used for user authentication?
   A) Finger print C) Facial imaging
   B) Voice print D) Pattern drawing
4. _____ is an attempt made by an attacker who pretends to be a trusted individual or institution.
   A) Social engineering C) Attacker
   B) Hacker D) Vulnerability
5. _____can improve reliability by detecting latent errors at the interfaces between component subsystems.
   A) Protection C) Trap door
   B) Vulnerability D) Intrusion detection
6. A _____ is a program that masquerades as something beneficial but actually causes damage or invades privacy.

A) Trojan horse B) Phishing

C) Spyware D) Vulnerability

7. A key, time-tested guiding principle for protection is the_____

A) principle of less privilege C) principle of privilege

B) principle of high privilege D) less principle

II. Answer the following question briefly

1. List and discus the components security of a system which depends on!
2. Describe briefly about trap door within example!
3. What mean social engineering? How you describe by your own words?
4. List and describe system vulnerability!
5. Compare and contrast decryption and encryption briefly!

Chapter 8
File Systems

Data: -a collection of facts, observations, or other information related to a particular question or problem; as, the historical data show that the budget deficit is only a small factor in determining interest rates. The term in this sense is used especially in reference to experimental observations collected in the course of a controlled scientific investigation.

Metadata: -Metadata is "data that provides information about other data". In other words, it is "*data about data*." Many distinct types of metadata exist, including descriptive metadata, structural metadata, administrative metadata, reference metadata and statistical metadata.

Files: - *A* file is a named collection of related information that is recorded on secondary storage. From a user's perspective, a file is the smallest allotment of logical secondary storage; that is, data cannot be written to secondary storage unless they are within a file. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free form, such as text files, or may be formatted rigidly. In general, a file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user. The concept of a file is thus extremely general.

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

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

To define a file properly, we need to consider the operations that can be performed on files. The operating system can provide system calls to create, write, read, reposition, delete, and truncate files. Let's examine what the operating system must do to perform each of these six 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, we make a system call specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the file's location. The system must keep a *write* pointer to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.

Reading a file: - To read from a file, we use a system call that specifies the name of the file and where (in memory) the next block of the file should be put. Again, the directory is searched for the associated entry, and the system needs to keep a *read* pointer to the location in the file where the next read is to take place. Once the read has taken place, the read pointer is updated. Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process *current file-position pointer*. Both the read and write operations use this same pointer, saving space and reducing system complexity.

Repositioning within a file: - The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value. Repositioning within a file need not involve any actual I/0. This file operation is also known as a file ***seek***.

Deleting a file: - To delete a file, we search the directory for the named file. Having found the associated directory entry, we 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 file be reset to length zero and its file space released.

File management system

A file management system is that set of system software that provides services to users and applications in the use of files. Typically, the only way that a user or application may access files is through the file management system. This relieves the user or programmer of the necessity of developing special-purpose software for each application and provides the system with a consistent, well-defined means of controlling its most important asset. the following objectives for a file management system:

- To meet the data management needs and requirements of the user, which include storage of data and the ability to perform the aforementioned operations
- To guarantee, to the extent possible, that the data in the file are valid

- To optimize performance, both from the system point of view in terms of overall throughput and from the user's point of view in terms of response time
- To provide I/O support for a variety of storage device types
- To minimize or eliminate the potential for lost or destroyed data
- To provide a standardized set of I/O interface routines to user processes
- To provide I/O support for multiple users, in the case of multiple-user systems

File Types

When we design a file system-indeed, an entire operating system-we always consider whether the operating system should recognize and support file types. If an operating system recognizes the type of a file, it can then operate on the file in reasonable ways. For example, a common mistake occurs when a user tries to print the binary-object form of a program. This attempt normally produces garbage; however, the attempt can succeed *if* the operating system has been told that the file is a binary-object program.

A common technique for implementing file types is to include the type as part of the *file name*. The name is *split into two parts*-a *name and an extension*, usually separated by a period character. For example, most operating systems allow users to specify a file name as a sequence of characters followed by a period and terminated by an extension of additional characters. File name examples include resume.doc, Server.java.

The system uses the extension to indicate the type of the file and the type of operations that can be done on that file. Only a file with a *.com, .exe,* or *.bat* extension can be *executed,* for instance. The *.com* and *.exe* files are two forms of binary executable files, whereas a *.bat* file is a batch file containing, in ASCII format, commands to the operating system. MS-DOS recognizes only a few extensions, but application programs also use extensions to indicate file types in which they are interested. For example, assemblers expect source files to have an *.asm* extension, and the Microsoft Word processor expects its files to end with a *.doc(docx)* extension. These extensions are not required, so a user may specify a file without the extension (to save typing), and the application will look for a file with the given name and the extension it expects. Because these extensions are not supported by the operating system, they can be considered as "hints" to the applications that operate on them.

File type	Usual extension	Function
Executable	Exe, com, bin	Ready to run machine language program
Object	Obj, o	compiled, machine language, not linked
Source code	Pl, java, py, cpp, c, asm, a	source code in various languages
Batch	Bat, sh	commands to the command interpreter

Text	Txt, doc	Textual data, documents
Word processor	Docx, rtf, doc, wp,	various word processor formats
Library	Lib, a, so, dll	libraries of routines for programmers
Print or view	Ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
Archive	Arch, zip, rar	related files grouped into one file sometimes compressed, for archiving or storage
Multimedia	Mpeg, mov, wav, mp3, mp4, avi	binary file containing audio or A/V information

Common file types

Files structure

File types also can be used to indicate the internal structure of the file. File has its own structure. Further, certain files must conform to a required structure that is understood by the operating system. For example, the operating system requires that an executable file have a specific structure so that it can determine where in memory to load the file and what the location of the first instruction is. Some operating systems extend this idea into a set of system-supported file structures, with sets of special operations for manipulating files with those structures. For instance, DEC's VMS operating system has a file system that supports three defined file structures.

Some operating systems impose (and support) a minimal number of file structures. This approach has been adopted in UNIX, MS-DOS, and others. UN1X considers each file to be a sequence of 8-bit bytes; no interpretation of these bits is made by the operating system. This scheme provides maximum flexibility but little support. Each application program must include its own code to interpret an input file as to the appropriate structure. However, all operating systems must support at least one structure-that of an executable file-so that the system is able to load and run programs. The Macintosh operating system also supports a minimal number of file structures. It expects files to contain two parts: a *resource fork* and a *data fork*. The resource fork contains information of interest to the user. For instance, it holds the labels of any buttons displayed by the program. A foreign user may want to re-label these buttons in his own language, and the Macintosh operating system provides tools to allow modification of the data in the resource fork. The data fork contains program code or data-the traditional file contents. To accomplish the same task on a UNIX or MS-DOS system, the programmer would need to change and recompile the source code, unless she created her own user-changeable data file. Clearly, it is useful for an operating system to support structures that will be used frequently and that will save the programmer substantial effort. Too few structures make programming inconvenient, whereas too many causes operating-system bloat

and programmer confusion.

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

The logical record size, physical block size, and packing technique determine how many logical records are in each physical block. The packing can be done either by the user's application program or by the operating system. In either case, the file may be considered a sequence of blocks. All the basic I/O functions operate in terms of blocks. The conversion from logical records to physical blocks is a relatively simple software problem.

Because disk space is always allocated in blocks, some portion of the last block of each file is generally wasted. If each block were 512 bytes, for example, then a file of 1,949 bytes would be allocated four blocks (2,048 bytes); the last 99 bytes would be wasted. The waste incurred to keep everything in units of blocks (instead of bytes) is internal fragmentation All file systems suffer from internal fragmentation; the larger the block size, the greater the internal fragmentation.

Access Methods

Files store information. When it is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways. Some systems provide only one access method for files. Other systems, such as those of IBM, support many access methods, and choosing the right one for a particular application is a major design problem.

1. **Sequential Access: -**The simplest access method is sequential access. Information in the file is processed in order, one record after the other. This mode of access is by far the most common; for example, editors and compilers usually access files in this fashion.
2. **Direct Access: -**Another method is direct access (or relatively access) A file is made up of fixed length logical record that allow programs to read and write records rapidly in no particular order. The direct-access method is based on a disk model of a file, since disks allow random access to any file block. For direct access, the file is viewed as a numbered sequence of blocks or records. Thus, we may read block 14, then read block 53, and then write block 7. There are no restrictions on the order of reading or writing for a direct-access file.

Direct-access files are of great use for immediate access to large amounts of information. Databases are often of this type. When a query concerning a particular subject arrives, we compute

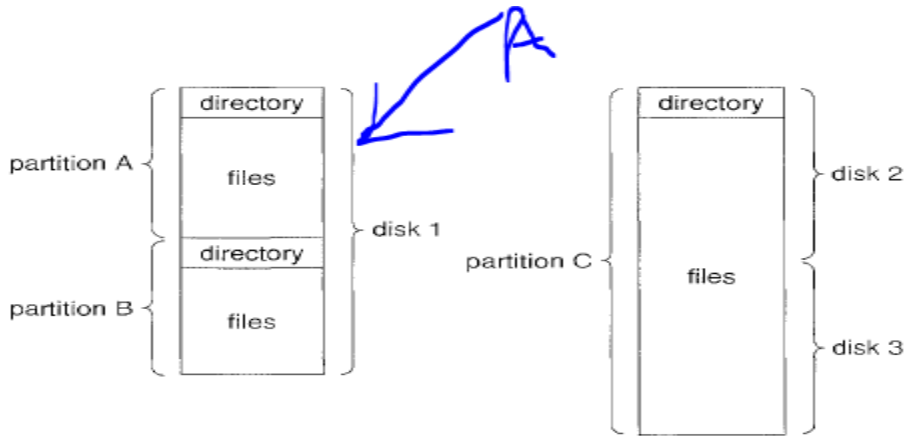
which block contains the answer and then read that block directly to provide the desired information.

Directory and Disk Structure

Next, we consider how to store files. Certainly, no general-purpose computer stores just one file. There are typically thousand, millions, and even billions of files within a computer. Files are stored on random-access storage devices, including hard disks, optical disks, and solid state (memory based) disks. *A storage device* can be used in its entirety for a file system. It can also be subdivided for finer grained control. For example, a disk can be partitioned into quarters, and each quarter can hold a file system. Storage devices can also be collected together into RAID sets that provide protection from the failure of a single disk. Sometimes, disks are subdivided and also collected into RAID sets.

Partitioning is useful for limiting the sizes of individual file systems, putting multiple file-system types on the same device, or leaving part of the device available for other uses, such as swap space or unformatted (raw) disk space. Partitions are also known as *slices* or (in the IBM world) *minidisks*. A file system can be created on each of these parts of the disk. Any entity containing a file system is generally known as a *Volume*. The volume may be a subset of a device, a whole device, or multiple devices linked together into a RAID set. Each volume can be thought of as a virtual disk. Volumes can also store multiple operating systems, allowing a system to boot and run more than one operating system.

Each volume that contains a file system must also contain information about the files in the system. This information is kept in entries in a device directory or volume table of contents. The device directory (more commonly known simply as that directory) records information -such as name, location, size, and type-for all files on that volume. Figure below shows a typical file-system organization.



Storage Structure

Common file system in a Solaris operating system: -

- **Tmpfs**: -a "temporary" file system. that is created in volatile main memory and has its contents erased if the system reboots or crashes
- **Objfs**: -a "virtual" file system (essentially an interface to the kernel that looks like a file system) that gives debuggers access to kernel symbols
- **Dfs:** -a virtual file system that maintains "contract" information to manage which processes start when the system boots and must continue to run during operation
- **Lofs**: -a "loop back" file system that allows one file system to be accessed in place of another one
- **Prods**: -a virtual file system that presents information on all processes as a file system
- **ufs, zfs:** -general-purpose file systems

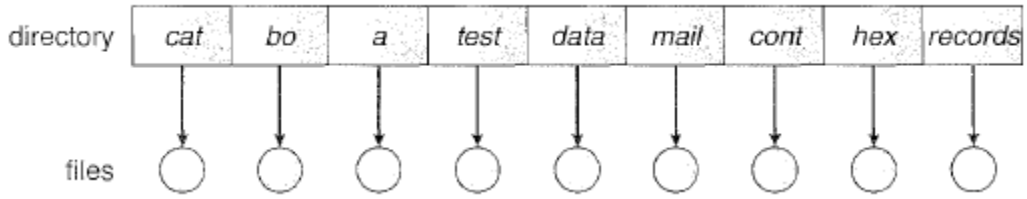
Directory Overview

The directory can be viewed as a *symbol table* that translates file names into their directory entries. If we take such a view, we see that the directory itself can be organized in many ways. We want to be able to insert entries, to delete entries, to search for a named entry, and to list all the entries in the directory. When considering a particular directory structure, we need to keep in mind the operations that are to be performed on a directory:

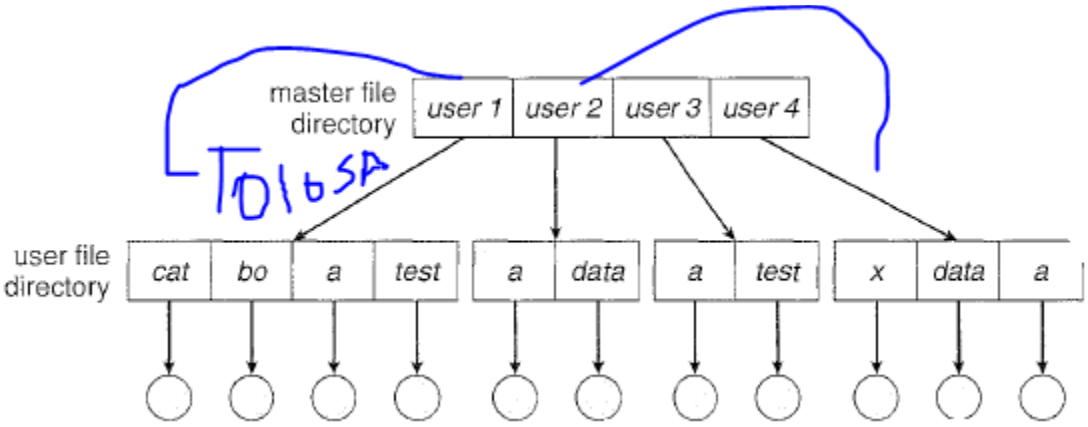
1. **Search for a file**. We need to be able to search a directory structure to find the entry for a particular file. Since files have symbolic names, and similar names may indicate a relationship between files, we may want to be able to find all files whose names match a particular pattern.
2. **Create a file**. New files need to be created and added to the directory.
3. **Delete a file**. When a file is no longer needed, we want to be able to remove it from the directory.
4. **List a directory**. We need to be able to list the files in a directory and the contents of the directory entry for each file in the list.
5. **Rename a file**. Because the name of a file represents its contents to its users, we must be able to change the name when the contents or use of the file changes. Renaming a file may also allow its position within the directory structure to be changed.
6. **Traverse the file system**. We may wish to access every directory and every file within a directory structure. For reliability, it is a good idea to save the contents and structure of the entire file system at regular intervals. Often, we do this by copying all files *to* magnetic tape. This technique provides a backup copy in case of system failure. In addition, if a file is no longer in use, the file can be copied *to* tape and the disk space of that file released for reuse by another file.

In. the following is most common schemes for defining the logical structure of a directory.

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.



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 list only the files of a single user. W11en 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.



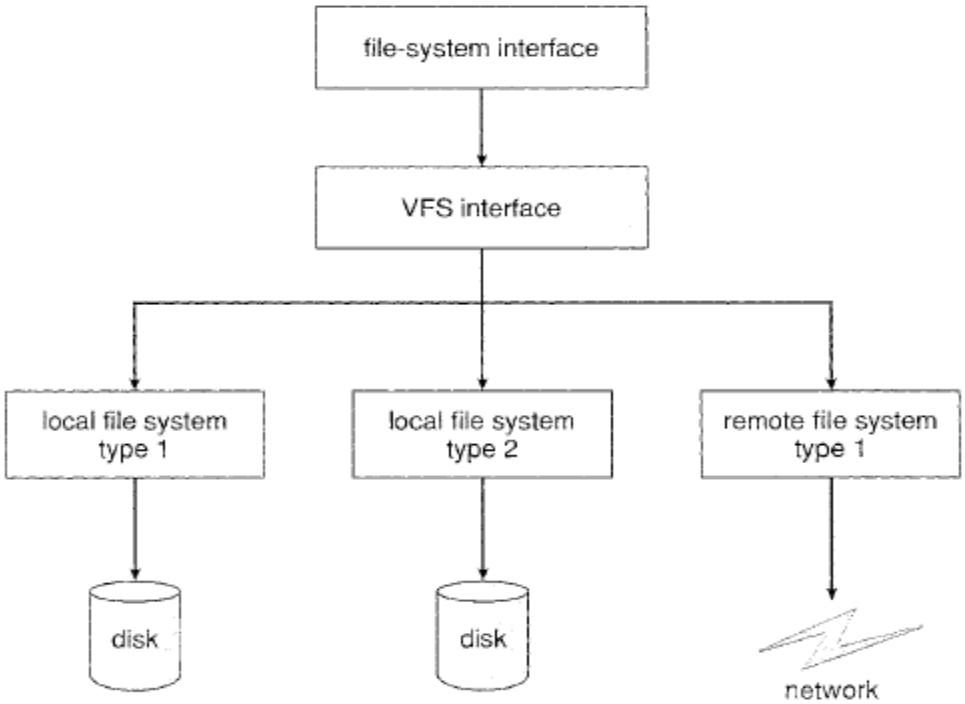
File System Mounting (initiating)

The mount procedure is *straightforward*. The operating system is given the name of the device and *mount point* the location within the file structure where the file system is to be attached. Some operating systems require that a file system type be provided, while others inspect the structures of the device and determine the type of file system. Typically, a mount point is an empty directory. For instance, on a UNIX system, a file system containing a user's home directories might be mounted as */home;* then, to access the directory structure within that file system, we could precede the directory names with */home,* as in */home/Husen-Adem/.* Mounting that file system under */users* would result in the path name */users/Husen-Adem,* which we could use *to* reach the same directory. Next, the operating system verifies that the device contains a valid file system. It does so by asking the device driver to read the device directory and verifying that the directory has the expected format. Finally, the operating system notes in its directory structure that a file system is mounted at the specified mount point. This scheme enables the operating system to traverse its directory structure, switching among file systems, and even file systems of varying types, as appropriate.

Virtual File Systems

An obvious but suboptimal method of implementing multiple types of file systems is to write directory and file routines for each type. Instead, however, most operating systems, including UNIX, use object-oriented techniques to simplify, organize, and modularize the implementation. The use of these methods allows very dissimilar file-system types to be implemented within the same structure, including network file systems, such as **NFS**. Users can access files that are contained within multiple file systems on the local disk or even on file systems available across the network. Virtual file system has two important function: -

1. It separates file-system-generic operations from their implementation by defining a clean VFS interface. Several implementations for the VFS interface may coexist on the same machine, allowing transparent access to different types of file systems mounted locally.
2. It provides a mechanism for uniquely representing a file throughout a network. The **VFS** is based on a file-representation structure, called a that contains a numerical designator for a network-wide unique file. (UNIX) Inodes are unique within only a single file system.) This network-wide uniqueness is required for support of network file systems. The kernel maintains one Vnode structure for each active node (file or directory).



File Protection

When information is stored in a computer system, we want to keep it safe from physical damage (the issue of *reliability)* and improper access (the issue of *protection).* Protection can be provided in many ways. For a small single-user system, we might provide protection by physically removing

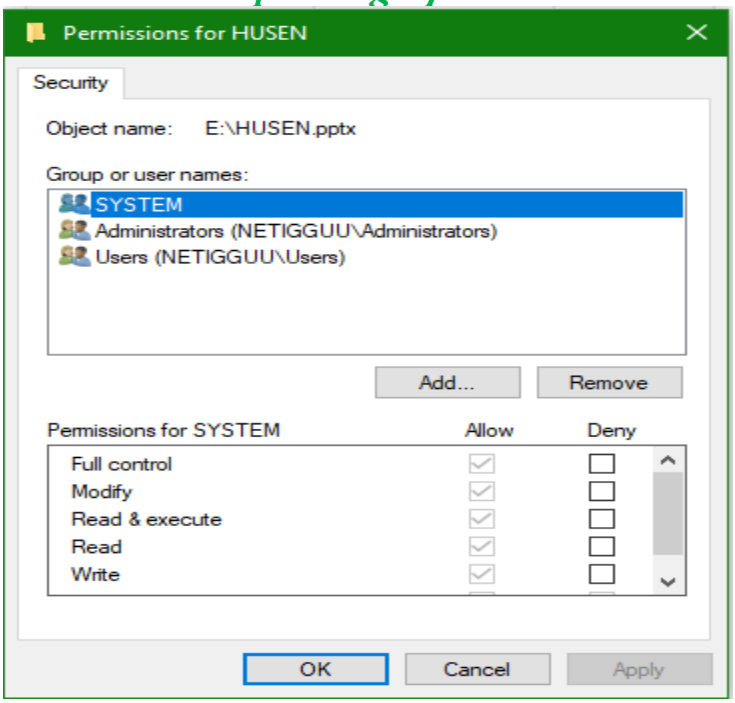
the floppy disks and locking them in a desk drawer or file cabinet. In a multiuser system, however, other mechanisms are needed.

Types of Access The need to protect files is a direct result of the ability to access files. Systems that do not permit access to the files of other users do not need protection. Thus, we could provide complete protection by prohibiting access. Alternatively, we could provide free access with no protection. Both approaches are too extreme for general use. What is needed is *controlled 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 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.

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.



Windows access control management

RECOVERY (file recovery)

A system crash can cause inconsistencies among on-disk file-system data structures, such as directory structures, free-block pointers, and free FCB pointers. Many file systems apply changes to these structures in place. A typical operation, such as creating a file, can involve many structural changes within the file system on the disk Directory structures are modified, FCBs are allocated, data blocks are allocated, and the free counts for all of these blocks are decreased. These changes can be interrupted by a crash, and inconsistencies among the structures can result. For example, the free FCB count might indicate that an FCB had been allocated, but the directory structure might not point *to* the FCB. Compounding this problem is the caching that operating systems do to optimize I/0 performance. Some changes may go directly *to* disk, while others may be cached. If the cached changes do not reach disk before a crash occurs, more corruption is possible. In addition *to* crashes, bugs in file-system implementation, disk controllers, and even user applications can corrupt a file system. File systems have varying methods to deal with corruption, depending on the file-system data structures and algorithms.

NTF (network file system)

Network file systems are commonplace. They are typically integrated with the overall directory structure and interface of the client system. NFS is a good example of a widely used, well implemented *client-server network* file system. Here, we use it as an example to explore the implementation details of network file systems.

NFS is both an implementation and a specification of a software system for accessing remote files across LANs (or even WANs). NFS is part of ONC+, which most UNIX vendors and some PC operating systems support. The implementation described here is part of the Solaris operating system, which is a modified version of UNIX SVR4 running on Sun workstations and other hardware. It uses either the TCP or UDP /IP protocol (depending on the interconnecting network). The specification and the implementation are intertwined in our description of NFS. Whenever detail is needed, we refer to the Sun implementation; whenever the description is general, it applies to the specification also.

Exercise

I. Select alternative answer from the given option

1. A _____is that set of system software that provides services to users and applications in the use of files.
   A) File types C) File Security
   B) File management system D) File control
2. The directory can be viewed as a _____that translates file names into their directory entries
   A) Symbol table C) Folder
   B) Home D) Director
3. Which one is not types of file operations?
   A) Read C) Edit
   B) Write D) Delete
4. Repositioning within a file need not involve called _____
   A) Seek C) Write
   B) Read D) Rewrite
5. A _____can be used in its entirety for a file system.
   A) Memory C) ROM
   B) Storage device D) RAM

II. Give the brief answer for the following question

1. Undoubtable state about file mounting and try with example!
2. Clearly state how to access file and file structure
3. Do you think that virtual file system till now working? If yes how it works explain?
4. State about file directory and illustrate or show by diagrammatically!