Chapter 1

Introduction

1.1 Operating System and Function

An *operating system (OS)* is a program that controls the execution of application programs and acts as an interface between applications and the computer hardware.

Examples: Microsoft Windows, Apple macOS, Linux OS, Unix OS.

An OS has three main objectives:

i. An operating system should make the computer more convenient to use.

ii. An operating system should use the hardware efficiently.

iii. An OS should allow the development, testing, and introduction of new functions.

An operating system has two basic functions:

1. **Resource management**
   - Process management
   - Memory management
   - I/O device management
   - File management
2. **Virtual machine management**

1. The Operating System as Resource Manager

Computer systems consist of different hardware and software as resources such as processors, memory, timers, disks, n/w interfaces and so on. Each of these resources has its own functionality and usage which needs to be managed and piled up to make whole computer system work efficiently and complete the given task. So, OS is the one which manages all the resources providing an ordered and controlled allocation of resources among various programs.

It acts as an interface between the user and hardware devices where application software is used to perform specific task and hardware helps in achieving them.

Why resource management?

The management and protection of resources becomes a basic requirement when we have multiple users. This is because we may have many applications trying to gain access from system's resources like networking devices, memory etc. along with sharing of information like files and databases but these applications cannot access same device and processor at same time without causing chaos. For example, consider three programs which request to use a printer at same time. Accessing same printer at same time results to undesired output. Hence, this requires an operating system to control this traffic and ensure that the resource is used by only one program at a particular time.

How resource management?

Resource management includes multiplexing (sharing of resources) by time and by space. According to time multiplexing, different programs use particular resource at specific time as allocated to resources determining how the time has been multiplexed. Similarly, when the resources are multiplexed by space, the sharing resources are divided into units which are allocated to different programs. For example, main memory is divided among several programs where the programs are resided at. Another example is (hard) disk where OS allocates disk space and keeps track of disk usage by particular program resembling a resource manager.

Thus, as a resource manager, OS manages the following:

- Memory management
- I/O device management
- Process management
- File management

Memory management

An operating system manages memory by keeping track of memory in use and unused memory where it allocate and deallocate spaces for the processes available. However, a decision is made by OS regarding when and how to load each process into memory.

Process management

OS manages a process by its creation, deletion, suspension, and scheduling of processes. It uses traffic controller to constantly check processor and status of process, job scheduler to select job from queue for execution, and dispatcher to allocate processor for the particular process chosen by scheduler.

Device management

OS manages devices to each process by constantly keeping track of I/O devices and deciding which device is to be allocated in which process.

File system management

OS manages files by keeping track of all information about files that how they are opened and closed. OS creates and deletes file directories, manipulates (read, write, extend, rename, copy, protect) them and also provide higher level services such as backup, accounting, etc.

2. Operating System as an Extended (Virtual) Machine

In every computer system, the architecture (instruction set, memory, I/O, and bus structure) at machine language level are very primitive and difficult to program, especially for input/output. Consider an example of modern SATA (Serial ATA) hard disk that are used on most computers. It is too difficult for the programmer to use this hard disk at the hardware level because a programmer would have to know detailed information (according to Anderson's book of 2007, information for interfacing SATA hard disk consisted of 450 pages). So, user or an average programmer do not want to be involved into programming of such complex devices.

"Abstraction" is the key concept to manage this complexity. An OS provides a simple, high-level abstraction such as a collection of named files. Using this abstraction, programs can create, write, and read files without worrying the details of hardware. Such file is a useful information like digital photos, emails, web page, songs, etc. and it is much easier to deal with these files than dealing with SATA or any other disk.

Interfacing the real processors, memories, disks and any other devices are very complicated. On addition to it, it is more difficult for the programmer to design a software to use these interfaces. Thus, OS hides this complexity of hardware and presents a nice, clean and *beautiful interface* to user. Through abstraction, the operating systems presents the user with the equivalent of an *extended machine* or *virtual machine* that is easier to work with than the underlying hardware.

1.2 Evolution of Operating Systems

The evolution of operating systems can be understood under the following headings:

1.2.1 Serial Processing

The earliest computers (late 1940s to the mid-1950s) had no OS; the programmer interacted directly with the computer hardware. These computers were run from a console consisting of display lights, toggle switches, some form of input device, and a printer. Programs in machine code were loaded via the input device (e.g., a card reader). If an error halted the program, the error condition was indicated by the lights. If the program proceeded to a normal completion, the output appeared on the printer.

These early systems presented two main problems:

- **Scheduling:** When users required computer time, they had to sign a sign-up sheet indicating the amount of time they needed. A user might sign up for an hour and finish in 40 minutes; this would result in wasted computer processing time. On the other hand, the user might run into problems, not finish in the allotted time, and be forced to stop before resolving the problem.

- **Setup time:** A single program, called a job, could involve loading the compiler plus the high-level language program (source program) into memory, saving the compiled program (object program), and then loading and linking together the object program and common functions. Each of these steps could involve mounting or dismounting tapes or setting up card decks. If an error occurred, the user typically had to go back to the beginning of the setup sequence. Thus, a considerable amount of time was spent just in setting up the program to run.

This mode of operation is *serial processing*, reflecting the fact that users have access to the computer in series.

1.2.2 Simple Batch Systems

To solve the problems associated with scheduling and setup time, the concept of a batch OS was developed. The first batch OS was developed by General Motors in mid-1950s for use on an IBM 701.

With this type of OS, the user has no direct access to the processor. Instead, the user submits the job on cards or tape to a computer operator, who batches the jobs together sequentially and places the entire batch on an input device, for use by the monitor (a piece of software). The monitor processes each job in the order it was loaded. When one job is finished, the monitor runs the next job in line from the batch until all jobs are completed. Batch is a group of jobs with similar needs.

With a batch operating system, processor time alternates between execution of user programs and execution of the monitor. These systems presented two overheads: Some main memory is given over to the monitor and some processor time is consumed by the monitor. Despite this, the simple batch system improves utilization of the processor.

1.2.3 Multiprogrammed Batch Systems

Even with the simple batch OS, the processor time is often idle due to the fact that the I/O devices are slow compared to the processor. To solve this problem, the concept of "multiprogramming" or "multitasking" was introduced. Multiprogramming is the central theme of modern operating systems.

Multiprogramming increases CPU utilization by organizing jobs in such a manner that the CPU always has one job to execute. If a computer is required to run several programs at the same time, the processor could be kept busy for most of the time by switching its attention from one program to the next. Additionally, I/O transfer could overlap the processor activity i.e., while one program is awaiting for an I/O transfer, another program can use the processor. So the CPU never sits idle or if comes in an idle state then after a very small time it is again busy.

1.2.4 Time-Sharing Systems

This systems also use multiprogramming. In time-sharing systems, the processor time is shared among multiple users. Multiple users simultaneously access the system. Time-sharing systems offer the users an opportunity to interact directly with the computer. Using a terminal and keyboard, each user submits a job request by pressing a transmit key and wait their turn for a response from the processor. The intention of time sharing is to minimized response time back to the user, reduce idle time, and still maximize processor usage. Today, UNIX systems still use the concept of "time sharing".

1.3 Types of Operating System

Modern computer operating systems may be classified into following types[1]:

1. **Batch Processing Operating System**

   In this type of operating system, the users submit jobs to a central place where these jobs are collected into a batch, and subsequently placed on an input queue at the computer where they will be run. The users of a *batch operating system* do not interact with the computer directly. Each user prepares his job on an off-line device like punch cards and submits it to the computer operator. Here jobs means

rogramming"
is the central

y organizing
o to execute.
e same time,
by switching
ionally, I/O
while one
ram can use
s in an idle

me-sharing
iple users.
me-sharing
ly with the
omits a job
urn for a
aring is to
time, and
ill use the

ified into

jobs to a
atch, and
computer
perating
ach user
ards and
means

program plus *input data* plus *control instructions*. To speed up processing, jobs with similar needs are batched together and run as a group. The programmers leave their programs with the operator and the operator then sorts the programs with similar requirements into batches.

The problems with batch systems are as follows:

- No interaction between the user and the job.
- CPU is often idle.
- Difficult to provide the desired priority.

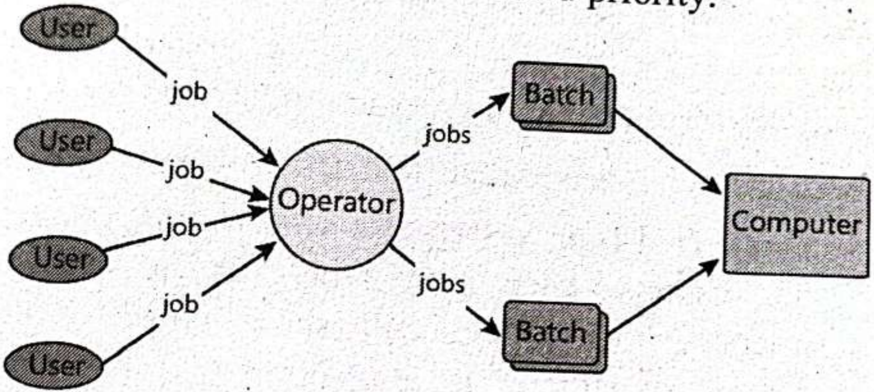


Figure 1.1: *Batch operating system process*

Examples of use: Payroll, stock control and billing systems.

2. Time Sharing Operating System

In *time sharing operating system*, a computer provides computing services to several or many users concurrently online. The various users share the central processor, the memory, and other resources of the computer system as facilitated, controlled, and monitored by the operating system.

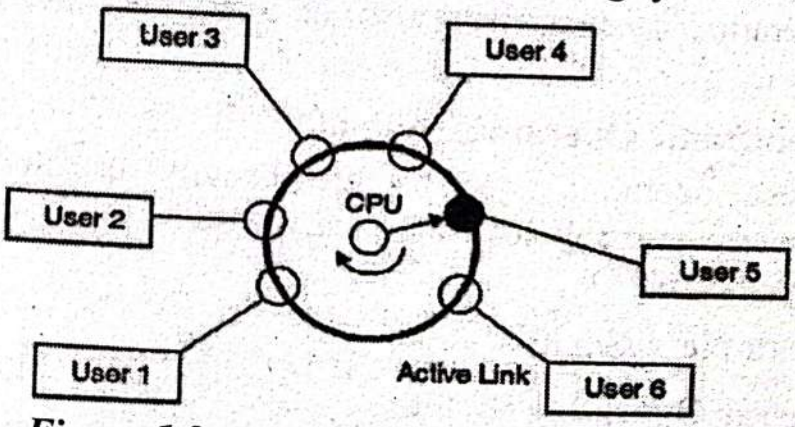


Figure 1.2: *Time sharing operating system*

Examples of use: A mainframe computer

3. **Real-Time Operating System**

This type of operating system is designed to service those applications where response time is critical in order to prevent error, misrepresentation or even disaster. A very important part of real-time operating system is managing the resources of the computer so that a particular operation executes in precisely the same amount of time every time it occurs. Consider an example of a car running on an assembly line. Certain actions are to be taken at certain instants of time. If the actions are taken too early or too late, then the car will be ruined. Therefore, for such systems, the deadlines must be met in order to produce the correct result.

There are two types of real-time operating systems:

- **Hard real-type system:** In a *hard real-type system*, if the response takes more time than the specified time interval, the system will show failed results. The secondary storage is also limited in 'hard real type systems.
- **Soft real-type system:** The *soft real-type system* does not fail the program if the response takes more time than the specified time, it will just show the output but it can compromise with the accuracy of the response.

Examples of use: Air-traffic control system, machine tool control system, airline reservations system, monitoring system of a nuclear power station, etc.

Operating system may also be classified into following categories[2]:

1. **Mainframe Operating System**

These operating systems are heavily oriented towards processing many jobs at once, most of which need large amounts of I/O.

Example: OS/390

2. **Server Operating System**

These operating systems run on servers (personal computers, workstations, or mainframes). They serve multiple users a once over a network and allow the users to share hardware and software resources.

Examples: Solaris, FreeBSD, Linux and Windows Server 201x.

3. **Multiprocessor Operating System**

   A *multiprocessing operating system* is one which consists of more than one independent processing unit. Because of multiple CPUs, this kind of operating system perform parallel execution. We use this kind of operating system when we have many jobs to perform and the single CPU switching takes much more time to execute all the processes. Many popular operating systems such as Windows, Linux run on multiprocessors.

4. **Personal Computer Operating System**

   These operating systems provide good support to a single user and are widely used for word processing, spreadsheets, games, Internet access, etc.

   Examples: Windows 7, Windows 8, Linux, Apple's OS X

5. **Handheld Computer Operating System**

   These operating systems are found in handheld computers (smartphones and tablets).

   Examples: Google's Android, Apple's iOS

6. **Embedded Operating System**

   These operating systems are designed for specific purposes and are found in embedded systems. Embedded system is a computer system that has a dedicated function within a large mechanical or electronic system.

   Examples: Embedded Linux, QNX

7. **Sensor-Node Operating System**

   These type of operating systems are found in networks of tiny sensor nodes i.e., tiny computers. Sensor networks are used to protect the perimeters of buildings, detect fire in forests, forecast weather, etc.

   Example: Tiny OS

8. Real-Time Operating System

9. Smart Card Operating System

These operating systems run on smart cards. Smart cards are credit-card-sized devices containing a CPU chip which are used for electronic payments, proprietary system, etc.

Some Other Terminologies

1. Interactive Operating System

In an *interactive operating system*, the user directly interacts with the OS. Interactive operating systems take the input from the user i.e., from a human being and then produces an output. The input which is sent to the computer can be any form like pressing the button or selecting something from the cursor, or by typing on the keyboard. Today mostly users use Windows or Mac OS in their computer system, which can be treated as the best example of an interactive system (or graphical interactive system). Integrated Development Environments (IDEs) and a Web browser is also an example of a complex interactive system.

Figure 1.3: *Interactive operating system*

Example of use: Automated teller machine (ATM)

2. Multiprogramming Operating System

This type of system allows more than one active user program (or part of user program) to be stored in main memory simultaneously. So, a time-sharing system is a multiprogramming system. However, a multiprogramming system is not necessarily a time-sharing system.

3. Distributed Operating System

A *distributed computer system* is a collection of autonomous computer systems capable of communication and cooperation via their hardware and software interconnections.

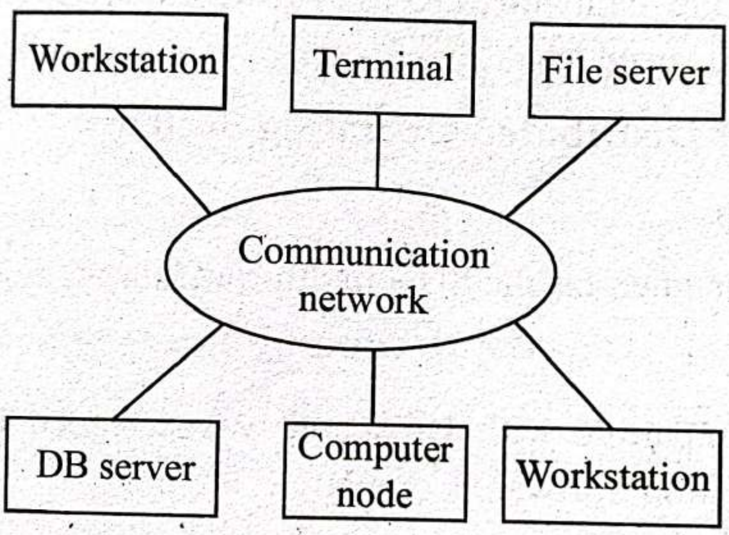


***Figure 1.4:** Distributed computer system*

A *distributed operating system* governs the operation of a distributed computer system and provides a virtual machine abstraction to its users. The key objective of a distributed operating system is transparency. Distributed OS is more reliable or fault tolerant i.e., it performs even if certain part of the hardware starts malfunctioning. Here users access remote resources in the same manner as they access local resource. Distributed operating systems usually provide the means for system wide sharing of resources, such as computational capacity, files, and I/O devices. In addition to typical operating-system services provided at each node for the benefit of local clients, a distributed operating system may facilitate access to remote resources, communication with remote processes, and distribution of computations.

To understand why distributed OS is most demanding, let's see Facebook, that currently, has more than 1.5 billion active monthly users, google performs at least 1 trillion searches per year, about 48 hours of video is uploaded in YouTube every minute. Here a single system would be unable to handle the processing. To cope with the extremely higher demand of users in both processing power and data storage, there is need of distributed operating systems.

To summarize, the advantages of distributed operating systems include fault tolerant, high availability, scalability, high computational speed, etc.

Examples of Distributed Operating Systems:

- **Solaris**

  It is designed for the SUN multiprocessor workstations

- **OSF/1**

  It's compatible with Unix and was designed by the Open Foundation Software Company.

1.4 Operating System Structure

A modern OS must be engineered carefully if it has to function properly. The six designs that have been tried in practice are monolithic systems, layered systems, microkernels, client-server systems, virtual machines, and exokernels.

1. Monolithic Structure

The *monolithic operating system* is the earliest and most common operating system architecture. In this type of operating system, every component is contained in the kernel and can directly communicate with any other (i.e. simply by using function calls). Examples of monolithic operating system include OS/360, VMS, Linux, etc.

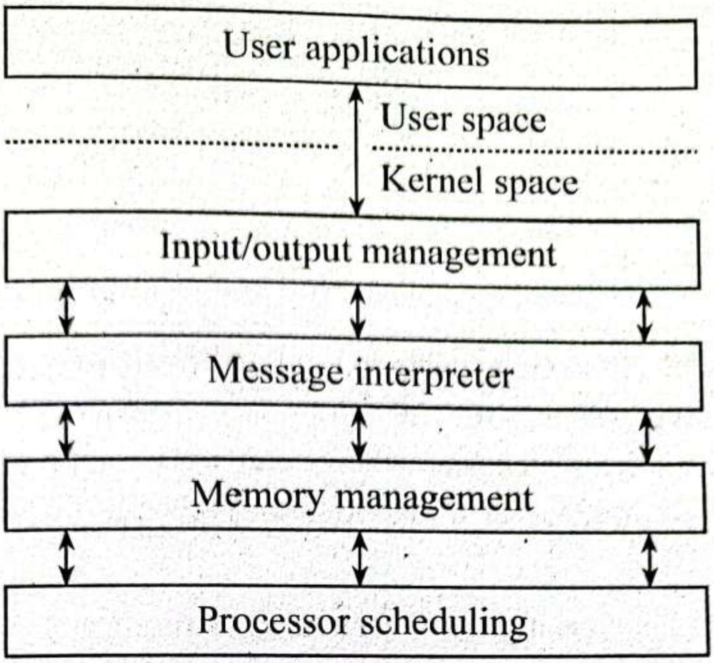


***Figure 1.5:** Monolithic kernel architecture*

Advantages of monolithic kernel:

- Highly efficient due to direct interconnection between components.

Disadvantages of monolithic kernel:

- It is difficult to isolate source of bugs and other errors because monolithic kernels group components together.
- Monolithic kernels are susceptible to damage from errant or malicious code.

2. Layered Structure

In layered operating system, the components that perform similar functions are grouped into layers. Each layer has a specific well-defined task to perform. Each layer communicates exclusively with those immediately above and below it. Lower-level layers provide services to higher-level layers.

There are six layers in the system:

Layer	Function
5	The operator
4	User programs

Layer	Function
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

Layer 0 (Processor allocation and multiprogramming) – This layer deals with the allocation of processor, switching between the processes when interrupts occur or when the timers expire. Layer 0 provides that basic multiprogramming of the CPU.

Layer 1 (Memory and drum management) – This layer deals with allocating memory to the processes in the main memory. The drum is used to hold parts of the processes (pages) for which space couldn't be provided in the main memory.

Layer 2 (Operator-process communication) – In this layer, each process communicates with the operator (user) through the console. Each process has its own operator console and can directly communicate with the operator.

Layer 3 (Input/output management) – This layer handles and manages all the I/O devices, and it buffers the information streams that are made available to it. Each process can communicate directly with the abstract I/O devices with all of its properties.

Layer 4 (User programs) – The programs used by the user are operated in this layer, and they don't have to worry about I/O management, operator/processes communication, memory management, or the processor allocation.

Layer 5 (The operator) – The system operator process is in the outer most layer.

An earlier example of a layered operating system is the operating system. Today many operating systems like Windows and Linux implement some level of layers.

Advantages of layered operating systems:

1. Each layer can be tested and debugged separately.
2. Designers can change each layer's implementation without needing to modify the other layers.

Disadvantages of layered operating systems:

1. Since a user process's request may need to pass through many layers before it is serviced, performance of the system is degraded compared to that of a monolithic kernel.
2. Since all layers have unrestricted access to the system, layered kernels are also susceptible to damage from errant and malicious code.

3. Microkernel

Microkernel is one of the classifications of the kernel where the user services and kernel services are implemented in different address space.

As compared to monolithic kernel, in microkernel almost all the functions and services are removed from the kernel mode and relocated into the user mode. As a result, the kernel size is minimum. Since almost all the functions and services operated in user mode, here the kernel mode does the following tasks only:

- Interrupt handling,
- Low-level process management, and
- Message-passing handling.

A common communication method in microkernel is message passing. In microkernel, exchange of information among the processes are done using Inter Process Communication (IPC). A message (simply data) is transferred from one process to another using IPC. Here messages are passed using message registers.

Famous examples of a microkernel system include Integrity, K42, PikeOS, Symbian, and MINIX 3.

The primary purpose of this system is to provide high reliability. Because of the high reliability that it provides, the applications of microkernels can be seen in real-time, industrial, avionics (electronics fitted in aircraft and aviation), and military applications that are mission-critical and require high reliability.

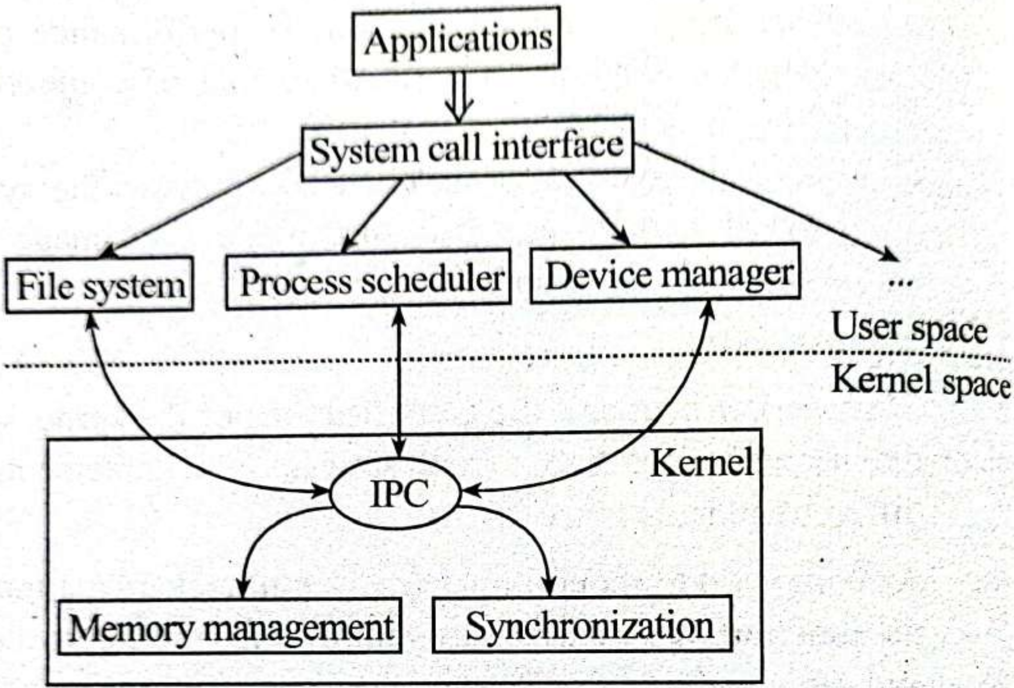


Figure 1.6: Microkernel operating system architecture

Advantages of microkernels:

- Microkernels exhibit a high degree of modularity making them extensible, portable, and scalable.
- Failure of one or more components does not cause the operating system to fail.

Disadvantages of microkernels:

- Providing services in a microkernel system are expensive compared to the normal monolithic system.
- The performance of a microkernel system may be degraded due to intermodule communication.

Comparison b
Kernel

S.N.	Microl
1.	In micr services services are address spa
2.	Microkerne size.
3.	Slow execu
4.	The micro extendible.
5.	If a service effect on microkerne
6.	To write more code
7.	*Example*: L4Linux, S Mac OS X

4. Client-S

A slight
two classe
some serv
model is l

Communi
message
processes
sends it t
work and

If the cli
some opt
they are
network l

Comparison between Microlithic Kernel and Monolithic Kernel

S.N.	Microlithic Kernel	Monolithic Kernel
1.	In microkernel, user services and kernel, services are kept in separate address space.	In monolithic kernel, both user services and kernel services are kept in the same address space.
2.	Microkernels are smaller in size.	Monolithic kernel is larger than microkernel.
3.	Slow execution.	Fast execution.
4.	The microkernel is easily extendible.	The monolithic kernel is hard to extend.
5.	If a service crashes, it does effect on working of microkernel.	If a service crashes, the whole system crashes in monolithic kernel.
6.	To write a microkernel, more code is required.	To write a monolithic kernel, less code is required.
7.	*Example*: QNX, Symbian, L4Linux, Singularity, K42, Mac OS X	*Example*: Linux, BSDs (FreeBSD, OpenBSD, NetBSD), Microsoft Windows, Solaris, OS-9, DOS, OpenVMS

4. Client-Server Model

A slight variation of the microkernel idea is to distinguish two classes of processes, the servers, each of which provides some service, and the clients, which use these services. This model is known as the *client-server model.*

Communication between clients and servers is obtained by message passing. To receive a service, one of the client processes constructs a message saying what it wants and sends it to the appropriate service. The service then does it work and sends back the answer.

If the clients and servers are on the same machine, then some optimizations are possible. But generally speaking, they are on different systems and are connected via a network link like LAN or WAN.

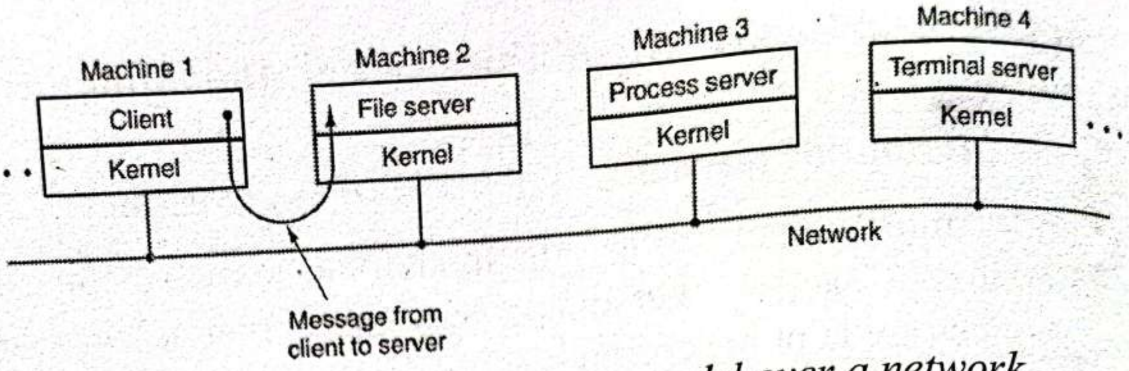


Figure 1.7: The client-server model over a network

The best example of this model is when you are scrolling Facebook, you are the client, and you are requesting the Facebook page from where it is hosted i.e. server. The internet is basically the example since much of the web operates this way.

Advantages of client-server model:

1. All the required data is concentrated in a single place i.e., the server. So, it is easy to protect the data and provide authorization and authentication.
2. The server need not be located physically close to the clients. Yet the data can be accessed efficiently.
3. It is easy to replace, upgrade or relocate the nodes in the client server model because all the nodes are independent and request data only from the server.

Disadvantages of client-server model:

1. If all the clients simultaneously request data from the server, it may get overloaded. This may lead to congestion in the network.
2. If the server fails for any reason, then none of the requests of the clients can be fulfilled. This leads of failure of the client server network.
3. The cost of setting and maintaining a client-server model are quite high.

5. Virtual Machine Structure

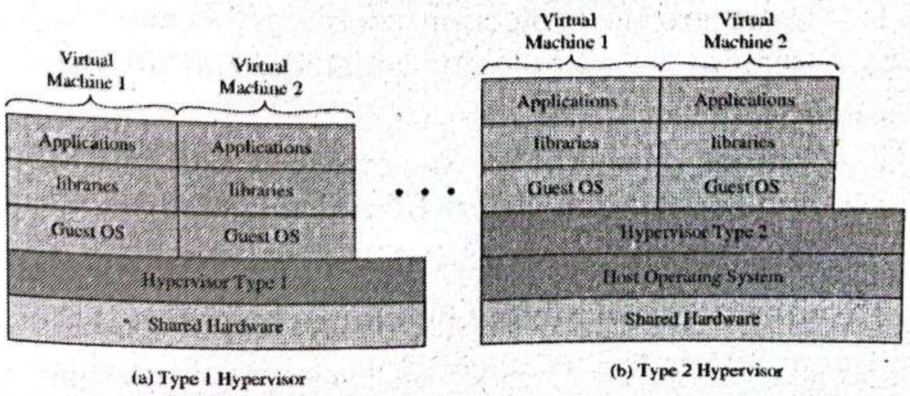
A *virtual machine (VM)* is a virtual environment that functions as a virtual computer system with its own CPU, memory, network interface, and storage, created on a physical hardware system. VMs allow multiple different OS to run simultaneously on a single computer.

Let us take an example: We have a laptop where windows 10 is installed. With the use of VMware workstation software (which is installed on our windows OS) we can run various other OS like: Linux, Windows 7, Windows 10, etc. This means we can run various OS simultaneously inside our single Laptop using the technique of virtual machine.

Virtual machine technology is used for many use cases across on-premises and cloud environments. More recently, public cloud services are using virtual machines to provide virtual application resources to multiple users at once, for even more cost efficient and flexible compute.

Virtualization is a form of abstraction (hiding irrelevant details to the users). In OS, the OS abstracts the disk I/O commands from a user through the use of program layers and interfaces. In the similar way virtualization abstracts the physical hardware from the virtual machines it supports.

The *virtual machine monitor* (also called *hypervisor*) is the software that provides this abstraction. It creates the illusion of multiple(virtual) machines on the same physical hardware. There are two types of hypervisor: Type 1 and Type 2. Type 1 hypervisors run on the bare metal. Examples of type 1 hypervisors are VMware ESXi, Microsoft Hyper-V. Type 1 hypervisor can directly control the physical resources of the host. Type 2 hypervisors run on the top of the OS (as shown in figure). It relies on the OS to handle all of the hardware interactions. Examples of type 2 hypervisors are VMware Workstation and Oracle VM Virtual Box.



(a) Type 1 Hypervisor

(b) Type 2 Hypervisor

***Figure 1.8:** Types of hypervisor*

The following table shows the [illegible] Type 1 and Type 2 hypervisor:

Type 1 hypervisor	Type 2 hypervisor
1. A hypervisor that runs directly on the host's hardware to control the hardware and to manage the guest operating systems.	1. A hypervisor that runs on a conventional operating system just as other computer programs do.
2. Called a native or bare metal hypervisor.	2. Called a host OS hypervisor.
3. Runs directly on the host's hardware.	3. Runs on an operating system similar to other computer programs.
Examples: AntsleOs, Xen, XCP-ng, Microsoft Hyper-V, VMware ESX/ESXi, Oracle VM Server for x86.	Examples: VMware Workstation, VMware Player, VirtualBox, Parallel Desktop for Mac

The Java Virtual Machine (JVM)

Another area where virtual machines are used, but in a different way, is for running Java programs. Sun Microsystems invented a virtual machine called the JVM. The Java compiler produces code for JVM, which then typically is executed by a software JVM interpreter. The advantage of this approach is that the JVM code can be shipped over the Internet to any computer that has a JVM interpreter and run there.

Advantages of virtual machine structure:

1. There are no protection problems because each virtual machine is completely isolated from all other virtual machines.
2. Virtual machine can provide an instruction set architecture that differs from real computers.
3. Easy maintenance, availability, & convenient recovery.

Disadvantages of virtual machine structure:

1. A VM is less efficient than an actual machine when it accesses the host hard drive indirectly.

The following table shows the differences between Type 1 and Type 2 hypervisor:

Type 1 hypervisor	Type 2 hypervisor
1. A hypervisor that runs directly on the host's hardware to control the hardware and to manage the guest operating systems.	1. A hypervisor that runs on a conventional operating system just as other computer programs do.
2. Called a native or bare metal hypervisor.	2. Called a host OS hypervisor.
3. Runs directly on the host's hardware.	3. Runs on an operating system similar to other computer programs.
Examples: AntsleOs, Xen, XCP-ng, Microsoft Hyper-V, VMware ESX/ESXi, Oracle VM Server for x86.	Examples: VMware Workstation, VMware Player, VirtualBox, Parallel Desktop for Mac

The Java Virtual Machine (JVM)

Another area where virtual machines are used, but in a different way, is for running Java programs. Sun Microsystems invented a virtual machine called the JVM. The Java compiler produces code for JVM, which then typically is executed by a software JVM interpreter. The advantage of this approach is that the JVM code can be shipped over the Internet to any computer that has a JVM interpreter and run there.

Advantages of virtual machine structure:

1. There are no protection problems because each virtual machine is completely isolated from all other virtual machines.
2. Virtual machine can provide an instruction set architecture that differs from real computers.
3. Easy maintenance, availability, & convenient recovery.

Disadvantages of virtual machine structure:

1. A VM is less efficient than an actual machine when it accesses the host hard drive indirectly.

2. Great use of disk space and RAM consumption, since it takes all the files for each OS installed on each VM.
3. Difficulty in direct access to hardware.

6. Exokernels

Exokernels are a subset of virtual machines. In this, the disks are actually partitioned, and resources are allocated while setting it up. In other words, giving each user a subset of the resources.

The advantage of the exokernel scheme is that it saves a layer of mapping. In the other designs, each virtual machine thinks it has its own disk, with blocks running from 0 to some maximum, so the virtual machine monitor must maintain tables to remap disk addresses (and all other resources). With the exokernel, this remapping is not needed. The exokernel need only keep track of which virtual machine has been assigned which resource[3].

1.5 Operating System Services

An operating system provides services to both the users and to the programs. It provides programs an environment to execute. It provides users the services to execute the programs in a convenient manner. Figure below shows one view of the various operating-system services and how they interrelate.

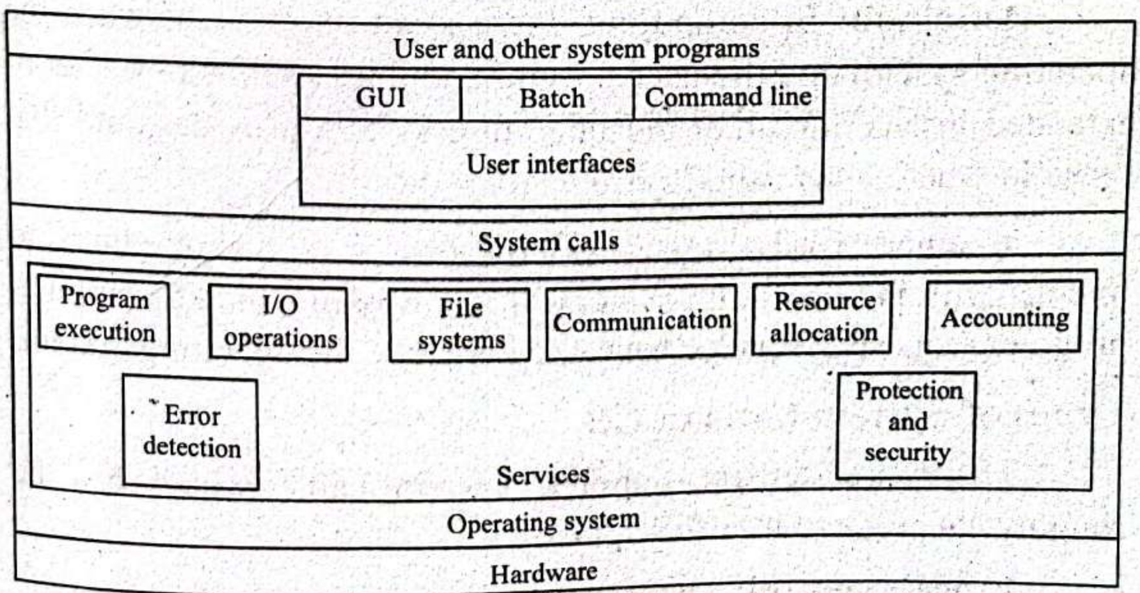


Figure 1.9: A view of operating system services.

1.5.1 System Calls

The user interface of an operating system is primarily for dealing with abstractions. All this happens by various system calls. System calls differ from one OS to another, but the underlying concept remains the same.

What are System Calls in OS?

Let's say we have a company that offers various tools and resources to clients.

1. Clients need these tools and resources as a service.
2. A manager communicates with the clients and asks the company to serve as per the clients' requirements.
3. The clients ask for tools and resources to the manager, according to their needs.
4. The manager follows a process to get the service from the company.

Here, the manager works as a mediator between the clients and the company.

In this scenario, the company is the operating system. The tools and resources are the resources of the OS. The clients are the users. And the manager is a mediator between the company and the clients, which takes up the role of system calls.

Definition: In computing, there is an interface between the operating system and the user program, and it is defined by a set of extended instructions that the operating system provides, and that set of instructions is called *system calls*.

In other words, a *system call* is a way for programs to interact with the operating system. It is a way in which a computer program requests a service from the kernel of the operating system.

Types of System Calls in OS

There are several hundred system calls, which can be roughly categorized into five types:

1. **Process control:** These include *CreateProcess, ExitProcess*

2. **File management:** These include *CreateFile, ReadFile, DeleteFile*
3. **Device management:** These include *RequestDevice, ReadDevice*
4. **Information maintenance:** This includes *GetLocalTime*
5. **Communication:** This includes *CreatePipe*

Examples of System Calls for Process Management

A lot of services such as memory management, file system management, etc. are provided by the system calls, and one of them is process management.

A set of system calls given below is used to manage a process:

- **fork( )**: To create a process we use a method called **fork( )**. It creates a child process identical to the parent in every way.
- **exec( )**: To run a program, that is, to execute a program, **exec( )** method is used.
- **wait( )**: To make a process wait, **wait( )** is used.
- **getpid( )**: Every process has a unique process id, to get that unique id **getpid( )** is used.
- **getppid( )**: To get the unique process id of a parent process, **getppid( )** is used.
- **exit( )**: It is used to terminate a process with an exit status.

Examples of System Calls for File Management

For file management, following system calls are mainly used:

- **open( )**: This system call is used to open a file for reading, writing, or both.
- **read( )**: To read the content from a file into the buffer, we use a read() system call.
- **write( )**: It is used to write content into the file from the buffer.
- **close( )**: This system call is used to close the opened file.

Examples of System Calls for Directory Management

For directory management, following system calls are mainly used:

- **mkdir( )**: Used to create a new directory.
- **rmdir( )**: Used to remove a directory.
- **link( )**: Used to create a link to an existing file.
- **opendir( )**: Used to open a directory for reading.
- **closedir( )**: This is used to close a directory.

Examples of Miscellaneous System Calls

Till now, we have seen some of the service-specific system calls. Following is a list which contains some of the miscellaneous system calls:

- **chdir( )**: Used to change the working directory.
- **chmod( )**: Used to change a file's protection bits.
- **kill( )**: Used to send a signal to a process.
- **time( )**: It gets the elapsed time since January 1, 1970.
- **ulimit( )**: Used to get and set user limits.
- **acct( )**: It is used to enable or disable process accounting.
- **alarm( )**: It is used to set a process alarm clock.
- **modload( )**: It loads dynamically loadable kernel module.
- **moduload( )**: It unloads the kernel module.
- **modpath( )**: It is used to change the path from which the modules are loaded.

1.5.2 Shell and Shell Script

A *shell* is a special user program which provides an interface to the user to use operating system services. Shell accepts human readable commands from the user and convert them into something which kernel can understand.

Example of shell commands:

1. **Displaying the file contents on the terminal**
   - **cat :** It is generally used to concatenate the files. It gives the output on the standard output.

- **more :** It is a filter for paging through text one screenful at a time.
- **less :** It is used to view the files instead of opening the file. Similar to *more* command but it allows backward as well as forward movement.
- **head :** Used to print the first N lines of a file. It accepts N as input and the default value of N is 10.
- **tail :** Used to print the last N-1 lines of a file. It accepts N as input and the default value of N is 10

2. File and directory manipulation commands

- **mkdir :** Used to create a directory if not already exist. It accepts directory name as input parameter.
- **cp :** This command will copy the files and directories from source path to destination path. It can copy a file/directory with new name to the destination path. It accepts source file/directory and destination file/ directory.
- **mv :** Used to move the files or directories. This command's working is almost similar to *cp* command but it deletes copy of file or directory from source path.
- **rm :** Used to remove files or directories.
- **touch :** Used to create or update a file.

3. Extract, sort and filter data commands

- **grep :** This command is used to search for the specified text in a file.
- **sort :** This command is used to sort the contents of files.
- **wc :** Used to count the number of characters, words in a file.
- **cut :** Used to cut a specified part of a file.
- **| :** Pipe is a command in Linux that lets you use two or more commands such that output of one command serves as input to the next. In short, the output of each process directly as input to the next one like a pipeline. The symbol '|' denotes a pipe. **Example:** cat filename | less

4. Basic term

- **ls :** To get the list of all the files or folders.
- **cd :** Used to change the directory.
- **du :** Show disk usage.
- **pwd :** Show the present working directory.
- **man :** Used to show the manual of any command present in Linux.
- **rmdir :** It is used to delete a directory if it is empty.
- **ln file1 file2 :** Creates physical link.
- **ln -s file1 file2 :** Creates symbolic link.

5. File Permissions commands

The *chmod* and *chown* commands are used to control access to files in UNIX and Linux systems.

- **chown :** Used to change the owner of file.
- **chmod :** Used to modify the access/permission of a user.

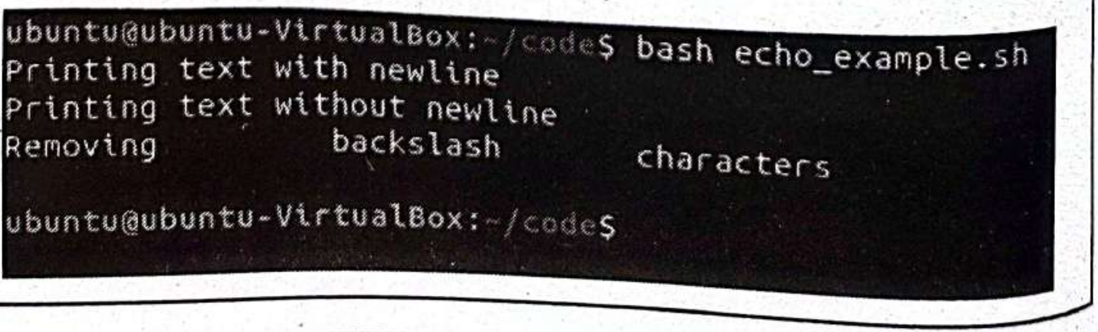
A shell script is a list of commands in a computer program that is run by the Unix shell which is a command line interpreter. Shell scripts have several required constructs that tell the shell environment what to do and when to do it.

Example of shell script:

```
#!/bin/bash
echo "Printing text with newline"
echo -n "Printing text without newline"
echo -e "\nRemoving \t backslash \t characters\n"
```

Run the file with bash command.

```
$ bash echo_example.sh
```



```
ubuntu@ubuntu-VirtualBox:~/code$ bash echo_example.sh
Printing text with newline
Printing text without newline
Removing         backslash        characters

ubuntu@ubuntu-VirtualBox:~/code$
```

ANS

1. What

Ans: Buffe
a sing
to sta
the ne
are bo
data i
Thus,
while
output
until t

SPOO
Rather
memo
reader
table
reques
for out
written
job wi

2. What
archit

Ans: A laye
betwee
microk
operati

ANSWERS TO SOME IMPORTANT QUESTIONS

1. **What is buffering and spooling? Explain.**

Ans: **Buffering:** It is overlapping input, output and processing of a single job. After data has been read and the CPU is about to start operating on it, the input device is instructed to begin the next input immediately, so the CPU and input devices are both busy. By the time that CPU is ready for the next data item, the input device will have finished reading it. Thus, the CPU can begin processing the newly read data while the input device again starts to read the next data. For output, the CPU creates the data that is put into the buffer until the output device can accept it.

SPOOLING (Simultaneous Peripheral Operations On-line): Rather than the cards being read from card readers directly into memory and then job being processed, cards are read from card reader onto disk. Location of the card image is recorded in the table kept by OS. When a job is executed, OS satisfies its request for card reader input by reading from a disk. Similarly, for output of a line, the line is copied into a system buffer and is written on the disk. Spooling overlaps the input/output of one job with the computation of other jobs.]

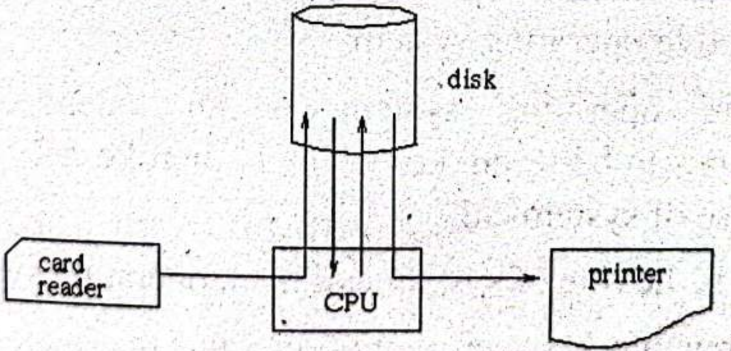


Fig.: SPOOLING

2. **What is the difference between a purely layered architecture and a microkernel architecture?**

Ans: A layered architecture enables communication exclusively between operating system components in adjacent layers. A microkernel architecture enables communication between all operating system components via the microkernel.

REFERENCES

[1] William Stallings. Operating Systems: Internals and Design Principles. 7th ed. New Jersey: Pearson Education, Inc., 2012, p.48.

[2] Andrew S. Tanenbaum, Herbert Bos. Modern Operating Systems. 4th ed. New Jersey: Pearson Education, Inc., 2015, pp.35-38.

[3] Andrew S. Tanenbaum, Herbert Bos. Modern Operating Systems. 4th ed. New Jersey: Pearson Education, Inc., 2015, p.73.

EXERCISE

1. What are the functions of an operating system? Explain about microkernel.
2. What are pipe and shell? Describe the role of an operating system as a resource manager.
3. How operating system creates abstraction? Explain with reference to OS to an extended machine.
4. Define system call and explain its working mechanism with suitable example.
5. List the essential properties for the batch-oriented and interactive operating system.
6. Explain operating system as an extended machine? Distinguish between kernel and microkernel. Explain the purpose of system call.
7. What is the role of system call in an operating system?
8. Differentiate between monolithic and microkernel structure of OS?
9. Explain the virtual machine structure. What are the benefits over other OS structure?
10. Is layered structure of OS better than monolithic structure? If yes, explain with an example. If no, why?

Chap

2

2.1 In

Th

OS. All

systems s

as IBM's

concept o

A p

-
-
-
-

The

terminate

that each

coordinate

allocate sy

2.2 Pro

A *pr*

that is crea

process or

run a par

correspond

control blo

processes a

Since

elements, a

Chapter 2 Process Management

2.1 Introduction to Process

The *process* is the most fundamental concept in a modern OS. All multiprogramming operating systems, from single-user systems such as Windows for end users to mainframe systems such as IBM's mainframe operating system, z/OS are built around the concept of process.

A process is a

- a program in execution
- an instance of a program running on a computer
- the entity that can be assigned to and executed on a processor
- a unit of activity characterized by the execution of a sequence of instructions, a current state, and an associated set of system resources.[1]

The principal function of the OS is to create, manage, and terminate processes. While processes are active, the OS must see that each is allocated time for execution by the processor, coordinate their activities, manage conflicting demands, and allocate system resources to processes.

2.2 Process Control Block (PCB)

A *process control block (PCB)* is a data structure (a table) that is created and managed by the OS for every process. Every process or program that runs needs a PCB. When a user requests to run a particular program, the operating system constructs a corresponding process control block for that program. The process control block is the key tool that enables the OS to support multiple processes and to provide multiprocessing.

Since a process is uniquely characterized by the following elements, a process control block contains all these elements:

- **Identifier:** to distinguish it from other processes
- **State:** for defining the state of the process (for example, the process is currently executing, it is in the running state)
- **Priority:** shows the priority level relative to other processes
- **Program counter:** contains the address of the next instruction in the program to be executed.
- **Memory pointers:** includes pointers to the program code and data associated with this process, plus any memory blocks shared with other processes.
- **Context data:** are data present in the registers in the processor while the process is executing.
- **I/O status information:** contains outstanding I/O requests I/O devices assigned to the process, a list of files in use by the process, etc.
- **Accounting information:** processor time, clock time used account numbers, etc..

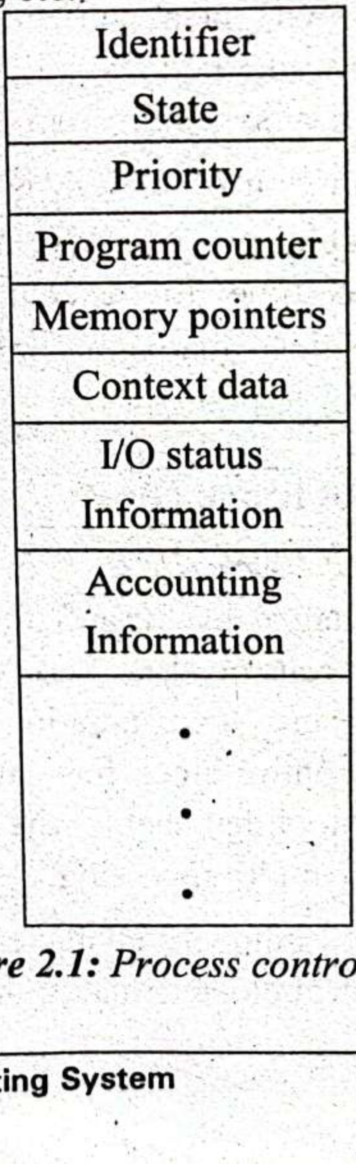


***Figure 2.1:** Process control block*

2.3 The Cr

The lif
termination.

Process Crea

Process

1. **System**
typicall
foregro
perform
that ru
particul
2. **Execut**
proces
3. **A user**
In inter
comma
start a r
4. **Initiati**
When
remotel
process

When
another proce
When one pr
parent proces
process.

Process Terr

Process
conditions:

1. **Norma**
of their
2. **Error**
process
illegal
dividin

2.3 The Creation and Termination of Processes

The life of a process is bounded by its *creation* and *termination*.

Process Creation

Processes are created by any of the following events:

1. **System initialization:** When an operating system is booted, typically numerous processes are created. These may be foreground processes (processes that interact with users and perform work for them) or background processes (processes that run in the background and are not associated with particular users, but instead have some specific function).
2. **Execution of a process-creation system call by a running process.**
3. **A user requests to create a new process**

   In interactive systems, a user can start a program by typing a command or clicking on an icon. Either of these actions will start a new process and run the selected program in it.
4. **Initiation of a batch job**

   When users submit batch jobs to the system (possibly remotely), as in the case of inventory management, processes are created in response to the submission of jobs.

When the OS creates a process at the explicit request of another process, the action is referred to as *process spawning*. When one process spawns another, the former is referred to as the *parent process*, and the spawned process is referred to as the *child process*.

Process Termination

Processes terminate usually due to one of the following conditions:

1. **Normal exit:** Most processes terminate after the completion of their task.
2. **Error exit:** One of the reasons of the termination of a process is an *error* caused by the process (executing an illegal instruction, referencing non-existent memory, or dividing by zero).

3. **Fatal error:** Another reason for the termination of a process is the discovery of a *fatal error*.

For example, if a user types the command

```
cc foo.c
```

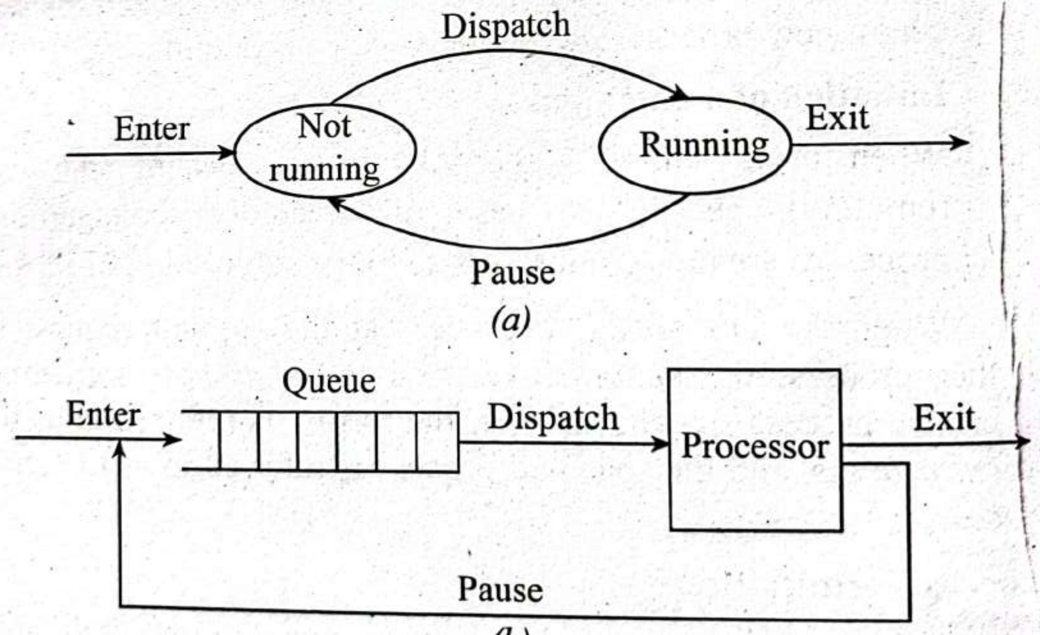
to compile the program *foo.c* and no such file exists, the compiler simply announces this fact and exits. Screen-oriented interactive processes, when given bad parameters, however, pop up a dialog box and ask the user to try again.

4. **Killed by another process:** A process might terminate if the process executes a system call telling the operating system to kill some other process.

2.4 Process States

From creation to completion, the process passes through various states. There are different models to explain this:

2.4.1 Two-State Process Model



(a)

(b)

Figure 2.2: *Two-state process model (a) State transition diagram (b) Queuing diagram*

In this model, a process may be in one of two states: *running* or *not running*. When the OS creates a new process, it creates a process control block for that process and enters that process into the system in the "not running" state. Now that process waits to get

executed by
will be inte
of CPU to
OS will se
moves from
of the other

As
process th
processes.
system. In
from the q

2.4.2 Fi

If a
queuing d
Some pro
while oth
Thus, usi
process a
have to s
that has b

Th
running"
inclusion
obtain a
briefly d

1. R

2. R
op

3. B
e

4. N
b
T
n

executed by OS. From time to time, the currently running process will be interrupted and the dispatcher (a program that gives control of CPU to the process selected by CPU scheduler) portion of the OS will select some other process to run. The former process moves from the "running" state to the "not running" state, and one of the other processes moves to the "running" state.

As shown in the queueing diagram (Figure 2.2 (b)), a process that is interrupted is transferred to the queue of waiting processes. If the process has completed or aborted, it exits the system. In either scenario, the dispatcher takes another process from the queue to execute.

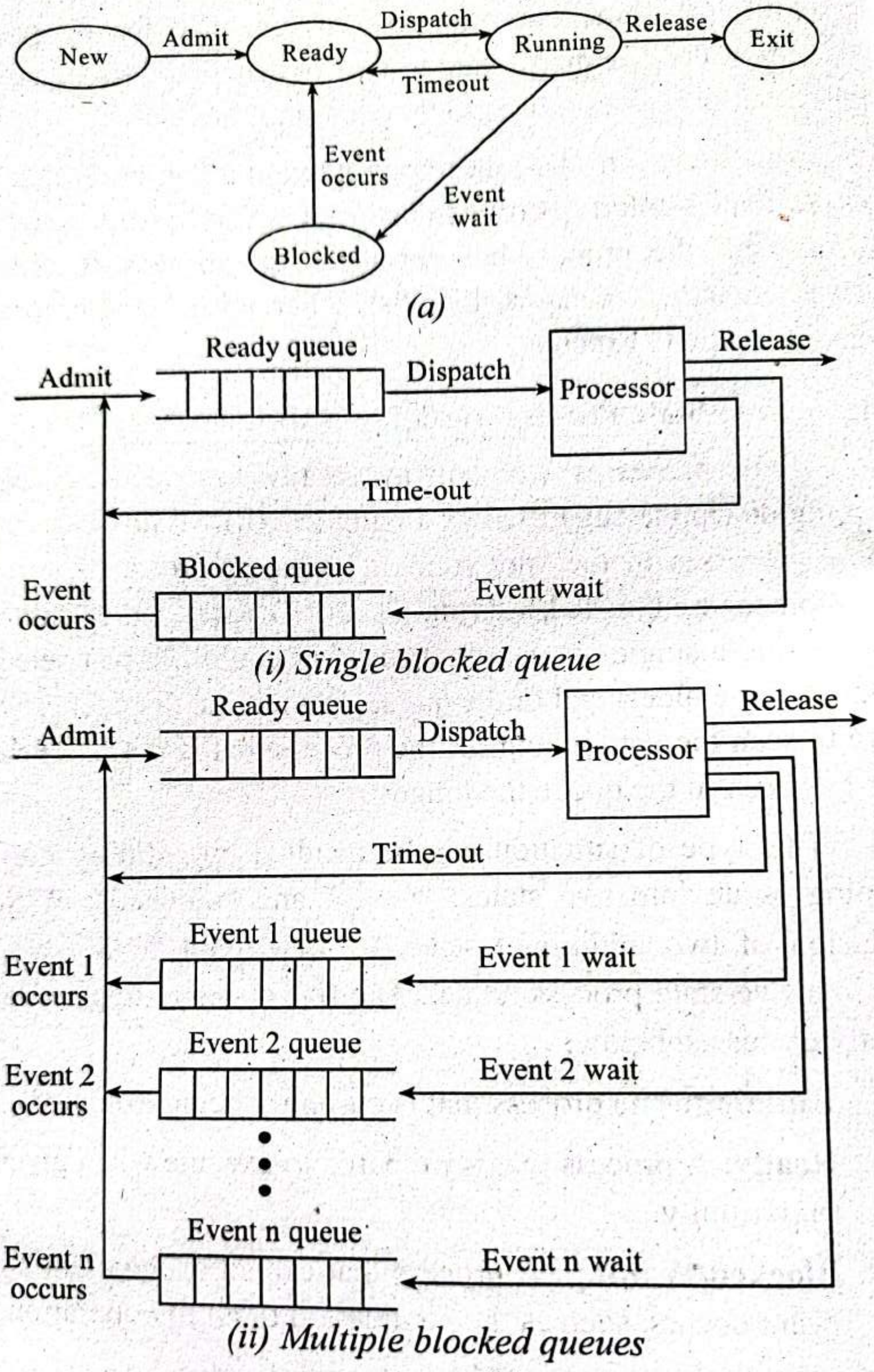
2.4.2 Five-State Process Model

If all processes were always ready to execute, then the queuing discipline suggested by Figure 2.2 (b) would be effective. Some processes in the "not running" state are ready to execute, while others are blocked, waiting for an I/O operation to complete. Thus, using a single queue, the dispatcher could not just select the process at the oldest end of the queue. Rather, the dispatcher would have to scan the list looking for the process that is not blocked and that has been in the queue the longest.

This type of situation can be handled by splitting the "not running" state into two states: "ready" and "blocked". With the inclusion of two additional states – "new" and "exit", we will obtain a five-state process model. The five states in this model are briefly discussed below:

1. **Running:** The process that is currently being executed.
2. **Ready:** A process that is prepared to execute when given the opportunity.
3. **Blocked/Waiting:** A process that cannot execute until some event occurs, such as the completion of an I/O operation.
4. **New:** A process that has just been created but has not yet been admitted to the pool of executable processes by the OS. Typically, a new process has not yet been loaded into main memory, although its process control block has been created.

5. **Exit:** A process that has been released from the pool of executable processes by the OS, either because it halted or because it aborted for some reason.



(a)

(i) Single blocked queue

(ii) Multiple blocked queues

(b)

Figure 2.3: *Five-state process model (a) State transition diagram (b) Queuing diagram: (i) single blocked queue (ii) multiple blocked queues.*

Compariso

Parameter
Definition
Nature
Lifespan

2.5 Thr

A p
called a *th*
is a single
not a prog
registers,
address s
(files, I/O

Compar

1. Proce same
2. Proce consu
3. Proc slow
4. It term

Comparison Between Process and Program

Parameter	Process	Program
Definition	Process is an instance of an executing program.	Program contains a set of instructions designed to complete a specific task.
Nature	Process is a active entity as it is created during execution and loaded into the main memory. .	Program is a passive entity as it resides in the secondary memory.
Lifespan	The process has a shorter and very limited lifespan as it gets terminated after the completion of the task.	A program has a longer lifespan as it is stored in the memory until it is not manually deleted.

2.5 Threads

A process is divided into smaller tasks and each task is called a *thread*. A *thread* (sometimes called a *lightweight process*) is a single sequential execution stream within a process. It itself is not a program because it cannot run its own. A thread has its own registers, program counter, stack, stack pointer. A thread shares address space, program code, global variables, heap, OS resources (files, I/O devices) with other threads.

Comparison between Process and Threads

Process	Thread
1. Processes cannot share the same memory.	1. Threads can share memory and files.
2. Process creation is time consuming.	2. Thread creation is not time consuming.
3. Process execution is very slow.	3. Thread execution is very fast.
4. It takes more time to terminate a process.	4. It takes less time to terminate threads.

Process	Thread
5. It takes more time to switch between two processes.	5. It takes less time to switch between two threads.
6. Process is loosely coupled i.e., lesser resources sharing.	6. Threads are tightly coupled i.e., more resources sharing.
7. Communication between processes is difficult.	7. Communication between threads is easy and efficient.
8. System calls are required for communication.	8. System calls are not required.
9. It is a heavy weight process as it requires more resources.	9. It is a light weight process as it requires fewer resources.
10.They are not suitable for exploiting parallelism.	10.They are suitable for exploiting parallelism.

2.5.1 Multithreading

Multithreading refers to the ability of an OS to support multiple, concurrent paths of execution within a single process.

In other words, multithreading is the use of multiple threads in a single program, all running at the same time and performing different task. Most software applications that run on modern computers are multithreaded. For example, JAVA browser is a multithreaded application wherein we can scroll a page while it is downloading an applet or image, play animation and sound parallelly, print a page in the background while we download a new page.

More examples of use of multithreading:

1. A Web server accepts client requests for web pages, images, sound, and so forth. A busy web server may have several (perhaps thousands of) clients concurrently accessing it. If the web server ran as a traditional single-threaded process, it would be able to service only one client at a time, and a client might have to wait a very long time for its request to be serviced. So, a web server uses multiple threads.

2. A word processor uses multiple threads: a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.

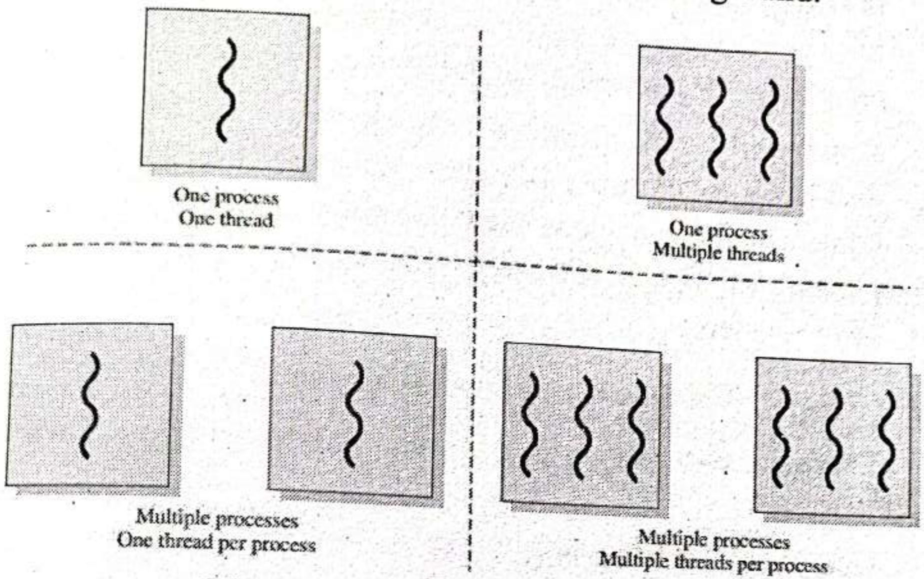


Figure 2.4: Threads and processes

MS-DOS uses one process, one thread approach. JAVA Runtime uses one process, multiple threads approach. UNIX uses multiple processes, one thread per process approach. W2K, Solaris, and Linux use multiple processes, multiple threads per process approach.

Advantages of multithreading:

1. **Responsiveness:** If the process is divided into multiple threads, if one thread completes its execution, then its output can be immediately returned.
2. **Faster context switch:** Context switch time between threads is lower compared to process context switch. Process context switching requires more overhead from the CPU.
3. **Effective utilization of a multiprocessor system:** If we have multiple threads in a single process, then we can schedule multiple threads on multiple processors. This will make process execution faster.
4. **Resource sharing:** Resources like code, data, and files can be shared among all threads within a process.

Note: Stack and registers can't be shared among the threads. Each thread has its own stack and registers.

5. **Communication:** Communication between multiple threads is easier, as the threads share common address space. While in process we have to follow some specific communication technique for communication between two processes.

6. **Enhanced throughput of the system:** If a process is divided into multiple threads, and each thread function is considered as one job, then the number of jobs completed per unit of time is increased, thus increasing the throughput of the system.

2.5.2 User-Level and Kernel-Level Threads

There are two categories of thread implementation. They are as follows:

1. **User-Level Threads (ULTs)**

User-level threads are threads implemented in the user space and the kernel doesn't know anything about them. When threads are managed in the user space, each process must have its own private thread table. This private table consists of the information of the program counter, stack pointer, registers, etc. and is managed by the run-time system. This is shown in figure.

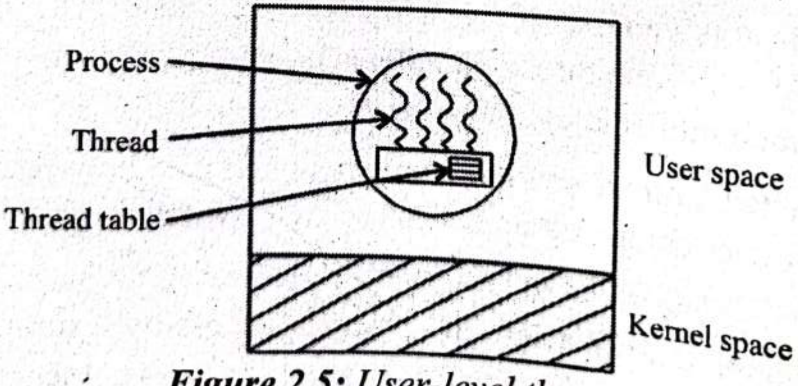


Figure 2.5: User-level thread

Advantage of user-level threads:

- User-level threads are very efficient because a mode switch is not required to switch from one thread to another.

Disadvantage of user-level threads:

- Only a single user-level thread within a process can execute at a time, and if one thread blocks, the entire process is blocked.

2. Kernel-Level Threads (KLTs)

Kernel-level threads are threads within a process that are maintained by the kernel. There is no thread table in each process. The kernel has a thread table. This table keeps track of all the threads in a system.

The kernel's thread table holds each thread registers, state and other information: Note that the information here is the same as with the user-level threads but it is now in the kernel instead of user space. This is shown in Figure 2.6.

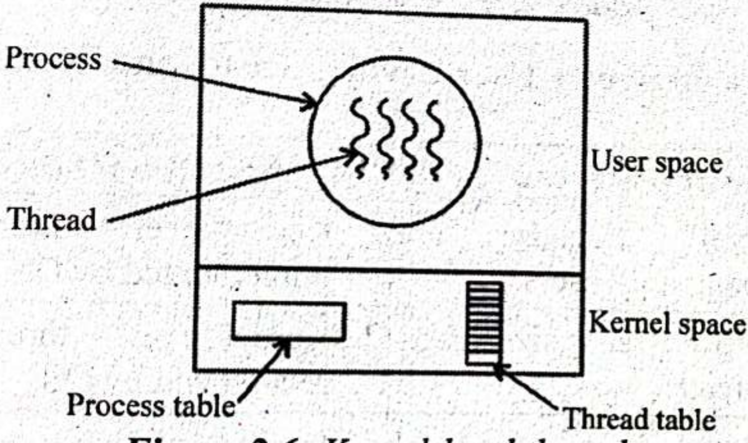


Figure 2.6: Kernel-level thread

Advantages of kernel-level threads:

- The blocking of a thread does not block the entire process.
- It supports multiprocessing because multiple threads within the same process can execute in parallel on a multiprocessor.

Disadvantage of kernel-level threads:

- Switching between threads is time consuming because the kernel must do the switching.

2.5.3 Combined/ Hybrid Approach

In this combined/hybrid approach, we combine both the user-level and kernel-level approaches. In such a system, thread creation is done completely in the user space. Some OS provide this facility. For example, Solaris 2 (a UNIX version) is an example of such an OS. It is shown in Figure 2.7.

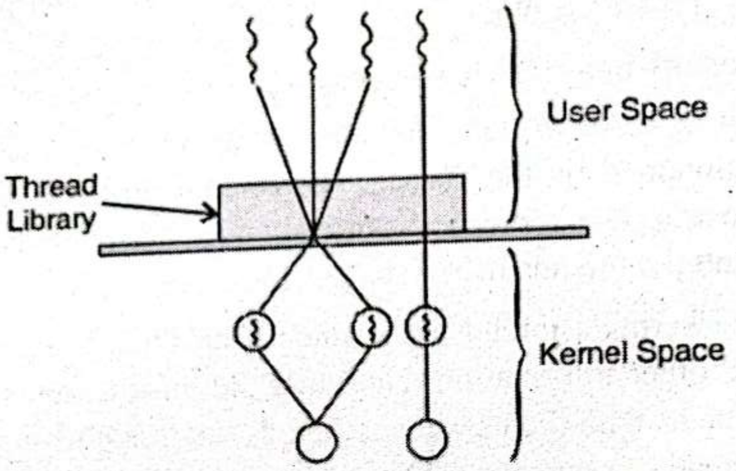


Figure 2.7: *A combined approach*

Difference Between User-Level Thread and Kernel-Level Thread

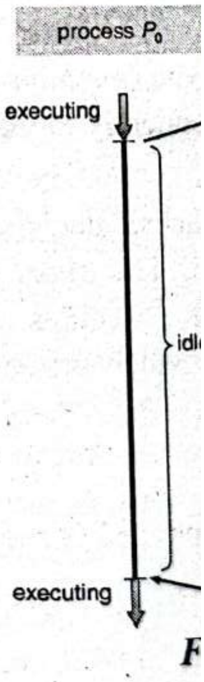
User-level thread	Kernel-level thread
1. User-level threads are implemented by users.	1. Kernel-level threads are implemented by OS.
2. OS doesn't recognize user-level threads.	2. Kernel-level threads are recognized by OS.
3. Implementation of user-level threads is easy.	3. Implementation of kernel-level thread is complicated.
4. Context switch time is less.	4. Context switch time is more.
5. Context switch requires no hardware support.	5. Hardware support is needed.
6. If one user-level thread performs a blocking operation, then the entire process will be blocked.	6. If one kernel-level thread performs a blocking operation, then another thread can continue execution.
7. User-level threads are designed as dependent threads. *Example:* Java threads, POSIX threads.	7. Kernel-level threads are designed as independent threads. *Example*: Solaris threads

2.6 Context Switcl

The *context swi*
operating system to sv
execute its function us
to another process req
process and a state res
time is pure overhe
work while switchi
machine. Furtherm
dependent on hardwar

The steps involved in

1. Save the conter
the CPU. Updat
2. Move the PCB
such as the "rea
3. Select a new pı
selected process
(This includes ı
4. Update the mer
5. Restore the con
when it is loac
loading the prev



2.6 Context Switching

The *context switching* is a technique or method used by the operating system to switch a process from one state to another to execute its function using CPUs in the system. Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. Context-switch time is pure overhead, because the system does no useful work while switching. Its speed varies from machine to machine. Furthermore, context-switch times are highly dependent on hardware support.

The steps involved in context switching are:

1. Save the content of the process that is currently running on the CPU. Update the PCB and other important fields.
2. Move the PCB of the above process into the relevant queue such as the "ready" queue, I/O queue etc.
3. Select a new process for execution. Update the PCB of the selected process.

   (This includes updating the process state to "running")
4. Update the memory management data structures as required.
5. Restore the context of the process that was previously running when it is loaded again on the processor. This is done by loading the previous values of the PCB and registers.

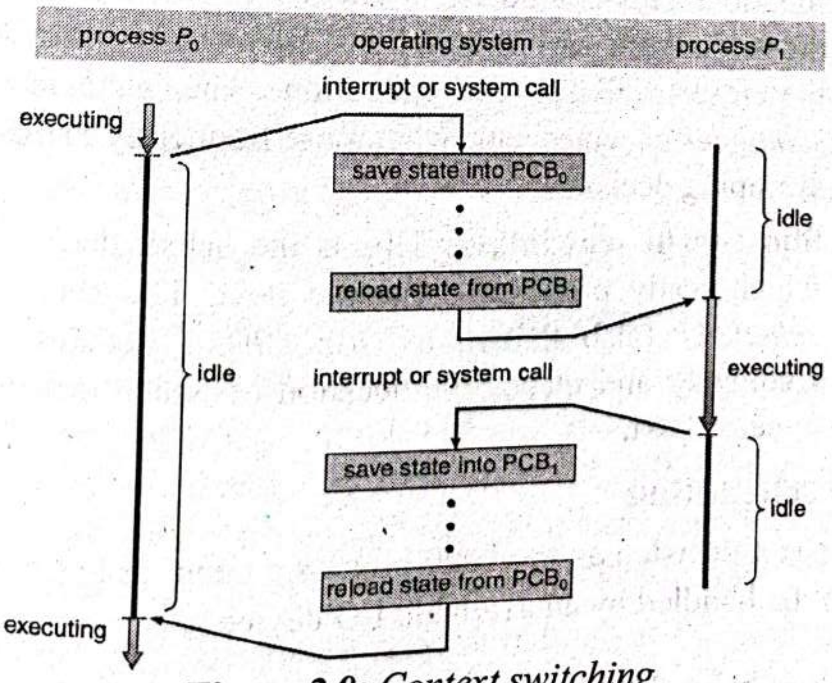


Figure 2.9: Context switching

2.7 Scheduling

The key to multiprogramming is *scheduling*. Scheduling refers to the set of policies and mechanisms that an OS supports for determining the order of execution of the pending jobs and processes. A *scheduler* is an operating system program (module) that selects the next job to be admitted for execution. The main objective of scheduling is to increase CPU utilization and to increase the throughput.

2.7.1 Types of Scheduling

Scheduling can be categorized into two categories:

1. **Processor scheduling**

   The aim of *processor scheduling* is to assign processes to be executed by the processor or processors over time.

   i. **Long-term scheduling:** This is a decision whether to add a new process to the set of processes that are currently active. The *long-term scheduler* executes relatively infrequently and makes the decision of whether or not to take on a new process and which one to take.

   ii. **Medium-term scheduling:** It is part of the swapping function. This is a decision whether to add a process to those that are at least partially in main memory and therefore available for execution. The *medium-term scheduler* executes somewhat more frequently to make a swapping decision.

   iii. **Short-term scheduling:** This is the actual decision of which ready process to execute next. The *short-term scheduler* (also known as *dispatcher*) executes most frequently and makes the decision of which process to execute next.

2. **I/O scheduling**

   This is a decision as to which process's pending I/O request shall be handled by an available I/O device.

2.7.2 Schedu

A schedu
accepted perfor
five criteria:

1. **CPU utili**
during wl
or system
all the tin
of the sys
better it is

2. **Through**
completed

3. **Turnarou**
elapsed f
created) t
the sum o
in the rea
write that

Turnarou
arrival tir

Lower is

4. **Waiting**
the job (
ready sta
be given

Waiting t

Lower th

5. **Respons**
for two
system.
systems.
interval f

In real-ti
the time

2.7.2 Scheduling Criteria

A scheduler algorithm is evaluated against some widely accepted performance criteria. We need to consider the following five criteria:

1. **CPU utilization:** It is defined as the average fraction of time during which CPU is busy, executing either user programs or system modules. The key idea is that if the CPU is busy all the time then the utilization factor of all the components of the system will also be high. Higher the CPU utilization, better it is.

2. **Throughput:** It is defined as the average amount of work completed per unit time. Higher is the throughput, better it is.

3. **Turnaround time (TAT):** It is defined as the total time elapsed from the time the job is submitted (or process is created) to the time the job (or process) is completed. It is the sum of periods spent waiting to get into memory, waiting in the ready queue, CPU time and I/O operations. So, we can write that

   Turnaround time (TAT) = (Process finish time – Process arrival time)

   Lower is the average turnaround time, better it is.

4. **Waiting time (WT):** It is defined as the total time spent by the job (or process) while waiting in suspended state or ready state, in a multiprogramming environment. So it may be given by a formula

   Waiting time (WT) = (Turnaround time – Processing time)

   Lower the average waiting time, the better it is.

5. **Response time (RT):** This parameter is usually considered for two systems - time sharing and real-time operating system. However, its characteristics differ in these two systems. In the time-sharing system, it may be defined as the interval from the time the last result appears on the terminal.

   In real-time systems, it may be defined as the interval from the time an internal or external event is signalled to the time the first instruction of the respective [illegible] routine (ISR) is executed.

6. **Fairness:** A good scheduler should make sure that each process gets its fair share of the CPU.

2.7.3 CPU Scheduling Algorithms

Scheduling algorithms can be divided into two categories with respect to how they deal with clock interrupts: *nonpreemptive scheduling algorithm* and *preemptive scheduling algorithm.*

The scheduling in which a running process cannot be interrupted by any other process is called non-preemptive scheduling. Any other process which enters the queue has to wait until the current process finishes its CPU cycle.

In contrast, the scheduling in which a running process can be interrupted if a high priority process enters the queue and is allocated to the CPU is called preemptive scheduling.

Comparison between Non Preemptive and Preemptive Scheduling

Non-preemptive scheduling	Preemptive scheduling
1. In nonpreemptive scheduling, if once a process has been allocated CPU, then the CPU cannot be taken away from that process.	1. In preemptive scheduling, the CPU can be taken away before the completion of the process.
2. No preference is given when a higher priority job comes.	2. It is useful when a higher priority job comes as here the CPU can be snatched from a lower priority process.
3. The treatment of all processes is fairer.	3. The treatment of all processes is not fairer as CPU snatching is done either due to time constraints or due to a higher priority process request for its execution.

Non-preemptive scheduling	Preemptive scheduling
4. Process cannot be interrupted till it terminates or switches to waiting state.	4. Process can be interrupted in between.
5. It is a cheaper scheduling method. **Example:** First-Come-First-Served (FCFS), Shortest Job First(SJF) CPU scheduling algorithm	5. It is a costlier scheduling method. **Example:** Round-Robin, Shortest Remaining Time First (SRTF) CPU scheduling algorithm

Following are the different types of scheduling algorithms:

1. First-Come-First-Served (FCFS)

It is the simplest of all the scheduling algorithms, also known as *first-in-first-out (FIFO)* or a *strict queueing scheme*. With this scheme, the process that requests the CPU first is allocated the CPU first. Its implementation involves a ready queue that works in first-in-first-out order. When the CPU is free, it is assigned to a process which is in front of the ready queue. A FCFS scheduling algorithm is *nonpreemptive* because the CPU has been allocated to a process that keeps the CPU busy until it is released.

Advantages:

- It is the simplest algorithm and is easy to implement.
- It is suitable for batch systems.

Disadvantages:

- The average waiting time is not minimal.
- It is not suitable for time sharing systems like UNIX.

2. Shortest Job First (SJF)

In this scheduling algorithm, the CPU is assigned the process with least CPU-burst time. The processes are available in the ready queue. When two processes have the same CPU-burst time, FCFS is used to for decision. This algorithm can be either preemptive or nonpreemptive. A preemptive SJF algorithm will preempt the currently executing, where as a non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is also known Shortest-Remaining-time (SRT) First Scheduling.

Advantages:

- Shortest jobs are favored.
- It is an optimal algorithm because it gives minimum average waiting time.

Disadvantages:

- It is difficult to know the length of the next CPU-burst time.
- It cannot be implemented at the level of short-term CPU scheduling time.
- There is a risk of starvation of longer processes.
- It is not desirable for time-sharing or transaction processing environment because of the lack of preemption.

3. Shortest Remaining Time First (SRTF)

Shortest remaining time first (SRTF) is a preemptive version of SJF. In this case, the scheduler always chooses the process that has the shortest remaining processing time. When a new process joins the ready queue, the scheduler compares the remaining time of executing process and new process. If the new process has the least CPU-burst time, the scheduler selects that job and allocates CPU, otherwise, it continues with the old process.

Advantages:

- SRTF does not have the bias in favor of long processes found in FCFS.
- Unlike round robin, no additional interrupts are generated, reducing overhead.

Disadvantages:

- There is a risk of starvation of longer processes.

4. Roun

Round
algorit
design
FCFS
system
called
quantu
The re
schedu
to each

The pe
upon:

i. Si
ii. Nu

Advar

- Ro
un
- It c

Disad

- Th
- Th
- Lo
sw

5. **Highe**

This al
respons
turnarc
time an

Mather

R = w

where
process

4. **Round Robin**

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

The performance of this pure preemptive algorithm depends upon:

i. Size of time quantum

ii. Number of context switches

Advantages:

- Round robin algorithm utilizes the system resources uniformly.
- It doesn't face the issues of starvation or convoy effect.

Disadvantages:

- The average waiting time is not minimal.
- This method spends more time on context switching
- Lower time quantum results in higher the context switching overhead in the system.

5. **Highest Response Ratio Next (HRRN)**

This algorithm executes that job first which has the highest response ratio. *Response ratio*, also known as *normalized turnaround time* is defined as the ratio between turnaround time and response time.

Mathematically,

$$R = \frac{w + s}{s}$$

where R = response ratio, w = time spent waiting for the processor, s = expected service time.

This approach is attractive because it accounts for the age of the process.

6. **Priority Scheduling**

In this scheduling algorithm, each process is given a priority. The scheduler always picks up the highest priority process for its execution from the ready queue. If the processes have equal priority, then FCFS is used to make decision.

This type of scheduling algorithm may be of two types:

1. **Preemptive (priority-based) scheduling**

   In preemptive scheduling, the tasks are mostly assigned with their priorities. Sometimes it is important to run a task with a higher priority before another lower priority task, even if the lower priority task is still running. The lower priority task holds for some time and resumes when the higher priority task finishes its execution.

2. **Nonpreemptive scheduling**

   In nonpreemptive scheduling, the CPU has been allocated to a specific process. The process that keeps the CPU busy, will release the CPU either by switching context or terminating. It is the only method that can be used for various hardware platforms.

Advantages of priority scheduling:

- Processes are executed on the basis of priority so high priority does not need to wait for long which saves time.
- Suitable for applications with fluctuating time and resource requirements.

Disadvantages of priority scheduling:

- If the system eventually crashes, all low priority processes get lost.
- This scheduling algorithm may leave some low priority processes waiting indefinitely.

7. **Rate Monotonic Algorithm (RMA)**

- RMA is static-priority preemptive scheme where all priorities are determined before runtime.

- The priority of task is the length of period of respective task where task with short period are highest priority.
- Used to schedule periodic tasks:
  1. Deadlines for task are end of each period; D = T.
  2. The tasks are independent and do not block each other.
  3. Scheduling overhead due to context switches and swapping etc. are assumed to be zero.

Example:

	Capacity	**Period (T)**
P1	**3**	**20**
P2	**2**	**5**
P3	**2**	**10**

Here, the capacity denotes unit of execution of each process for given period i.e.,

P1 executes 3 units in each 20 units time,

P2 executes 2 units in each 5 units time,

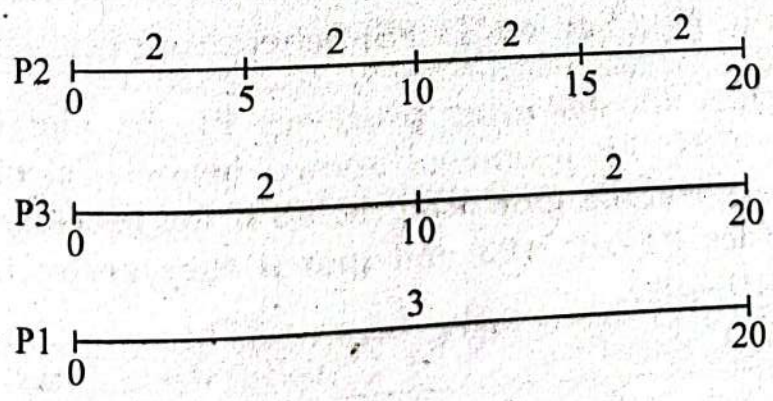
P3 executes 2 units in each 10 unit time.

Process with least period has highest priority.

i.e. P2 > P3 > P1

Total time required:

LCM of 20, 5, 10 = 20



Gantt chart:

P2	P2	P3	P3	P1	P2	P2	P1	P1		P2	P2	P3	P3		P2	P2			

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

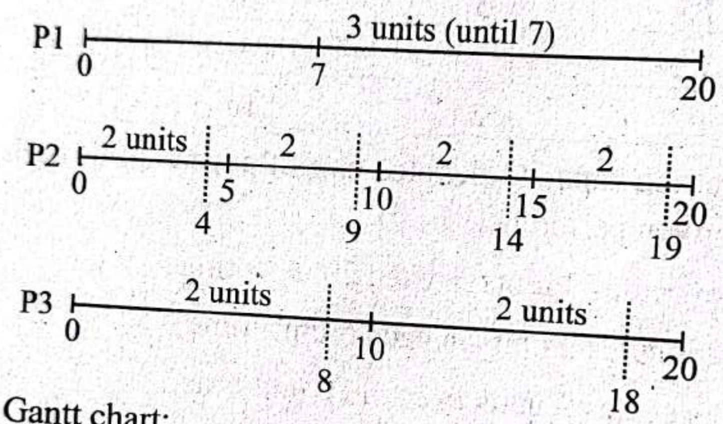
8. Earliest Deadline First (EDF)

- The tasks are assigned priority based on deadline.
- The deadline (earliest) has highest priority.
- EDF is capable of achieving full processor utilization

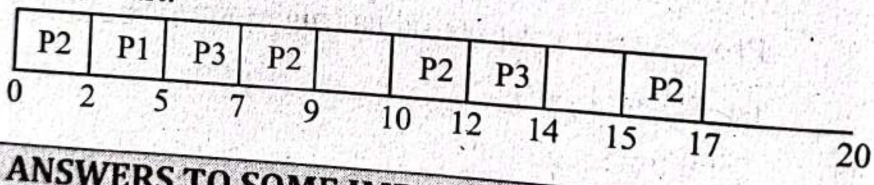
Example:

	Capacity	Deadline	Period
P1	**3**	**7**	**20**
P2	**2**	**4**	**5**
P3	**2**	**8**	**10**

P2 > P1 > P3 (Priority)



Gantt chart:



ANSWERS TO SOME IMPORTANT QUESTIONS

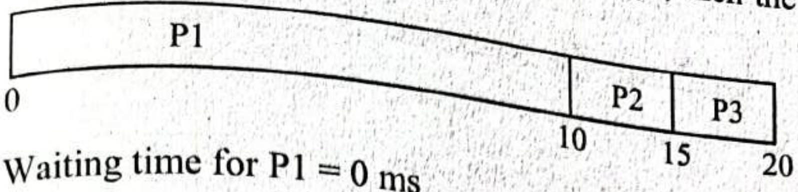
A. First in First Served (FCFS) Scheduling Problems

1. **Consider the following processes P1, P2 and P3 with their CPU burst-time as shown below. Calculate the average waiting time using FCFS if the processes arrive in order P1→P2→P3 and that if they arrive in order P2→P3→P1.**

Process	CPU Burst Time (ms)
P1	10
P2	5
P3	5

Solution:

If the processes arrive in order P1→P2→P3, then the Gantt chart is as shown below.



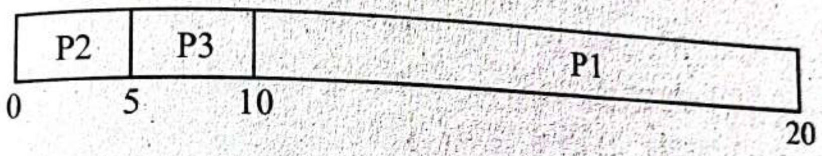
Waiting time for P1 = 0 ms

Waiting time for P2 = 10 ms

Waiting time for P3 = 15 ms

$$\therefore \text{Average waiting time (AWT)} = \frac{0+10+15}{3} = 8.33 \text{ ms}$$

If the processes arrive in order P2→P3→P1, then the Gantt chart is as shown below.



Waiting time for P2 = 0 ms

Waiting time for P3 = 5 ms

Waiting time for P1 = 10 ms

$$\therefore \text{Average waiting time (AWT)} = \frac{0+5+10}{3} = 5 \text{ ms}$$

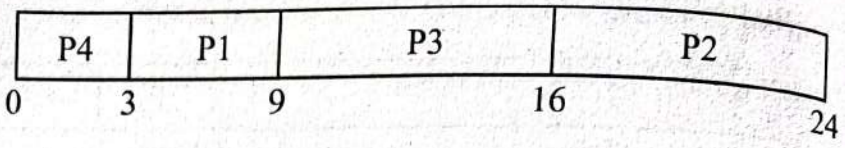
B. Shortest Job First (SJF) Scheduling Problems

2. **Consider four processes P1, P2, P3 and P4 with their CPU burst-times given below: Using non-preemptive SJF, calculate the AWT and ATAT.**

Process	CPU Burst Time (ms)
P1	6
P2	8
P3	7
P4	3

Solution:

Gantt chart is as shown below.



$$\text{Average waiting time (AWT)} = \frac{0 + 3 + 9 + 16}{4} = 7 \text{ ms}$$

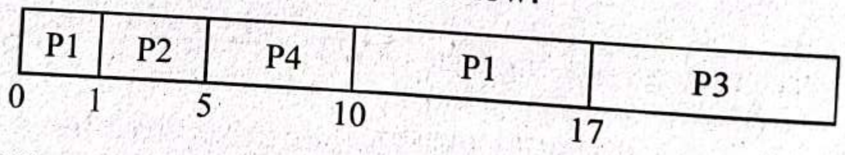
$$\text{Average turnaround time (ATAT)} = \frac{3 + 9 + 16 + 24}{4} = 13 \text{ ms}$$

3. **Consider four processes P1, P2, P3 and P4 with their CPU burst-times given below. Using preemptive SJF, calculate the AWT.**

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Solution:

The Gantt chart is as shown below.



When P2 arrives, the remaining time for P1 (i.e., 7 ms) is larger than the time required for P2 (4 ms), so P1 is preempted and P2 is executed. After completion of P2, all the processes have arrived, so the burst time is compared and maintained in Gantt chart

$$\text{AWT} = \frac{(10 - 1) + (1 - 1) + (17 - 2) + (5 - 3)}{4} = 6.5 \text{ ms}$$

C. Priority Scheduling Problems

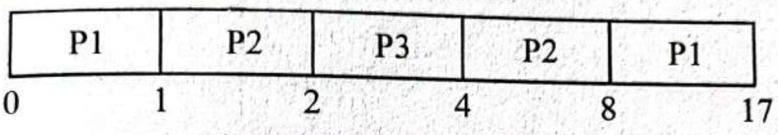
4. **Consider a set of 3 processes P1, P2 and P3 with their priorities and arrival times as given below. Calculate AWT for preemptive priority based algorithm.**

Process	Burst Time	Priority	Arrival Time
P1	10	3	0
P2	5	2	1
P3	2	1	2

where 1 denotes highest priority and 3 denotes lowest priority.

Solution:

The GANTT chart is as shown below.



$$AWT = \frac{0 + (8 - 1) + 1 + (4 - 2) + 2}{3} = 4 \text{ ms}$$

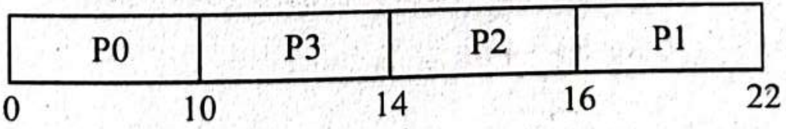
∴. Consider a set of 4 processes P0, P1, P2, P3. There arrival times and burst times are given below. Calculate AWT for non-preemptive priority based algorithm.

Process	AT	Burst Time	Priority
P0	0	10	5
P1	1	6	4
P2	3	2	2
P3	5	4	0

Where 0- highest priority, 5-lowest priority

Solution:

The Gantt chart is shown below:



Process	AT	Burst Time (BT)	Final Time (FT)	TAT=FT-AT	WT=TAT-BT
P0	0	10	10	10	0
P1	1	6	22	21	15
P2	3	2	16	13	11
P3	5	4	14	9	5

$$ATAT = \frac{10 + 21 + 13 + 9}{4} = 13.25 \text{ ms}$$

and

$$AWT = \frac{0 + 15 + 11 + 5}{4} = 7.75 \text{ ms}$$

D. Round Robin Scheduling Problem

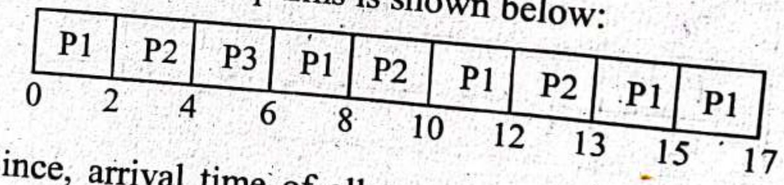
6. Consider a set of 3 processes P1, P2 and P3 with their CPU burst times in milliseconds.

Process	Burst Time
P1	10
P2	5
P3	2

Assume time quantum, q=2 ms. Find AWT for round-robin scheduling algorithm.

Solution:

Gantt chart with q=2ms is shown below:



Since, arrival time of all processes are same, we calculate waiting time as follows:

Waiting time for P1= 0 + (6 – 2) + (10 – 8) + (13 – 12)=7 ms

Waiting time for P2= 2 + (8 – 4)+(12 – 10)=8 ms

Waiting time for P3= 4 ms

$$\text{Now, } AWT = \frac{7 + 8 + 4}{3} = 6.33 \text{ ms}$$

E. Highest-Response Ration Next (HRN) Scheduling Problems

7. Consider the processes with following arrival time, burst time and priorities:

Process	Arrival time	Burst time	Priority
P1	0	7	3 (High)
P2	2	4	1 (Low)
P3	3	4	2

Solution:

At time 0 only process P1 is available, so P1 is considered for execution.

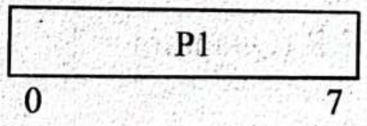
Since it is non-preemptive, it executes process P1 completely. It takes 7 ms to complete process P1 execution.

Now, among P2 and P3, the process with highest response ratio is chosen for execution.

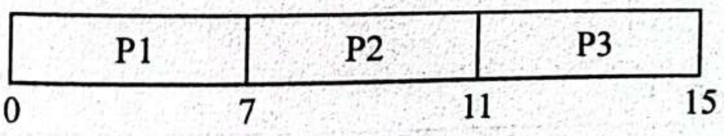
$$\text{Response ratio} = \frac{\text{waiting time + service time}}{\text{service time}}$$

$$\text{Response ratio for P2} = \frac{(7-2)+4}{4} = 2.25$$

$$\text{Response ratio for P3} = \frac{(7-3)+4}{4} = 2$$



As process P2 is having highest response ratio than that of P3, process P2 will be considered for execution and then followed by P3.



Average waiting time = [0 + (7–2)+(11–3)]/3 = 4.33

Average turnaround time = [7 +(11–2)+(15–3)]/3 = 9.33

8. a. Consider the following set of processes, with arrival time and the length of CPU burst time given in table as below:

Process	Arrival time	Burst time
A	**0**	**3**
B	**1**	**6**
C	**4**	**4**
D	**6**	**2**

i. **Draw Gantt chart illustrating the execution of these processes using FCFS, SRTN and RR (Q=2) scheduling.**

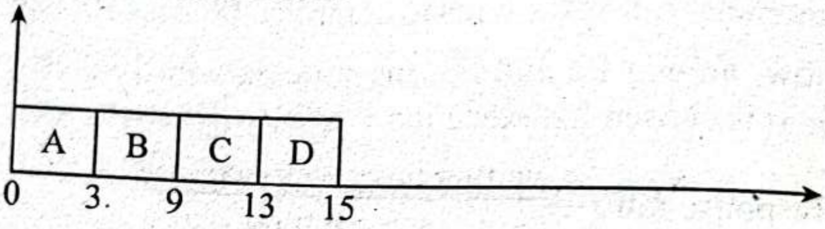
ii. **What is the waiting time and turnaround time of each process for each of the scheduling algorithm?**

[2076 Baishakh]

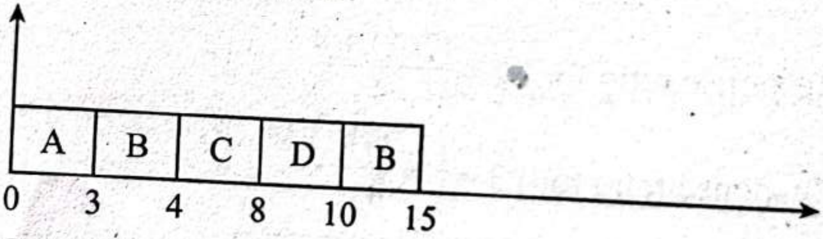
Solution:

i.

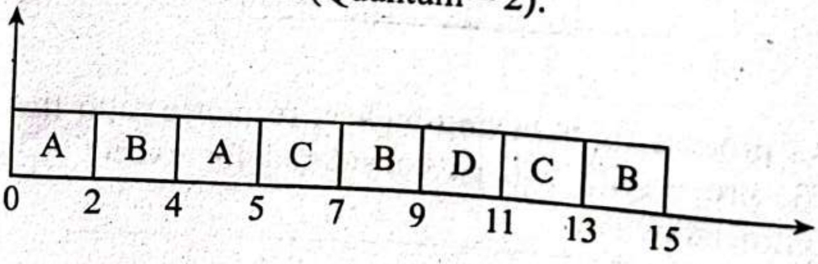
a. Gantt chart for FCFS:



b. Gantt chart for SRTN:



c. Gantt chart for RR (Quantum = 2):



ii.

a. For FCFS:

Process	AT	BT	FT	TAT=FT–AT	WT=TAT–BT
A	0	3	3	3	0
B	1	6	9	8	2
C	4	4	13	9	5
D	6	2	15	9	7
				Total = 29 ATAT=7.25	Total = 14 AWT=3.5

b.

c.

9. Cor

the

Dra

hig

ave

eac

b. For SRTN:

Process	AT	BT	FT	TAT (FT–AT)	WT (TAT–BT)
A	0	3	3	3	0
B	1	6	15	14	8
C	4	4	8	4	0
D	6	2	10	4	2
				Total = 25 ATAT = 6.25	Total = 10 AWT = 2.5

c. For RR:

Process	AT	BT	FT	TAT (FT–AT)	WT (TAT–BT)
A	0	3	5	5	2
B	1	6	15	14	8
C	4	4	13	9	5
D	6	2	11	5	3
				Total = 23 ATAT = 8.25	Total = 18 AWT = 4.5

9. **Consider the following set of process, with the length of the CPU burst time and arrival time given in millisecond.**

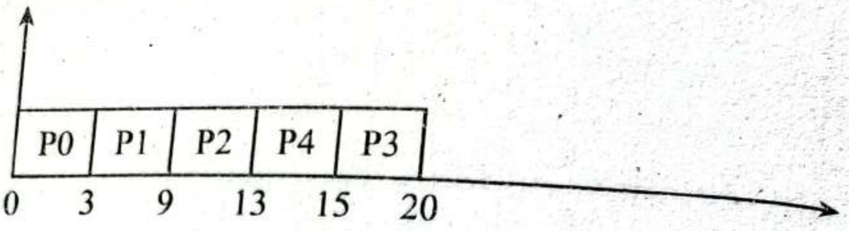
Process	Arrival time	Burst time
P0	0	3
P1	2	6
P2	4	4
P3	6	5
P4	8	2

Draw Gantt chart illustrating RR (Quantum = 2) and highest rank ratio next (HRRN) scheduling. Also find average waiting time and average turnaround time for each of the algorithm. [2077 Chaitra]

Solution:

For HRN algorithm:

Gantt chart:



Explanation: HRRN is non-preemptive algorithm. P0 arrive at t = 0, at time 3, P1 is only waiting, so P1 executes. At time 9, processes P2, P3 and P4 are waiting.

Now,

Response ratio (RR) at time 9 for:

Process P2 = $\frac{(9-4)+4}{4}$ = 2.25 (maximum)

Process P3 = $\frac{(9-6)+5}{5}$ = 1.6

Process P4 = $\frac{(9-8)+2}{2}$ = 1.5

Hence, P2 executes.

Similarly, at time 13, process P3 and P4 are waiting.

Now,

RR at 13 for:

Process P3 = $\frac{(13-6)+5}{5}$ = 2.4

Process P4 = $\frac{(13-8)+2}{2}$ = 3.5 (maximum)

Hence, P4 executes.

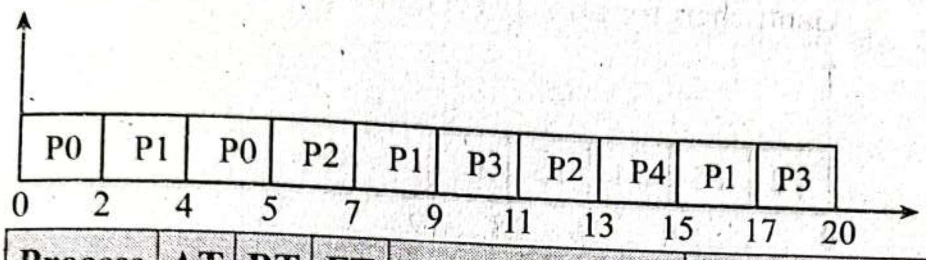
Process	AT	BT	FT	TAT (FT–AT)	WT (TAT–BT)
P0	0	3	3	3	0
P1	2	6	9	7	1
P2	4	4	13	9	5
P3	6	5	20	14	9
P4	8	2	15	7	5
				Total TAT = 40	Total WT = 20

$\therefore$ Average TAT $= \frac{40}{5} = 8$

$\therefore$ Average WT $= \frac{20}{5} = 4$

For round robin algorithm (Q = 2):

Gantt chart:



Process	AT	BT	FT	TAT (FT–AT)	WT (TAT–BT)
P0	0	3	5	5	2
P1	2	6	17	15	9
P2	4	4	13	9	5
P3	6	5	20	14	9
P4	8	2	15	7	5
				Total TAT = 50	Total WT = 30

$\therefore$ Average TAT $= \frac{50}{5} = 10$

$\therefore$ Average WT $= \frac{30}{5} = 6$

10. Consider the following set of processes, with the length of CPU burst time in millisecond. The processes are assumed to have arrived in the order P1, P2, P3, P4, P5 all at time 0 [lowest number being highest priority].

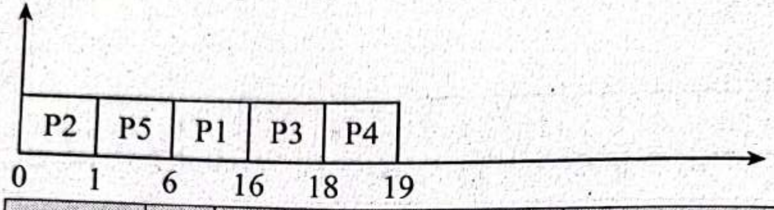
Process	Burst time	Priority	AT
P1	10	3	0
P2	1	1	0
P3	2	4	0
P4	1	5	0
P5	5	2	0

Draw Gantt chart illustrating priority and RR (quantum = 1) scheduling. Also find average waiting time and average turn-around time for each of the algorithms.

[2075 Bhadra]

Solution:

Gantt chart for priority scheduling:

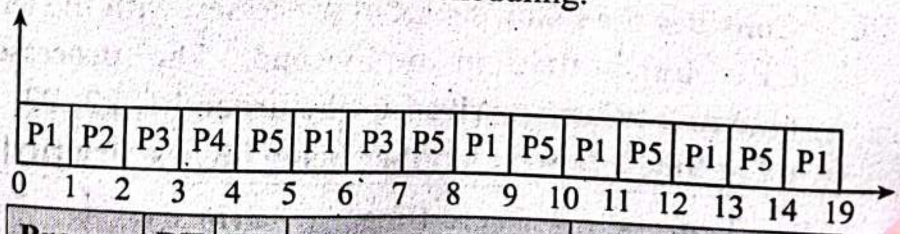


Process	BT	FT	TAT (FT–AT)	WT
P1	10	16	16	6
P2	1	1	1	0
P3	2	18	18	16
P4	1	19	19	18
P5	5	6	6	1
			Total TAT = 60	Total WT = 41

$\therefore$ Average TAT $= \frac{60}{5} = 12$

$\therefore$ Average WT $= \frac{41}{5} = 8.2$

Gantt chart for round-robin scheduling:



Process	BT	FT	TAT	WT (TAT–BT)
P1	10	19	19	9
P2	1	2	2	1
P3	2	7	7	5
P4	1	4	4	3
P5	5	14	14	9
			Total TAT = 46	Total WT = 27

$\therefore$ Average TAT $= \frac{46}{5} = 9.2$

$\therefore$ Average WT $= \frac{27}{5} = 5.4$

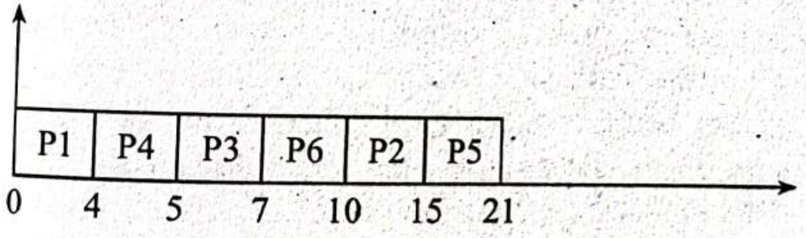
11. Let us consider following processes with given arrival time and length of the CPU burst given in milliseconds.

Process	Arrival time	Burst time
P1	0	4
P2	1	5
P3	2	2
P4	3	1
P5	4	6
P6	6	3

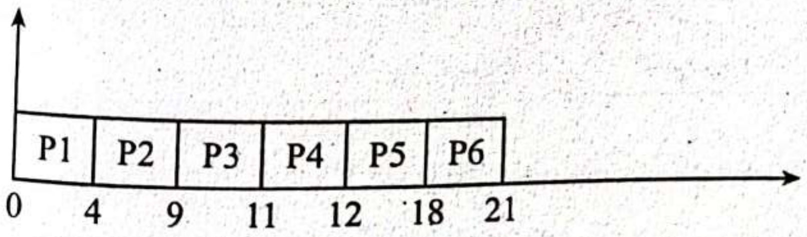
a. Draw the Gantt chart showing the execution for FCFS, SRTN and RR (Quantum = 2) [2078 Baishakh]

Solution:

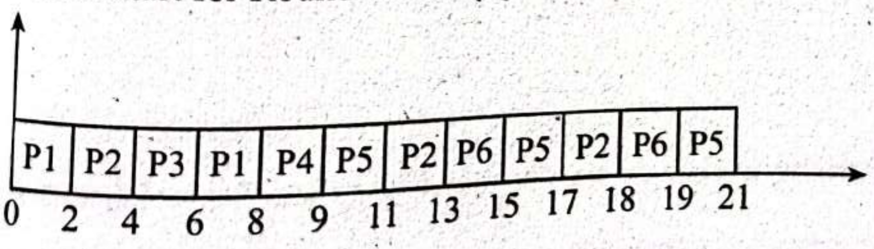
Gantt chart for SRTN:



Gantt chart for FCFS:



Gantt chart for Round Robin (Quantum = 2):



EXERCISE

1. Describe the five-state process model.
2. Differentiate between:
   i. Process and thread
   ii. User-level thread and kernel-level thread.
   iii. Preemptive and non-preemptive scheduling
3. Explain multithreading technique.

Chapter 3

Process Communication And Synchronization

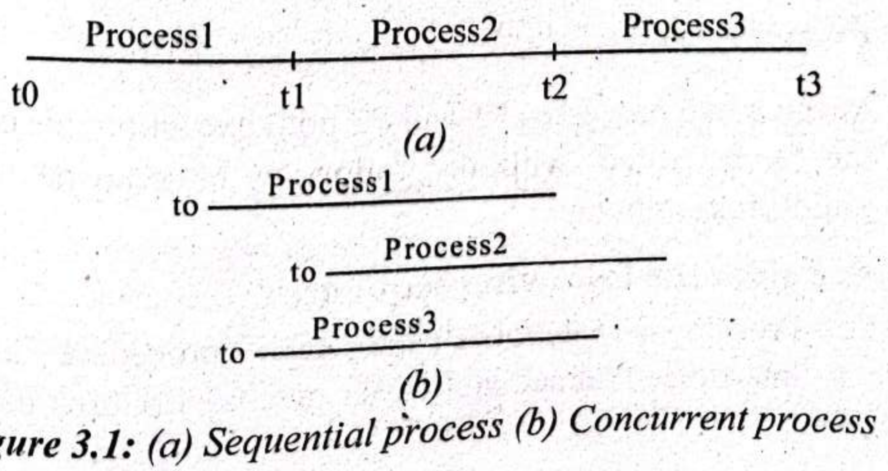
3.1 Principles of Concurrency

Before understanding what concurrency is, it is important to understand the following terms:

- **Multiprogramming:** It is the management of multiple processes within a uniprocessus system.
- **Multiprocessing:** It is the management of multiple processes within a multiprocessor.
- **Distributed processing:** It is the management of multiple processes executing on multiple, distributed computer systems.

Concurrency is the fundamental concept to all these areas and OS design.

Concurrency is the tendency for things to happen at the same time in a system. *Concurrency processing* is a computing model in which multiple processors execute instructions simultaneously for better performance. Here the tasks are broken into subtasks (as shown in figure b) that are then assigned to separate processors to perform simultaneously, instead of sequentially as they would have to be carried out by a single processor (sequential process is shown in figure a). The independent process, as the name implies, does not share any kind of information or data with each other. They just compete with each other for resources like CPU, I/O devices etc.



(a)

(b)

Figure 3.1: *(a) Sequential process (b) Concurrent process*

There are several design and management issues raised by concurrency. These are:

i. The OS must be able to keep track of the various processes.

ii. The OS must allocate and deallocate various resources for each active process; sometimes multiple processes want access to the same resource.

iii. The OS must protect the data and physical resources of each process against interference from other processes.

iv. The process must function independent of the execution speed.

In concurrent processing, following problems may arise:

1. Sharing of global resources

If two processes both make use of the same global variables and both perform reads and writes on that variable, then the order in which the various reads and writes are executed is critical.

Example:

Consider two processes P1 and P2 executing the function echo:

```
void echo ()
{
 chin=getchar();
 chout=chin;
 putchar(chout);
}
```

Imagine two processes P1 and P2 both executing this code at the "same" time, with the following interleaving due to multi-programming.

Consider the following sequence:

1. Process P1 invokes the echo procedure and is interrupted immediately after getchar() returns its value

in chin. At this point, the most recently entered character, say x, is stored in variable chin.

2. Process P2 is activated and invokes the echo procedure, which runs to conclusion, inputting and then displaying a single character, say y, on the screen.
3. Process P1 is resumed. By this time, the value x has been overwritten in chin and therefore lost. Instead, chin contains y, which is transferred to chout and displayed.

The essence of the problem is the shared global variable chin. P1 sets chin, but this write is subsequently lost during the execution of P2. The general solution is to allow only one process at a time to enter the code that accesses chin: such code is often called a critical section. When one process is inside a critical section of code, other processes must be prevented from entering that section. This requirement is known as mutual exclusion.

This solution can be explained as following sequence:

1. Process P1 invokes the echo procedure and is interrupted immediately after the conclusion of the input function. At this point, the most recently entered character, x, is stored in variable chin.
2. Process P2 is activated and invokes the echo procedure. However, because P1 is still inside the echo procedure, although currently suspended, P2 is blocked from entering the procedure. Therefore, P2 is suspended awaiting the availability of the echo procedure.
3. At some later time, process P1 is resumed and completes execution of echo. The proper character, x, is displayed.
4. When P1 exits echo, this removes the block on P2. When P2 is later resumed, the echo procedure

This problem was stated with the assumption that there was a single-processor, multiprogramming OS. The example demonstrates that the problems of concurrency occur even when there is a single processor. In a multiprocessor system, the same problems of protected shared resources arise, and the same solution works. First, suppose there is no mechanism for controlling access to the shared global variable:

Process P1	Process P2
.	.
chin=getchar();	.
.	chin=getchar();
chout=chin;	chout=chin;
putchar(chout);	.
.	putchar(chout);

1. Processes P1 and P2 are both executing, each on a separate processor. P1 invokes the echo procedure.
2. While P1 is inside the echo procedure, P2 invokes echo. Because P1 is still inside the echo procedure (whether P1 is suspended or executing), P2 is blocked from entering the procedure. Therefore, P2 is suspended awaiting the availability of the echo procedure.
3. At a later time, process P1 completes execution of echo, exits that procedure, and continues executing. Immediately upon the exit of P1 from echo, P2 is resumed and begins executing echo.

In the case of a uniprocessor system, the reason we have a problem is that an interrupt can stop instruction execution anywhere in a process. In the case of a multiprocessor system, we have that same condition and, in addition, a problem can be caused because two processes may be executing simultaneously and both trying to access the same global variable. However, the solution to both types of problem is the same: control access to the shared resource.

2. Optimal resource allocation

It is difficult for the OS to optimally manage the allocation of resources optimally.

3. Locating programming errors

It becomes very difficult to locate a programming error because results are typically not deterministic and reproducible.

The key issues that arise when concurrent processes interact are *mutual exclusion* and *deadlock.*

- **Mutual exclusion:** It is a condition in which there is a set of concurrent processes, only one of which is able to access a given resource or perform a given function at any time. This technique can be used to resolve conflicts (competition for resources) and to synchronize processes.

  There are two approaches to support mutual exclusion:

  - One approach uses special-purpose machine instructions.
  - Another approach involves providing features (semaphores and message facilities) within the OS.

- **Deadlock:** It is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.

Why do processes need to be synchronized?

Process synchronization means sharing system resources by processes in such a way that, concurrent access to 'shared data' is handled thereby minimizing the chance of inconsistent data.

Process synchronization needs to be implemented to prevent data inconsistency among processes. The main purpose of synchronization is the sharing of resources without interference using mutual exclusion. The other purpose is the coordination of the process interactions in an operating system. Semaphores and monitors are the most powerful and most commonly used mechanisms to solve synchronization problems.

3.2 Race Condition

The situation where several processes access and manipulate shared data concurrently is called *race condition*. The final value of the shared data depends upon which process finishes last.

A race condition occurs when multiple processes or threads read and write data items so that the final result depends on the order of execution of instructions in the multiple processes. To avoid race condition we need Mutual Exclusion.

3.3 Critical Section

Critical section or *critical region* is that part of the program where the shared memory is accessed. Moreover, it is a code segment where the shared variables can be accessed in an atomic manner, i.e., only one process can execute its critical section at a time. All other processes have to wait to execute in their critical section. Critical section is given as:

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

Figure 3.3 Critical section

Critical section problem needs a solution to synchronize the different processes. A solution to the critical section problem must satisfy the following three conditions:

1. **Mutual exclusion**
   Out of a group of cooperating processes, only one process can be in its critical section at a given point of time.
2. **Progress**
   If no process is in its critical section, and if one or more threads want to execute their critical section then any one of these threads must be allowed to get into its critical section.
3. **Bounded waiting**
   After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process's request is

granted. So after the limit is reached, the system must grant the process permission to get into its critical section.

3.4 Mutual Exclusion using Critical Section

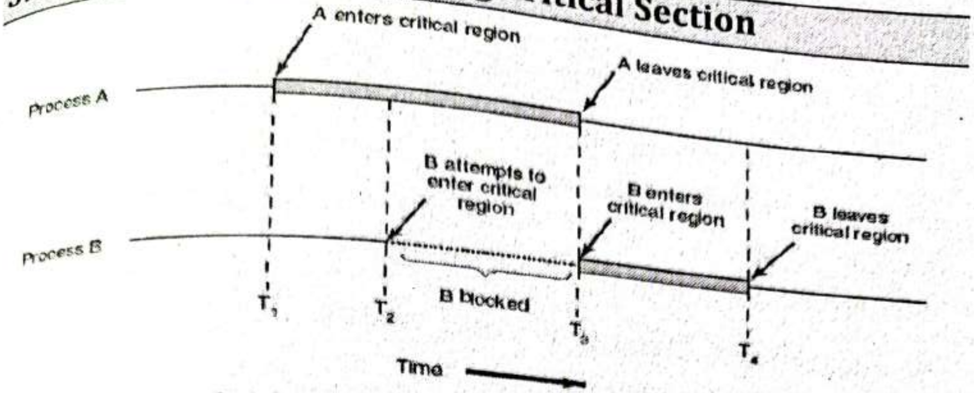


Figure 3.4 Mutual exclusion using critical section

Mutual Exclusion also known as Mutex was first identified by Dijkstra. It is the requirement that when a process P is accessing a shared resource R, no other process should be able to access R until P has finished with R. Examples of such resources include files, I/O devices such as printers, and shared data structures. It is property of concurrency control which is used to prevent race conditions.

In figure 3.4, process A enters its critical region at time T1 . A little later, at time T2 process B attempts to enter its critical region but fails because another process is already in its critical region and we allow only one at a time. Consequently, B is temporarily suspended until time T3 when A leaves its critical region, allowing B to enter immediately. Eventually B leaves (at T4) and we are back to the original situation with no processes in their critical regions.

3.4.1 Requirements of Mutual Exclusion

Any facility that provides mutual exclusion should meet these requirements:

1. No two processes may be simultaneously inside their critical regions.
2. No assumptions may be made about speeds or the number of CPUs.

3. No process running outside its critical region may block any process.

4. No process should have to wait forever to enter its critical region.

3.4.2 Mutual Exclusion with Busy Waiting

In this section we will examine various proposals for achieving mutual exclusion, so that while one process is busy updating shared memory in its critical region, no other process will enter its critical region and cause trouble.

Proposals for achieving mutual exclusion:

- Disabling interrupts
- Shared Lock variables
- Strict alternation
- Peterson's solution
- The TSL instruction

❑ **Disabling interrupts:**

Each process disables all interrupts just after entering in its critical section and re-enable all interrupts just before leaving critical section. With interrupts turned off the CPU could not be switched to other process. Hence, no other process will enter its critical and mutual exclusion achieved. This approach is generally unattractive because it is unwise to give user processes the power to turn off interrupts. Suppose that one of them did it, and never turned them on again. That could be the end of the system.

❑ **Shared lock variable:**

Consider having a single, shared (lock) variable, initially 0. When a process wants to enter its critical region, it first tests the lock. If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0. Thus, a 0 means that no process is in its critical region, and a 1 means that some process is in its critical region.

Unfortunately, this idea contains some drawbacks. Suppose that one process reads the lock and sees that it is 0. Before it can set the lock to 1, another process is scheduled, runs, and sets the lock to 1. When the first process runs again, it will also set the lock to 1, and two processes will be in their critical regions at the same time.

Strict alternation

Two Process Solution :

```
#define FALSE 0
#define TRUE 1
while (TRUE)
{
while (turn != 0) ;
critical _region();
turn = 1;
noncritical_region();
}
```

```
#define FALSE 0
#define TRUE 1
while (TRUE)
{
        while (turn != 1);
        critical _region();
        turn = 0;
        noncritical _region();
}
```

Figure 3.5 : *A proposed solution to the critical region problem, (a) Process 0 (b) Process 1.*

The integer variable turn, initially 0, keeps track of whose turn it is to enter the critical region and examine or update the shared memory. Initially, process 0 inspects turn, finds it to be 0, and enters its critical region. Process 1 also finds it to be 0 and therefore sits in a tight loop continually testing turns to see when it becomes 1. When process 0 leaves the critical region, it sets turn to 1, to allow process 1 to enter its critical region. Suppose that process 1 finishes its critical region quickly, so that both processes are in their noncritical regions, with turn set to 0. Now process 0 executes its whole loop quickly, exiting its critical region and setting turn to 1. At this point turn is 1 and both processes are executing in their noncritical regions.

❑ Peterson's Solution

Peterson's algorithm is shown in Figure 3.6.

```
#define FALSE  0
#defineTRUE    1
#define N      2        /* number of processes */
int turn;               /* whose turn is it? */
int interested[N];      /* all values initially 0̆ (FALSE) */
void enter_region(int process);  /* process is O or 1 */
{
int other;              /* number of the other process */
other = 1` - process;   /* the opposite of process */
interested[process] = TRUE;
                        /* show that you are interested */
turn = process;         /* set flag */
while (turn == process && interested[other] == TRUE);
                        /* null statement */
}
void leave_region(int process)
                        /* process: who is leaving */
{
interested[process] = FALSE;
              /* indicate departure from critical region /
}
```

Figure 3.6 : Peterson's solution for achieving mutual exclusion

Before using the shared variables (i.e., before entering its critical region), each process calls *enter_region* with its own process number, 0 or 1, as parameter. This call will cause it to wait, if need be, until it is safe to enter. After it has finished with the shared variables, the process calls *leave_region* to indicate that it is done and to allow the other process to enter, if it so desires. Initially neither process is in its critical region. Now process 0 calls *enter_region*. It indicates its interest by setting its array element and sets turn to 0. Since process 1 is not interested, enter region returns immediately. If process 1 now makes a call to *enter_region*, it will hang there until interested [0] goes to FALSE, an event that only happens when process 0 calls *leave_region* to exit the critical region. Now consider the case that both processes call *enter_region* almost simultaneously. Both will store their process number in *turn*. Whichever store is done last is the one that counts; the first one is overwritten and lost. Suppose that process 1 stores last, so *turn* is 1. When both processes come to the while statement, process 0 executes it zero times and enters its critical region. Process 1 loops and does not enter its critical region until process 0 exits its critical region.

❑ The TSL Instruction (Test and Set Lock)

Test and Set Lock (TSL) is a synchronization mechanism. It uses a test and set instruction to provide the synchronization among the processes executing concurrently.

```
enter_region:
   TSL REGISTER,LOCK    |copy lock to register and set lock to 1
   CMP REGISTER, #0     |was lock zero?
   JNE enter_region     |if it was nonzero, lock was set, so loop
   RET                  |return to caller; critical region entered
leave_region:
   MOVE LOCK, #0        | store a 0 in lock
   RET                  |return to caller
```

***Figure 3.7 :** Entering and leaving a critical region using the TSL instruction.*

To use the TSL instruction, we will use a shared variable, lock, to coordinate access to shared memory. When lock is 0, any process may set it to 1 using the TSL instruction and then read or write the shared memory. When it is done, the process sets lock back to 0 using an ordinary move instruction. The first instruction copies the old value of lock to the register and then sets lock to 1. Then the old value is compared with 0. If it is nonzero, the lock was already set, so the program just goes back to the beginning and tests it again. Sooner or later it will become 0 (when the process currently in its critical region is done with its critical region), and the subroutine returns, with the lock set. Clearing the lock is very simple. The program just stores a 0 in lock. No special synchronization instructions are needed.

An alternative instruction to TSL is XCHG , which exchanges the contents of two locations atomically, for example, a register and a memory word. The code is shown in Figure 3.8, and, as can be seen, is essentially the same as the solution with TSL . All Intel x86 CPUs use XCHG instruction for low-level synchronization.

The TSL Instruction (2)

```
enter_region:
     MOVE REGISTER,#1            | put a 1 in the register
     XCHG REGISTER,LOCK          | swap the contents of the register and lock variable
     CMP REGISTER,#0             | was lock zero?
     JNE enter_region            | if it was non zero, lock was set, so loop
     RET                         | return to caller; critical region entered

leave_region:
     MOVE LOCK,#0                | store a 0 in lock
     RET                         | return to caller
```

Figure 3.8 : Entering and leaving a critical region using the XCHG instruction.

3.5 Sleep () and Wakeup ()

Both Peterson's solution and the solutions using TSL or XCHG are correct, but both have the defect of requiring *busy waiting*. In essence, what these solutions do is this: when a process wants to enter its critical region, it checks to see if the entry is allowed. If it is not, the process just sits in a tight loop waiting until it is.

In busy waiting solutions, processes waiting to enter the critical sections waste processor time checking to see if they can proceed waster of processor time and possibility of deadlock(starvation), sleep and wakeup eliminate this problem.

For mutual exclusion one of the simplest is the pair sleep and wakeup. Sleep is a system call that causes the caller to block, that is, be suspended until another process wakes it up. The wakeup call has one parameter, the process to be awakened. Alternatively, both sleep and wakeup each have one parameter, a memory address used to match up sleeps with wakeups.

3.6 Producer-Consumer Problem

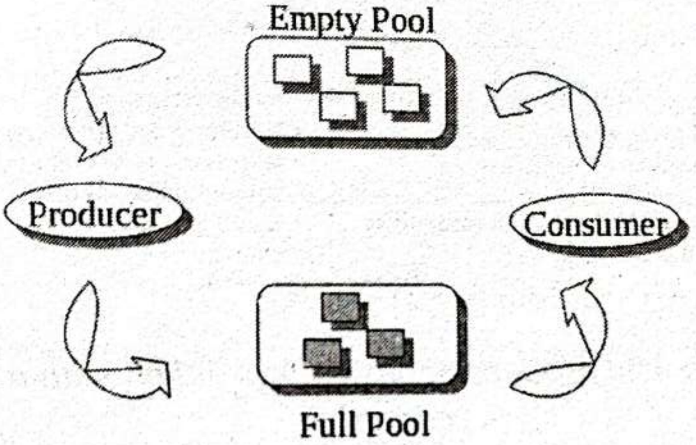


Figure 3.9: Producers produce a product and consumers consume the product, but both use of one of the containers each time.

In the *producer-consumer problem* (also known as the *bounded-buffer problem*), two processes share a common, fixed-size buffer. One of them, the producer, puts information into the buffer, and the other one, the consumer, takes it out. Trouble arises when the producer wants to put a new item in the buffer, but it is already full. The solution is for the producer to go to sleep, to be awakened when the consumer has removed one or more items. Similarly, if the consumer wants to remove an item from the buffer and sees that the buffer is empty, it goes to sleep until the producer puts something in the buffer and wakes it up. This approach sounds simple enough, but it leads to the same kinds of race conditions we saw earlier with the spooler directory.

```
#define N 100                                  /* number of items in the buffer */
int count = 0;

void producer(void)
{
     int item;

     while (TRUE) {                            /* repeat forever */
          item = produce_item();               /* generate next item */
          if (count == N) sleep();             /* if buffer is full, go to sleep */
          insert_item(item);                   /* put item in buffer */
          count = count + 1;                   /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);    /* was buffer empty? */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                            /* repeat forever */
          if (count == 0) sleep();             /* if buffer is empty, got to sleep */
          item = remove_item();                /* take item out of buffer */
          count = count - 1;                   /* decrement count of items in buffer */
          if (count == N - 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                  /* print item */
     }
}
```

Figure 3.10: *The producer-consumer problem with a fatal race condition.*

To keep track of the number of items in the buffer, we will need a variable count. If the maximum number of items the buffer can hold is *N*, the producer's code will first test to see if *count* is *N*. If it is, the producer will go to sleep; if it is not, the producer will add an item and increment *count*. The consumer's code is similar: first test *count* to see if it is 0. If it is, go to sleep; if it is nonzero, remove an item and decrement the counter. Each of the processes also tests to see if the other should be awakened, and if so, wakes it up.

3.7 Semaphores

A semaphore is a special kind of integer variable, usually stored in shared memory, so all processes can access it. A semaphore could have the value 0, indicating that no wakeups were saved, or some positive value if one or more wakeups were pending. It is used as a flag that is only accessed through two atomic operations: *down* (or *wait* or *sleep* or *P (Proberen)*) and *up* (or *signal* or *wakeup* or *V (Vergogen)*). This atomicity is absolutely essential to solving synchronization problems and avoiding race conditions.

Semaphores can be used to implement critical sections or to enforce certain scheduling constraints. Critical sections are typically built by using semaphores that are initialized to 1. In a case, one process can call wait() in order to enter the critical section, preventing any other processes from passing the wait condition. Once the process finishes the critical section, it calls signal() which will allow the next process to enter the critical section. In some cases, it is useful to start the semaphore with a value greater than 1. This allows multiple processes to enter the critical section at once. While this can be a problem if the critical section is supposed to be protecting data, it can be useful in other cases such as where a limited number of resources are available for simultaneous use, such as in the Multiple Consumers and Producers Problem. The definitions of wait and signal are as follows –

Wait

The wait operation decrements the value of its argument S, if it is positive. If S is negative or zero, then no operation is performed.

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

Signal

The signal operation increments the value of its argument S.

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

3.7.1 Properties of Semaphores

1. **Machine independent:** No need to code at assemble level as in tsl.
2. Works with any number of processes.
3. Can have different semaphores for different critical sections.
4. A process can acquire multiple needed resources by executing multiple downs.
5. Simply binary semaphore or more than one if desired using a counting semaphore.

3.7.2 Types of Semaphores

- **Counting semaphore:** Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.
- **Binary semaphore:** This is also known as mutex lock. A binary semaphore is initialized as free (1) and can vary from 0 to 1. It is used to implement the solution of critical section problems with multiple processes.

  Here, are some major differences between counting and binary semaphore:

Counting semaphore	Binary semaphore
No mutual exclusion	Mutual exclusion
Any integer value	Value only 0 and 1
More than one slot	Only one slot
Provide a set of processes	It has a mutual exclusion mechanism.

3.7.3 Semaphore Implementation

We can define a semaphore as a class:

```
class Semaphore
{
 int value;                // semaphore value
 ProcessQueue L;          // process queue
 //operations
   wait();
```

```
        signal();
    }
```

Semaphore operations now defined as

```
S.wait():
S.value --;
    if (S.value < 0) {
    add this process to S.L;
    block(); // block a process
    }
    S.signal():
    S.value ++;
      if (S.value <= 0) {
      remove a process P from S.L;
      wakeup(); // wake a process
      }
```

In addition, there are two simple utility operations:

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

3.7.4 Disadvantages of Semaphores

Here, are drawbacks of semaphores:

- One of the biggest limitations of a semaphore is priority inversion.
- The operating system has to keep track of all calls to wait and signal semaphore.
- Semaphore is more prone to programmer error.
- It may cause deadlock or violation of mutual exclusion due to programmer error.

3.8 MUTEX

A mutex is a shared variable that can be in one of two states: *unlocked* or *locked*. Consequently, only one bit is required to represent it, but in practice an integer often is used, with 0 meaning unlocked and all other values meaning locked.

Two procedures are used with mutexes. When a thread (or process) needs access to a critical region, it calls *mutex_lock*. If the mutex is currently unlocked (meaning that the critical region available), the call succeeds and the calling thread is free to ente the critical region. On the other hand, if the mutex is alread locked, the calling thread is blocked until the thread in the critica region is finished and calls *mutex_unlock*. If multiple threads ar blocked on the mutex, one of them is chosen at random and allowed to acquire the lock.

```
mutex_lock:
        TSL REGISTER,MUTEX      | copy mutex to register and set mutex to 1
        CMP REGISTER,#0         | was mutex zero?
        JZE ok                  | if it was zero, mutex was unlocked, so return
        CALL thread_yield       | mutex is busy; schedule another thread
        JMP mutex_lock          | try again
ok:     RET                     | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0           | store a 0 in mutex
        RET                     | return to caller
```

Figure 3.11: *Implementation of mutex lock and mutex unlock.*

(Courtesy of Andrew S. Tanenbaum, Herbert Bos; *Modern Operating Systems*)

Difference between Semaphore and Mutex

Parameters	Semaphore	Mutex
Mechanism	It is a type of signaling mechanism.	It is a locking mechanism.
Data Type	Semaphore is an integer variable.	Mutex is just an object.
Modification	The wait and signal operations can modify a semaphore.	It is modified only by the process that may request or release a resource.
Resource management	If no resource is free, then the process requires a resource that should execute wait operation. It should wait until the count of the semaphore is greater than 0.	If it is locked, the process has to wait. The process should be kept in a queue. This needs to be accessed only when the mutex is unlocked.
Thread	You can have multiple program threads.	You can have multiple program threads in mutex but not simultaneously.

Parameters	Semaphore	Mutex
Ownership	Value can be changed by any process releasing or obtaining the resource.	Object lock is released only by the process, which has obtained the lock on it.
Types	Types of Semaphore are counting semaphore and binary semaphore.	Mutex has no subtypes.
Operation	Semaphore value is modified using wait () and signal () operation.	Mutex object is locked or unlocked.

3.9 Solving Consumer Producer Problem using Semaphore

```
#define N 100                              /* number of slots in the buffer */
typedef int semaphore;                     /* semaphores are a special kind of int */
semaphore mutex = 1;                       /* controls access to critical region */
semaphore empty = N;                       /* counts empty buffer slots */
semaphore full = 0;                        /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                        /* TRUE is the constant 1 */
          item = produce_item( );          /* generate something to put in buffer */
          down(&empty);                    /* decrement empty count */
          down(&mutex);                    /* enter critical region */
          insert_item(item);               /* put new item in buffer */
          up(&mutex);                      /* leave critical region */
          up(&full);                       /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                        /* infinite loop */
          down(&full);                     /* decrement full count */
          down(&mutex);                    /* enter critical region */
          item = remove_item( );           /* take item from buffer */
          up(&mutex);                      /* leave critical region */
          up(&empty);                      /* increment count of empty slots */
          consume_item(item);              /* do something with the item */
     }
}
```

(Courtesy of Andrew S. Tanenbaum, Herbert Bos; *Modern Operating Systems*)

This solution uses three semaphores: one called full for counting the number of slots that are full, one called empty for counting the number of slots that are empty, and one called mutex to make sure the producer and consumer do not access the buffer at the same time. Full is initially 0, empty is initially equal to the number of slots in the buffer, and mutex is initially 1. Semaphores that are initialized to 1 and used by two or more processes to ensure that only one of them can enter its critical region at the same time are called binary semaphores. If each process does a down just before entering its critical region and an up just after leaving it, mutual exclusion is guaranteed.

3.10 Monitors

A monitor is a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package. Processes may call the procedures in a monitor whenever they want to, but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor.

```
monitor example
integer i; condition c;
procedure producer();
.
.
.
end;
procedure consumer();
.
end;
end monitor;
```

Figure 3.12: A monitor

Figure illustrates a monitor written in an imaginary language, Pidgin Pascal. C cannot be used here because monitors are a language concept and C does not have them.

3.10.1 Characteristics of Monitors

- Only one process may be executed in monitor at a time.
- A process enters the monitor by invoking one of its procedures.
- Local data variables are accessible only by monitor procedure and not by the external procedures.
- Monitors are a programming language concept.
- They control not only the time but also the nature of operations.

3.10.2 Solving Consumer Producer Problem using Monitors

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count – 1;
        if count = N – 1 then signal(full)
    end;
    count := 0;
end monitor;

procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;
procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```

Figure 3.13: *An outline of the producer-consumer problem with monitors.*

(Courtesy of Andrew S. Tanenbaum, Herbert Bos; *Modern Operating Systems*)

3.11 Message Passing

Message passing model allows multiple processes to read and write data to the message queue without being connected to each other. Messages are stored on the queue until their recipient retrieves them. Message queues are quite useful for interprocess communication and are used by most operating systems. As such, they can easily be put into library procedures, such as

send (destination, &message);

and

receive (source, &message);

The former call sends a message to a given destination and the latter one receives a message from a given source (or from ANY, if the receiver does not care). If no message is available, the receiver can block until one arrives. Alternatively, it can return immediately with an error code.

3.11.1 Issues in Message Passing

Send and receive could be blocking or non-blocking:

- **Blocking send**: when a process sends a message it blocks until the message is received at the destination.
- **Non-blocking send**: After sending a message the sender proceeds with its processing without waiting for it to reach the destination.
- **Blocking receive:** When a process executes a receive it waits blocked until the receive is completed and the required message is received.
- **Non-blocking receive**: The process executing the receive proceeds without waiting for the message(!).

Blocking Receive/non-blocking send is a common combination.

3.11.2 Solvir Passin

```
#define N 100

void producer(void
{
    int item;
    message m;

    while (TRUE)
        item = p
        receive(
        build_m
        send(co
    }
}

void consumer(voi
{
    int item, i;
    message m;

    for (i = 0; i <
    while (TRUE
        receive
        item = 
        send(pr
        consum
    }
}
```

Figure 3.14:

(Courtesy of Ar *Systems*)

3.12 The Di

Figu

processes to read
eing connected to
ntil their recipient
il for interprocess
systems. As such,
:h as

en destination and
ı source (or from
ge is available, the
ely, it can return

-blocking:

message it blocks
tion.

essage the sender
ing for it to reach

cutes a receive it
d and the required

cuting the receive
!).

is a common

3.11.2 Solving Consumer Producer Problem using Message Passing :

```
#define N 100                              /* number of slots in the buffer */

void producer(void)
{
    int item;
    message m;                             /* message buffer */

    while (TRUE) {
        item = produce_item();             /* generate something to put in buffer */
        receive(consumer, &m);             /* wait for an empty to arrive */
        build_message(&m, item);           /* construct a message to send */
        send(consumer, &m);                /* send item to consumer */
    }
}

void consumer(void)
{
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
    while (TRUE) {
        receive(producer, &m);             /* get message containing item */
        item = extract_item(&m);           /* extract item from message */
        send(producer, &m);                /* send back empty reply */
        consume_item(item);                /* do something with the item */
    }
}
```

***Figure 3.14:** The producer-consumer problem with N messages.*

(Courtesy of Andrew S. Tanenbaum, Herbert Bos; *Modern Operating Systems*)

3.12 The Dining Philosopher's Problem

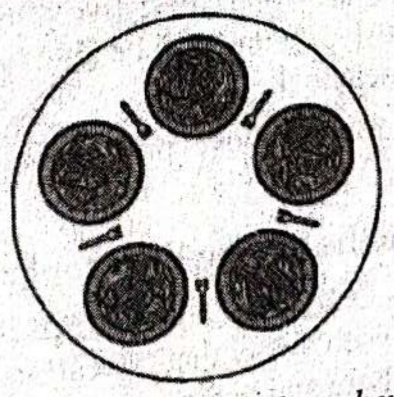


***Figure 3.15:** For dining philosopher's problem*

3.11 Message Passing

Message passing model allows multiple processes to read and write data to the message queue without being connected to each other. Messages are stored on the queue until their recipient retrieves them. Message queues are quite useful for interprocess communication and are used by most operating systems. As such they can easily be put into library procedures, such as

send (destination, &message);

and

receive (source, &message);

The former call sends a message to a given destination and the latter one receives a message from a given source (or from ANY, if the receiver does not care). If no message is available, the receiver can block until one arrives. Alternatively, it can return immediately with an error code.

3.11.1 Issues in Message Passing

Send and receive could be blocking or non-blocking:

- **Blocking send**: when a process sends a message it blocks until the message is received at the destination.
- **Non-blocking send**: After sending a message the sender proceeds with its processing without waiting for it to reach the destination.
- **Blocking receive:** When a process executes a receive it waits blocked until the receive is completed and the required message is received.
- **Non-blocking receive**: The process executing the receive proceeds without waiting for the message(!).

Blocking Receive/non-blocking send is a common combination.

3.11.2 Solving Consumer Producer Problem using Message Passing :

```
#define N 100                               /* number of slots in the buffer */

void producer(void)
{
    int item;
    message m;                              /* message buffer */

    while (TRUE) {
        item = produce_item();              /* generate something to put in buffer */
        receive(consumer, &m);              /* wait for an empty to arrive */
        build_message(&m, item);            /* construct a message to send */
        send(consumer, &m);                 /* send item to consumer */
    }
}

void consumer(void)
{
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m); /* send N empties */
    while (TRUE) {
        receive(producer, &m);              /* get message containing item */
        item = extract_item(&m);            /* extract item from message */
        send(producer, &m);                 /* send back empty reply */
        consume_item(item);                 /* do something with the item */
    }
}
```

Figure 3.14: *The producer-consumer problem with N messages.*

(Courtesy of Andrew S. Tanenbaum, Herbert Bos; *Modern Operating Systems*)

3.12 The Dining Philosopher's Problem

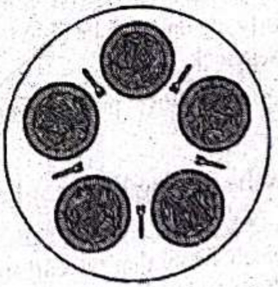


Figure 3.15: *For dining philosopher's problem*

- Five philosophers are seated around a round table with the plate of spaghetti.
- Each philosopher requires two forks to eat the spaghetti.
- The condition may arise when neighbouring philosopher also require fork to have spaghetti.
- According to this problem, the life of philosopher consists of eating and thinking.
- When philosopher is hungry, he tries to acquire his left and right fork. If he is successful, then he eats for a while, then puts down the fork and continues to think.
- The question is: Can philosopher eat without getting stuck?

The solution:

```
#define N 5                              /* number of philosophers */

void philosopher(int i)                  /* i: philosopher number, from 0 to 4 */
{
     while (TRUE) {
          think();                       /* philosopher is thinking */
          take_fork(i);                  /* take left fork */
          take_fork((i+1) % N);          /* take right fork; % is modulo operator */
          eat();                         /* yum-yum, spaghetti */
          put_fork(i);                   /* put left fork back on the table */
          put_fork((i+1) % N);           /* put right fork back on the table */
     }
}
```

Figure 3.16: A nonsolution to the dining philosopher's problem

(Courtesy of Andrew S. Tanenbaum, Herbert Bos; *Modern Operating Systems*)

- The procedure take_fork waits until the specified fork is available and then seizes it.
- But, the solution is wrong i.e. if five philosophers take their left simultaneously, no one will be able to take right fork and there will be a deadlock.

Thus, the solution to above problem is the use of binary semaphore.

3.12.1 A Solution to the Dining Philosophers Problem using Semaphore

- The scenario is such a way that the call to think( ) is done by a binary semaphore and before acquiring fork, a philosopher would do a down on mutex and perform 'up' on a mutex after placing the forks.
- The program using semaphore is stated below which makes use of an array, state to keep track of philosopher (eating, thinking or hungry).
- A philosopher may only move into eating state if neither neighbour is eating.
- Philosopher's i's neighbours are defined by LEFT and RIGHT i.e. if i is 2, LEFT is 1 and RIGHT is 3.

```
#define N           5                   /* number of philosophers */
#define LEFT        (i+N-1)%N           /* number of i's left neighbor */
#define RIGHT       (i+1)%N             /* number of i's right neighbor */
#define THINKING    0                   /* philosopher is thinking */
#define HUNGRY      1                   /* philosopher is trying to get forks */
#define EATING      2                   /* philosopher is eating */
typedef int semaphore;                  /* semaphores are a special kind of int */
int state[N];                           /* array to keep track of everyone's state */
semaphore mutex = 1;                    /* mutual exclusion for critical regions */
semaphore s[N];                         /* one semaphore per philosopher */
void philosopher(int i)                 /* i: philosopher number, from 0 to N-1 */
{
     while (TRUE) {                     /* repeat forever */
          think( );                     /* philosopher is thinking */
          take_forks(i);                /* acquire two forks or block */
          eat( );                       /* yum-yum, spaghetti */
          put_forks(i);                 /* put both forks back on table */
     }
}
void take_forks(int i)                  /* i: philosopher number, from 0 to N-1 */
{
     down(&mutex);                      /* enter critical region */
     state[i] = HUNGRY;                 /* record fact that philosopher i is hungry */
     test(i);                           /* try to acquire 2 forks */
     up(&mutex);                        /* exit critical region */
     down(&s[i]);                       /* block if forks were not acquired */
}
void put_forks(i)                       /* i: philosopher number, from 0 to N-1 */
{
     down(&mutex);                      /* enter critical region */
     state[i] = THINKING;               /* philosopher has finished eating */
     test(LEFT);                        /* see if left neighbor can now eat */
     test(RIGHT);                       /* see if right neighbor can now eat */
     up(&mutex);                        /* exit critical region */
}

void test(i)                            /* i: philosopher number, from 0 to N-1 */
{
     if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
          state[i] = EATING;
          up(&s[i]);
     }
}
```

Figure 3.17: A solution to the dining philosophers problem using semaphore.

(Courtesy of Andrew S. Tanenbaum, Herbert Bos; *Modern Operating Systems*)

3.12.2 A Solution to the Dining Philosophers Problem using Monitors

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

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

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

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

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

Figure 3.18: A solution to the dining philosopher's problem using monitors

(Courtesy of Andrew S. Tanenbaum, Herbert Bos; *Modern Operating Systems*)

3.13 The Readers and Writers Problem

The dining philosophers problem is useful for modeling processes that are competing for exclusive access to a limited number of resources, such as I/O devices. Another famous problem is the readers and writers problem, which models access to a database. Imagine, for example, an airline reservation system, with many competing processes wishing to read and write it. It is acceptable to have multiple processes reading the database at the same time, but if one process is updating (writing) the database, no other processes may have access to the database, not even readers.

3.13.1 A Solution to the Readers and Writers Problem using Semaphore

```
typedef int semaphore;          /* use your imagination */
semaphore mutex = 1;            /* controls access to 'rc' */
semaphore db = 1;               /* controls access to the database */
int rc = 0;                     /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                  /* repeat forever */
        down(&mutex);               /* get exclusive access to 'rc' */
        rc = rc + 1;                /* one reader more now */
        if (rc == 1) down(&db);     /* if this is the first reader ... */
        up(&mutex);                 /* release exclusive access to 'rc' */
        read_data_base();           /* access the data */
        down(&mutex);               /* get exclusive access to 'rc' */
        rc = rc – 1;                /* one reader fewer now */
        if (rc == 0) up(&db);       /* if this is the last reader ... */
        up(&mutex);                 /* release exclusive access to 'rc' */
        use_data_read();            /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {                  /* repeat forever */
        think_up_data();            /* noncritical region */
        down(&db);                  /* get exclusive access */
        write_data_base();          /* update the data */
        up(&db);                    /* release exclusive access */
    }
}
```

Figure 3.19: A solution to the readers and writers problem using semaphore

(Courtesy of Andrew S. Tanenbaum, Herbert Bos; *Modern Operating Systems*)

In this solution, the first reader to get access to the database does a down on the semaphore db. Subsequent readers merely increment a counter, *rc*. As readers leave, they decrement the counter, and the last to leave does an up on the semaphore, allowing a blocked writer, if there is one, to get in.

The solution presented here implicitly contains a subtle decision worth noting. Suppose that while a reader is using the database, another reader comes along. Since having two readers at the same time is not a problem, the second reader is admitted. Additional readers can also be admitted if they come along. Now suppose a writer shows up. The writer may not be admitted to the data- base, since writers must have exclusive access, so the writer is suspended. Later, additional readers show up. As long as at least one reader is still active, subsequent readers are admitted. As a consequence of this strategy, as long as there is a steady supply of readers, they will all get in as soon as they arrive. The writer will be kept suspended until no reader is present. If a new reader arrives, say, every 2 sec, and each reader takes 5 sec to do its work, the writer will never get in.

To avoid this situation, the program could be written slightly differently: when a reader arrives and a writer is waiting, the reader is suspended behind the writer instead of being admitted immediately. In this way, a writer has to wait for readers that were active when it arrived to finish but does not have to wait for readers that came along after it. The disadvantage of this solution is that it achieves less con- currency and thus lower performance.

3.13.2 A Solution to the Readers and Writers Problem using Monitors

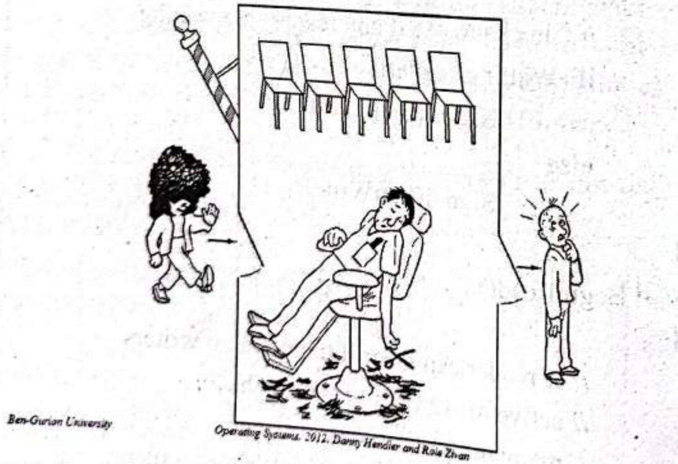
```
// STATE VARIABLES
// Number of active readers; initially = 0
int NReaders = 0;

// Number of waiting readers; initially = 0
int WaitingReaders = 0;

// Number of active writers; initially = 0
int NWriters = 0;

// Number of waiting writers; initially = 0
int WaitingWriters = 0;
```

```
Condition canRead = NULL;
Condition canWrite = NULL;
void BeginWrite()
{
        // A writer can enter if there are no other
        // active writers and no readers are waiting
        if (NWriters == 1 || NReaders > 0) {
                ++WaitingWriters;
                wait(CanWrite);
                --WaitingWriters;
        }
        NWriters = 1;
}
void EndWrite()
{
        NWriters = 0;
        // Checks to see if any readers are waiting
        if (WaitingReaders)
                Signal(CanRead);
        else
                Signal(CanWrite);
}
void BeginRead()
{
        // A reader can enter if there are no writers
        // active or waiting, so we can have
        // many readers active all at once
        if (NWriters == 1 || WaitingWriters > 0) {
                ++WaitingReaders;
                // Otherwise, a reader waits (maybe many do)
                Wait(CanRead);
                --WaitingReaders;
        }
        ++NReaders;
        Signal(CanRead);
}
void EndRead()
{
        // When a reader finishes, if it was the last reader,
        // it lets a writer in (if any is there).
        if (--NReaders == 0)
                Signal(CanWrite);
}
```

Figure 3.20: A solution to the readers and writers problem using monitors

3.14 The Sleeping Barber Problem

Sleeping Barber Problem



A barber shop (saloon) has one barber, one barber chair and N chairs for waiting customers to sit. If there is no customer, the barber goes to sleep. When a customer arrivers, he has to wake up the sleeping barber. If additional customers arrive while the barber is cutting a customer's hair, then they sit if chairs are empty else they leave the shop. Customers are serviced in FCFS order. The problem is to program the barber and the customers without getting into race condition.

```
#define CHAIRS 5                    /* # chairs for waiting customers */

typedef int semaphore;              /* use your imagination */

semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */

void barber(void)
{
     while (TRUE) {
          down(&customers);         /* go to sleep if # of customers is 0 */
          down(&mutex);             /* acquire access to 'waiting' */
          waiting = waiting – 1;    /* decrement count of waiting customers */
          up(&barbers);             /* one barber is now ready to cut hair */
          up(&mutex);               /* release 'waiting' */
          cut_hair();               /* cut hair (outside critical region) */
     }
}

void customer(void)
{
     down(&mutex);                  /* enter critical region */
     if (waiting < CHAIRS) {        /* if there are no free chairs, leave */
          waiting = waiting + 1;    /* increment count of waiting customers */
          up(&customers);           /* wake up barber if necessary */
          up(&mutex);               /* release access to 'waiting' */
          down(&barbers);           /* go to sleep if # of free barbers is 0 */
          get_haircut();            /* be seated and be serviced */
     } else {
          up(&mutex);               /* shop is full; do not wait */
     }
}
```

Figure 3.21: A solution to the sleeping barber problem using semaphore

(Courtesy of Andrew S. Tanenbaum, Herbert Bos; *Modern Operating Systems*)

EXERCISE

1. What is a monitor? Solve dining philosophers' problems using semaphore.
2. What are the conditions to get mutual exclusion? Define semaphore and solve the producer-consumer problem using semaphore.

3. What is the race condition and critical section problem? Explain all possible approaches to handle the situation "while one process is busy updating shared memory, no other process will enter its critical section and cause trouble".

4. Define critical section and mutual exclusion with respect to multiple-process systems. Solve producer and consumer problem using semaphore.

5. Define race condition with example. Explain Peterson's algorithm.

6. Why do we need pipe() function? Define semaphore and explain the major operations in semaphore. Can semaphore be used in distributed systems? Explain why or why not.

7. What is TSL instruction? Why it is used? Solve producer-consumer problem using monitors.

8. Why do processes need to be synchronized? Explain Peterson's solution and TSL instruction approaches used in mutual exclusion with busy waiting.

9. Explain critical section problems. Why is it important for a thread to execute a critical section as quickly as possible?

10. Define semaphore and explain the major operation in semaphore including pseudocode.

11. Define race condition. What are the requirements of mutual exclusion? Solve producer-consumer problem using semaphore and message passing.

12. What is the race condition? Explain how sleep() and wakeup() solutions are better than busy waiting solutions for critical section problems.

13. What is TSL? Why is it used? Explain the major operations of semaphore with a simple implementation as a class.

Chapter 4 Memory Management

4.1 Introduction

Main memory (RAM) has to be wisely managed, so memory management is one of the most important tasks of an operating system. In some earlier systems, the concept of "partitioning" was employed for memory management but now, in almost all modern multiprogramming systems, memory management is done using the scheme of virtual memory (based on one or both techniques: *paging* and *segmentation*).

Memory management includes the methods to manage primary memory since most of the processes run in primary memory. *Protection* and *sharing of memory* are two important memory management functions.

4.2 Memory Management

When memory is assigned dynamically, the operating system must manage it. There are two ways to manage memory: *bitmaps* and *linked lists*. These are also the approaches by which the free blocks in the disk are managed.

4.2.1 Memory Management with Bitmaps

In this concept, memory is divided into allocation units. Corresponding to each allocation unit is a bit in the bitmap, which is 0 if the unit is free and 1 if it is occupied (or vice-versa). The size of the bitmap depends only on the size of memory and the size of the allocation unit.

Consider as shown in Figure 4.1 (a), some portion of memory having five processes and three holes. The memory allocation units are depicted by the tick marks. The shaded regions of the memory shown mean they are free which is represented by 0 in the bitmap. Figure 4.1 (b) denotes the corresponding bitmap.

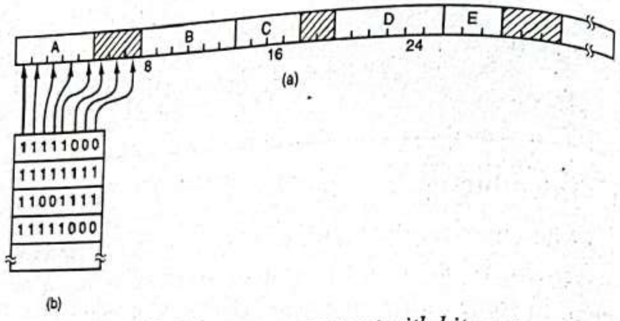


Figure 4.1: Memory management with bitmaps

4.2.2 Memory Management with Linked Lists

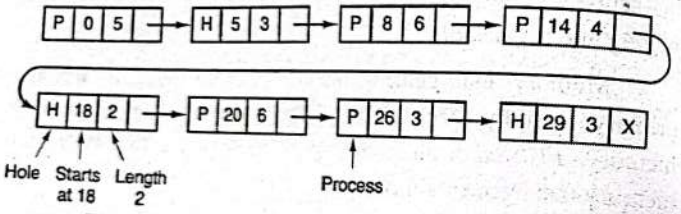


Figure 4.2: Memory management with linked lists

The same memory having five processes and three holes (Figure 4.1 (a)), is represented in Figure 4.2 as a linked list of segments. In this concept, a linked list of allocated and free memory segments is maintained. A segment either contains a process or is an empty hole. Each entry in the list specifies a hole (H) or process (P), the address at which it starts, the length, and a pointer to the next item.

4.3 Resident Monitor

A *resident monitor* is a type of system software program that was used in many early computers from the 1950s to 1970s. It can be considered a precursor to the operating system. The name is derived from a program which is always present in the computer's memory, thus being "resident". Because memory was very limited on those systems, the resident monitor was often little more than a stub that would gain control at the end of a job and load a non-resident portion to perform required job cleanup and setup tasks.

On a general-use computer using punched card input, the resident monitor governed the machine before and after each job control card was executed, loaded and interpreted each control card, and acted as a job sequencer for batch processing operations. The resident monitor could clear memory from the last used program (with the exception of itself), load programs, search for program data and maintain standard input-output routines in memory.

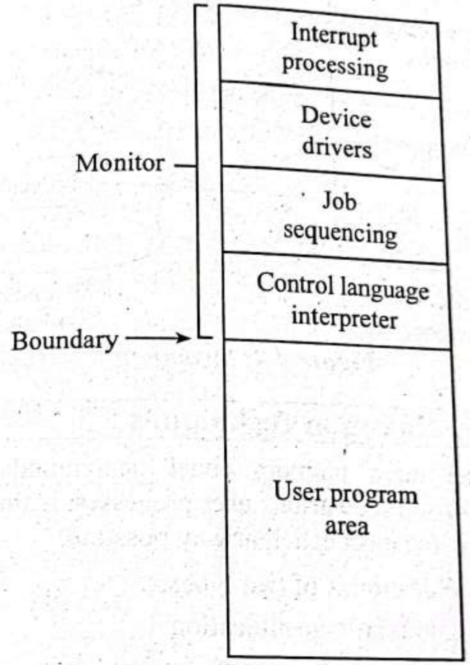


Figure 4.3: Memory layout of resident monitor

4.4 Swapping in Operating System

Swapping is a memory management scheme in which any process can be temporarily swapped from main memory to secondary memory so that the main memory can be made available for other processes. It is used to improve main memory utilization. In secondary memory, the place where the swapped-out process is stored is called swap space.

The concept of swapping is divided into two more concepts: swap-in and swap-out.

- Swap-out is a method of removing a process from RAM and adding it to the hard disk.
- Swap-in is a method of removing a program from a hard disk and putting it back into the main memory or RAM.

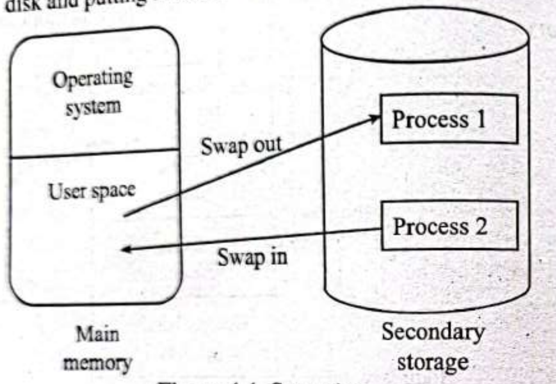


Figure 4.4: Swapping

4.5 Memory Allocation Techniques

Since the main memory must accommodate both the operating system and the various user processes, main memory has to be allocated in the most efficient way possible.

Memory allocation is of two types:

1. Contiguous storage allocation
2. Non-contiguous storage allocation

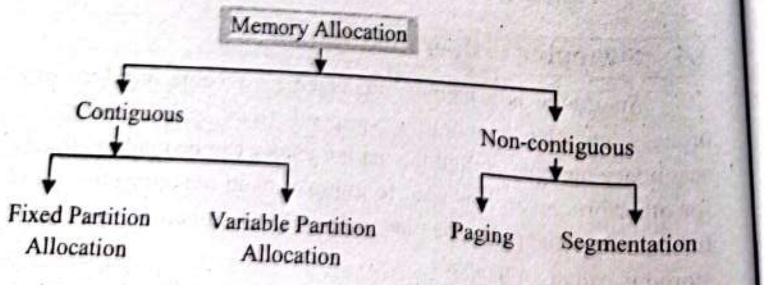


Figure 4.5: Memory allocation techniques

4.5.1 Contiguous Storage Allocation

In contiguous storage allocation, each process occupies a single contiguous section of memory.

In a multiprogramming environment, several programs reside in primary memory at a time and the CPU passes its control rapidly between these programs. To support multiprogramming, one idea is to divide the main memory into several partitions each of which is allocated to a single process. Depending on how and when partitions are created, there are two types of memory partitioning:

- Fixed (or static) partitioning
- Variable (or dynamic) partitioning

4.5.1.1 Fixed Partition Allocation (Multiprogramming with Fixed Partition)

In fixed partitioning, main memory is divided into a number of static partitions at system generation time. There are two alternatives for fixed partitioning: *equal-size partitions* and *unequal-size partitions*.

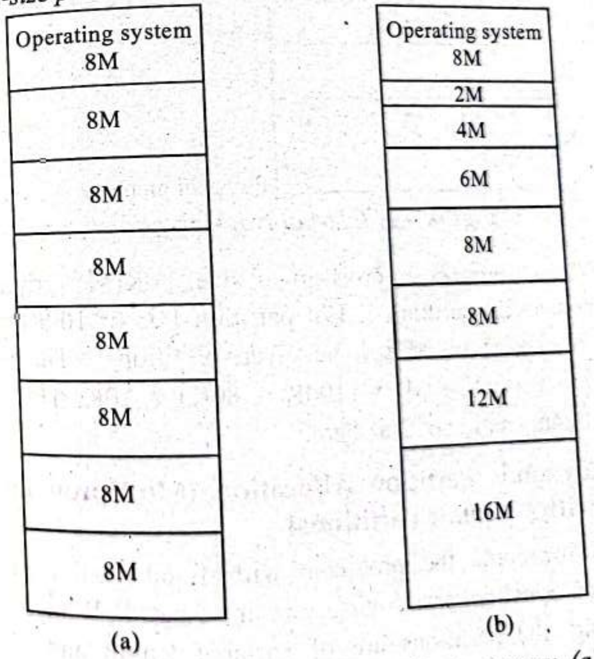


Figure 4.6: Fixed partitioning of a 64-Mbyte memory. (a) Equal-size partitions (b) Unequal-size partitions

Advantages of fixed partitioning:

- Simple to implement.
- Little operating system overhead.

Disadvantages of fixed partitioning:

- Maximum number of active processes is fixed.
- Wastage of memory by programs that are smaller than their partitions (known as *internal fragmentation*).

To understand **internal fragmentation** in detail, consider an example, in which we have a physical memory with the following fixed partitions.

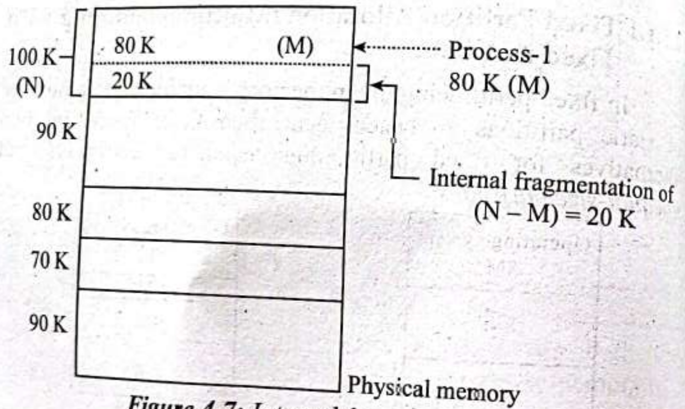


Figure 4.7: Internal fragmentation

When a process or program of size, 80K(M) arrives, it is accommodated in partition-1. But partition-1 is of 100K(N) size. So, M<N. Therefore, M can be given partition-1. The leftover unused space is (N – M) = (100K – 80K) = 20K. This causes internal fragmentation of 20K here.

4.5.1.2 Dynamic Partition Allocation (Multiprogramming with Dynamic Partitions)

To overcome the problems with fixed partitioning, the concept of dynamic partitioning was introduced. With dynamic partitioning, the partitions are of variable length and number. When a process is brought into main memory, it is allocated exactly as much memory as it requires and no more.

Advantages:

- More efficient use of main memory
- No internal fragmentation

Disadvantages:

- Suffers from external fragmentation

External fragmentation:

External fragmentation exists when there is enough total memory space to satisfy a requesting process, but the available spaces are not contiguous; storage is fragmented into a large number of small holes (free spaces).

To solve this problem, *compaction* is employed. Compaction is a technique by which the processes are relocated in such a way that the small chunks of free memory are made contiguous to each other and clubbed together into a single free partition, that may be big enough to accommodate additional process. As an example, consider a memory that has three holes of size 30K, 20K, 50K that have been compacted into one large hole or block of 60K. This is shown in Figure 4.8.

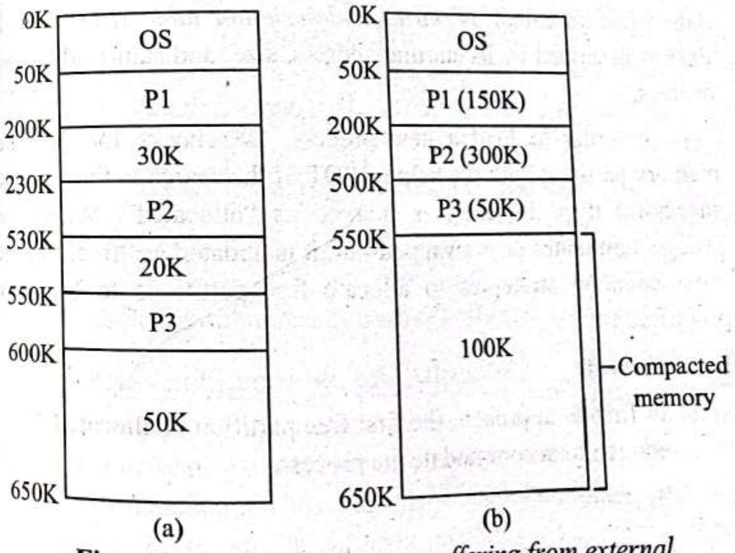


Figure 4.8: (a) Main memory suffering from external fragmentation (b) Main memory after compaction

Comparison between fixed partitioning and dynamic/variable partitioning:

Fixed partitioning	Dynamic/variable partitioning
1. In fixed partitioning, main memory is divided into a number of static partitions.	1. In dynamic partitioning, the partitions are of variable length and number.
2. Operating system decides the partition size only once at the system boot time.	2. Operating system needs to decide about partition size, every time a new process is chosen by long-term scheduler.
3. The degree of multiprogramming is fixed.	3. The degree of multiprogramming varies.
4. It may lead to internal fragmentation.	4. It may lead to external fragmentation.

4.5.1.3 Memory Placement Algorithms

After the partitions are defined, the OS keeps track of status (allocated/free) of the memory partitions and this is done through a data structure called as *partition description table (PDT)*. Each PDT is described by its starting address, size, and status (allocated or free).

In order to load a new process, OS checks for the free memory partition with the help of PDT, if the search is found to be successful then the entry is marked as "allocated". When the process terminates or is swapped-out, it is updated as "free". Three most common strategies to allocate free partitions to the new processes are:

1. **First-fit**

   In first-fit approach, the first free partition is allocated large enough to accommodate the process.

2. **Best-fit**

   In best-fit approach, the smallest free partition is allocated that meets the requirement of the process.

3. **Worst-fit**

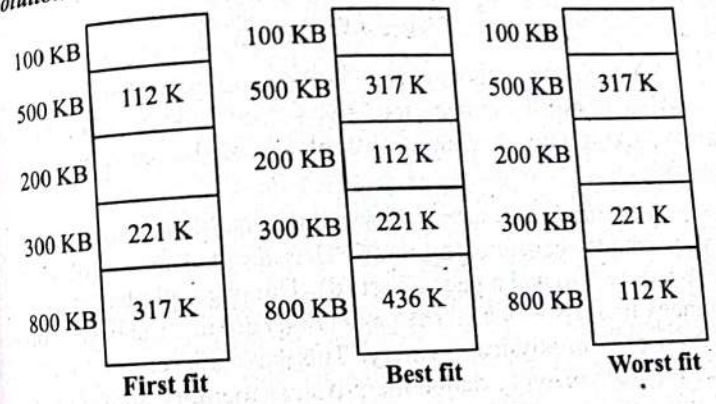
   In worst-fit approach, the largest available partition is allocated to the newly entered process.

In order to find out the free partitions, all of these approaches require scanning PDT. The first-fit terminates after finding the first such partition whereas the best-fit continuous searching for the near exact size. Hence, the first-fit executes faster whereas the best-fit achieves higher utilization of memory.

Example:

Given memory partitions of 100K, 500K, 200K, 300K, and 800K (in order); how would each of first-fit, best-fit, and worst-fit algorithms place processes of 112K, 317K, 221K, and 436K (in order)? Which algorithm makes the most efficient use of memory?

Solution:



Best-fit algorithm makes the most efficient use of memory.

4.5.2 Non-Contiguous Storage Allocation

Employing compaction technique to avoid external fragmentation can be expensive. Another possible solution to external fragmentation is to have non-contiguous storage allocations. There are three main methods of non-contiguous storage allocations:

4.5.2.1 Paging

A memory-management scheme that permits the physic address space of a process to be noncontiguous is called Paging. Paging avoids external fragmentation as well as the need fo compaction.

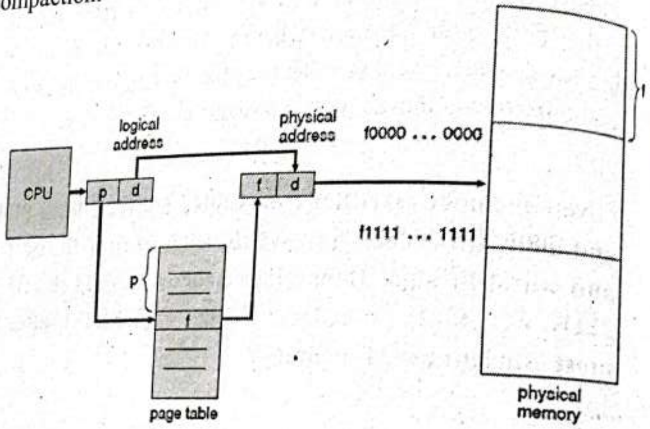


Figure 4.9: Paging

The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called frames and breaking logical memory into blocks of the same size called pages.

The hardware support for paging is illustrated in figure. Every address generated by the CPU is divided into two parts: a page number (p) and a page offset (d). The page number is used as an index into a page table. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.

Advantages of paging:

1. Paging reduces external fragmentation, but still suffer from internal fragmentation.
2. Paging is simple to implement and assumed as an efficient memory management technique.
3. Due to equal size of the pages and frames, swapping becomes very easy.

Drawbacks of paging:

1. Size of Page table can be very big and therefore it wastes main memory.
2. CPU will take more time to read a single word from the main memory.

Solution to these drawbacks:

Translation Look Aside Buffer (TLB)

A *translation look aside buffer* can be defined as a memory cache which can be used to reduce the time taken to access the page table again and again.

It is a memory cache which is closer to the CPU and the time taken by CPU to access TLB is lesser then that taken to access main memory. In other words, we can say that TLB is faster and smaller than the main memory but cheaper and bigger than the register. TLB follows the concept of locality of reference which means that it contains only the entries of those many pages that are frequently accessed by the CPU. In translation look aside buffers, there are tags and keys with the help of which, the mapping is done. TLB hit is a condition where the desired entry is found in translation look aside buffer. If this happens then the CPU simply access the actual location in the main memory. However, if the entry is not found in TLB (TLB miss) then CPU has to access page table in the main memory and then access the actual frame in the main memory.

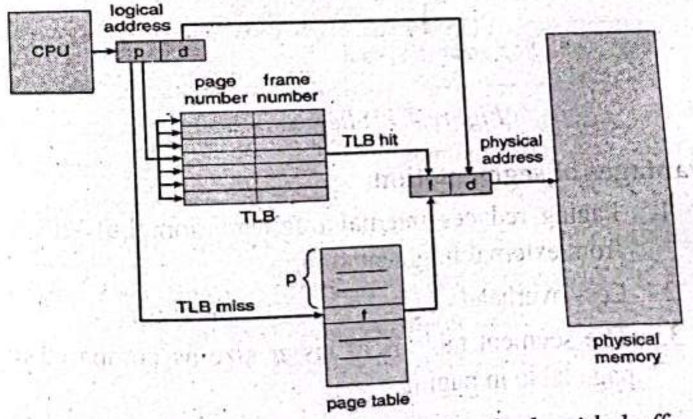


Figure 4.10: Paging using translation look aside buffer

4.5.2.2 Segmentation

Segmentation is a memory management technique in which the memory is divided into the variable size parts. Each part is known as a segment which can be allocated to a process. The details about each segment are stored in a table called a segment table. Segment table is stored in one (or many) of the segments. Segment table contains mainly two information about segment:

1. **Base:** It is the base address of the segment
2. **Limit:** It is the length of the segment.

The figure given below shows how address mapping is done in segmentation.

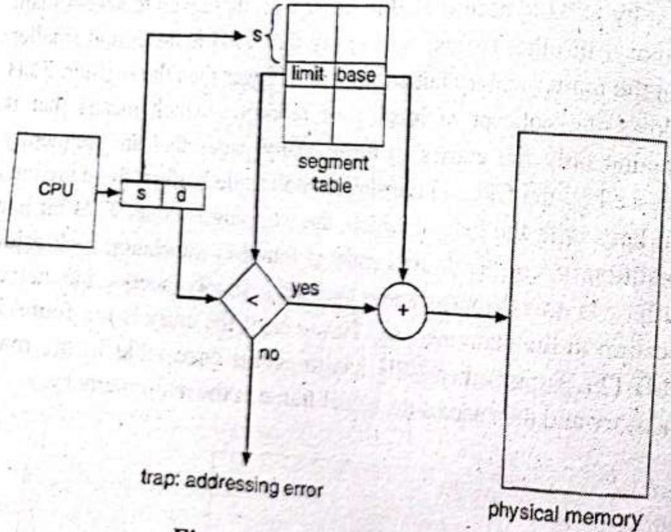


Figure 4.11: Segmentation

Advantages of segmentation:

1. Paging reduces internal fragmentation, but still suffer from external fragmentation.
2. Less overhead
3. The segment table is of lesser size as compared page table in paging.

Drawbacks of segmentation:

1. It can have external fragmentation.
2. It is difficult to allocate contiguous memory to variable sized partition.
3. It is costly memory management algorithms.

4.5.2.3 Paged Segmentation (Combined Approach)

Both paging and segmentation have their advantages and disadvantages. So, it is better to combine these two schemes to improve on each. This combined scheme is "Page the segments". Each segment in this scheme is divided into pages and each segment maintains a page table. So, the logical address is divided into 3 parts: (S, P, D) where S is the segment number, P is the page number and D is the offset or displacement.

Comparison between Paging and Segmentation.

Paging	Segmentation
1. In paging scheme, the main memory is partitioned into frames (or blocks).	1. In segmentation scheme, the main memory is partitioned into segments.
2. The logical address space is divided into pages by the compiler or . memory management unit (MMU).	2. The logical address space divided into segments as specified by the programmer.
3. This scheme suffers from internal fragmentation or page breaks.	3. This scheme suffers from external fragmentation.
4. The operating system maintains a free frames list; there is no need to search for free frames.	4. The operating system maintains the particulars of available memory.
5. The operating system maintains a page map table for mapping between frames and pages.	5. The operating system maintains a segment map table for mapping purpose.

Paging	Segmentation
6. This scheme does not support the user's view of memory.	6. This scheme supports the user's view of memory:
7. Processor uses the page number and displacement to calculate absolute address (p, d).	7. Processor uses the segment number and displacement to calculate the absolute address (s, d).

4.5.2.4 Demand Paging

It is a scheme in which a page is not loaded into the main memory from the secondary memory, until it is needed. So, in demand paging, pages are loaded only on demand and not in advance. The advantage here is that lesser I/O is needed, less memory is needed, faster response and more users serviced. In this scheme, rather than swapping the entire program in memory, only those pages are swapped which are required currently by the system.

Implementation of demand paging

To implement demand paging, it is necessary for the OS to keep track of which pages are currently in use. The page map table (PMT) contains an entry bit for each virtual page of the related process.

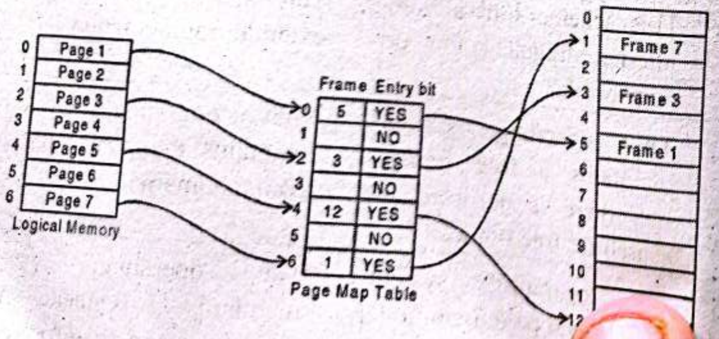


Figure 4.12: Implementation of demand pagin

If a program tries to access a page that was not swapped in memory, then *page fault* occurs. When the running program experiences a page fault, it must be suspended until the missing page is swapped in main memory.

4.5.2.5 Thrashing

Thrashing is when the page fault and swapping happens very frequently at a higher rate, and then the operating system has to spend more time swapping these pages. This state in the operating system is known as thrashing. Because of thrashing, the CPU utilization is going to be reduced or negligible.

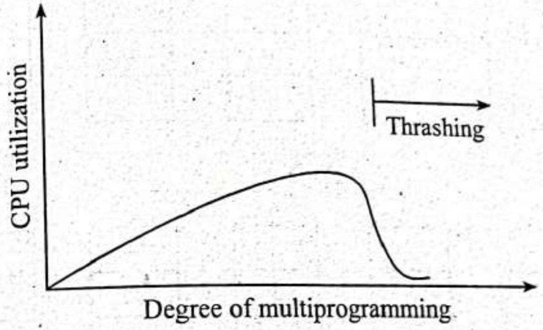


Figure 4.13: Thrashing

Causes of thrashing:

1. **High degree of multiprogramming:** The CPU scheduler sees the decreasing CPU utilization and increases the degree of multiprogramming as a result. The new process tries to get started by taking frames from running processes, causing more page faults and a longer queue for the paging device. As a result, CPU utilization drops even further, and the CPU scheduler tries to increase the degree of multiprogramming even more.

2. **Lack of frames:** If a process has less number of frames then less pages of that process will be able to reside in memory and hence it would result in more frequent swapping. This may lead to thrashing. Hence sufficient amounts of frames must be allocated.

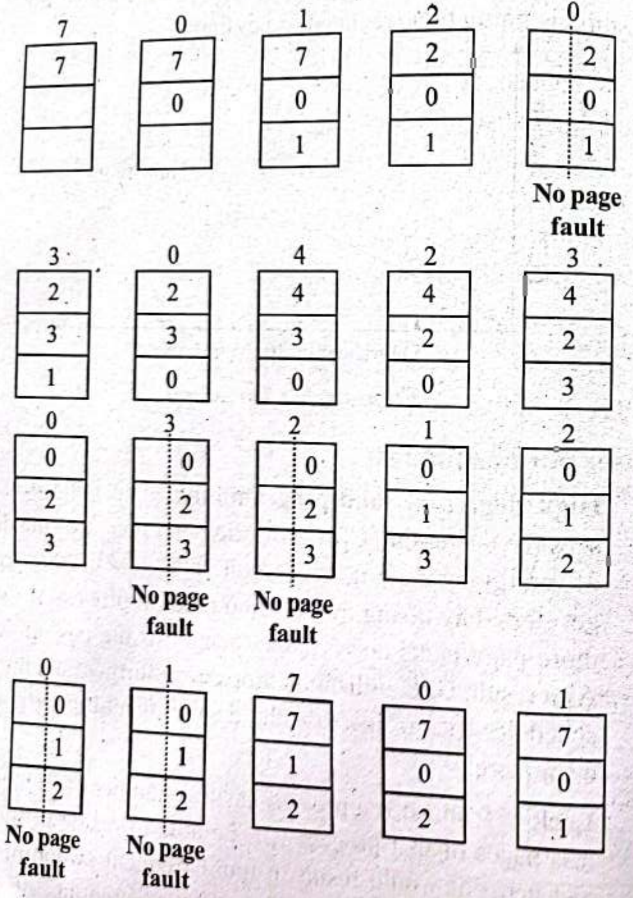
ANSWERS TO SOME IMPORTANT QUESTIONS

1. **Consider the following page reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1. How many page faults would occur for the FIFO replacement algorithm having three frames?**

Solution:

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

No. of frames: 3



Total no. of page faults = 15

Total no. of no page faults = 5

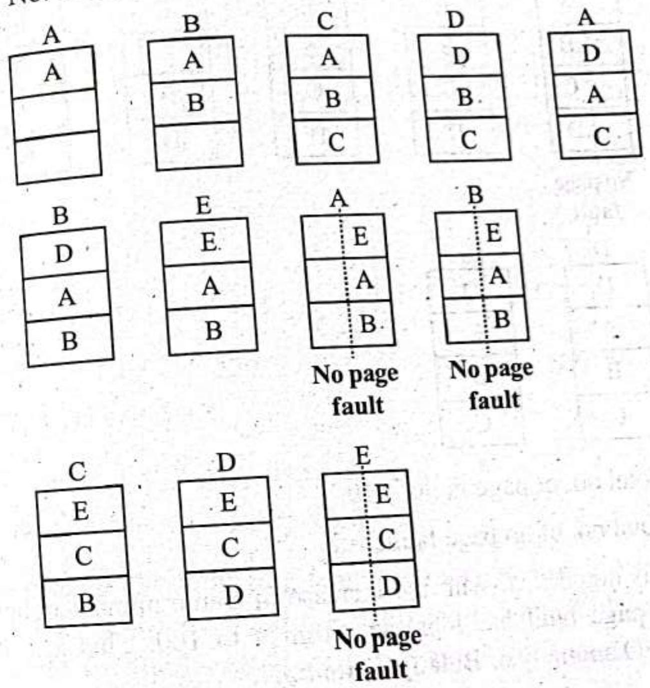
2. **Consider the page reference string: A B C D A B E A B C D E. How many page faults would occur for the FIFO page replacement algorithm having three frames and four frames?**

Solution:

Using three frames:

Reference string: A B C D A B E A B C D E

No. of frames: 3

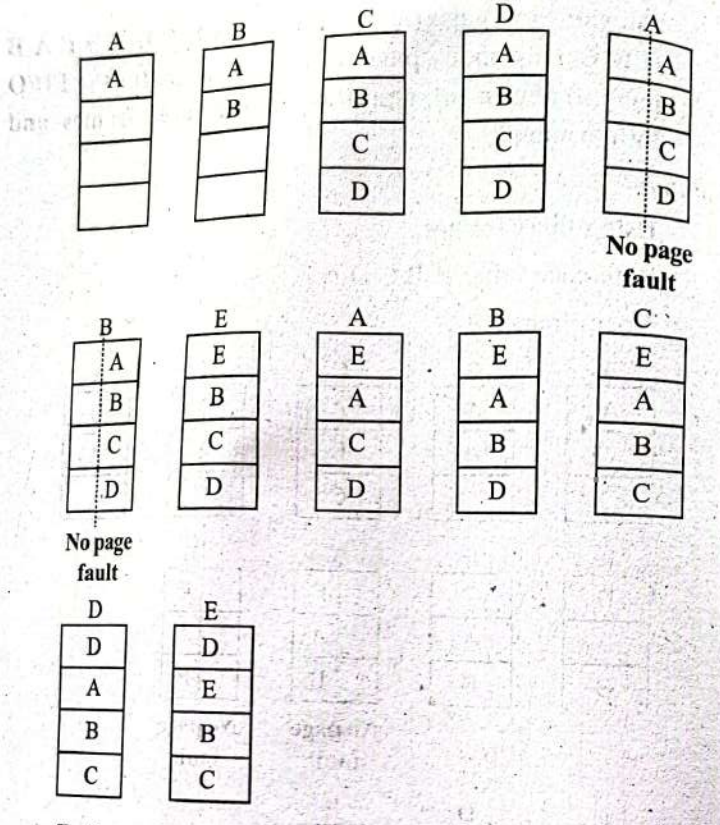


Total no. of page faults = 9

Total no. of no page faults = 3

Using four frames:

Now, no. of frames = 4



Total no. of page faults = 10

Total no. of no page faults = 2

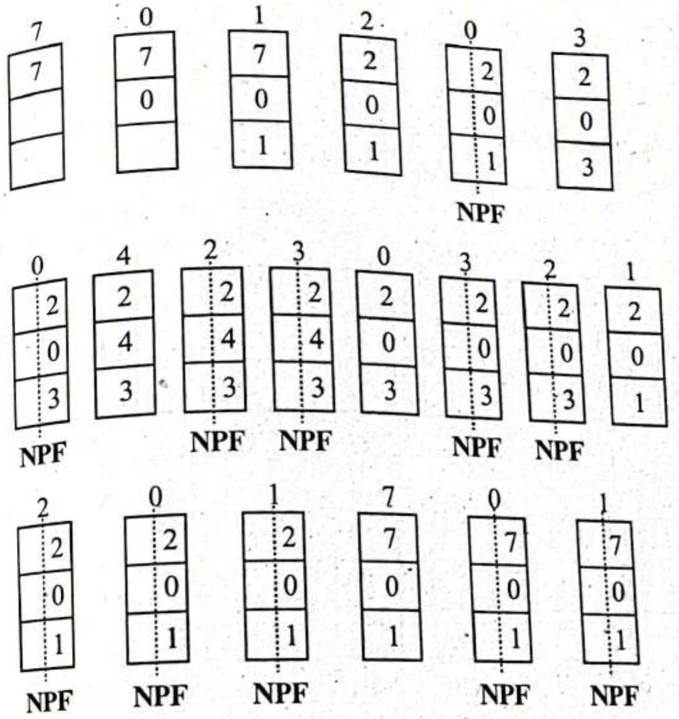
It is noted that, with the increase in frame number, number of page fault has increased from 9 to 100. This is called FIFO anomaly or **Belady's anomaly.**

3. **Consider the following page reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1. How many page faults would occur for the optimal page replacement algorithm having three frames.**

Solution:

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

No. of frames = 3



NPF = No page fault

Total number of page fault = 9

Total number of No Page fault = 11

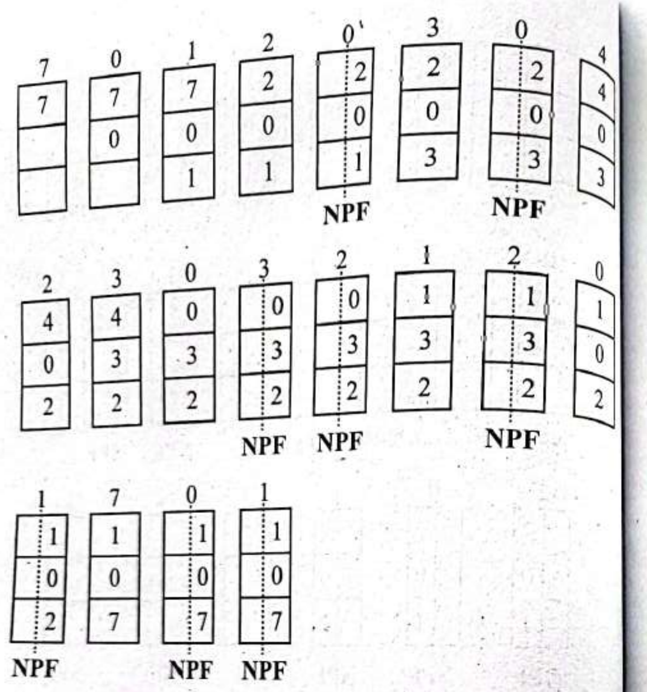
4. **Consider the following page reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1. How many page faults would occur for the Least Recently Used (LRU) having three frames.**

Solution:

Example:

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

No. of frames: 3

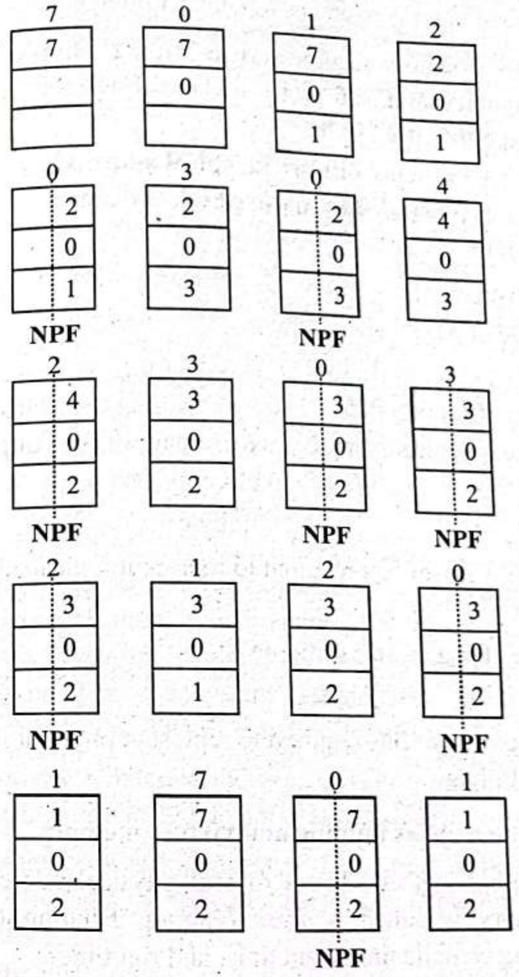


Total no. of page faults = 12

Total no. of no page faults = 8

5. **Consider the following page reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1 . How many page faults would occur for the Least Frequently Used (LFU) having three frames.**

Solution:



No. of page faults = 13

6. **Differentiate compaction and coalescing technique.**

Ans: The process of merging adjacent holes to form a single larger hole is called coalescing. By coalescing holes, the larger possible contiguous blocks of storage can be obtained.

Similarly, the technique of storage compaction involves moving all occupied areas of storage to one end. This creates a single large free storage hole instead of the numerous small holes common in variable partition multiprogramming.

7. **On a simple paging system with 2^{24} bytes of physical memory and 256 pages of logical address space and a page size of 2^{10} bytes.**
   a. **How many bits are in logical address?**
   b. **How many bits are in physical frame?**

Solution:

a. Given,

Size of page = size of frame

$= 2^{10}$ bytes $= 2^n$

No. of pages = 256

Size of logical memory = No. of pages × size of page

$= 256 \times 2^{10}$

$= 2^{18}$ bytes

Hence, no. of bits required to represent logical address space = m = 18 bits

b. Size of page = size of frame

$= 2^{10}$ bytes

Hence, no. of bits required to represent physical frame size = n = 10 bits

8. **Define address binding and virtual memory.**

Ans: Mapping logical address to real physical addresses in the memory is called *address binding*. Binding takes place during compile time, load time, and run time.

Virtual memory is a storage mechanism which offers user an illusion of having a very big main memory. It is done by treating a part of secondary memory as the main memory.

Chapter 5 File System

5.1 Introduction

A file management system is a set of system software. They provide services to users and applications in the use of files, including file access, directory maintenance, and access control.

5.2 File

File is a collection of related information defined by its creator. A file is an abstraction mechanism. It provides a way to store information on the disk and read it back later. File is used to represent and organize the system's non-volatile storage resources, including hard disks, cd-roms, and optical disks.

5.2.1 File System

Files are managed by the operating system. The part of the operating system dealing with how they are structured, named, accessed, used, protected, implemented, and managed is known as the file system. The file system permits users to create data collections, called files, with desirable properties, such as:

- **Long-term existence**: Files are stored on disk or other secondary storage and do not disappear when a user logs off.
- **Sharable between processes**: Files have names and can have associated access permissions that permit controlled sharing.
- **Structure:** Depending on the file system, a file can have an internal structure that is convenient for particular applications. In addition, files can be organized into a hierarchical or more complex structure to reflect the relationships among files.

5.2.2 Files Naming

When a process creates a file, it gives the file a name. When the process terminates, the file continues to exist and can be accessed by other processes using its name. It is the mechanism to store and retrieve information from the disk. It represents programs (both source and object) and data.

Naming conventions

- We can use set of a fixed number of characters (letters, digits, special characters)
- We should take account for case sensitivity

  For example: a UNIX system can have all of the following as three distinct files: maria, Maria, and MARIA. In MS. DOS, all these names refer to the same file.

File extension

Many operating systems support two-part file names, with the two parts separated by a period, as in prog.c. The part following the period is called the **file extension.**

5.2.3 File Structure

Files can be structured in any of several ways. Three common possibilities are as follows:

- **Byte sequence**

  The file is an unstructured sequence of bytes. In effect, the operating system does not know or care what is in the file. All it sees are bytes. Any meaning must be imposed by user-level programs. Both UNIX and Windows use this approach. It provides maximum flexibility but minimal support. It is advantages to users who want to define their own semantics on files.

- **Record sequence**

  In this model, a file is a sequence of fixed-length records, each with some internal structure. Central to the idea of a file being a sequence of records is the idea that the read operation re-turns one record and the write operation overwrites or appends one record. It is used in cp/m with a 128-character record.

- **Tree**

  In this organization, a file consists of a tree of records, not necessarily all the same length, each containing a key field in a fixed position in the record. The tree is sorted on the key field, to allow rapid searching for a particular key. It is useful for searching. It is used in some mainframes.

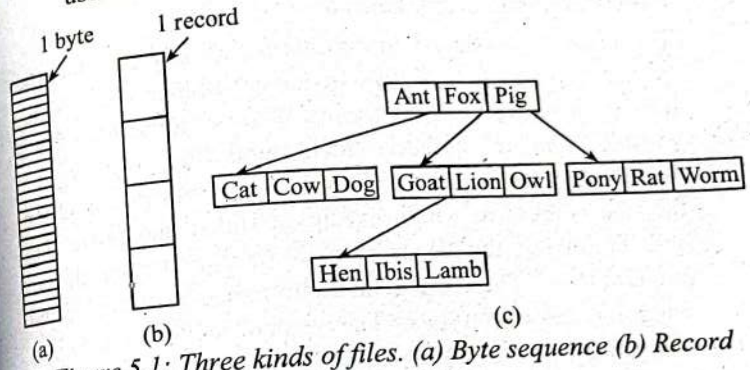


Figure 5.1: Three kinds of files. (a) Byte sequence (b) Record sequence (c) Tree

5.2.4 File Types

Many operating systems support three types of files:

1. **Regular files**

   Regular files are the ones that contain user information. Regular files may contain ASCII characters, binary data, executable program binaries, program input or output. Directories are system files for maintaining the structure of the file system.

   There are following types of regular file:

   - **Text files (ASCII)**

     They can be displayed and printed as is, and they can be edited with any text editor. Furthermore, if large numbers of programs use ASCII files for input and output, it is easy to connect the output of one program to the input of another, as in shell pipelines. It is useful for interprocess communication via pipes in Unix.

- **Binary files**

  They are in binary and not easily readable. Listing them on the printer gives an incomprehensible listing full of random junk. Usually, they have some internal structure known to programs that use them depending upon the type of file (executable or archive)

2. **Executable file (a.out format)**

   The operating system will execute a file only if it has the proper format. It has five sections: header, text, data, relocation bits, and symbol table. The header starts with a so-called magic number, identifying the file as an executable. Then come the sizes of the various pieces of the file, the address at which execution starts, and some flag bits. Following the header are the text and data of the program itself. These are loaded into memory and relocated using the relocation bits. The symbol table is used for debugging.

3. **Archive file**

   It consists of a collection of library procedures (modules) compiled but not linked. Each one is prefaced by a header telling its name, creation date, owner, protection code, and size. Just as with the executable file, the module headers are full of binary numbers. Copying them to the printer would produce complete gibberish.

5.2.5 File Access

The information in the file can be accessed in several ways as follows:

- **Sequential access**

  The simplest access method is Sequential Access. Early operating systems provided only sequential access. Information in the file is processed in order, one record after the other. In these systems, a process could read all the bytes or records in a file in order, starting at the beginning, but could not skip around and read them out of order. Sequential files were convenient when the storage medium was magnetic tape rather than disk.

- **Random access**

  Files whose bytes or records can be read in any order are called random-access files. They are required by many applications. For example, database systems. If an airline customer calls up and wants to reserve a seat on a particular flight, the reservation program must be able to access the record for that flight without having to read the records for thousands of other flights first.

5.2.6 File Attributes

Every file has a name and its data. In addition, all operating systems associate other information with each file, for example, the date and time the file was last modified and the file's size. We will call these extra items the file's attributes. Some people call them metadata.

***Table 5.1:** Some possible file attributes*

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access

Attribute	Meaning
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file was last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

5.2.7 File Operations

Files exist to store information and allow it to be retrieved later. Different systems provide different operations to allow storage and retrieval. Below are the most common system calls relating to files.

- **Create:** The file is created with no data. The purpose of the call is to announce that the file is coming and to set some of the attributes.
- **Delete:** When the file is no longer needed, it has to be deleted to free up disk space. There is always a system call for this purpose.
- **Open:** Before using a file, a process must open it. The purpose of the open call is to allow the system to fetch the attributes and list of disk addresses into main memory for rapid access on later calls.
- **Close:** When all the accesses are finished, the attributes and disk addresses are no longer needed, so the file should be closed to free up internal table space.
- **Read:** Data are read from file. Usually, the bytes come from the current position. The caller must specify how many data are needed and must also provide a buffer to put them in.
- **Write:** Data are written to the file again, usually at the current position. If the current position is the end of the file, the file's size in-creases. If the current position is in the middle of the file, existing data are overwritten and lost forever.
- **Append:** This call is a restricted form of write. It can add data only to the end of the file. Systems that provide a minimal set of system calls rarely have append, but many systems provide multiple ways of doing the same thing, and these systems sometimes have append.
- **Seek:** This call repositions the file pointer to a specific place in the file. After this call has completed, data can be read from, or written to, that position.
- **Get attributes:** Processes often need to read file attributes to do their work. For example, the UNIX make program is commonly used to manage software development projects consisting of many source files. When make is called, it examines the modification times of all the source and object files and arranges for the minimum number of compilations required to bring everything up to date. To do its job, it must look at the attributes, namely, the modification times.
- **Set attributes:** Some of the attributes are user settable and can be changed after the file has been created. This system call makes that possible. The protection-mode information is an obvious example. Most of the flags also fall in this category.
- **Rename:** It frequently happens that a user needs to change the name of an existing file. This system call makes that possible.

5.2.8 File Management Systems

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 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. Following are 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

5.2.9 File System Architecture

Different systems will be organized differently, but the organization given below is reasonably representative. At the lowest level, device drivers communicate directly with peripheral devices or their controllers or channels. A device driver is responsible for starting I/O operations on a device and processing the completion of an I/O request. For file operations, the typical devices controlled are disk and tape drives. Device drivers are usually considered to be part of the operating system.

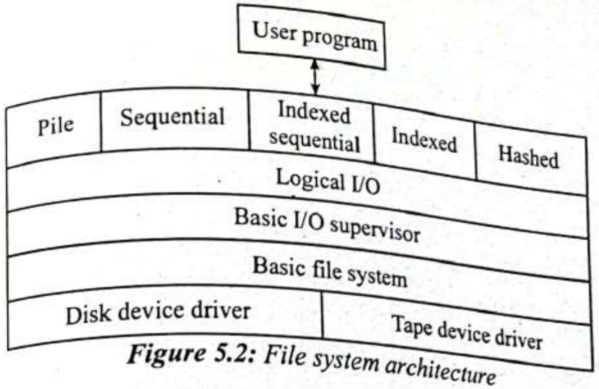


Figure 5.2: *File system architecture*

5.2.10 File Organization

File organization refers to the logical structuring of the records as determined by the way in which they are accessed. The physical organization of the file on secondary storage depends on the blocking strategy and the file allocation strategy. In choosing a file organization, several criteria are important:

1. Short access time
2. Ease of update
3. Economy of storage
4. Simple maintenance
5. Reliability

The number of alternative file organizations that have been implemented or just proposed is unmanageably large, even for a book devoted to file systems. We will outline five fundamental organizations. Most structures used in actual systems either fall into one of these categories, or can be implemented with a combination of these organizations. The five organizations are as follows:

1. The pile
2. The sequential file
3. The indexed sequential file
4. The indexed file
5. The direct, or hashed, file

5.2.11 File Directories

It is the system files for maintaining the structure of the file system. It is the binary file containing a list of files contained in it (including other directories). Directories may contain any kind of files, in any combination and refer to directory itself and its parent directory.

Contents:

Associated with any file management system and collection of files is a file directory. The directory contains information about the files, including attributes, location, and ownership.

Basic Information

- **File Name:** Name as chosen by creator (user or program). Must be unique within a specific directory
- **File Type:** For example: text, binary, load module, etc.
- **File Organization:** For systems that support different organizations

Address Information

- **Volume:** Indicates device on which file is stored
- **Starting Address:** Starting physical address on secondary storage (e.g., cylinder, track, and block number on disk)
- **Size Used:** Current size of the file in bytes, words, or blocks
- **Size Allocated:** The maximum size of the file

Access Control Information

- **Owner:** User who is assigned control of this file. The owner may be able to grant/deny access to other users and to change these privileges.
- **Access Information:** A simple version of this element would include the user's name and password for each authorized user.
- **Permitted Actions:** Controls reading, writing, executing, and transmitting over a network

Usage Information

- **Date Created:** When file was first placed in directory
- **Identity of Creator:** Usually but not necessarily the current owner
- **Date Last Read Access:** Date of the last time a record was read
- **Identity of Last Reader:** User who did the reading
- **Date Last Modified:** Date of the last update, insertion, or deletion
- **Identity of Last Modifier:** User who did the modifying
- **Date of Last Backup:** Date of the last time the file was backed up on another storage medium
- **Current Usage:** Information about current activity on the file, such as process or processes that have the file open, whether it is locked by a process

  Directories can be created by mkdir and deleted by rmdir, if empty. Non-empty directories can be deleted by rm -r. In directory, each entry made up of a file-inode pair used to associate inodes and directory locations

Character-special files

- It allows the device drivers to perform their own i/o buffering. It is used for unbuffered data transfer to and from a device. Character special file generally have names beginning with r (for raw), such as /dev/rsd0a.

Block-special files

- Block special files expect the kernel to perform buffering for them. It is used for devices that handle i/o in large chunks, known as blocks.

Structure

- The simplest form of structure for a directory is that of a list of entries, one for each file. This structure could be represented by a simple sequential file, with the name of the file serving as the key. In some earlier single-user systems, this technique has been used. However, it is inadequate when multiple users share a system and even for single users with many files.

To understand the requirements for a file structure, it is helpful to consider the types of operations that may be performed on the directory:

- **Search:** When a user or application references a file, the directory must be searched to find the entry corresponding to that file.
- **Create file:** When a new file is created, an entry must be added to the directory.
- **Delete file:** When a file is deleted, an entry must be removed from the directory.
- **List directory:** All or a portion of the directory may be requested. Generally, this request is made by a user and results in a listing of all files owned by that user, plus some of the attributes of each file (e.g., type, access control information, usage information).
- **Update directory:** Because some file attributes are stored in the directory, a change in one of these attributes requires a change in the corresponding directory entry.

To keep track of files, file systems normally have directories or folders, which are themselves files. We will discuss directories, their organization, their properties, and the operations that can be performed on them.

Levels of Directory

1. **Single directory for all the users**

   It is used in most primitive systems. It can cause conflicts and confusion when large multiuser system is considered hence may not be appropriate for any large multiuser system.

2. **One directory per user**

   This system eliminates name confusion across users as there are one directory per user. But it may not be satisfactory if users have many files.

3. **Hierarchical directories**

   It has a tree-like structure where root directory sits at the top of the tree. All directories spring out of the root. It allows logical grouping of files. In this system, every process can have its own working directory to avoid affecting other processes. It is used in most of the modern systems

Information in directory entry

- **File name:** It is the only information kept in human readable form.
- **File type:** It is needed for those systems that support different file types.
- **Location:** Pointer to the device and location of file on that device
- **Size:** Current size of the file May also include the maximum possible size for the file
- **Current position:** Pointer to the current read/write position in the file
- **Protection:** Access control information
- **Usage count:** Number of processes currently using the file
- **Time, date, and process identification:** May be kept for creation, last modification, and last use, Useful for protection and usage monitoring

5.2.12 Path Names

When the file system is organized as a directory tree, some way is needed for specifying file names. Two different methods are commonly used they are:

- **Absolute path name:** each file is given an absolute path name consisting of the path from the root directory to the file. As an example, the path /usr/ast/mailbox means that the root directory contains a subdirectory usr, which in turn contains a subdirectory ast, which contains the file mailbox. Absolute path names always start at the root directory and are unique.

- **Relative path name:** This is used in conjunction with the concept of the working directory (also called the current directory). A user can designate one directory as the current working directory, in which case all path names not beginning at the root directory are taken relative to the working directory. For example, if the current working directory is /usr/ast, then the file whose absolute path is /usr/ast/mailbox can be referenced simply as mailbox.

5.2.13 File Sharing

In a multiuser system, there is almost always a requirement for allowing files to be shared among a number of users. Two issues arise: access rights and the management of simultaneous access.

Access Rights

The file system should provide a flexible tool for allowing extensive file sharing among users. The file system should provide a number of options so the way in which a particular file is accessed can be controlled. Typically, users or groups of users are granted certain access rights to a file.

The following list is representative of access rights that can be assigned to a particular user for a particular file:

- **None:** The user may not even learn of the existence of the file, much less access it.
- **Knowledge:** The user can determine that the file exists and who its owner is.
- **Execution:** The user can load and execute a program but cannot copy it. Proprietary programs are often made accessible with this restriction.
- **Reading:** The user can read the file for any purpose, including copying and execution.
- **Appending:** The user can add data to the file, often only at the end, but cannot modify or delete any of the file's contents.

- **Updating:** The user can modify, delete, and add to the file's data.
- **Changing protection:** The user can change the access rights granted to other users. Typically, this right is held only by the owner of the file. In some systems, the owner can extend this right to others.
- **Deletion:** The user can delete the file from the file system.

Simultaneous Access

When access is granted to append or update a file to more than one user, the operating system or file management system must enforce discipline. A brute-force approach is to allow a user to lock the entire file when it is to be updated. A finer grain of control is to lock individual records during update. Essentially, this is the readers/writers problem. Issues of mutual exclusion and deadlock must be addressed in designing the shared access capability.

5.3 File System Implementation

Users are concerned with how files are named, what operations are allowed on them, what the directory tree looks like, and similar interface issues. Implementers are interested in how files and directories are stored, how disk space is managed, and how to make everything work efficiently and reliably. In the following sections we will examine a number of these areas to see what the issues and trade-offs are.

5.3.1 File System Layout

File systems are stored on disks. Most disks can be divided up into one or more partitions, with independent file systems on each partition. Sector 0 of the disk is called the MBR (Master Boot Record) and is used to boot the computer. The end of the MBR contains the partition table. This table gives the starting and ending addresses of each partition. One of the partitions in the table is marked as active. When the computer is booted, the BIOS reads in and executes the MBR.

The first thing the MBR program does is locate the active partition, read in its first block, which is called the boot block, and execute it. The program in the boot block loads the operating system contained in that partition. For uniformity, every partition starts with a boot block, even if it does not contain a bootable operating system. Besides, it might contain one in the future. Other than starting with a boot block, the layout of a disk partition varies a lot from file system to file system. The first one is the superblock. It contains all the key parameters about the file system and is read into memory when the computer is booted or the file system is first touched. Typical information in the superblock includes a magic number to identify the file-system type, the number of blocks in the file system, and other key administrative information. Next might come information about free blocks in the file system, for example in the form of a bitmap or a list of pointers. This might be followed by the i-nodes, an array of data structures, one per file, telling all about the file. After that might come the root directory, which contains the top of the file-system tree. Finally, the remainder of the disk contains all the other directories and files.

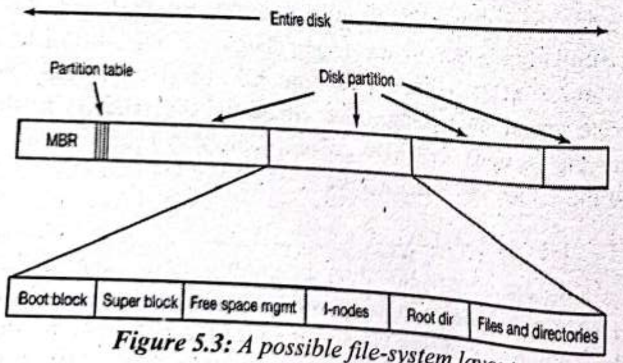


Figure 5.3: A possible file-system layout

5.3.2 Implementing Files

The most important issue in implementing file storage is keeping track of which disk blocks go with which file. Various methods are used in different operating systems. In this section, we will examine a few of them.

5.3.2.1 Contiguous Allocation

The simplest allocation scheme is to store each file as a contiguous run of disk blocks. Thus, on a disk with 1-KB blocks, a 50-KB file would be allocated 50 consecutive blocks. With 2-KB blocks, it would be allocated 25 consecutive blocks. In the example below, the first 40 disk blocks are shown, starting with block 0 on the left. Initially, the disk was empty. Then a file A, of length four blocks, was written to disk starting at the beginning (block 0). After that a six-block file, B, was written starting right after the end of file A. Note that each file begins at the start of a new block, so that if file A was really 3½ blocks, some space is wasted at the end of the last block. In the figure, a total of seven files are shown, each one starting at the block following the end of the previous one. Shading is used just to make it easier to tell the files apart. It has no actual significance in terms of storage.

Advantages:

- Simplest allocation technique
- Simple to implement; files can be accessed by knowing the first block of the file on the disk
- Improves performance as the entire file can be read in one operation

Disadvantages:

- File size may not be known in advance
- Disk fragmentation which can be partially solved by compaction

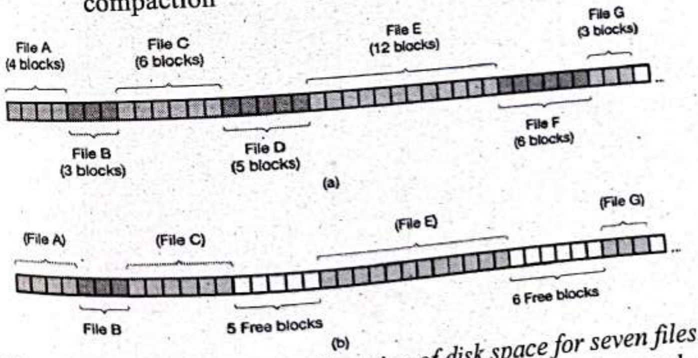


Figure 5.4: (a) Contiguous allocation of disk space for seven files. (b) The state of the disk after files D and F have been removed.

5.3.2.2 Linked list allocation

The second method for storing files is to keep each one as a linked list of disk blocks. The first word of each block is used as a pointer to the next one. The rest of the block is for data. Unlike contiguous allocation, every disk block can be used in this method. No space is lost to disk fragmentation (except for internal fragmentation in the last block). Also, it is sufficient for the directory entry to merely store the disk address of the first block. The rest can be found starting there. On the other hand, although reading a file sequentially is straightforward, random access is extremely slow. To get to block n, the operating system has to start at the beginning and read the n − 1 blocks prior to it, one at a time. Clearly, doing so many reads will be painfully slow. Also, the amount of data storage in a block is no longer a power of two because the pointer takes up a few bytes. While not fatal, having a peculiar size is less efficient because many programs read and write in blocks whose size is a power of two. With the first few bytes of each block occupied by a pointer to the next block, reads of the full block size require acquiring and concatenating information from two disk blocks, which generates extra overhead due to the copying.

Advantages:

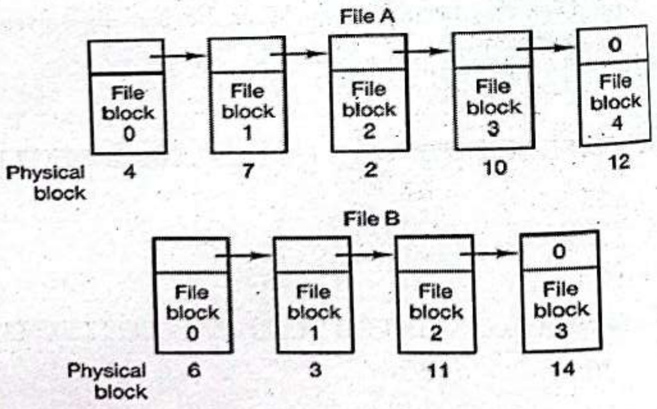


Figure 5.5: Linked list allocation

- Only the address of the first block appears in the directory entry
- Every disk block can be used
- No space is lost to disk fragmentation
- No disk fragmentation

Disadvantage:

- Random access is extremely slow
- Data in a block is not a power of 2 (to accommodate the link to next block)

5.3.2.3 Linked list allocation using an index

Both disadvantages of the linked-list allocation can be eliminated by taking the pointer word from each disk block and putting it in a table in memory. In both figures, we have two files. File A uses disk blocks 4, 7, 2, 10, and 12, in that order, and file B uses disk blocks 6, 3, 11, and 14, in that order. Using the table, we can start with block 4 and follow the chain all the way to the end. The same can be done starting with block 6. Both chains are terminated with a special marker (e.g.,−1) that is not a valid block number. Such a table in main memory is called a FAT (File Allocation Table). Using this organization, the entire block is available for data. Furthermore, random access is much easier. Although the chain must still be followed to find a given offset within the file, the chain is entirely in memory, so it can be followed without making any disk references. Like the previous method, it is sufficient for the directory entry to keep a single integer (the starting block number) and still be able to locate all the blocks, no matter how large the file is. The primary disadvantage of this method is that the entire table must be in memory all the time to make it work. With a 1-TB disk and a 1-KB block size, the table needs 1 billion entries, one for each of the 1 billion disk blocks. Each entry has to be a minimum of 3 bytes. For speed in lookup, they should be 4 bytes. Thus, the table will take up 3 GB or 2.4 GB of main memory all the time, depending on whether the system is optimized for space or time. Not wildly practical. Clearly the FAT idea does not scale well to large disks. It was the original

MS-DOS file system and is still fully supported by all versions of Windows though.

Advantages:

- Memory contains a table pointing to each disk block called a **FAT (File Allocation Table)**.
- The entire block is available for data (solution of 2^{nd} problem).
- Random access is easier because the chain must still be followed to find without making any disk references.
- Large file can be access easily

Disadvantage:

- The entire table must be in memory all the time to make it work.
- With a 20-GB disk and a 1-KB block size, the table needs 20 million entries, one for each of the 20 million disk blocks. Each entry has to be a minimum of 3 bytes. For speed in lookup, they should be 4 bytes. Thus, the table will take up 60 MB or 80 MB of main memory all the time, depending on whether the system is optimized for space or time.

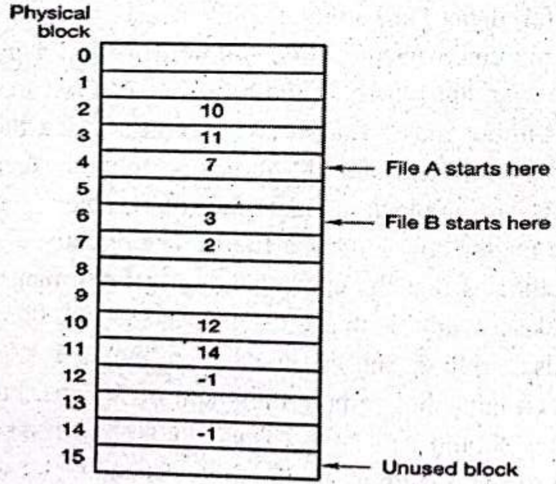


Figure 5.6: Linked list allocation using a file allocation table in main memory.

5.3.2.4 Inodes (Index node)

Our last method for keeping track of which blocks belong to which file is to associate with each file a data structure called an i-node (index-node), which lists the attributes and disk addresses of the file's blocks. A simple example is depicted in Figure below. Given the i-node, it is then possible to find all the blocks of the file. The big advantage of this scheme over linked files using an in-memory table is that the i-node need be in memory only when the corresponding file is open. If each i-node occupies n bytes and a maximum of k files may be open at once, the total memory occupied by the array holding the i-nodes for the open files is only kn bytes. Only this much space need be reserved in advance. This array is usually far smaller than the space occupied by the file table described in the previous section. The reason is simple. The table for holding the linked list of all disk blocks is proportional in size to the disk itself. If the disk has n blocks, the table needs n entries. As disks grow larger, this table grows linearly with them. In contrast, the i-node scheme requires an array in memory whose size is proportional to the maximum number of files that may be open at once. It does not matter if the disk is 100 GB, 1000 GB, or 10,000 GB. One problem with i-nodes is that if each one has room for a fixed number of disk addresses, what happens when a file grows beyond this limit? One solution is to reserve the last disk address not for a data block, but instead for the address of a block containing more disk-block addresses. Even more advanced would be two or more such blocks containing disk addresses or even disk blocks pointing to other disk blocks full of addresses.

Advantages:

- i-node need be in memory only when the corresponding file is open.
- the total memory occupied by the array holding the i-nodes for the open files is less.

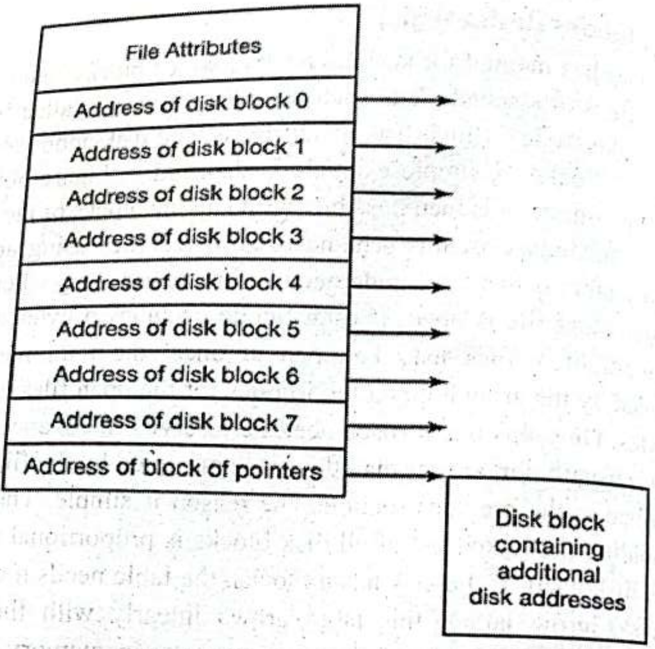


Figure 5.7: disk space allocation using Inodes

5.3.3 Implementing Directories

Before a file can be read, it must be opened. When a file is opened, the operating system uses the path name supplied by the user to locate the directory entry on the disk. The directory entry provides the information needed to find the disk blocks. Depending on the system, this information may be the disk address of the entire file (with contiguous allocation), the number of the first block (both linked-list schemes), or the number of the i-node. In all cases, the main function of the directory system is to map the ASCII name of the file onto the information needed to locate the data. A closely related issue is where the attributes should be stored. Every file system maintains various file attributes, such as each file's owner and creation time, and they must be stored somewhere. One obvious possibility is to store them directly in the directory entry. Some systems do precisely that. In this simple design, a directory consists of a list of fixed-size entries, one per file, containing a (fixed-length) file name, a structure of the file attributes, and one or more disk addresses (up to some maximum) telling where the disk blocks are. For systems that use i-nodes, another possibility for storing the attributes is in the i-nodes, rather than in the directory entries. In that case, the directory entry can be shorter: just a file name and an i-node number.

Two methods are used to keep the information of attributes of the file:

- Store at directory itself
- Store file name at directory and attribute using pointer

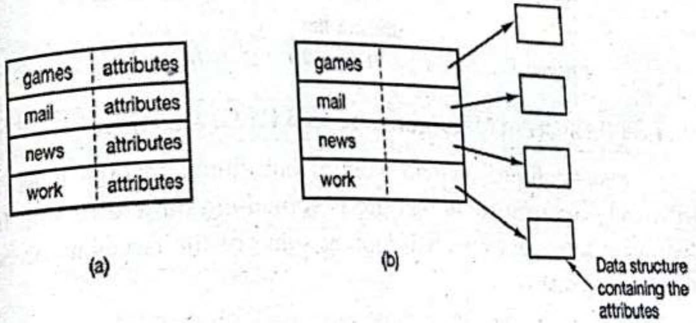


Figure 5.8: (a) A simple directory containing fixed-size entries with the disk addresses and attributes in the directory entry. (b) A directory in which each entry just refers to an i-node.

5.3.4 Shared Files

When several users are working together on a project, they often need to share files. As a result, it is often convenient for a shared file to appear simultaneously in different directories belonging to different users. Figure shows the file system again, only with one of C's files now present in one of B's directories as well. The connection between B's directory and the shared file is called a link. The file system itself is now a Directed Acyclic Graph, or DAG, rather than a tree. Having the file system be a DAG complicates maintenance.

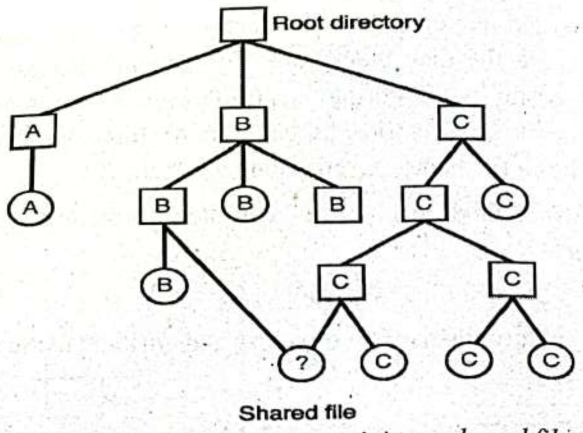


Figure 5.9: File system containing a shared file

5.4. File System Management and Optimization

Making the file system work is one thing; making it work efficiently and robustly in real life is something quite different. In the following sections we will look at some of the issues involved in managing disks.

5.4.1 Disk Management

Files are normally stored on disk, so management of disk space is a major concern to file-system designers. Two general strategies are possible for storing an n byte file: n consecutive bytes of disk space are allocated, or the file is split up into a number of (not necessarily) contiguous blocks. As we have seen, storing a file as a contiguous sequence of bytes has the obvious problem that if a file grows, it may have to be moved on the disk. The same problem holds for segments in memory, except that moving a segment in memory is a relatively fast operation compared to moving a file from one disk position to another. For this reason, nearly all file systems chop files up into fixed-size blocks that need not be adjacent.

5.4.2 File System Performance

Access to disk is much slower than access to memory because of seek time and latency time (Rotational). Reading a 32-bit memory word might take 10 nsec. Reading from a hard disk might proceed at 100 MB/sec, which is four times slower per 32-bit word, but to this must be added 5–10 msec to seek to the track and then wait for the desired sector to arrive under the read head.

Disk performance can be increase using following concept

- Block Cache or Buffer Cache
- Block Read Ahead
- Reduce Disk Arm Motion

- **Block Cache**

  The most common technique used to reduce disk accesses is the block cache or buffer cache. 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.

- **Block Read Ahead**

  A second technique for improving perceived file-system performance is to try to get blocks into the cache before they are needed to increase the hit rate.

- **Reduce Disk Arm motion**

  Caching and read ahead are not the only ways to increase file-system performance. Another important technique is to reduce the amount of disk-arm motion by putting blocks that are likely to be accessed in sequence close to each other, preferably in the same cylinder. When an output file is written, the file system has to allocate the blocks one at a time, on demand. If the free blocks are recorded in a bitmap, and the whole bitmap is in main memory, it is easy enough to choose a free block as close as possible to the previous block. With a free list, part of which is on disk, it is much harder to allocate blocks close together.

EXERCISE

1. *Explain i-node approach of file system implementation with its advantages and disadvantages.*
2. *Discuss various file allocation and access methods. Compare their advantages and disadvantages.*
3. What is file attribute? Write the difference between single level directory system and hierarchical directory system. Explain how OS manage free blocks of secondary storage.
4. In what ways is file system management similar to virtual memory management? What are the advantages and disadvantages of a contiguous file allocation scheme? Which file organization technique is most appropriate for tape storage? Why?
5. What is file system layout? Explain how OS manages free blocks of secondary storage.
6. Describe file allocation methods.

Chapter 6 I/O Management & Disk Scheduling

In addition to providing abstractions such as processes, address spaces, and files, an operating system also controls all the computer's I/O (Input/ Output) devices. It must issue commands to the devices, catch interrupts, and handle errors. It should also provide an interface between the devices and the rest of the system that is simple and easy to use. To the extent possible, the interface should be the same for all devices which we call device independence. All this is I/O management.

6.1 Principles of I/O Hardware

Different people look at I/O hardware in different ways. Electrical engineers look at it in terms of chips, wires, power supplies, motors, and all the other physical components that comprise the hardware. Programmers look at the interface presented to the software—the commands the hardware accepts, the functions it carries out, and the errors that can be reported back. We are concerned with programming I/O devices, not designing, building, or maintaining them, so our interest is in how the hardware is programmed, not how it works inside.

6.1.1 I/O Devices

I/O devices can be roughly divided into two categories:

- **Block devices:**

  A block device is one that stores information in fixed-size blocks, each one with its own address. Common block sizes range from 512 to 65,536 bytes. All transfers are in units of one or more entire (consecutive) blocks. The essential property of a block device is that it is possible to read or write each block independently of all the other ones. Hard disks, Blu-ray discs, and USB sticks are common block devices.

- ***Character devices:***
  A character device delivers or accepts a stream of characters, without regard to any block structure. It is not addressable and does not have any seek operation. Printers, network interfaces, mice (for pointing), rats (for psychology lab experiments), and most other devices that are not disk-like can be seen as character devices.

  This classification scheme is not perfect. Some devices do not fit in. Clocks, for example, are not block addressable nor do they generate or accept character streams. All they do is cause interrupts at well-defined intervals. Memory-mapped screens do not fit the model well either. Nor do touch screens, for that matter. Still, the model of block and character devices is general enough that it can be used as a basis for making some of the operating system software dealing with I/O device independent.

6.1.2 Device Controllers

I/O units often consist of a mechanical component and an electronic component. It is possible to separate the two portions to provide a more modular and general design. The electronic component is called the device controller or adapter. On personal computers, it often takes the form of a chip on the parent board or a printed circuit card that can be inserted into a (PCIe) expansion slot. The mechanical component is the device itself.

6.1.3 Memory-Mapped I/O

Each controller has a few registers that are used for communicating with the CPU. By writing into these registers, the operating system can command the device to deliver data, accept data, switch itself on or off, or otherwise perform some action. By reading from these registers, the operating system can learn what the device's state is, whether it is prepared to accept a new command, and so on. Each control register is assigned a unique memory address to which no memory is assigned. This system is called memory-mapped I/O. In most systems, the assigned addresses are at or near the top of the address space.

6.1.4 Direct Memory Access (DMA)

No matter whether a CPU does or does not have memory-mapped I/O, it needs to address the device controllers to exchange data with them. The CPU can request data from an I/O controller one byte at a time, but doing so wastes the CPU's time, so a different scheme, called DMA (Direct Memory Access) is often used. The detail of DMA is described in the subsequent topic.

6.2 Principal of I/O Software

First, we will look at goals of I/O Software and then at the different ways I/O can be done from the point of view of the operating system.

6.2.1 Goals of I/O Software

- **Device independence**

  What it means is that we should be able to write programs that can access any I/O device without having to specify the device in advance. For example: Read and Write from Disk, CD-ROM without modifying programs

- **Uniform naming**

  The name of a file or a device should simply be a string or an integer and not depend on the device in any way. In UNIX, all disks can be integrated in the file-system hierarchy in arbitrary ways so the user need not be aware of which name corresponds to which device.

- **Error handling**

  Errors should be handled as close to the hardware as possible. If the controller discovers a read error, it should try to correct the error itself if it can. If it cannot, then the device driver should handle it, perhaps by just trying to read the block again.

- **Synchronous read/write at application level**

  Most physical I/O is asynchronous—the CPU starts the transfer and goes off to do something else until the interrupt arrives. User programs are much easier to write if the I/O operations are blocking—after a read system call the program is automatically suspended until the data are available in the buffer. It is up to the operating system to make operations that are actually interrupt-driven look blocking to the user programs.

- **Buffering**

  Often data that come off a device cannot be stored directly in their final destination. So the data must be put into an output buffer in advance to decouple the rate at which the buffer is filled from the rate at which it is emptied, in order to avoid buffer underruns. Buffering involves considerable copying and often has a major impact on I/O performance

- **Sharable vs. dedicated devices**

  Some I/O devices, such as disks, can be used by many users at the same time. No problems are caused by multiple users having open files on the same disk at the same time. Other devices, such as printers, have to be dedicated to a single user until that user is finished. Then another user can have the printer. Having two or more users writing characters intermixed at random to the same page will definitely not work. Introducing dedicated (unshared) devices also introduces a variety of problems, such as deadlocks. Again, the operating system must be able to handle both shared and dedicated devices in a way that avoids problems.

6.2.2 I/O Handling Techniques

6.2.2.1 Programmed I/O

In programmed I/O, the processor keeps on scanning whether any device is ready for data transfer. If an I/O device is ready, the processor fully dedicates itself in transferring the data between I/O and memory. It transfers data at a high rate, but it can't get involved in any other activity during data transfer. This is the major drawback of programmed I/O.

6.2.2.2 Interrupt-Driven I/O

With programmed I/O, the processor has to wait a long time. The processor, while waiting, must repeatedly examine the status of the I/O module. As a result, the performance level of the entire system is degraded.

In Interrupt driven I/O, whenever the device is ready for data transfer, then it raises an interrupt to processor. Processor completes executing its ongoing instruction and saves its current state. It then switches to data transfer which causes a delay. Here, the processor doesn't keep scanning for peripherals ready for data transfer. But it is fully involved in the data transfer process.

6.2.2.3 Direct Memory Access

Interrupt-driven I/O, though more efficient than simple programmed I/O, still requires the active involvement of the processor to transfer data between memory and an I/O module, and any data transfer must traverse a path through the processor.

When large volumes of data are to be moved, a more efficient technique is required: direct memory access (DMA). DMA is the method of data transfer which takes places between I/O devices and Memory without the involvement of processor. The hardware unit that controls the activity of accessing memory directly is called a DMA controller. The DMA controller transfers the data in three modes: Burst Mode, Cycle Stealing Mode and Transparent Mode. Below we have the diagram of DMA controller that explains its working:

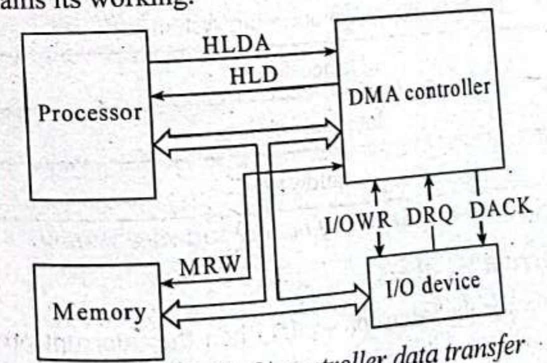


Figure 6.1: *DMA controller data transfer*

- Whenever an I/O device wants to transfer the data to or from memory, it sends the DMA request (DRQ) to the DMA controller. DMA controller accepts this DRQ and asks the CPU to hold for a few clock cycles by sending it the Hold request (HLD).
- CPU receives the Hold request (HLD) from DMA controller and relinquishes the bus and sends the Hold acknowledgement (HLDA) to DMA controller.
- After receiving the Hold acknowledgement (HLDA), DMA controller acknowledges I/O device (DACK) that the data transfer can be performed and DMA controller takes the charge of the system bus and transfers the data to or from memory.
- When the data transfer is accomplished, the DMA raise an interrupt to let know the processor that the task of data transfer is finished and the processor can take control over the bus again and start processing where it has left.

6.3 I/O Software Layers

I/O software is typically organized in four layers, as shown in Figure with description below. Each layer has a well-defined function to perform and a well-defined interface to the adjacent layers. The functionality and interfaces differ from system to system.

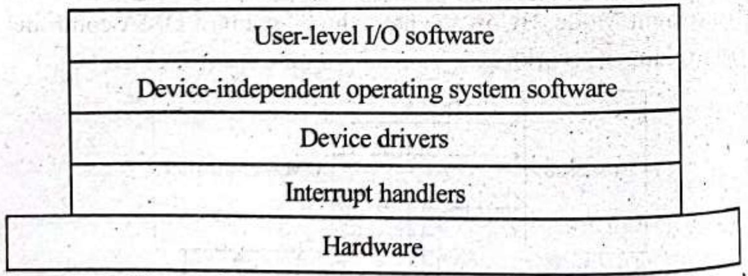


Figure 6.2: Layers of the I/O software system

1. **Interrupt Handlers**

   Whenever the interrupt occurs, then the interrupt procedure does whatever it has to in order to handle the interrupt.

2. **Device Drivers**

   Basically, device drivers is a device-specific code just for controlling the input/output device that are attached to the computer system.

3. **Device-Independent Input/Output Software**

   In some of the input/output software is device specific, and other parts of that input/output software are device-independent.

   The exact boundary between the device-independent software and drivers is device dependent, just because of that some functions that could be done in a device-independent way sometime be done in the drivers, for efficiency or any other reasons.

   Here are the list of some functions that are done in the device-independent software:

   - Uniform interfacing for device drivers
   - Buffering
   - Error reporting
   - Allocating and releasing dedicated devices
   - Providing a device-independent block size

4. **User-Space Input/Output Software**

   Generally, most of the input/output software is within the operating system (OS), and some small part of that input/output software consists of libraries that are linked with the user programs and even whole programs running outside the kernel.

6.3.1 Interrupt Handlers

The interrupt handler determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a return from interrupt instruction to return the CPU to the execution state prior to the interrupt. We say that the device controller raises an interrupt by asserting a signal on the interrupt request line, the CPU catches the interrupt and dispatches it to the interrupt handler, and the handler clears the interrupt by servicing the device. Interrupt driven I/O is described in the subsequent topic.

6.3.2 Device Drivers

Each I/O device attached to a computer needs some device-specific code for controlling it. This code, called the device driver, is generally written by the device's manufacturer and delivered along with the device. Since each operating system needs its own drivers, device manufacturers commonly supply drivers for several popular operating systems. Device drivers are normally positioned below the rest of the operating system, as is illustrated in figure below.

6.3.3 Device-Independent I/O Software

Although some of the I/O software is device specific, other parts of it are device independent. The exact boundary between the drivers and the device-independent software is system (and device) dependent, because some functions that could be done in a device-independent way may actually be done in the drivers, for efficiency or other reasons. The functions shown below are typically done in the device-independent software.

Uniform interfacing for device drivers
Buffering
Error reporting
Allocating and releasing dedicated devices
Providing a device-independent block size

Figure 6.3: Functions of the 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

- **Uniform interfacing for device drivers**

  A major issue in an operating system is how to make all I/O devices and drivers look more or less the same. One aspect of this issue is the interface between the device drivers and the rest of the operating system.

  In Figure (a) we illustrate a situation in which each device driver has a different interface to the operating system. What this means is that the driver functions available for the system to call differ from driver to driver. It might also mean that the kernel functions that the driver needs also differ from driver to driver. Taken together, it means that interfacing each new driver requires a lot of new programming effort.

  In contrast, in Figure (b), we show a different design in which all drivers have the same interface. Now it becomes much easier to plug in a new driver, providing it conforms to the driver interface. It also means that driver writers know what is expected of them. In practice, not all devices are absolutely identical, but usually there are only a small number of device types and even these are generally almost the same.

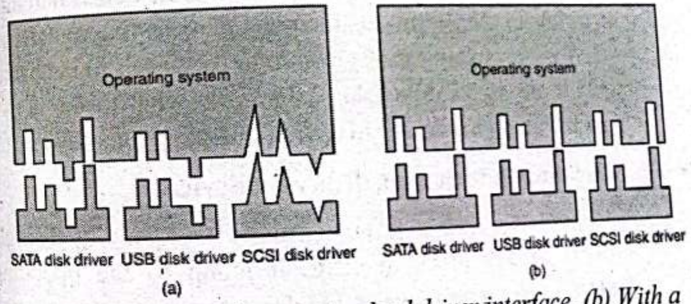


Figure 6.4: (a) Without a standard driver interface. (b) With a standard driver interface.

(Courtesy of Andrew S. Tanenbaum, Herbert Bos; *Modern Operating Systems*)

- **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.

- **Error reporting**

When error occurs, the operating system must handle them as best it can. Many errors are device specific and must be handled by the appropriate driver, but the framework for error handling is device independent.

One class of I/O errors is programming errors. These occur when a process asks for something impossible, such as writing to an input device (keyboard, scanner, mouse, etc.) or reading from an output device (printer, plotter, etc.). Other errors are providing an invalid buffer address or other parameter, and specifying an invalid device (e.g., disk 3 when the system has only two disks), and so on. The action to take on these errors is straightforward: just report back an error code to the caller.

Another class of errors is the class of actual I/O errors, for example, trying to write a disk block that has been damaged or trying to read from a camcorder that has been switched off. In these circumstances, it is up to the driver to determine what to do. If the driver does not know what to do, it may pass the problem back up to device-independent software.

- **Allocating and releasing dedicated devices**

Some devices, such as printers, can be used only by a single process at any given moment. It is up to the operating system to examine requests for device usage and accept or reject them, depending on whether the requested device is available or not. A simple way to handle these requests is to require processes to perform opens on the special files for devices directly. If the device is unavailable, the open fails. Closing such a dedicated device then releases it.

An alternative approach is to have special mechanisms for requesting and releasing dedicated devices. An attempt to acquire a device that is not available blocks the caller instead of failing. Blocked processes are put on a queue. Sooner or later, the requested device becomes available and the first process on the queue is allowed to acquire it and continue execution.

- **Providing a device-independent block size**

Different disks may have different sector sizes. It is up to the device-independent software to hide this fact and provide a uniform block size to higher layers, for example, by treating several sectors as a single logical block. In this way, the higher layers deal only with abstract devices that all use the same logical block size, independent of the physical sector size. Similarly, some character devices deliver their data one byte at a time (e.g., mice), while others deliver theirs in larger units (e.g., Ethernet interfaces). These differences may also be hidden.

6.3.4 User-Space I/O Software

Although most of the I/O software is within the operating system, a small portion of it consists of libraries linked together with user programs, and even whole programs running outside the kernel. System calls, including the I/O system calls, are normally made by library procedures. When a C program contains the call the library procedure write might be linked with the program and contained in the binary program present in memory at run time.

```
count = write(fd, buffer, nbytes);
```

In other systems, libraries can be loaded during program execution. Either way, the collection of all these library procedures is clearly part of the I/O system.

Another important category is the spooling system. Spooling is a way of dealing with dedicated I/O devices in a multiprogramming system.

6.4 Disk

6.4.1 Disk Hardware

Disks come in a variety of types. The most common ones are the magnetic hard disks. They are characterized by the fact that reads and writes are equally fast, which makes them suitable as secondary memory (paging, file systems, etc.). Arrays of these disks are sometimes used to provide highly reliable storage. For distribution of programs, data, and movies, optical disks (DVDs and Blu-ray) are also important. Solid-state disks are increasingly popular as they are fast and do not contain moving parts.

Magnetic Disks

Magnetic disks are organized into cylinders, each one containing as many tracks as there are heads stacked vertically. The tracks are divided into sectors, with the number of sectors around the circumference. The number of heads varies from 1 to about 16.

Modern disks are divided into zones with more sectors on the outer zones than the inner ones. Figure (a) illustrates a tiny disk with two zones. The outer zone has 32 sectors per track; the inner one has 16 sectors per track.

To hide the details of how many sectors each track has, most modern disks have a virtual geometry that is presented to the operating system. The software is instructed to act as though there are x cylinders, y heads, and z sectors per track. The controller then remaps a request for (x, y, z) onto the real cylinder, head, and sector. A possible virtual geometry for the physical disk of Figure (a) is shown in Figure (b). In both cases the disk has 192 sectors, only the published arrangement is different than the real one.

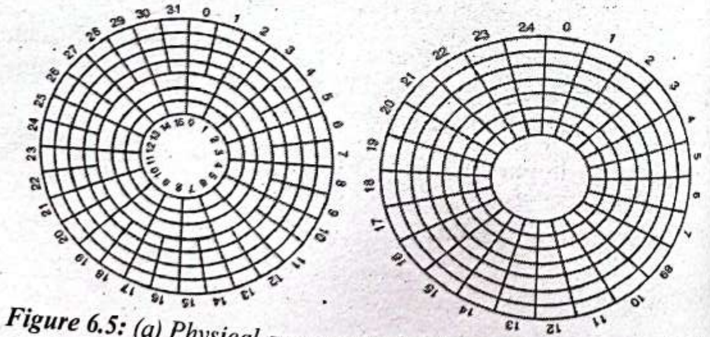


Figure 6.5: (a) Physical geometry of a disk with two zones. (b) A possible virtual geometry for this disk.

(Courtesy of Andrew S. Tanenbaum, Herbert Bos; *Modern Operating Systems*)

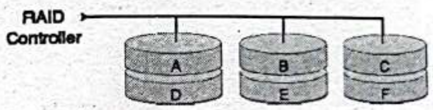
RAID

RAID or **R**edundant **A**rray of **I**ndependent **D**isks, is a technology to connect multiple secondary storage devices and use them as a single storage media. RAID consists of an array of disks in which multiple disks are connected together to achieve different goals (like: fault-tolerance and performance). RAID levels define the use of disk arrays.

Out of the seven RAID levels described, only four are commonly used: RAID levels 0, 1, 5, and 6.

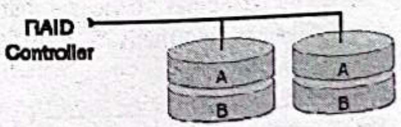
RAID Level 0

In this level, a striped array of disks is implemented. The data is broken down into blocks and the blocks are distributed among disks. Each disk receives a block of data to write/read in parallel. It enhances the speed and performance of the storage device. There is no parity and backup in Level 0.



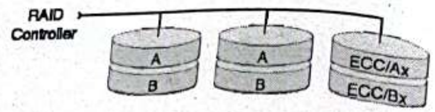
RAID Level 1

RAID 1 uses mirroring techniques. When data is sent to a RAID controller, it sends a copy of data to all the disks in the array. RAID level 1 is also called **mirroring** and provides 100% redundancy in case of a failure.



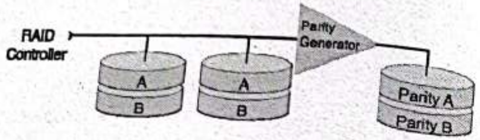
RAID Level 2

RAID 2 records Error Correction Code using Hamming distance for its data, striped on different disks. Like level 0, each data bit in a word is recorded on a separate disk and ECC codes of the data words are stored on a different set disk. Due to its complex structure and high cost, RAID 2 is not commercially available.



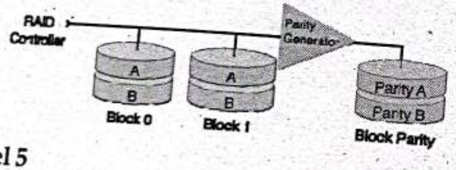
RAID Level 3

RAID 3 stripes the data onto multiple disks. The parity bit generated for data word is stored on a different disk. This technique makes it to overcome single disk failures.



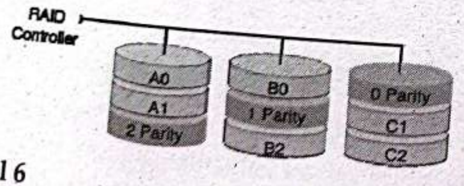
RAID Level 4

In this level, an entire block of data is written onto data disks and then the parity is generated and stored on a different disk. Note that level 3 uses byte-level striping, whereas level 4 uses block-level striping. Both level 3 and level 4 require at least three disks to implement RAID.



RAID Level 5

RAID 5 writes whole data blocks onto different disks, but the parity bits generated for data block stripe are distributed among all the data disks rather than storing them on a different dedicated disk.



RAID Level 6

In the RAID 6 scheme, two different parity calculations are carried out and stored in separate blocks on different disks. Thus, a RAID 6 array whose user data require N disks consists of N + 2 disks. The advantage of RAID 6 is that it provides extremely high data availability.

6.4.2 Disk Formatting

Disk formatting is a process to configure any data storage device to use for the first time. During this procedure, any existing data on the device will be erased. The configuration process involves wiping all the data on a storage device like a hard drive, a solid-state drive, or a flash drive before installing an operating system.

Types of Disk Formatting

1. Low Level Formatting (Physical Formatting)
2. Logical Formatting (High Level Formatting)

1. Low Level Formatting

Before a disk can store data, it must be divided into sectors that the disk controller can read and write. This process is called low-level formatting, or physical formatting. Low-level formatting fills the disk with a special data structure for each sector. The data structure for a sector typically consists of a header, a data area (usually 512 bytes in size), and a trailer. The header and trailer contain information used by the disk controller, such as a sector number and an error-correcting code (ECC).

2. Logical Formatting

To use a disk to hold files, the OS still needs to record its own data structures on the disk. It does so in two steps.

The first step is to partition the disk into one or more groups of cylinders. After partitioning, the second step is logical formatting (or creation of a file system). In this step, the OS stores the initial file-system data structures onto the disk. These data structures mav include maps of free and allocated space (a FAT or ino n initial empty directory.

6.4.3 Disk ARM Scheduling

Disk scheduling is a technique used by the operating system to schedule multiple requests for accessing the disk.

The time required to read or write a disk block is determined by 3 factors:

- **Seek time:** The time to move the arm to the proper cylinder.
- **Rotational delay:** The time for the proper sector to rotate under the head.
- **Actual data transfer time:** Disk scheduling algorithms are used to reduce the total seek time of any request.

Several algorithms exist to schedule the servicing of disk I/O requests. Following are the scheduling algorithms:

1. **First-in, First-out (FIFO)**

   The simplest form of scheduling is first-in-first-out (FIFO) scheduling, which processes items from the queue in sequential order. This strategy has the advantage of being fair, because every request is honored, and the requests are honored in the order received.

2. **Shortest Seek Time First (SSTF)**

   The shortest-service-time-first (SSTF) policy is to select the disk I/O request that requires the least movement of the disk arm from its current position. It reduces the total seek time as compared to FCFS. However, there is an overhead of finding out the closest request.

3. **Elevator Algorithm (SCAN)**

   SCAN algorithm is also known as the elevator algorithm because it operates much the way an elevator does. With SCAN, the arm is required to move in one direction only until it reaches the last track in that direction or until there are no more requests in that direction. After reaching the other end, head reverses its direction and move towards the starting end servicing all the request in between.

4. **Modified Elevator (Circular SCAN, C-SCAN)**

   Circular SCAN algorithm is an improved version of the SCAN algorithm. The C-SCAN (circular SCAN) policy restricts scanning to one direction only. Thus, when the last track has been visited in one direction, the arm is returned to the opposite end of the disk and the scan begins again. This reduces the maximum delay experienced by new requests.

Selecting a Disk-Scheduling Algorithm

- SSTF common, natural appeal
- SCAN and C-SCAN perform better if heavy load on disk
- Performance depends on number and types of requests
- Requests for disk service influenced by file-allocation method
- Disk-scheduling should be separate module of OS, allowing replacement with different algorithm if necessary

6.4.4 Error-Handling

Bad sectors are those sectors that do not correctly read back the value just written to them. If the defect is very small, say, only a few bits, it is possible to use the bad sector and just let the ECC correct the errors every time. If the defect is bigger, the error cannot be masked.

There are two general approaches to bad blocks: deal with them in the controller or deal with them in the operating system.

6.4.5 Stable Storage

Stable storage is a classification of computer data storage technology that guarantees atomicity for any given write operation. For some applications, it is essential that data never be lost or corrupted, even in the face of disk and CPU errors. It is robust against some hardware and power failures.

Stable storage uses a pair of identical disks with the corresponding blocks working together to form one error-free block. In the absence of errors, the corresponding blocks on both drives are the same. To achieve this goal, the following three operations are defined:

1. **Stable writes:**

   A stable write consists of first writing the block on drive 1, then reading it back to verify that it was written correctly. If it was not, the write and reread are done again up to n times until they work.

2. **Stable reads:**

   A stable read first reads the block from drive 1. If this yields an incorrect ECC, the read is tried again, up to n times. If all of these give bad ECCs, the corresponding block is read from drive 2.

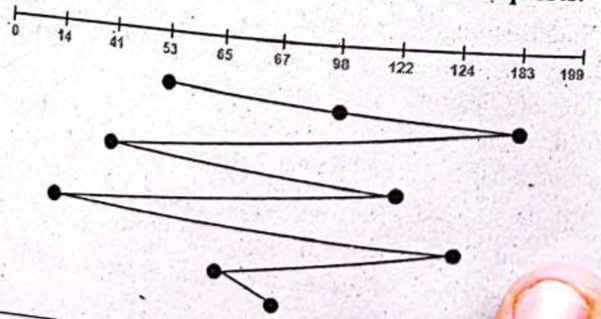
3. **Crash recovery:**

   After a crash, a recovery program scans both disks comparing corresponding blocks. If a pair of blocks are both good and the same, nothing is done. If one of them has an ECC error, the bad block is overwritten with the corresponding good block. If a pair of blocks are both good but different, the block from drive 1 is written onto drive 2.

ANSWERS TO SOME IMPORTANT QUESTIONS

1. **Consider a disk queue with requests for I/O to blocks on cylinders 98, 183, 41, 122, 14, 124, 65, 67. The FCFS scheduling algorithm is used. The head is initially at cylinder number 53. The cylinders are numbered from 0 to 199. Find the total head movement (in number of cylinders) incurred while servicing these requests.**

Solution:



Total head movements incurred while servicing these requests

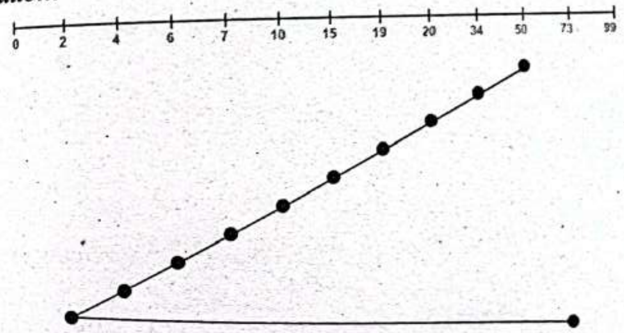
= (98 – 53) + (183 – 98) + (183 – 41) + (122 – 41) + (122 – 14) + (124 – 14) + (124 – 65) + (67 – 65)

= 45 + 85 + 142 + 81 + 108 + 110 + 59 + 2

= 632

2. **Consider a disk system with 100 cylinders. The requests to access the cylinders occur in following sequence- 4, 34, 10, 7, 19, 73, 2, 15, 6, 20 Assuming that the head is currently at cylinder 50, what is the time taken to satisfy all requests if it takes 1 ms to move from one cylinder to adjacent one and shortest seek time first policy is used?**

Solution:



Total head movements incurred while servicing these requests

= (50 – 34) + (34 – 20) + (20 – 19) + (19 – 15) + (15 – 10) + (10 – 7) + (7 – 6) + (6 – 4) + (4 – 2) + (73 – 2)

= 16 + 14 + 1 + 4 + 5 + 3 + 1 + 2 + 2 + 71

= 119

Time taken for one head movement = 1 msec. So,

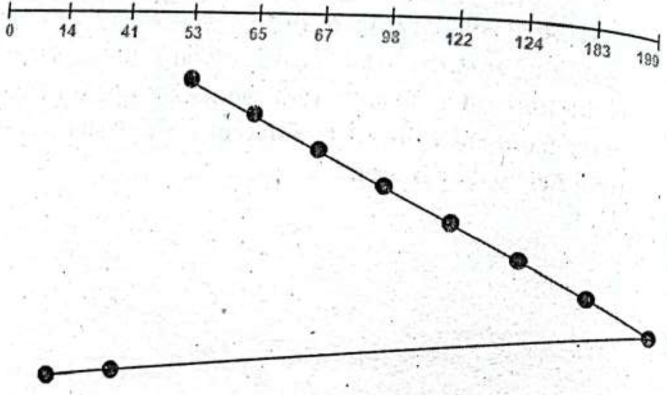
Time taken for 119 head movements

= 119 × 1 msec

= 119 msec

3. ***Consider a disk queue with requests for I/O to blocks on cylinders 98, 183, 41, 122, 14, 124, 65, 67. The SCAN scheduling algorithm is used. The head is initially at cylinder number 53 moving towards larger cylinder numbers on its servicing pass. The cylinders are numbered from 0 to 199. Find the total head movement (in number of cylinders) incurred while servicing these requests.***

Solution:



Total head movements incurred while servicing these requests

= (65 – 53) + (67 – 65) + (98 – 67) + (122 – 98) + (124 – 122) + (183 – 124) + (199 – 183) + (199 – 41) + (41 – 14)

= 12 + 2 + 31 + 24 + 2 + 59 + 16 + 158 + 27

= 331

Alternatively we can also calculate the total head movements as follows:

Total head movements incurred while servicing these requests

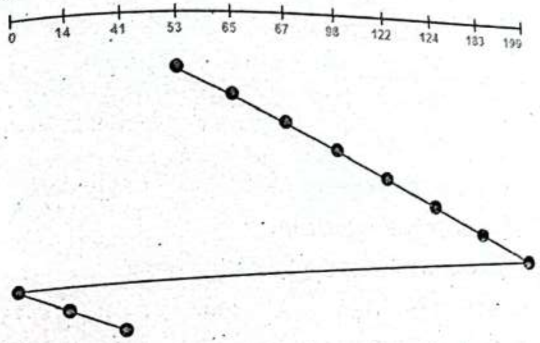
= (199 – 53) + (199 – 14)

= 146 + 185

= 331

4. **Consider a disk queue with requests for I/O to blocks on cylinders 98, 183, 41, 122, 14, 124, 65, 67. The C-SCAN scheduling algorithm is used. The head is initially at cylinder number 53 moving towards larger cylinder numbers on its servicing pass. The cylinders are numbered from 0 to 199. Find the total head movement (in number of cylinders) incurred while servicing these requests.**

Solution:



Total head movements incurred while servicing these requests

= (65 – 53) + (67 – 65) + (98 – 67) + (122 – 98) + (124 – 122) + (183 – 124) + (199 – 183) + (199 – 0) + (14 – 0) + (41 – 14)

= 12 + 2 + 31 + 24 + 2 + 59 + 16 + 199 + 14 + 27

= 386

Alternatively,

Total head movements incurred while servicing these requests

= (199 – 53) + (199 – 0) + (41 – 0)

= 146 + 199 + 41

= 386

5. **Suppose that the head of a moving head disk has 200 tracks numbers 0-199, is currently serving the request at track 143 and has just finished a request at track 125. The queue of requests is kept in FIFO order 86, 147, 91, 177, 94, 150, 102, 175, 130.**

What is the total number of head movements needed to satisfy these requests for the following disk scheduling algorithm?
FCFS, SSTF, SCAN, LOOK, C-SCAN.

Solution:

FIFO (FCFS) scheduling algorithm

Here the head is move in the order

143→86→147→ 91→ 177→ 94→150→102→175→ 130
↑ Starting head position

= |86-143| + |147-86| + |91-147| + |177-91| + |94-177| + |150-94| + |102-150| + |175-102| + |130-175|

= 565 cylinders

Average head movements =565/9=62.77 cylinders

SSTF scheduling algorithm

The head will move in the following order

143→147→150 →130 →102 →94 →91 →86 →175 → 177
↑ Starting head position

Total number of head movement is:

= |147-143| + |150-147| + |130-150| + |102-130| + |94-102| + |91-94| + |86-91| + |175-86| + |175-177|

= 4 + 3 + 20 + 28 + 8 + 3 + 5 + 89 + 2

= 162 cylinder

Average head movements = $\frac{162}{9}$ = 18 cylinders

SCAN Scheduling Algorithm

The head is move in order

143→147→150→175→177→199→130→102→94→91→ 86
↑ Starting head position

Total number of head movements are:

= |143-147 | + |150-147| + |175-150| + |177-175| + |199-177| + |130-199| + |102-130| + |94-102| + |91-94| + |86-91|

= 169 cylinders

Average head movements =169/9=18.77 cylinders

LOOK Scheduling Algorithm

The head is move in order

143→ 147→150→175→ 177→ 130→ 102→ 94→ 91→ 86
↑ Starting head position

Total number of head movements is:

= |147-147 | + |150-147| + |175-150| + |177-175| + |130-177| + |102-130| + |94-102| + |91-94| + |86-91|

= 125 cylinders

Average head movements =125/9=13.88 cylinders

C-SCAN Scheduling Algorithm

The head is move in order

143→147→150→175→177→199→0→86→91→94→102→130
↑ Starting head position

Total number of head movements are:

= |147-147 | + |150-147| + |175-150| + |177-175| + |199-0| + |0-86| + |91-86| + |94-91| + |102-94| + |130-102|

= 385 cylinders

Average head movements = $\frac{385}{9}$ = 18.77 cylinders

6. **Suppose the disk requests comes in the order of request as: 82, 170, 43, 160, 24, 16, 190 and current position of Read/write head is 50. If a seek takes 6 m sec/cylinder. How much seek and seek time is needed for: FCFS, SSTF, SCAN and C-LOOK disk arm scheduling algorithm.** [2078 Baishakh]

Solution:

∴ Movement of disk head is as follows:

1. **FCFS**

50 → 82 → 170 → 43 → 160 → 24 → 16 → 190
↑

∴ Total head movement is given by:

$= (50 \text{ to } 82) + (82 \text{ to } 170) + (170 \text{ to } 43) + (43 \text{ to } 160)$
$+ (160 \text{ to } 24) + (24 \text{ to } 16) + (16 \text{ to } 190)$

$= |82 - 50| + |170 - 82| + |170 - 43| + |160 - 43| + |160$
$- 24| + |24 - 16| + |16 - 190|$

$=$ 682 cylinders

$\therefore$ Average head movement $= \dfrac{682}{7} = 97.42$ cylinders.

2. **SSTF**

$50 \rightarrow 43 \rightarrow 24 \rightarrow 16 \rightarrow 82 \rightarrow 160 \rightarrow 170 \rightarrow 190$
$\uparrow$
Starting head position

$\therefore$ Total head movement is given by:

$= (50 \text{ to } 43) + 43 \text{ to } 24) + (24 \text{ to } 16) + (16 \text{ to } 82) + (82$
$\text{to } 160) + (160 \text{ to } 170) + (170 \text{ to } 190)$

$= |50 - 43| + |43 - 24| + |24 - 16| + |16 - 82| + |82 -$
$160| + |160 - 170| + |170 - 190|$

$= 218$

3. **SCAN**

$50 \rightarrow 82 \rightarrow 160 \rightarrow 170 \rightarrow 190 \rightarrow 43 \rightarrow 24 \rightarrow 16$
$\uparrow$

$\therefore$ Total head movement:

$= (50 \text{ to } 82) + (82 \text{ to } 160) + (160 \text{ to } 170) + (170 \text{ to }$
$190) + (43 \text{ to } 24) + (24 \text{ to } 16)$

$= |50 - 82| + |82 - 160| + |160 - 170| + |50 - 190| + |43$
$- 24| + |24 - 16|$

$= 287$

4. **C-LOOK**

$50 \rightarrow 82 \rightarrow 160 \rightarrow 170 \rightarrow 190 \rightarrow 16 \rightarrow 24 \rightarrow 43$
$\uparrow$

$\therefore$ Total head movement:

$= |50 - 82| + |82 - 160| + |160 - 170| + |170 - 190| +$
$|190 - 16| + |16 - 24| + |24 - 43|$

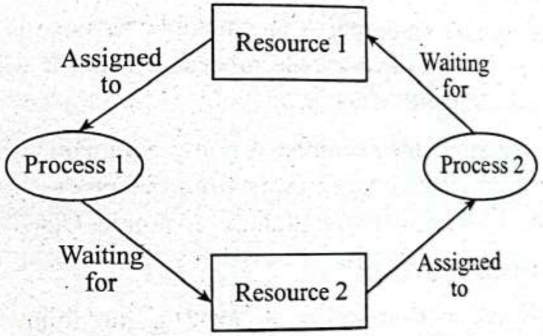
$= 313$

Chapter 7 Deadlock

7.1 Principles of Deadlock

In a multiprogramming system, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, this waiting process will never get the resources it has requested because the resources are held by other waiting processes. This situation is called *deadlock*.

In short, deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.



As illustrated in the diagram, process 1 is holding resource 1 and waiting for resource 2. Resource 2 has been assigned to process 2, and process 2 is waiting for resource 1. At this point, both processes are blocked and the condition is called a *deadlock*.

Let's have a clear insight of deadlock considering the following scenario.

- Process 1 is assigned the printer and will release it after printing one file.
- Process 2 is assigned the tape drive and will release it after reading one file.

- Process 1 tries to use the tape drive, but is told to wait until process 2 releases the tape drive.
- Process 2 tries to use the printer, but is told to wait until process 1 releases the printer.

This results in a deadlock.

Deadlock may involve reusable resources or consumable resources.

- **Reusable resource:** A reusable resource is one that is not depleted or destroyed by use. Example: I/O channel, a region of memory
- **Consumable resource:** A consumable resource is one that is destroyed when it is acquired by a process. Example: messages and information in I/O buffers

A resource can be a hardware or a software. There are two types of resources:

- **Preemptable resource:** A preemptable resource is one that can be taken away from the process owning it with no ill effects. Example: Memory.
- **Non-preemptable resource:** A non-preemptable resource is one that cannot be taken away from the process owning it without potentially causing failure. Example: CD

Conditions for Deadlock

According to Coffman et al. (1971), the following four conditions must hold for deadlock:

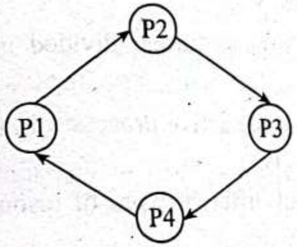
1. **Mutual exclusion:** This condition says that at least one resource must be in non-sharable mode i.e., only one process at a time can use the resource. If another process requests the resource, then the requesting process must be delayed until the resource has been released.

   Printer is an example of a resource that can be used by only one process at a time.

2. **Hold and wait:** This condition says that a process must be holding at least one resource and waiting for another resource currently held by some other process.

3. **No preemption:** This condition says that the resources can't be preempted once the process is assigned the resources. That is, the process must release its resources only voluntarily after it has completed its task.

4. **Circular wait:** This condition says that there must be a chain of processes such that each member of the chain is waiting for a resource held by the next member of the chain. That is, there must exist a set ($P_0, P_1, \ldots, P_n$) of waiting processes such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2, …, P_{n-1} is waiting for a resource held by P_n, and P_n is waiting for a resource held by P_0.



All four of these conditions must be present for a deadlock to occur. If one of them is absent, deadlock is not possible.

7.2 Deadlock Handling Strategies

In general, four strategies are used for dealing with deadlock:

1. Deadlock ignorance
2. Deadlock detection and recovery
3. Deadlock avoidance
4. Deadlock prevention

7.2.1 Deadlock Ignorance

It simply means to ignore the problem.

The *ostrich algorithm* is a strategy of ignoring potential problems on the basis that they may be very rare. It is defined as "*to stick one's head in the sand and pretend there is no problem*". It is used when it is more cost-effective to allow the problem to occur than to attempt its prevention.

7.2.2 Deadlock Detection

In this technique, the system does not attempt to prevent deadlocks from occurring. Instead, it lets them occur, tries to detect when this happens, and then takes some necessary action to recover.

A deadlock is described in terms of a directed graph called as a *resource allocation graph (RAG)*. If every resource has only single instances, then we define a RAG graph to detect deadlock. When we have multiple instances of a resource type then RAG graph is not applicable. This graph consists of two sets:

i. **a set of vertices, V**

The set of vertices is further divided into two categories, namely

a. the set of all the active processes in the system i.e., P = $\{P_1, P_2, ..., P_n)$

b. the set of all different types of resources i.e., R = $\{R_1, R_2, ..., R_m)$

ii. **a set of edges, E**

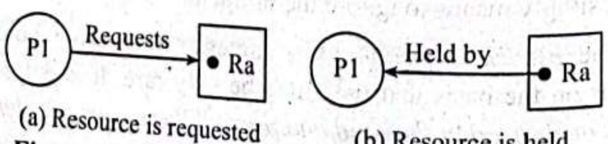
The set of edges is further divided into two types, namely

a. **a request edge**

A directed edge from the process P_i to resource type R_j is called a *request edge*, and is denoted by $P_i \rightarrow R_j$. It means that i^{th} process is requesting one unit of the resource type j.

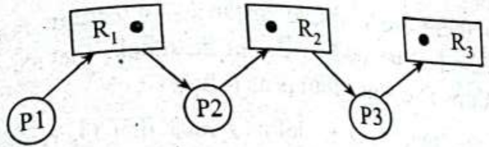
b. **an allocation edge/held edge**

A directed edge from the resource type R_i to process P_j is called an *allocation edge*, and is denoted by $R_i \rightarrow P_j$. It means that one unit of i^{th} resource is held by the process j.



(a) Resource is requested (b) Resource is held

Figure 7.1: Request edge (a) and allocation edge (b)



In the above graph, a process is represented by a circle, each resource is represented by a rectangle. The sets P, R, and E are given as

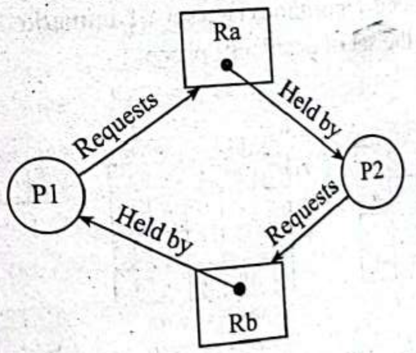
$P = \{P_1, P_2, P_3\}$

$R = \{R_1, R_2, R_3\}$

and $E = \{P_1 \rightarrow R_1, R_1 \rightarrow P_2, P_2 \rightarrow R_2, R_2 \rightarrow P_3, P_3 \rightarrow R_3\}$

The number of dots inside the rectangle represent the number of instances of the resource type, that is, 1-dot means that there is one instance of that resource, 2-dots means that there are two instances of that resource.

If RAG contains one or more cycles, a deadlock exists. Any process that is part of a cycle is deadlocked. If no cycles exist, the system is not deadlocked. For example: the following graph shows deadlock. There is a single instance of resources Ra and Rb. Process P1 holds Rb and requests Ra, while P2 holds Ra but requests Rb. Here RAG contains a complete cycle and hence occurs deadlock where process P1 and P2 are deadlocked.



It is simple to detect the deadlocked processes by visual inspection from a simple graph, but for use in actual systems we need a formal algorithm for detecting deadlocks. Many algorithms for detecting cycles in directed graphs are known.

When there are multiple instances of certain resources, a different method is needed to detect deadlock, that is, a matrix-based algorithm. The algorithm is as follows:

A request matrix Q is defined such that Q_{ij} represents the number of resources of type j requested by process i. The algorithm proceeds by marking processes that are not part of a deadlocked set. Initially, all processes are unmarked. Then the following steps are performed:

1. Mark each process that has a row in the Allocation matrix of all zeros. A process that has no allocated resources cannot participate in a deadlock.
2. Initialize a temporary vector W to equal the Available vector.
3. Find an index i such that process i is currently unmarked and the ith row of Q is less than or equal to W. That is, $Q_{ik} <= W_k$, for $1<= k <= m$. If no such row is found, terminate the algorithm.
4. If such a row is found, mark process i and add the corresponding row of the allocation matrix to W. That is, set $W_k = W_k + A_{ik}$, for $1 <= k <= m$. Return to step 3.

There is a deadlock if and only if there is an unmarked process at the end algorithm. The set of unmarked rows exactly corresponds to the set of deadlocks process.

For example:

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	0	1

Available vector

Figure 7.2: Example for deadlock detection

The algorithm is:

1. Mark P4, because P4 has no allocated resources.

Set W = (0 0 0 0 1).

2. The request of process P3 is less than or equal to W, so mark P3 and set W = W + (0 0 0 1 0) = (0 0 0 1 1).
3. No other unmarked process has a row in Q that is less than or equal to W. Therefore, we terminate the algorithm. The algorithm concludes with P1 and P2 unmarked, indicating these processes are deadlocked.

7.2.2.1 Recovery from Deadlock

When the deadlock is detected by the system, there are two possible methods of recovery from deadlocks. They are:

1. Process termination
2. Resource preemption

1. **Process Termination**

This method can be implemented by one of the following two ways:

i. By aborting all deadlocked processes.

ii. By aborting one process at a time until the deadlock cycle is eliminated.

It may not be easy to abort a process. If the process is in the midst of updating a file, then terminating it will leave that file in an incorrect state. If the process is in the midst of printing data on a printer, the system must reset the printer to a correct state before printing the next job.

If we employ partial termination method, then we must decide which deadlock process (or processes) should be terminated. The factors that should be taken into account in making decision are:

- What the priority of the process is
- How long the process has computed
- How many resources have been used
- What types of resource have been used
- How many more resources the process needs
- Whether the process is interactive or batch

2. ***Resource Preemption***

In this method, we successively preempt (snatch) some resources from one or more of the deadlocked processes and then give these resources to other processes until the deadlock cycle is broken. For preemption of resources, we need to consider three different issues:

i. Selecting a victim

Victim is that process which is selected for preemption. The selection is based on following parameters:

- the cost
- the time, resources associated with a process
- how much time is left for the process for its completion
- the number of resources a process is currently holding and how much it requires more

ii. Rollback

When a resource is preempted from a process, the process cannot continue with its normal execution. At this stage, such process must be roll back to some safe state and restart it from that state. But since it is difficult to determine what a safe state is, the simplest solution is a total rollback. Total rollback means abort the process and then restart it.

iii. Starvation

Preemption of resources from a process may lead to starvation. System must take care that the resources must not be preempted always from the same process otherwise such victim process will require to wait for long time for completing its execution and so will starve to death.

7.2.3 Deadlock Avoidance

Deadlock avoidance involves the analysis of each new resource request to determine if it could lead to deadlock, and granting it only if deadlock is not possible. Two approaches to deadlock avoidance are described below:

i. **Process initiation denial**

Do not start a process if its demands might lead to deadlock.

ii. **Resource allocation denial**

Do not grant an incremental resource request to a process if this allocation might lead to deadlock.

Deadlock avoidance algorithm dynamically examines the resource allocation state to ensure that a circular wait condition can never exist. The resource-allocation state is defined by the number of available and allocated resources and the maximum demands of the processes. The following are the requirements of deadlock avoidance:

1. The maximum resource requirement must be stated in advance.
2. The processes under consideration must be independent. There are no synchronization requirements.
3. There must be a fixed number of resources to allocate.
4. No process may exist till it releases the resources it is holding.

Normally, deádlock avoidance checks that after allocating resources, will the system be in a safe state of not.

Safe and Unsafe State

A system is in safe state only if there exists a safe sequence. A sequence of processes <P1, P2,...,Pn> is a safe sequence for the current allocation state if for each Pi, the resources that Pi can still request can be satisfied by the currently available resources plus the resources held by al the Pj such j<i.

A safe state is not a deadlocked state and conversely a deadlocked state is an unsafe state.

The Banker's Algorithm

The banker's algorithm is a resource allocation and deadlock avoidance algorithm proposed by E. Dijkstra in 1965, that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an "s-state" check to test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue.

Banker's algorithm is named so because it is used in banking system to check whether loan can be sanctioned to a person or not. Suppose there are n number of account holders in a bank and the total sum of their money is S. If a person applies for a loan then the bank first subtracts the loan amount from the total money that bank has and if the remaining amount is greater than S then only the loan is sanctioned. It is done because if all the account holders come to withdraw their money then the bank can easily do it.

In other words, the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. If the customer's request does not cause the bank to leave a safe state, the cash will be allocated, otherwise the customer must wait until some other customer deposits enough.

Let 'n' be the number of processes in the system and 'm' be the number of resources types. The data structure used in Banker's algorithm is shown below:

1. **Available:**

   It is a i-d array of size 'm' indicating the number of available resources of each type.

   Available[j] = k means there are 'k' instances of resource type Rj

2. **Max:**

   It is a 2-d array of size 'n*m' that defines the maximum demand of each process in a system.

   Max[i, j] = k means process Pi may request at most 'k' instances of resource type Rj.

3. **Allocation:**

   It is a 2-d array of size 'n*m' that defines the number of resources of each type currently allocated to each process.

   Allocation[i, j] = k means process P_i is currently allocated 'k' instances of resource type R_j

4. **Need:**

   It is a 2-d array of size 'n*m' that indicates the remaining resource need of each process.

   Need [i, j] = k means process P_i currently need 'k' instances of resource type R_j

   Note: Need [i, j] = Max [i, j] – Allocation [i, j]

   Banker's algorithm consists of Safety algorithm and Resource request algorithm.

Safety Algorithm:

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

Let Work and Finish be vectors of length 'm' and 'n' respectively.

1. **Initialize:**

   Work = Available

   Finish[i] = false; for i=1, 2, 3, 4….n

2. **Find an i such that both**

   a. Finish[i] = false

   b. Needi <= Work

   if no such i exists goto step (4)

3. **Work = Work + Allocation[i]**

   Finish[i] = true

   goto step (2)

4. **if Finish [i] = true for all i**

   then the system is in a safe state

   This algorithm takes O(m x n^2) operations to decide whether a state is safe.

Resource-Request Algorithm:

Now the next algorithm is a resource-request algorithm and it is mainly used to determine whether requests can be safely granted or not.

Let Request$_i$ be the request array for process P$_i$. Request$_i$ [j] = k means process P$_i$ wants k instances of resource type R$_j$. When a request for resources is made by process P$_i$, the following actions are taken:

1. ***If Request$_i$ <= Need$_i$***

   Goto step (2) ; otherwise, raise an error condition, since the process has exceeded its maximum claim.

2. ***If Request$_i$ <= Available***

   Goto step (3); otherwise, Pi must wait, since the resources are not available.

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

   Available = Available – Request$_i$

   Allocation$_i$ = Allocation$_i$ + Request$_i$

   Need$_i$ = Need$_i$ – Request$_i$

If the resulting resource-allocation is safe, then the transaction is completed and process, P$_i$ is allocated its resources. However, if the new state is unsafe then P$_i$ must wait for Request$_i$ and the old resource allocation state is restored.

This algorithm is known as Banker's algorithm.

7.2.4 Deadlock Prevention

Havender in 1968 suggested that if at least one of the conditions required for deadlock is violated, then deadlock will not occur. There are different strategies that can be adopted for violating these conditions which are explained as follows:

1. Eliminating the mutual-exclusion condition

Mutual exclusion means a resource can never be used by more than one process simultaneously. If a resource could be used by more than one process at the same time, then the process would never have to wait for any resource, and there will be no deadlock. So, if we can violate resources behaving in the mutually exclusive manner, then the deadlock can be prevented. One approach used to violate this condition is "spooling". But in general, we can't deny mutual-exclusion condition because some resources are nonsharable (e.g., printer).

2. Eliminating the hold-and-wait condition

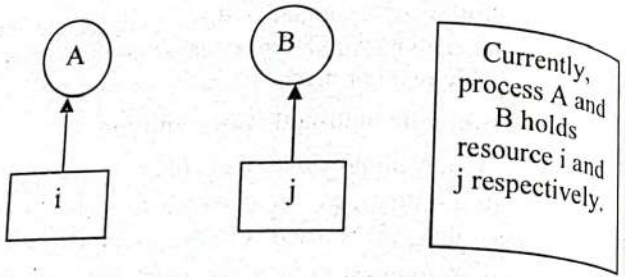
If we can prevent processes that hold resources from waiting for more resources, we can prevent deadlocks. To do so, one approach that can be used is to assign a process with all the required resources before the execution starts. Another approach is to allow a process to request resources only after it has released the resources it has. The disadvantages of both approaches are: low resource utilization and possibility of starvation.

3. Eliminating the no-preemption condition

In order to eliminate this condition, resources must be preempted from the process when resources are required by other high priority processes. This is not a good method at all since if we take a resource away which is being used by the process, then all the work which it has done till now can become inconsistent. For example, consider a printer which is being used by a process. If a printer is snatched from that process and assigned to some other process, then all the data which has been printed can become inconsistent and ineffective, and also the fact that the process can't start printing again from where it has left causes performance inefficiency. So, this method cannot generally be applied to resources like printers and tape drivers. This method is applied to resources whose state can be easily saved and restored later such as CPU registers and memory space.

4. Eliminating the circular wait condition

This condition can be prevented by providing a global numbering of resources. With this way, the processes can request resources whenever they require, but all requests must be made in the numerical order. A process can't request for a lesser priority resource. With this rule, the resource allocation graph can never have a cycle.



Currently, process A and B holds resource i and j respectively.

If i>j A is not allowed to request j and if j>I B is not allowed to request i.

Consider, for example, the numerically ordered resources, as shown below.

1. Floppy drive
2. Printer
3. Plotter
4. Tape drive
5. CD drive

For the above case, a process may request first a printer and then a tape drive, but it may not request first a plotter and then a printer.

Among all the methods, violating circular wait condition is the only method that can be implemented practically.

7.3 An Integrated Deadlock Strategy

J. Howard in 1973 suggested that, in dealing with deadlock, it is more efficient to use different strategies in different situations rather than using only one of the strategies. His suggestion is:

- Group resources into a number of different resource classes.
- Use the linear ordering strategy (global numbering of resources) for the prevention of circular wait to prevent deadlocks between resources.
- For each resource class, use the algorithm that is most suitable for that class.

Consider the following classes of resources:

- **Swappable space** (blocks of main memory on secondary storage)
- **Process resources** (such as tape drives and files)
- **Main memory**
- **Internal resources** (such as I/O channels)

Note that the classes of resources are in the order in which resources are assigned.

Following strategies could be employed for each class to deal with deadlock:

- **Swappable space**

  Deadlock prevention strategy can be used most of the time and deadlock avoidance strategy may be employed sometime.

- **Process resources**

  Deadlock avoidance strategy will be effective most of the time. Deadlock prevention strategy (by means of resource ordering) is also a possibility.

- **Main memory**

  Deadlock prevention (by preemption) is the most appropriate strategy.

- **Internal resources**

  Deadlock prevention (by means of resource ordering) can be adopted.

7.4 Other Issues

In this section we will discuss a few miscellaneous issues related to deadlocks. These include two-phase locking, non-resource deadlocks, and starvation.

7.4.1 Two-Phase Locking

In **two-phase locking** the first phase, the process tries to lock all the records it needs, one at a time. If it succeeds, it begins the second phase, performing its updates and releasing the locks. No real work is done in the first phase.

If during the first phase, some record is needed that is already locked, the process just releases all its locks and starts the first phase all over. In a certain sense, this approach is similar to requesting all the resources needed in advance, or at least before anything irreversible is done. In some versions of two-phase locking, there is no release and restart if a locked record is encountered during the first phase. In these versions, deadlock can occur.

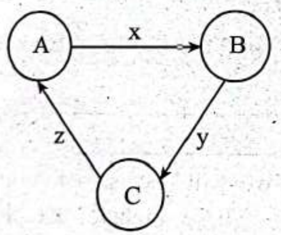
7.4.2 Communication Deadlocks

Processes in this deadlock wait to communicate with other processes in a group of processes. On receiving a communication from any of these processes, a waiting process can unblock. If each process in the set is waiting to communicate with another process in the set, and no process in the set ever begins any additional communication until it receives the communication for which it is waiting, the set is communication-Deadlocked.

Example:

Process A waits to get a message from process B, Process B waits to get a message from process C and Process C waits to get a message from process A and hence a deadlock.

The TFW diagram is shown below:



Users in a distributed system (DDBS) access the database's data objects by executing transactions. A transaction can be thought of as a series of reads and writes performed on data objects. A database's data objects can be thought of as resources that are acquired (through locking) and released (through unlocking) by transactions. In DDBS, a wait for graph is referred to as transaction-wait-for-graph (TWF Graph).

7.4.3 Livelock

Livelock occurs when two or more processes continually repeat the same interaction in response to changes in the other processes without doing any useful work. These processes are not in the waiting state, and they are running concurrently. This is different from a deadlock because in a deadlock all processes are in the waiting state.

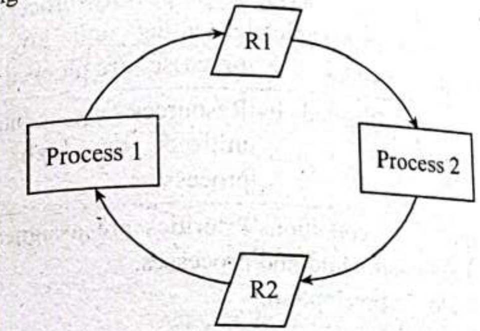


Figure 7.3: Process P1 holds resource R2 and requires R1 while process P2 holds resource R1 and requires R2

Each of the two processes needs the two resources and they use the polling primitive enter_reg to try to acquire the locks necessary for them. In case the attempt fails, the process just tries again.

If process A runs first and acquires resource 1 and then process B runs and acquires resource 2, no matter which one runs next, it will make no further progress, but neither of the two processes blocks. What actually happens is that it uses up its CPU quantum over and over again without any progress being made but also without any sort of blocking. Thus this situation is not that of a deadlock (as no process is being blocked) but we have something functionally equivalent to deadlock: LIVELOCK.

7.4.4 Starvation

Starvation occurs if a process is indefinitely postponed. Starvation is a problem which is closely related to both, Livelock and Deadlock. In a dynamic system, requests for resources keep on happening. Thereby, some policy is needed to make a decision about who gets the resource when. This process, being reasonable, may lead to some processes never getting serviced even though they are not deadlocked.

Difference between Deadlock and Starvation

S.No.	Deadlock	Starvation
1.	All processes keep waiting for each other to complete and none get executed.	High priority processes keep executing and low priority processes are blocked.
2.	Resources are blocked by the processes.	Resources are continuously utilized by high priority processes.
3.	Necessary conditions Mutual Exclusion, Hold and Wait, No preemption, Circular Wait	Priorities are assigned to the processes.
4.	Also known as Circular wait.	Also known as lived lock.
5.	It can be prevented by avoiding the necessary conditions for deadlock.	It can be prevented by Aging.

ANSWERS TO SOME IMPORTANT QUESTIONS

1. **Consider the following snapshot of a system.**

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

Let the available number of resources be given by available vectors as (3, 3, 2). Use Banker's algorithm to claim that the system is in safe state and show the safe sequence. [2075 Back]

Ans: Given,

Available matrix = [3, 3, 2]

$$\text{Allocation matrix} = \begin{bmatrix} 0 & 1 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{bmatrix}$$

$$\text{Maximum matrix} = \begin{bmatrix} 7 & 5 & 3 \\ 3 & 2 & 2 \\ 9 & 0 & 2 \\ 2 & 2 & 2 \\ 4 & 3 & 3 \end{bmatrix}$$

We know that,

Need matrix = Maximum matrix – Allocation matrix

$$= \begin{bmatrix} 7 & 5 & 3 \\ 3 & 2 & 2 \\ 9 & 0 & 2 \\ 2 & 2 & 2 \\ 4 & 3 & 3 \end{bmatrix} - \begin{bmatrix} 0 & 1 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{bmatrix} = \begin{bmatrix} 7 & 4 & 3 \\ 1 & 2 & 2 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{bmatrix}$$

$$\text{Therefore, need matrix is } \begin{bmatrix} 7 & 4 & 3 \\ 1 & 2 & 2 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{bmatrix}$$

Initially,

Work = available = [3, 3, 2]

Iteration I:

For P0:

Is Need <= Work ?

i.e. [7, 4, 3] <= [3, 3, 2] ? obviously not. It doesn't satisfy the condition.

So, resource is not given to P0.

Iteration II:

For P1:

Is Need <= Work ?

i.e., [1, 2, 2] <= [3, 3, 2] ? Yes, it satisfies the condition.

So, P1 executes and work is updated.

Therefore, new work = old work + allocation of P2

= [3, 3, 2] + [2, 0, 0]

= [5, 3, 2]

Iteration III:

For P2:

Is Need <= Work?

i.e., [6, 0, 0] <= [5, 3, 2] ? obviously not. It doesn't satisfy the condition.

So, resource is not given to P2.

Iteration IV:

For P3:

Is Need <= Work?

i.e., [0, 1, 1] <= [5, 3, 2] ? Yes, it satisfies the condition.

So, P3 executes and work is updated.

Therefore, new work = old work + allocation of P3

= [5, 3, 2] + [2, 1, 1]

= [7, 4, 3]

Iteration V:

For P4:

Is Need <= Work?

i.e., [4, 3, 1] <= [7, 4, 3] ? Yes, it satisfies the condition.

So, P4 executes and work is updated.

Therefore, new work = old work + allocation of P4

= [7, 4, 3] + [0, 0, 2]

= [7, 4, 5]

Iteration VI:

For P0:

Is Need <= Work?

i.e., [7, 4, 3] <= [7, 4, 5] ? Yes, it satisfies the condition.

So, P0 executes and work is updated.

Therefore, new work = old work + allocation of P0

= [7, 4, 5] + [0, 1, 0]

= [7, 5, 5]

Iteration VII:

For P1:

Is Need <= Work?

i.e., [1, 2, 2] <= [7, 5, 5] ? Yes, it satisfies the condition.

So, P1 executes and work is updated.

Hence, the system is in a safe state because all the process executes.

Sequence of execution: P1, P3, P4, P0, P2

EXERCISE

1. What is deadlock? Discuss the necessary conditions for deadlock with examples.
2. Write four conditions for deadlock.
3. What is deadlock? State the necessary conditions for deadlock to occur. Give reason, why all conditions are necessary.
4. What is deadlock? How it occurs? Explain various deadlock avoidance methods with examples.
5. Explain the necessary conditions of deadlock? How can a system detect deadlock and what does it do after detection?
6. What is deadlock? Explain the essential condition for deadlock. How do you detect deadlock? Explain with examples.

7. What is deadlock avoidance and detection? Explain all possible deadlock prevention techniques.

8. What is the difference between deadlock and indefinite postponement?

9. List four essential conditions for deadlock. Explain each of them briefly. What would be necessary (in OS) to prevent the deadlock? How is the deadlock recovered? Explain.

10. A system has 2 process and 3 resources. Each process need maximum of two resources, Is deadlock possible? Explain.

Chapter 8 Security

8.1 Computer Security Concepts

These days, computers may contain important and confidential information. These information may be personal financial information like credit card numbers, tax returns; company technical information like new chip design, new software; future plans and policies of a company like plans for a stock offering, merger of one company with another; classified information like investigation of origin of COVID-19 virus, classified emails from intelligence of one country to another country; and much more. Preventing these information from unauthorized access is a prime concern of all operating systems.

The *NIST Computer Security Handbook* defines the term computer security as:

"*Computer security* is the protection afforded to an automated information system in order to attain the applicable objectives of preserving the integrity, availability, and confidentiality of information system resources (includes hardware, software, firmware, information/data, and telecommunications)."

Information security may be defined as:

"The protection of information and information systems from unauthorized access, use, disclosure, disruption, modification, or destruction in order to provide confidentiality, integrity, and availability." (CNSS, 2010)

So, the security of an information system can be studied in three parts: *confidentiality*, *integrity*, and *availability*. These are the key objectives of computer security.

1. **Confidentiality**
   - *Confidentiality* is the concealment of information or resources.

- Confidentiality is concerned with having secret data remain secret. More specifically, if the owner of some data has decided these data are to be made available only to certain people and no others, the system should guarantee that release of the data to unauthorized people never occurs.[1]
- The "snowden case" is an example of a breach of confidentiality.
- Measures: Adopting access control mechanism such as cryptography.

2. **Integrity**

- *Integrity* refers to the trustworthiness of data or resources.
- Integrity means that unauthorized users should not be able to modify any data without the owner's permission. Data modification in this context includes not only changing the data, but also removing data and adding false data.[2]
- Examples of breach of integrity: Hacked sites that display the pirate banners, modified contents on electronic communication.
- Measures: Digital signatures

3. **Availability**

- *Availability* refers to the ability to use the information or resource desired.
- Availability means that systems work promptly and service is not denied to authorized users. For any computer system to serve its purpose, the information must be available when needed.
- DOS (denial of service), DDOS (distributed denial of service) attacks are examples of breach of availability.
- Measures: Application of front end hardware, upstream filtering

These three concepts form what is often referred to as the *CIA triad*.

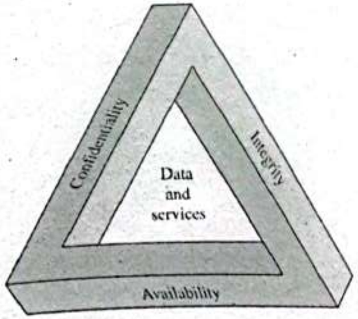


Figure 8.1: The security requirements triad

Some security scientists feel that, to understand a complete picture of security objectives, additional concepts are needed. Two of these are:

- **Authenticity:** Authenticity means that the parties involved are who they claim to be. This means verifying that users are who they say they are and that each input arriving at the system came from a trusted source.
- **Accountability:** This supports fault isolation, deterrence, nonrepudiation, intrusion detection and prevention, and after-action recovery and legal action.

8.2 Information Security Model

There are a number of different models proposed as a framework for information security but one of the best models is the *McCumber model* which was designed by John McCumber. In this model, the elements to be studied are organized in a cube structure, in which each axis indicates a dissimilar viewpoint of some information security issue and there are three major modules in each axis. This model with 27 little cubes all organized together looks similar like a Rubik's cube. There are three axes in the cube they are: goals desired, information states, and measures to be taken. At the intersection of three axes, you can research all angles of an information security problem.

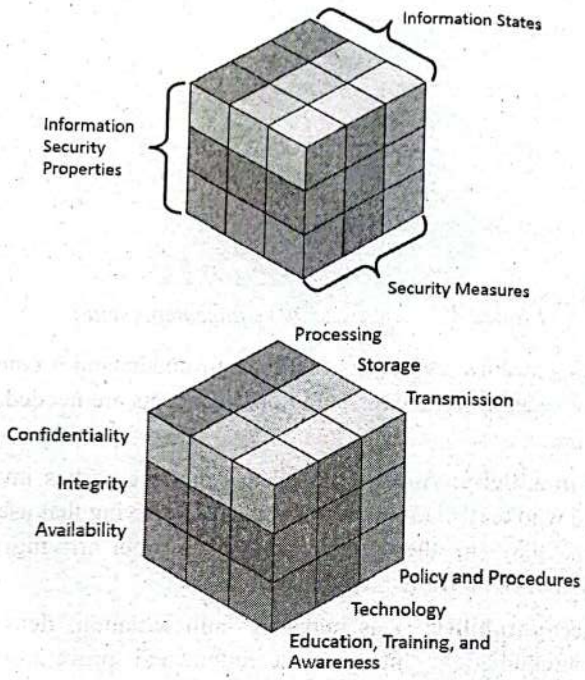


Figure 8.2: The McCumber model

Computer Security Terminology:

- **Security breach**

  A security breach is any incident that results in unauthorized access of data, applications, services, networks and/or devices by bypassing their underlying security mechanisms. Security breaches happen when the security policy, procedures and/or system are violated.

- **Adversary (threat agent)**

  An entity that attacks, or is a threat to, a system.

- **Attack**

  An assault on system security that derives from an intelligent threat; that is, an intelligent act that is an attempt (especially in the sense of a method or technique) to avoid security services and violate the security policy of a system.

- **Countermeasure**

  An action, device, procedure, or technique that reduces a threat, a vulnerability, or an attack by eliminating or preventing it, by minimizing the harm it can cause, or by discovering and reporting it so that corrective action can be taken.

- **Risk**

  An expectation of loss expressed as the probability that a particular threat will exploit a particular vulnerability with harmful results.

- **Security policy**

  A set of rules and practices that specify or regulate how a system or organization provides security services to protect sensitive and critical system resources.

- **System resource (asset)**

  Data contained in an information system; or a service provided by a system; or a system capability, such as processing power or communication bandwidth; or an item of system equipment (i.e., a system component—hardware, firmware, software, or documentation); or a facility that houses system operations and equipment.

- **Threat**

  A potential for violation of security, which exists when there is a circumstance, capability, action, or event, that could breach security and cause harm. That is, a threat is a possible danger that might exploit a vulnerability.

- **Vulnerability**

  A flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy.

8.3 Categories of Security

There are basically two categories of security: information security and network and internet security.

i. **Information Security**

It is the practice of defending information from unauthorized access, use, disclosure, disruption, modification, perusal, inspection, recording or destruction. It is a general term that can be used regardless of the form the data may take (e.g., electronic, physical).

ii. **Network and Internet Security**

Network security consists of the policies and practices adopted to prevent and monitor unauthorized access, misuse, modification, or denial of a computer network and network-accessible resources.

Categories of Security Attacks

- **Interruption:** This is an attack on *availability*. Example: Cutting of a communication line.
- **Interception:** This is an attack on *confidentiality*. Example: Wiretapping to capture data in a network.
- **Modification:** This is an attack of *integrity*. Example: Changing values in a data file.
- **Fabrication:** This is an attack on *authenticity*. Example: Inserting fake messages in a network.

8.4 Network Security Attacks

Network security attacks can be classified as passive attacks and active attacks.

1. **Passive Attacks**

A *passive attack* attempts to learn or make use of information from the system but does not affect system resources. Passive attacks are in the nature of eavesdropping on, or monitoring of, transmissions.

There are two types of passive attacks: *release of message contents* and *traffic analysis*.

- **Release of message contents**

  Consider that, a person wants to send confidential email to another person. If a third person learns the contents of the email sent, this is a "release of message contents" attack. So, an outsider learning the contents of a telephone conversation, an electronic mail message, or a transferred file is an example of this attack.

- **Traffic analysis**

  By using encryption, a message sent could be masked in order to prevent the extraction of the information from the message by the attacker, even if the message is captured. However, if the attacker succeeds to retrieve the information by analyzing the traffic and observing the pattern, it is known as *traffic analysis*.

2. **Active Attacks**

An *active attack* attempts to alter system resources or affect their operation. It involves some modification of the data stream or the creation of a false stream.

There are four types of active attacks: *replay*, *masquerade*, *modification of messages*, and *denial of service*.

- **Replay**

  It involves the passive capture of a data unit and its subsequent retransmission to produce an unauthorized effect.

- **Masquerade**

  It takes place when one entity pretends to be a different entity. A masquerade attack usually includes one of the other forms of active attack. For example, authentication sequences can be captured and replayed after a valid authentication sequence has taken place, thus enabling an authorized entity with few privileges to obtain extra privileges by impersonating an entity that has those privileges.[3]

- **Modification of messages**

  It simply means that some portion of a legitimate message is altered, or that messages are delayed or recorded, to produce an unauthorized effect. For example, a message stating, "Allow Gautam_Rabin to read confidential file accounts." is modified to say, "Allow Poudel_Hemanta to read confidential file accounts."

- **Denial of service**

  It is an attack in which a machine or network resource is made unavailable to its intended users by temporarily or indefinitely disrupting services of a host connected to the Internet. Denial of service is typically accomplished by flooding the targeted machine or resource with unnecessary requests in an attempt to overload systems and prevent some or all legitimate requests from being fulfilled.

Passive attacks are very difficult to detect, however there are measures to prevent the success of these attacks, usually by means of encryption. Whereas active attacks can be detected but quite difficult to prevent them.

How to achieve CIA goals?
Answer:

First thing is we need to implement "confidentiality". In confidentiality, we need: Authentication which includes: user IDs and passwords, using a physical object, biometric verification, two-factor authentication and data encryption.

Second thing is we need to implement "integrity". In integrity, we need to implement : file permissions and user access controls, version control may be used to prevent erroneous changes or accidental deletion by authorized users becoming a problem, checksums,

Final thing is we need to implement "availability". It consists of: maintaining all hardware, doing necessary system upgrades periodically, providing adequate communication bandwidth and preventing the occurrence of bottlenecks, redundancy.

8.5 System Access Threats

System access threats fall into two general categories: *intruders* and *malicious software*.

8.5.1 Intruders

Intruder (often called *hacker* or *cracker*) is one of the most common threats to security. The objective of the intruder is to gain access to a system or to increase the range of privileges accessible on a system.

The intruder performs any of the following activities:

- Performs a remote root compromise of an e-mail server.
- Defaces a Web server.
- Guesses and cracks passwords.
- Copies a database containing credit card numbers.
- Views sensitive data without authorization.
- Runs a packet sniffer on a workstation to capture usernames and passwords.
- Uses a permission error on an anonymous FTP server to distribute pirated software and music files.
- Dials into an unsecured modem and gains internal network access, etc.

Intruders may be classified into following types.

- **Masquerader:** An individual who is not authorized to use the computer and who penetrates a system's access controls to exploit a legitimate user's account. It is likely to be an outsider.
- **Misfeasor:** A legitimate user who accesses data, programs, or resources for which such access is not authorized, or who is authorized for such access but misuses his or her privileges. It is generally an insider.
- **Clandestine user:** An individual who seizes supervisory control of the system and uses this control to evade auditing and access controls or to suppress audit collection. It is either an outsider or an insider.

8.5.2 Malicious Software (Malware)

Malware is a program that is inserted into a system, usually covertly, with the intent of compromising the confidentiality, integrity, or availability of the victim's data, applications, or operating system or otherwise annoying or disrupting the victim (M. Souppaya, K. Scarfone; 2013).

The programs that exploit vulnerabilities in computing systems and are threats to computer systems are called *malicious software* or *malware*. Malicious software can be relatively harmless or may perform harmful functions (destroying files and data in main memory, bypassing controls to gain privileged access, etc.).

Malicious software can be divided into two categories: *that need a host program* and *that are independent*.

- **That need a host program**

  Also known as *parasitic*, these malware are essentially fragments of programs that cannot exist independently of some actual application program, utility, or system program. Examples: viruses, logic bombs, Trojan horse, and backdoors.

- **That are independent**

  They are self-contained programs that can be scheduled and run by the OS. Examples: worms and bot programs.

Malicious software can also be categorized as: *that do not replicate* and *that replicate*.

- **That do not replicate**

  These type of malware are programs or fragments of programs that are activated by a trigger. Examples: logic bombs, backdoors, and bots programs.

- **That replicate**

  These are fragments of programs or independent programs that, when executed, may produce one or more copies of itself to be activated later on the same system or some other system. Examples: viruses and worms.

1. **Virus**

A *virus* is a piece of software that can "infect" other programs, or indeed any type of executable content, by modifying them.

Viruses perform actions such as corrupting and deleting computer data, using our email to spread to other computers, and so on. The majority of viruses are spread through online downloads (illicit software, files, or programs), email attachments, and messenger apps.

John Aycock in his book "Computer Viruses and Malware" published in 2006 has stated that a computer virus has three parts:

- **Infection mechanism:** *Infection mechanism* or *infection vector* are the means by which a virus spreads or propagates, enabling it to replicate.
- **Trigger:** It is the event or condition that determines when the payload is activated or delivered.
- **Payload:** It is what the virus does, besides spreading.

If we classify a virus by its target, the types are followings:

- **Boot sector infector:** Infects a master boot record or boot record and spreads when a system is booted from the disk containing the virus. The Brain virus released in 1986 for the IBM PC is a boot sector infector.
- **File infector:** *File infector* infects programs or executable files.
- **Macro virus:** *Macro virus* infects files that are created using certain applications and programs that contain macros. The Melissa virus that was spread in 1999 is a macro virus which infected Word 97 and 98 documents on Windows and Macintosh systems.
- **Multipartite virus:** This type of virus infects files in multiple ways, and is typically, capable of infecting multiple types of files.

If we classify a virus by its concealment strategy, the types of virus are:

- **Encrypted virus:** An encrypted virus is one that contains encrypted malicious code and typically replicate by decrypting themselves and spreading. When encrypted, it becomes difficult to identify them using antivirus software.
- **Stealth virus:** This is a virus that tricks antivirus software by intercepting its requests to the operating system and due to this, some antivirus software may not detect it.
- **Polymorphic virus:** A polymorphic virus is a virus that changes its form each time it inserts itself into another program. That is, this virus undergoes mutation with every infection. This makes detection of the virus by "signature" impossible. Satan Bug is an example of polymorphic virus.
- **Metamorphic virus:** With every infection, a metamorphic virus mutates like a polymorphic virus. But unlike polymorphic virus, a metamorphic virus rewrites itself completely at each iteration, using multiple transformation techniques, making difficult to detect.

2. Worms

A *worm* is a program that can replicate itself and send copies from computer to computer across network connections. Upon arrival, the worm may be activated to replicate and propagate again. In addition to propagation, the worm usually performs some unwanted function.

An example of worm is "Code Red" that appeared in July 2001 which attacked computers running Microsoft Internet Information Server (IIS).

Worms often exploit network configuration errors or security loopholes in the operating system or applications. Many worms use multiple methods to spread across networks, including the following:

- **Email:** Carried inside files sent as email attachments
- **Internet:** Via links to infected websites; generally hidden in the website's HTML, so the infection is triggered when the page loads.
- **Downloads & FTP servers:** May initially start in downloaded files or individual FTP files, but if not detected, can spread to the server and thus all outbound FTP transmissions.
- **Instant messages (IM):** Transmitted through mobile and desktop messaging apps, generally as external links, including native SMS apps, WhatsApp, Facebook messenger, or any other type of ICQ or IRC message.
- **P2P/filesharing:** Spread via P2P file sharing networks, as well as any other shared drive or files, such as a USB stick or network server.
- **Networks:** Often hidden in network packets; though they can spread and self-propagate through shared access to any device, drive or file across the network.

The primary difference between a virus and a worm is that viruses must be triggered by the activation of their host; whereas worms are stand-alone malicious programs that can self-replicate and propagate independently as soon as they have breached the system.

3. Spywares

Spyware is a software that collects information from a computer and transmits it to another system by monitoring keystrokes, screen data, and/or network traffic; or by scanning files on the system for sensitive information. Spyware is secretly loaded onto a PC without the owner's knowledge and runs in the background doing things behind the owner's back.

A spyware typically does the following things:

1. Change the browser's home page.
2. Modify the browser's list of favorite (bookmarked) pages.

3. Add new toolbars to the browser.
4. Change the user's default media player.
5. Change the user's default search engine.
6. Add new icons to the Windows desktop.
7. Replace banner ads on Web pages with those the spyware picks.
8. Put ads in the standard Windows dialog boxes.
9. Generate a continuous and unstoppable stream of pop-up ads.

Spyware is mostly classified into four types:

- **System monitors:** *System monitors* render serious privacy risks because they secretly capture and transmit user's personal information, worse still, passwords used in online transactions.
- **Trojans**: *Trojans* can be used to obtain sensitive information, install malicious programs, hijack the computer, or compromise additional computers or networks.
- **Adware**: *Adware* tracks the users' online activities on the internet to deliver targeted pop-up advertisements.
- **Tracking cookies:** *Tracking cookies* are the uninformed small text files downloaded to a user's computer that preserves preferences on specific websites.

4. Trojans

Trojan horse or *Trojan* is a computer program that appears to have a useful function, but also has a hidden and potentially malicious function that evades security mechanisms, sometimes by exploiting legitimate authorizations of a system entity that invokes it.

Trojan horse programs can be used to accomplish functions indirectly that an unauthorized user could not accomplish directly. For example, to gain access to sensitive, personal information stored in the files of a user, an attacker could create a Trojan horse program that, when executed, scans the user's files for the desired sensitive information and sends a

copy of it to the attacker via a Web form or e-mail or text message. The author could then attract users to run the program by incorporating it into a game or useful utility program, and making it available via a known software distribution site or app store. This approach has been used recently with utilities that "claim" to be the latest anti-virus scanner, or security update, for systems, but which are actually malicious Trojans, often carrying payloads such as spyware that searches for banking credentials. Hence, users need to take precautions to validate the source of any software they install.

A Trojan acts like a bonafide application or file to trick us. Users are typically tricked by some form of social engineering into loading and executing Trojans on their systems. Once installed, a Trojan can enable cyber-criminals to spy on us, steal our sensitive data, and gain backdoor access to our system.

The beauty of the Trojan horse attack is that it does not require the author of the Trojan horse to break into the victim's computer. The victim does all the work.

A Trojan carries out one of the following three activities:

- It additionally performs a separate malicious activity while performing the function of the original program.
- It modifies the function to perform malicious activity or to disguise other malicious activity while performing the function of the original program.
- It performs a malicious function that completely replaces the function of the original program.

5. Ransomware

Ransomware is a type of malware that threatens to publish the victim's personal data or perpetually block access to it unless a ransom is paid. While some simple ransomware may lock the system so that it is not difficult for a knowledgeable person to reverse, more advanced malware uses a technique called cryptoviral extortion. It encrypts the victim's files, making them inaccessible, and demands a ransom payment to decrypt them.

In a properly implemented cryptoviral extortion attack, recovering the files without the decryption key is an intractable problem. Difficult to trace digital currencies such as paysafecard or Bitcoin and other cryptocurrencies are used for the ransoms, making tracing and prosecuting the perpetrators difficult.

There are several different ways that ransomware can infect our computer. One of the most common methods today is through malicious spam, or malspam, which is an unsolicited email that is used to deliver malware. The email might include hidden attachments, such as PDFs or Word documents. It might also contain links to malicious websites.

6. Zombie and Botnet

A *bot (robot)*, also known as a *zombie* or *drone*, is a program that secretly takes control of another Internet-attached computer and then uses that computer to launch attacks on other computers. Bots are automated. The collection of bots is called *botnet*.

Bots usually operate over a network; more than half of Internet traffic is bots scanning content, interacting with webpages, chatting with users, or looking for attack targets. Some bots are useful, such as search engine bots that index content for search or customer service bots that help users. Some bots are bad and are programmed to break into user accounts, scan the web for contact information for sending spam, or perform other malicious activities.

Bots can be:

- **Chatbots:** Bots that simulate human conversation by responding to certain phrases with programmed responses.
- **Googlebots:** Bots that scan content on webpages all over the Internet.
- **Social bots:** Bots that operate on social media platforms.
- **Malicious bots:** Bots that scrape content, spread spam content, or carry out credential stuffing attacks.

According to the Honeynet Project, 2005, bots are used for any of the following activities:

- DDOS attacks
- Spamming
- Sniffing traffic
- Keylogging
- Spreading new malware
- Installing advertisement add-ons and browser helper objects.
- Attacking IRC chat networks
- Manipulating online polls/games

8.6 Insider Attacks

Insider attacks are executed by programmers and other employees of the company running the computer to be protected or making critical software. These attacks differ from external attacks because the insiders have specialized knowledge and access that outsiders do not have. It includes logic bombs, trap doors, and login spoofing. Insider attacks are among the most difficult to detect and prevent.

8.6.1 Back Door (Trap Door)

The security hole caused by an insider is the *back door*. It is a secret entry point into a program that allows someone who is aware of the backdoor to gain access without going through the usual security access procedures. This problem is created by code inserted into the system by a system programmer to bypass some normal check. Programmers have used backdoors legitimately for many years to debug and test programs; such a backdoor is called a *maintenance hook*. Backdoors become threats when unethical programmers use them to gain unauthorized access.

For example, a programmer could add code to the login program to allow anyone to log in using the login name "zzzzz" no matter what was in the password file.

```
while (TRUE) {
      printf("login: ");
      get_string(name);
      disable_echoing( );
      printf("password: ");
      get_string(password);
      enable_echoing( );
      v = check_validity(name, password);
      if (v) break;
}
execute_shell(name);
```

(a)

```
while (TRUE) {
      printf("login: ");
      get_string(name);
      disable_echoing( );
      printf("password: ");
      get_string(password);
      enable_echoing( );
      v = check_validity(name, password);
      if (v || strcmp(name, "zzzzz") == 0) break;
}
execute_shell(name);
```

(b)

Figure 8.3: *(a) Normal code (b) Code with a back door inserted.*

(Courtesy of Andrew S. Tanenbaum, Herbert Bos; *Modern Operating Systems*)

It is difficult to implement operating system controls for backdoors in applications. Security measures must focus on the program development and software update activities, and on programs that wish to offer a network service.

8.6.2 Logic Bombs

The *logic bomb* is code embedded in the malware that is set to "explode" when certain conditions are met. It is written by one of a company's (currently employed) programmers and secretly inserted into the production system. Logic bomb executes their functions, or launch their payload, once a certain condition is met such as upon the termination of an employee.

Examples of conditions that can be used as triggers for a logic bomb are the presence or absence of certain files or devices on the system, a particular day of the week or date, a particular version or configuration of some software, or a particular user running the application. Once triggered, a bomb may alter or delete data or entire files, cause a machine halt, or do some other damage.

Time bombs are a type of logic bomb that are triggered by a certain time or date.

An example of logic bomb attack was the incident that occurred in 1982, during the Cold War between the US and the Soviet Union. The CIA was supposedly informed that a KGB operative had stolen the plans for an advanced control system along with its software from a Canadian company, to be used on a Siberian pipeline. The CIA apparently had a logic bomb coded in the system to sabotage the enemy.

8.6.3 Login Spoofing

Login spoofings are techniques used to steal a user's password. The user is presented with an ordinary looking login prompt for username and password, which is actually a malicious program under the control of the attacker. When the username and password are entered, this information is logged or in some way passed along to the attacker, breaching security.

To prevent this, some operating systems require a special key combination to be entered before a login screen is presented. Windows uses CTRL-ALT-DEL for this purpose. If a user sits down at a computer and starts out by first typing CTRL-ALT-DEL, the current user is logged out and the system login program is started. There is no way to bypass this mechanism.

8.7 Security Policy and Access Control

In general, *security policy* is a written document in an organization outlining how to protect the organization from threats, including computer security threats, and how to handle situations when they do occur. However in UNIX (including LINUX), access control implements a security policy that specifies who or what (e.g., in the case of a process) may have access to each specific system resource and the type of access that is permitted in each instance.

An access control policy dictates what types of access are permitted, under what circumstances, and by whom. Access control policies are generally grouped into the following categories:[4]

- **Discretionary access control (DAC):** This policy controls access based on the identity of the requestor and on access rules (authorizations) stating what requestors are (or are not) allowed to do.

- **Mandatory access control (MAC):** This policy controls access based on comparing security labels (which indicate how sensitive or critical system resources are) with security clearances (which indicate system entities are eligible to access certain resources).
- **Role-based access control (RBAC):** This policy controls access based on the roles that users have within the system and on rules stating what accesses are allowed to users in given roles.

An access control mechanism can employ two or even all three of these policies.

8.8 Protection Mechanism

8.8.1 Protection Domains

A computer system contains many resources, or ''objects'' that need to be protected. These objects can be hardware (e.g., CPUs, memory pages, disk drives, or printers) or software (e.g., processes, files, databases, or semaphores).

A domain is a set of (object, rights) pairs. A "right" means permission to perform one of the operations. A domain can be a single user or more general than just one user.

How objects are allocated to domains depends on the specifics of who needs to know what. One basic concept, however, is the **POLA (Principle of Least Authority)** or need to know. In general, security works best when each domain has the minimum objects and privileges to do its work—and no more. Figure 8.4 shows three domains, showing the objects in each domain and the rights (Read, Write, eXecute) available on each object. Note that Printer1 is in two domains at the same time, with the same rights in each. File1 is also in two domains, with different rights in each one.

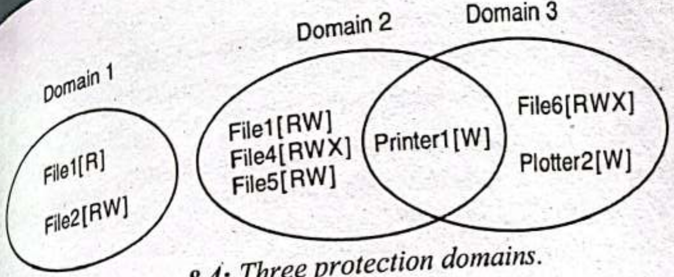


Figure 8.4: Three protection domains.

At every instant of time, each process runs in some protection domain. In other words, there is some collection of objects it can access, and for each object it has some set of rights. Processes can also switch from domain to domain during execution. The rules for domain switching are highly system dependent.

To make the idea of a protection domain more concrete, let us look at UNIX (including Linux). In UNIX, the domain of a process is defined by its UID and GID. When a user logs in, his shell gets the UID and GID contained in his entry in the password file and these are inherited by all its children. Given any (UID, GID) combination, it is possible to make a complete list of all objects (files, including I/O devices represented by special files, etc.) that can be accessed, and whether they can be accessed for reading, writing, or executing. Two processes with the same (UID, GID) combination will have access to exactly the same set of objects. Processes with different (UID, GID) values will have access to a different set of files, although there may be considerable overlap.

Furthermore, each process in UNIX has two halves: the user part and the kernel part. When the process does a system call, it switches from the user part to the kernel part. The kernel part has access to a different set of objects from the user part. For example, the kernel can access all the pages in physical memory, the entire disk, and all the other protected resources. Thus, a system call causes a domain switch.

When a process does an exec on a file with the SETUID or SETGID bit on, it acquires a new effective UID or GID. With a different (UID, GID) combination, it has a different set of files and operations available. Running a program with SETUID or SETGID is also a domain switch, since the rights available change.

An important question is how the system keeps track of which object belongs to which domain. Conceptually, at least, one can envision a large matrix, with the rows being domains and the columns being objects. Each box lists the rights, if any, that the domain contains for the object. The matrix for Figure 8.4 is shown in Figure 8.5. Given this matrix and the current domain number, the system can tell if an access to a given object in a particular way from a specified domain is allowed.

	Object							
Domain	File1	File2	File3	File4	File5	File6	Printer1	Plotter2
1	Read	Read Write						
2			Read	Read Write Execute	Read Write		Write	
3						Read Write Execute	Write	Write

Figure 8.5: *A protection matrix*

Domain switching itself can be easily included in the matrix model by realizing that a domain is itself an object, with the operation entered. Figure 8.6 shows the matrix of Fig. 8.5 again, only now with the three domains as objects themselves. Processes in domain 1 can switch to domain 2, but once there, they cannot go back. This situation models executing a SETUID program in UNIX. No other domain switches are permitted in this example.

	Object										
Domain	File1	File2	File3	File4	File5	File6	Printer1	Plotter2	Domain1	Domain2	Domain3
1	Read	Read Write								Enter	
2			Read	Read Write Execute	Read Write		Write				
3						Read Write Execute	Write	Write			

Figure 8.6: *A protection matrix with domains as objects*

8.8.2 Access Control Lists

An access control list (ACL) is a table that tells a computer operating system which access rights each user has to a particular system object, such as a file directory or individual file. Each object has a security attribute that identifies its access control list. The list has an entry for each system user with access privileges. The most common privileges include the ability to read a file (or all the files in a directory), to write to the file or files, and to execute the file (if it is an executable file, or program). Microsoft Windows NT/2000, Novell's NetWare, Digital's OpenVMS, and UNIX-based systems are among the operating systems that use access control lists.

An access-control list (ACL) is a list of permissions associated with a system resource (object). An ACL specifies which users or system processes are granted access to objects, as well as what operations are allowed on given objects. Each entry in a typical ACL specifies a subject and an operation. For example, if a file object has an ACL that contains John: read, write; Joe: read), this would give John permission to read and write the file and only give Joe permission to read it.

Two methods that are practical, however, are *storing the matrix by rows or by columns*, and then *storing only the non empty elements*. The first technique consists of associating with each object an (ordered) list containing all the domains that may access the object, and how. This list is called the *ACL (Access Control List)* and is illustrated in figure 8.7. In ACL we deal with storing the matrix by columns.

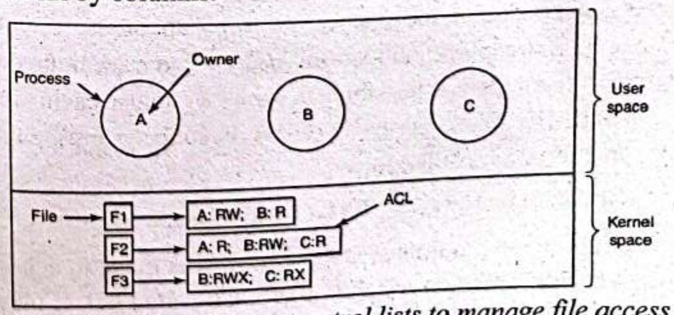


Figure 8.7: *Use of access control lists to manage file access.*

Here we see three processes, each belonging to a different domain, A, B, and C, and three files F1, F2, and F3. For simplicity, we will assume that each domain corresponds to exactly one user, in this case, users A, B, and C. Often in the security literature the users are called *subjects* or *principals*, to contrast them with the things owned, the *objects*, such as files. Each file has an ACL associated with it. File F1 has two entries in its ACL (separated by a semicolon). The first entry says that any process owned by user A may read and write the file. The second entry says that any process owned by user B may read the file. All other accesses by these users and all accesses by other users are forbidden. Note that the rights are granted by the user, not by process. As far as the protection system goes, any process owned by user A can read and write file F1. It does not matter if there is one such process or 100 of them. It is the owner, not the process ID, that matters.

File F2 has three entries in its ACL: A, B, and C can all read the file, and B can also write it. No other accesses are allowed. File F3 is apparently an executable program, since B and C can both read and execute it. B can also write it.

This example illustrates the most basic form of protection with ACLs. More sophisticated systems are often used in practice. To start with, we have shown only three rights so far: *read, write,* and *execute*. There may be additional rights as well like *destroy object* and *copy object* .

So far, our ACL entries have been for individual users. Many systems support the concept of a group of users. Groups have names and can be included in ACLs. Two variations on the semantics of groups are possible. In some systems, each process has a user ID (UID) and group ID (GID). In such systems, an ACL entry contains entries of the form :

UID1, GID1: rights1; UID2, GID2: rights2; ...

Under these conditions, when a request is made to access an object, a check is made using the caller's UID and GID. If they are present in the ACL, the rights listed are available. If the (UID, GID) combination is not in the list, the access is not permitted.

8.8.3 Capabilities

The other way of slicing up the matrix of figure 8.14 is by rows. When this method is used, associated with each process is a list of objects that may be accessed, along with an indication of which operations are permitted on each, in other words, its domain. This list is called a *capability list* (or *C-list*) and the individual items on it are called *capabilities*. A set of three processes and their capability lists is shown in Figure 8.8.

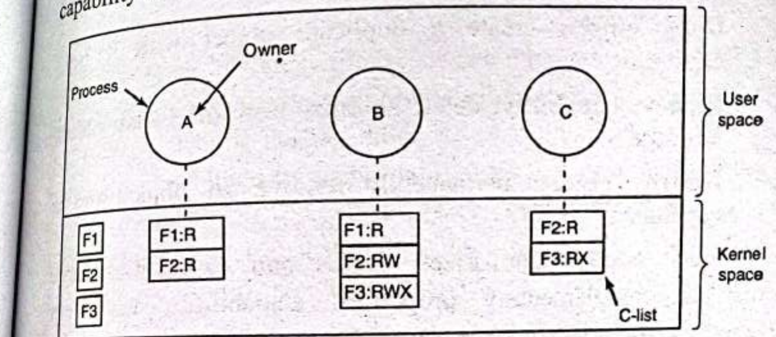


Figure 8.8: When capabilities are used, each process has a capability list.

Each capability grants the owner certain rights on a certain object. In Fig. 9-8, the process owned by user A can read files F1 and F2, for example. Usually, a capability consists of a file (or more generally, an object) identifier and a bitmap for the various rights. In a UNIX-like system, the file identifier would probably be the i-node number. Capability lists are themselves objects and may be pointed to from other capability lists, thus facilitating sharing of subdomains.

It is fairly obvious that capability lists must be protected from user tampering. Three methods of protecting them are known. The first way requires a *tagged architecture,* a hardware design in which each memory word has an extra (or tag) bit that tells whether the word contains a capability or not. The second way is to *keep the C-list inside the operating system.* Capabilities are then referred to by their position in the capability list. The third way is *to keep the C-list in user space,* but manage the capabilities cryptographically so that users cannot tamper with them. This approach is particularly suited to distributed systems.

In addition to the specific object-dependent rights, such as read and execute, capabilities (both kernel and cryptographically protected) usually have generic rights which are applicable to all objects. Examples of generic rights are :

1. **Copy capability:** create a new capability for the same object.
2. **Copy object:** create a duplicate object with a new capability.
3. **Remove capability:** delete an entry from the C-list; object unaffected.
4. **Destroy object:** permanently remove an object and a capability.

Very briefly summarized, ACLs and capabilities have somewhat complementary properties. Capabilities are very efficient because if a process says "Open the file pointed to by capability 3" no checking is needed. With ACLs, a (potentially long) search of the ACL may be needed. If groups are not supported, then granting everyone read access to a file requires enumerating all users in the ACL. Capabilities also allow a process to be encapsulated easily, whereas ACLs do not. On the other hand, ACLs allow selective revocation of rights, which capabilities do not. Finally, if an object is removed and the capabilities are not or vice versa, problems arise. ACLs do not suffer from this problem.

8.9 Cryptography

In its traditional definition *cryptography* is the *science of secret writing*. Cryptanalysis is the science of analyzing and breaking ciphers (ciphers are simply just a set of steps (an algorithm) for performing both an encryption, and the corresponding decryption). Cryptography is the principles, means, and methods of disguising information to ensure its integrity, confidentiality, and authenticity. The purpose of cryptography is to take a message or file, called the *plaintext*, and encrypt it into *ciphertext* in such a way that only authorized people know how to convert it back to *plaintext*. In general, the *ciphertext* is just an incomprehensible pile of bits.

The encryption and decryption algorithms (functions) should always be public. Trying to keep them secret almost never works and gives the people trying to keep the secrets a false sense of security. In the trade, this tactic is called *security by obscurity*. The secrecy depends on parameters to the algorithms called *keys*.

If P is the plaintext file, K_E is the encryption key, C is the ciphertext, and E is the encryption algorithm (i.e., function), then C = E(P, K_E). This is the definition of encryption. It says that the ciphertext is obtained by using the (known) encryption algorithm, E, with the plaintext, P, and the (secret) encryption key, K_E, as parameters. The idea that the algorithms should all be public and the secrecy should reside exclusively in the keys is called *Kerckhoffs' principle*.

Similarly, P = D(C, K_D) where D is the decryption algorithm and K_D is the decryption key. This says that to get the plaintext, P, back from the ciphertext, C, and the decryption key, K_D, one runs the algorithm D with C and K_D as parameters.

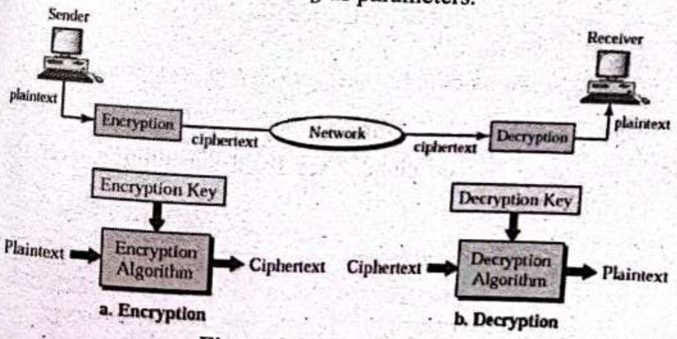


Figure 8.9: Cryptography

Cryptography is broadly classified into two categories: Symmetric key Cryptography and Asymmetric key Cryptography (popularly known as public key cryptography).

8.9.1 Symmetric Key Cryptography (or Symmetric Encryption)

Also known as secret key encryption, an encryption system in which the sender and receiver of a message share a single, common key that is used to encrypt and decrypt the message. Some of the examples of symmetric-key cryptography are described below:

1. **Transposition Cipher**

A transposition cipher is a method of encryption by which the positions held by units of plaintext (which are commonly characters or groups of characters) are shifted according to a regular system, so that the ciphertext constitutes a permutation of the plaintext.

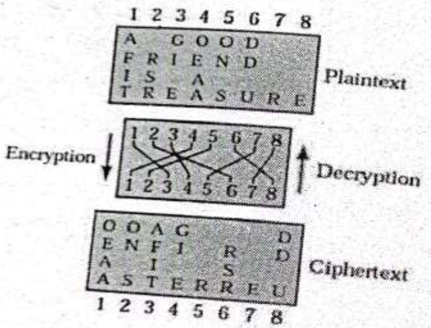


Figure 8.10: Symmetric key encryption, transposition cypher

2. **Monoalphabetic Cipher**

A monoalphabetic cipher uses fixed substitution over the entire message.

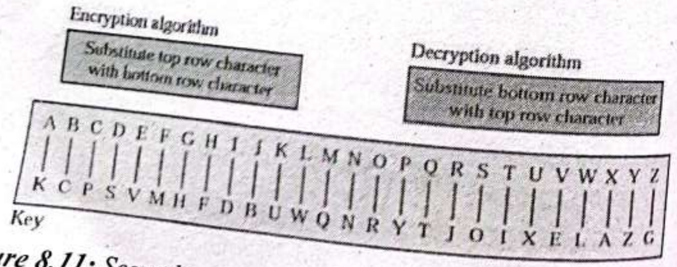

Encryption algorithm
Substitute top row character with bottom row character
Decryption algorithm
Substitute bottom row character with top row character
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
K C P S V M H F D B U W Q N R Y T J O I X E L A Z G
Key

Figure 8.11: Secret key encryption, monoalphabetic substitution

For example: Plaintext = CAT
Ciphertext = PKI

3. **Caesar Cipher**

It is one of the earliest known and simplest cipher. It is that type of substitution cypher in which each letter in the plaintext is shifted a certain number of places (depends on the value of key) down the alphabet. For example, with a shift of 1, A would be replaced by B, B would become C, and so on.

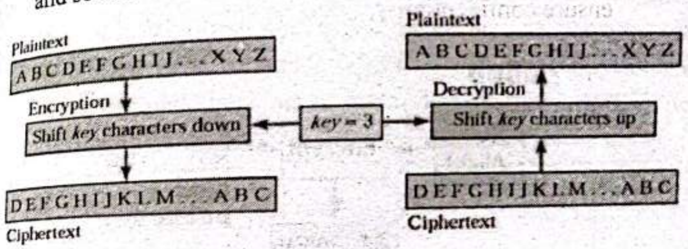


Figure 8.12: Key encryption: caesar cipher

8.9.2 Public Key Cryptography (Asymmetric Cryptography)

Secret-key systems are efficient because the amount of computation required to encrypt or decrypt a message is manageable, but they have a big drawback: *the sender and receiver must both be in possession of the shared secret key*. They may even have to get together physically for one to give it to the other. To get around this problem, public-key cryptography is used (Diffie and Hellman, 1976).

Public Key Cryptography, this system has the property that distinct keys are used for encryption and decryption and that given a well-chosen encryption key, it is virtually impossible to discover the corresponding decryption key. Under these circumstances, the encryption key can be made public and only the private decryption key kept secret.

In other words, in public key cryptography we use distinct keys for encryption and decryption.

Two of the best-known uses of public key cryptography are:

- □ **Public key encryption**

In public key encryption, message is encrypted with a recipient's public key. The message cannot be decrypted by anyone who does not possess the matching private key, who is thus presumed to be the owner of that key and the person associated with the public key. This is used in an attempt to ensure confidentiality.

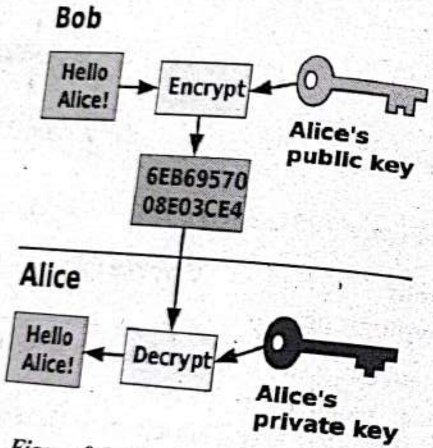


Figure 8.13: Public key encryption

As in above figure, in an asymmetric key encryption scheme, anyone can encrypt messages using the public key, but only the holder of the paired private key can decrypt. Security depends on the secrecy of the private key. The most commonly used public key algorithm is known as RSA.

- □ **Digital signatures**

Public key encryption is efficient if the message is short. If the message is long, a public key encryption is inefficient to use. The solution to this problem is to let the sender sign a digest of the document instead of the whole document. The sender creates a miniature version (digest) of the document and then signs it, the receiver checks the signature of the miniature version. The hash function is used to create a digest of the message. The hash function creates a fixed-size digest from the variable-length message. The two most common hash functions used: MD5 (Message Digest 5) and SHA-1 (Secure Hash Algorithm 1). The first one produces a 120-bit digest while the second one produces a 160-bit digest. A hash function must have two properties to ensure the success : First, the digest must be one way, i.e., the digest can only be created from the message but not vice versa. Second, hashing is a one-to-one function, i.e., two messages should not create the same digest.

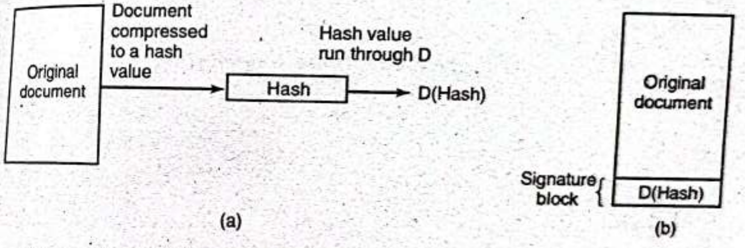


***Figure 8.14:** (a). Computing a signature block. (b). What the receiver gets.*

The miniature version (digest) of the message is created by using a hash function. The digest is encrypted by using the sender's private key. After the digest is encrypted, then the encrypted digest is attached to the original message and sent to the receiver. The receiver receives the original message and encrypted digest and separates the two. The receiver implements the hash function on the original message to create the second digest, and it also decrypts the received digest by using the public key of the sender. If both the digests are same, then all the aspects of security are preserved.

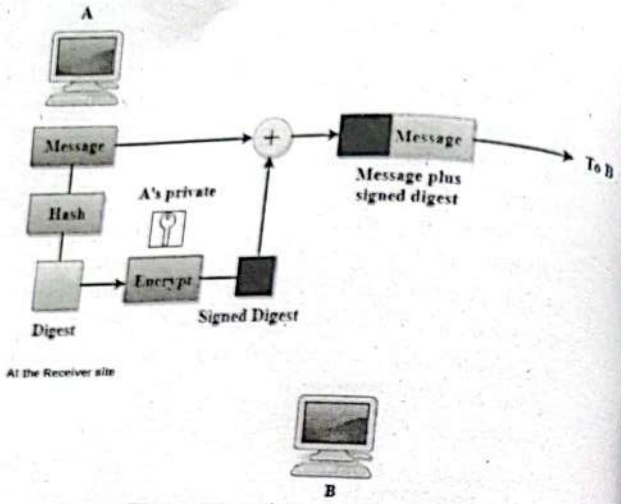


Figure 8.15: At the sender side

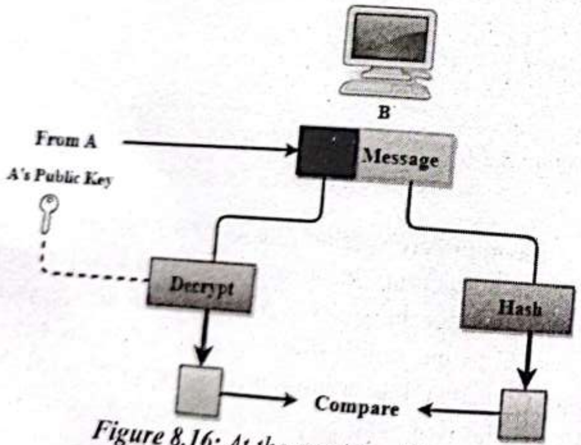


Figure 8.16: At the receiver side

Note that *Public Key Infrastructure (PKI)* is defined as a set of roles, policies, hardware, software and procedures needed to create, manage, distribute, use, store and revoke digital certificates and manage public-key encryption. The purpose of a PKI is to

facilitate the secure electronic transfer of information for a range of network activities such as e-commerce, internet banking and confidential email.

Table 8.1: Differences between secret key encryption and public key encryption

Basis for Comparison	Secret Key Encryption	Public Key Encryption
Define	Secret Key Encryption is defined as the technique that uses a single shared key to encrypt and decrypt the message.	Public Key Encryption is defined as the technique that uses two different keys for encryption and decryption.
Efficiency	It is efficient as this technique is recommended for large amounts of text.	It is inefficient as this technique is used only for short messages.
Other name	It is also known as Symmetric Key encryption.	It is also known as Asymmetric Key Encryption.
Speed	Its speed is high as it uses a single key for encryption and decryption.	Its speed is slow as it uses two different keys, both keys are related to each other through the complicated mathematical process.
Algorithms	The Secret key algorithms are DES, 3DES, AES & RCA.	The Public key algorithms are Diffie-Hellman, RSA.
Purpose	The main purpose of the secret key algorithm is to transmit the bulk data.	The main purpose of the public key algorithm is to share the keys securely.

8.10 Authentication

In security, authentication is the process of determining whet[illegible]one (or something) is, in fact, who (or what) it is declared to be. Every secured computer system must require all users to be authenticated at login time. After all, if the operating system cannot be sure who the user is, it cannot know which files and other resources he can access.

Many people nowadays (indirectly) log into remote computers to do Internet banking, engage in e-shopping, download music, and other commercial activities. All of these things require authenticated login, so user authentication is an important topic. Having determined that authentication is often important, the next step is to find a good way to achieve it. Most methods of authenticating users when they attempt to login are based on one of three general principles, namely identifying :

1. Something the user knows.
2. Something the user has.
3. Something the user is.

The most widely used form of authentication is to require the user to type a login name and a password. Password protection is easy to understand and easy to implement. The simplest implementation just keeps a central list of (login-name, password) pairs. The login name typed in is looked up in the list and the typed password is compared to the stored password. If they match, the login is allowed; if they do not match, the login is rejected.

8.10.1 Authentication Using a Physical Object

Another method for authenticating users is to check for some physical object. Using a bank's ATM (Automated Teller Machine) starts out with the user logging in to the bank's computer via a remote terminal (the ATM machine) using a plastic card and a password (currently a 4-digit PIN code in most countries, but this is just to avoid the expense of putting a full keyboard on the ATM machine).

Information-bearing plastic cards come in two varieties: *magnetic stripe cards* and *chip cards*. Magnetic stripe cards hold about 140 bytes of information written on a piece of magnetic tape glued to the back of the card. Chip cards contain a tiny integrated circuit (chip) on them. Chip cards can be subdivided into two categories: *stored value cards* and *smart cards*. Stored value cards contain a small amount of memory (usually less than 1 KB) using ROM technology to allow the value to be remembered when the card is removed from the reader and thus the power turned off. There is no CPU on this card, so the value stored must be changed by an external CPU (in the reader). The smart cards which currently have something like a 4-MHz 8-bit CPU, 16 KB of ROM, 4 KB of ROM, 512 bytes of scratch RAM, and a 9600-bps communication channel to the reader.

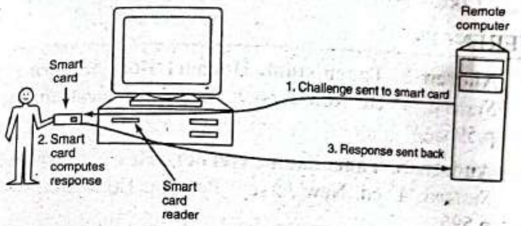


Figure 8.17: Use of a smart card for authentication

8.10.2 Authentication Using Biometrics

The third authentication method measures physical characteristics of the user like fingerprint or voiceprint reader. A typical biometrics system has two parts: *enrollment* and *identification*. During enrollment, the user's characteristics are measured and the results digitized. Then significant features are extracted and stored in a record associated with the user. The record can be kept in a central database (e.g., for logging in to a remote computer), or stored on a smart card that the user carries around and inserts into a remote reader (e.g., at an ATM machine). The other part is identification. The user shows up and provides a login name. Then the system makes the measurement again. If the new values match the ones sampled at enrollment time, the login is accepted; otherwise it is rejected.

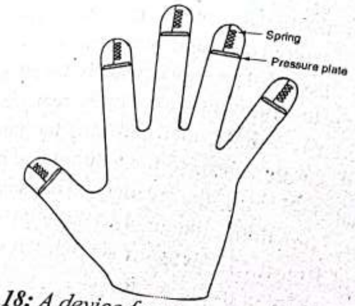


Figure 8.18: A device for measuring finger length.

REFERENCES

[1] **Andrew S. Tanenbaum, Herbert Bos.** *Modern Operating Systems.* 4th ed. New Jersey: Pearson Education, Inc., 2015, p.596.

[2] **Andrew S. Tanenbaum, Herbert Bos.** *Modern Operating Systems.* 4th ed. New Jersey: Pearson Education, Inc., 2015, p.596.

[3] **William Stallings.** *Operating Systems: Internals and Design Principles.* 7th ed. New Jersey: Pearson Education, Inc., 2012, p.616.

[4] **William Stallings.** *Operating Systems: Internals and Design Principles.* 7th ed. New Jersey: Pearson Education, Inc., 2012, pp.646-647.

[5] Souppaya, M., and Scarfone, K. Guide to Malware Incident Prevention and Handling for Desktops and Laptops. NIST Special Publication SP 800-83, July 2013.

https://antivirus.best/virus-types-differences

https://www.kaspersky.com/resource-center/threats/computer-viruses-vs-worms

https://www.kaspersky.com/resource-center/threats/types-of-spyware

https://www.cloudflare.com/learning/bots/what-is-a-bot/

https://www.avast.com/c-what-is-a-logic-bomb

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

EXERCISE

1. What is a trap door? Explain in firewalls and access control lists.
2. What are the attacks from inside? Explain about Malware and Spyware?
3. Explain protection domain and access control list (ACL).
4. Explain the domain-object and ACL. How are these mechanisms implemented for security?
5. Write short notes on: Information security model and security attack.
6. Write about types of network security attacks.
7. Write about Cryptography, Security Policy.
8. Explain the types of attacks. Explain how you can implement security and protection on all components of a system.
9. The use if internet is possible cause of a security breach. Describe the major threats by which a system connected to the internet is always prone to attack. Explain.
10. Explain about types of security attack.
11. Explain about Protection domain and cryptography.
12. Explain about Caesar cipher.
13. Explain about public key cryptography and protection domain.

Chapter 9

System Administration

9.1 Administration Tasks

System administrators are the persons who are responsible for setting up and maintaining the system or server. The duties/tasks of a system administrator are wide-ranging, and vary widely from one organization to another. Sysadmins are usually charged with installing, supporting, and maintaining servers or other computer systems, and planning for and responding to service outages and other problems. Other duties may include scripting or light programming, project management for systems-related projects.

Some of the tasks (duties and responsibilities) of system administrator are given below:

1. System startup and shutdown
2. Opening and closing user accounts
3. Helping users to set up their working environment
4. Maintaining user services
5. Allocating disk space and re-allocating quotas when the needs grow
6. Installing and maintaining software
7. Installing new devices and upgrading the configuration
8. Provisioning the mail and internet services
9. Ensuring security of the system
10. Maintaining system logs and profiling the users
11. System accounting
12. Reconfiguring the kernel whenever required
13. Database and Network Administration
14. Backup and disaster recovery

15. Web service administration & configurations
16. Implement the policies for the use of the computer system and network
17. Provide documentation and technical specifications to IT staff for planning and implementing new or upgrades of IT infrastructure

9.2 User Account Management

User accounts are the method by which an individual is identified and authenticated to the system. Managing user accounts and groups is an essential part of system administration within an organization.

The reasons for user accounts are:

- To verify the identity of each individual using a computer system.
- To permit the per-individual tailoring of resources and access privileges.

Resources can include files, directories, and devices. Controlling access to these resources is a large part of a system administrator's daily routine; often the access to a resource is controlled by groups. Groups are logical constructs that can be used to cluster user accounts together for a common purpose. For example, if an organization has multiple system administrators, they can all be placed in one system administrator group. The group can then be given permission to access key system resources. In this way, groups can be a powerful tool for managing resources and access. User accounts have several different components to them. First, there is the *username*. The *password* is next, followed by the *access control information*.

System administrators must be able to respond to the changes that are a normal part of day-to-day life in an organization. When a new person joins an organization, they are normally given access to various resources (depending on their responsibilities). They may be given a place to work, a phone, and a key to the front door. They may also be given access to one or more of the computers in your organization. It is the responsibility of a system administrator to see that this is done promptly and appropriately. Further the person may change his role or may terminate the job. In such cases, system administrators need to keep updated.

9.3 Start and Shutdown Procedures

Unix systems, on being powered on, usually require that a choice be made to operate either in single or in multiple-user mode. Most systems operate in multi-user mode. However, system administrators use single-user mode when they have some serious reconfiguration or installation task to perform. Family of Unix systems emanating from System V usually operate with a run level. The single-user mode is identified with *run levels*, otherwise there are levels from 0 to 6. The run level 3 is the most common for multi-user mode of operation.

On being powered on, Unix usually initiates the following sequence of tasks:

1. The Unix performs a sequence of self-tests to determine if there are any hardware problems.
2. The Unix kernel gets loaded from a root device.
3. The kernel runs and initializes itself.
4. The kernel starts the init process. All subsequent processes are spawned from init process.
5. The init checks out the file system using fsck.
6. The init process executes a system boot script.
7. The init process spawns a process to check all the terminals from which the system may be accessed. This is done by checking the terminals defined under */etc/ttytab* or a corresponding file. For each terminal a getty process is launched. This reconciles communication characteristics like baud rate and type for each terminal.
8. The *getty* process initiates a login process to enable a prospective login from a terminal.

During the startup, we notice that fsck checks out the integrity of the file system. In case the fsck throws up messages of some problems, the system administrator has to work around to ensure that there is a working configuration made available to the users.

9.4 Various Tools

9.4.1 AWK Tool

Awk is an extremely versatile programming language for working on files. It is designed for text processing and typically used as a data extraction and reporting tool. This language is a standard feature of most UNIX-like operating systems. AWK, takes its name from the initials of its authors — Aho, Weinberger and Kernighan. It is a scripting language that processes data files, especially text files that are organized in rows and columns.

The basic function of *awk* is to search files for lines (or other units of text) that contain certain patterns. When a line matches one of the patterns, *awk* performs specific actions on that line.

Syntactically, a rule consists of a *pattern* followed by an *action*. The action is enclosed in braces to separate it from the pattern. Newlines usually separate rules. Therefore, an awk program looks like this:

```
pattern { action }
pattern { action }
...
```

As in above syntax, the basic AWK program consists of patterns and actions — if the pattern is missing, the action is applied to all lines, or else, if the action is missing, the matched line is printed. There are two types of buffers used in AWK – the record buffer and field buffer. The latter is denoted as $1, $2… $n, where 'n' indicates the field number in the input file, i.e., $ followed by the field number (so $2 indicates the second field). The record buffer is denoted as $0, which indicates the whole record.

For example, to print the first field in a file, use the following command:

```
awk '{print $1}' filename
```

To print the third and first field in a file, use the command given below:

```
awk '{print $3, $1}' filename
```

AWK scripts are divided into the following three parts – BEGIN (pre-processing), body (processing) and END (post-processing).

```
BEGIN {begins actions}
Patterns{Actions}
END {ends actions}
```

BEGIN is the part of the AWK script where variables can be initialized and report headings can be created. The processing body contains the data that needs to be processed, like a loop. END or the post-processing part analyses or prints the data that has been processed.

```
#Begin Processing
BEGIN {print "To find the total marks & average"}
{
#body processing
tot = $2+$3+$4
avg=tot/3
print "Total of " $1 ":", tot
print "Average of "$1 ":", avg
}
#End processing
END{print "---Script Finished---"}
Input file is named as awkfile
Input file (awkfile)
Aby 20 21 25
Amy 22 23 20
```

```
Running the awk script as :
awk –f awscript awkfile

Output
To find the total marks & average
Total of Aby is : 66
Average of Aby is : 22
Total of Amy is : 65
Average of Amy is : 21.66
```

The system variables used by AWK are listed below.

- *FS*: Field separator (default=whitespace)
- *RS*: Record separator (default=\n)
- *NF:* Number of fields in the current record
- *NR*: Number of the current record
- *OFS*: Output field separator (default=space)
- *ORS*: Output record separator (default=\n)
- *FILENAME*: Current file name

9.4.2 Search Tool

The grep command stands for "global regular expression print", and it is one of the most powerful and commonly used commands in Linux. grep searches one or more input files for lines that match a given pattern and writes each matching line to standard output.

grep Command Syntax

```
grep [OPTIONS] PATTERN [FILE...]
```

the items in square brackets are optional.

- OPTIONS - Zero or more options. Grep includes a number of options that control its behavior.
- PATTERN - Search pattern.
- FILE - Zero or more input file names

For example, to display all the lines containing the string bash *from the /etc/passwd file, you would run the following command:*

```
grep bash /etc/passwd
```

Output :

```
root:x:0:0:root:/root:/bin/bash
linuxize:x:1000:1000:linuxize:/home/linuxize:/bin/bash
```

9.4.3 Sort Tools

SORT command is used to sort a file, arranging the records in a particular order. Using options in sort command, it can also be used to sort numerically. The sort command is a command line utility for sorting lines of text files. It supports sorting alphabetically, in reverse order, by number, by month and can also remove duplicates.

Suppose you create a data file with name *file.txt*

Command :

```
$ cat > file.txt
abhishek
chitransh
satish
rabin
naveen
divyam
binju
harsh
```

Sorting a file : Now use the sort command

Syntax : $ sort filename.txt

Command:

```
$ sort file.txt
```

Output :

```
abhishek
binju
chitransh
divyam
harsh
naveen
rabin
satish
```

9.4.4 Make Tool

GNU Make is a tool which controls the generation of executables and other non-source files of a program from the program's source files. Make gets its knowledge of how to build your program from a file called the *makefile*, which lists each of the non-source files and how to compute it from other files. When you write a program, you should write a makefile for it, so that it is possible to use Make to build and install the program. A *rule* in the makefile tells Make how to execute a series of commands in order to build a *target* file from source files. It also specifies a list of *dependencies* of the target file. This list should include all files (whether source files or other targets) which are used as inputs to the commands in the rule.

Here is what a simple rule looks like:

```
target: dependencies ...
        Command
        ...
```

When you run Make, you can specify particular targets to update; otherwise, Make updates the first target listed in the makefile. Of course, any other target files needed as input for generating these targets must be updated first.

Make uses the makefile to figure out which target files ought to be brought up to date, and then determines which of them actually need to be updated. If a target file is newer than all of its dependencies, then it is already up to date, and it does not need to be regenerated. The other target files do need to be updated, but in the right order: each target file must be regenerated before it is used in regenerating other targets.

EXERCISE

1. Suppose you are employed as a system administrator of ICT, Pulchowk campus. Detail your roles, also suggest the blowing ideas to maintain a secure and reliable system.
2. Write short notes:
   a. Roles of system administration
   b. Shell scripts
3. What are the roles of system administrators for an organization? How can you increase operating system performance if you are selected as a system administrator?
4. Write short notes on: Duties and responsibilities of system administrator.
5. What is system administration? How is a special user different from a general user? Explain.
6. Write short notes on:
   i. Roles of system administrator.
   ii. Duties and responsibilities of system administration.
   iii. Administration tasks.

BIBLIOGRAPHY

William Stallings. *Operating Systems: Internals and Design Principles.* 7th ed. New Jersey: Pearson Education, Inc., 2012.

Andrew S. Tanenbaum, Herbert Bos. *Modern Operating Systems.* 4th ed. New Jesey: Pearson Education, Inc., 2015.

Silberschatz, Galvin, Gagne. *Operating System Concepts.* 8th ed. New Jersey: John Wiley & Sons, Inc., 2009.

Milan Milenkovic. *Operating Systems: Concept and Design.* 2nd ed. New York: Mc-Graw Hill Education, 2001.

H. M. Deitel, P. J. Deitel, D. R. Choffnes. *Operating Systems.* 3rd ed. New Jersey: Pearson Education, Inc., 2004.

William Stallings, Lawrie Brown. *Computer Security: Principles and Practice.* 3rd ed. New Jersey: Pearson Education, Inc., 2015.

Matt Bishop. *Computer Security: Art and Science. 1st. ed.* New Jersey: Pearson Education, Inc., 2003.