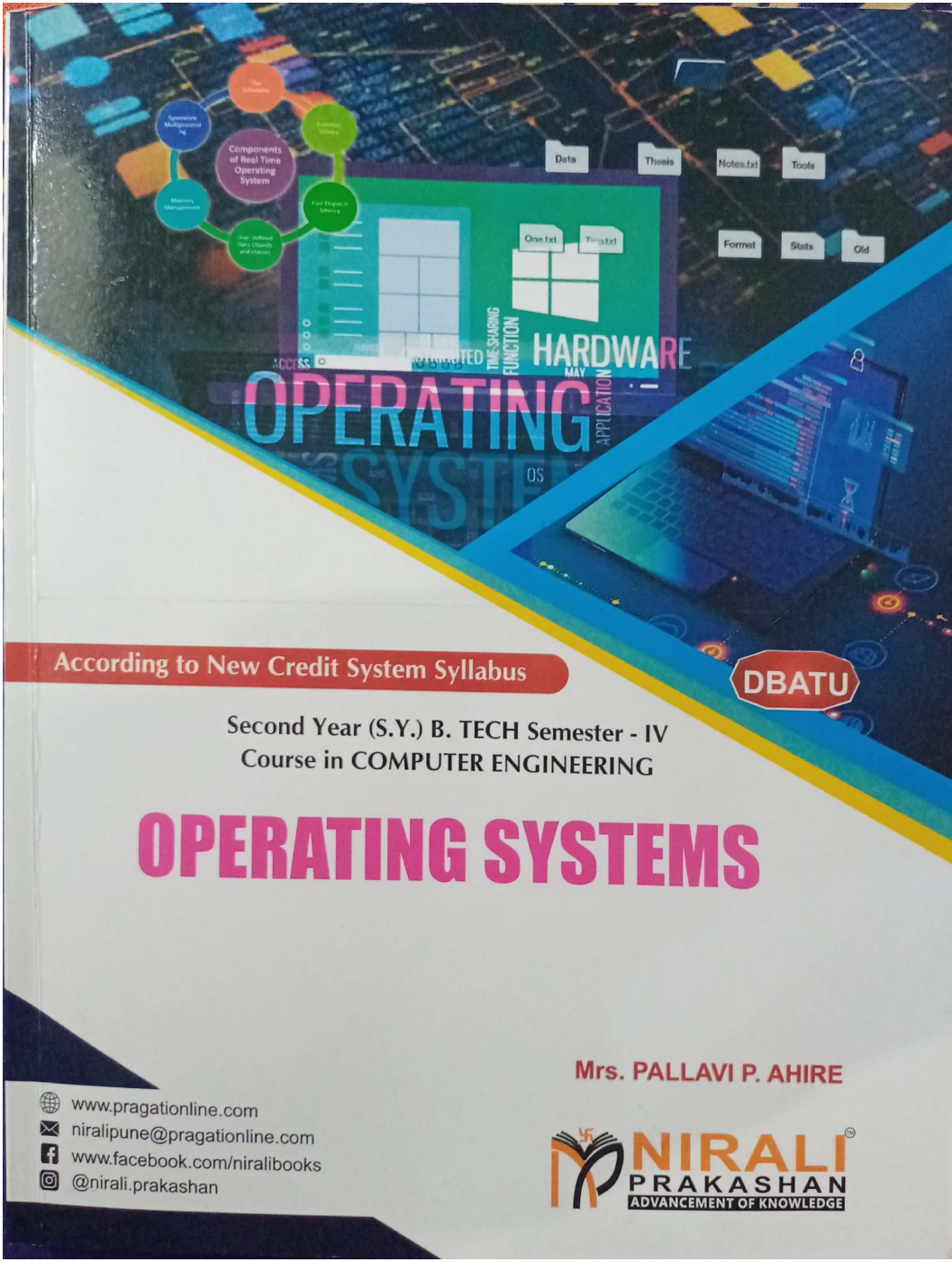


According to New Credit System Syllabus

DBATU

Second Year (S.Y.) B. TECH Semester - IV
Course in COMPUTER ENGINEERING

OPERATING SYSTEMS

Mrs. PALLAVI P. AHIRE

www.pragationline.com
niralipune@pragationline.com
www.facebook.com/niralibooks
@nirali.prakashan

NIRALI PRAKASHAN
ADVANCEMENT OF KNOWLEDGE

CONTENTS

✠ ✠ ✠

Unit I

INTRODUCTION AND OPERATING SYSTEM STRUCTURES

1.1 INTRODUCTION

- **Operating Systems** (OS) are an important part of any computer system. Also learning an OS is an essential part of any computer science education. This field is undergoing rapid changes as computers are now prevalent in virtually every application, like games, government and multinational firms.
- An OS is an intermediary between the user of computer and computer hardware. The purpose of an OS is to provide an environment in which a user can execute programs in a convenient and efficient manner.
- An OS is a program that manages the computer hardware. It also provides a basis for application program. Mainframe OS are designed primarily to optimize utilization of hardware. Personal Computer (PC) operating systems support complex games, business applications and everything in between, OS for handheld computers are designed to provide an environment in which a user can easily interface with computer to execute programs. Thus, some OS are designed to be convenient, others to be efficient and remaining are a combination of the two.

1.1.1 What Operating Systems Do?

- A computer system can be divided roughly into four components, the hardware, the operating system, the application programs and the users.

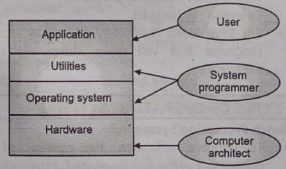


Fig. 1.1 : Functional layers in a computer system

- The above Fig. 1.1 shows the functional layers in operating system. The hardware, the CPU, memory and I/O devices, provides computing resources for the system.
- The application programs such as word processors, spread sheets, compilers and web browsers define the way in which these resources are used to solve users computing problems.
- The OS controls and co-ordinates the use of hardware among the various application programs for various users. We can also view a computer system as consisting of hardware, software and data. The OS provides the means for proper use of these resources in the operating of computer system.

  Example : An OS is similar to government.
- It performs no useful function by itself. It simply provides an environment within which other programs can do useful work.

User View

- The user view of computer varies according to the interface being used.
- Most computer users sit in front of a PC, consisting of a monitor, keyboard, mouse and system unit. Such a system is designed for one user to monopolize its resources. The goal is to maximize the work that user is performing. In this case, the OS is designed mostly for use with some attention paid to performance and none paid to resource utilization, how various hardware and software resources are shared. Performance is important to user but rather than resource utilization, such systems are optimized for single user experience.
- In other cases, a user sits at a terminal connected to a mainframe or minicomputer, other users are accessing the same computer through other terminals. These users share resources and may exchange information. The OS in such a case is designed to maximise resource utilisation to assure that all available CPU time, memory and I/O are used efficiently and that no individual user takes more than fair share.
- In other cases, users sit at workstations connected to network of other workstations and servers. These users have dedicated resources at their disposal, but they also share resources such as networking and servers file, computer and print servers. Therefore, their OS is designed to compromise between individual usability and resource utilization.

- Some computers have little or no user view. e.g. Embedded computers in home devices and automobiles may have numeric keypad and may turn indicator lights on or off to show status, but they and their OS are designed primarily to run without user intervention.

System View

- From computers point of view, OS is program most intimately involved with hardware. In this context OS is 'Resource Allocator'. A computer system has many resources that are required to solve a problem, CPU time, memory space, File storage, I/O devices.
- The OS acts as a manager of these resources, it must decide how to allocate resources to specific program and users so that it can operate the computer system efficiently and fairly. Resource allocation is especially important where many users access the same mainframe or minicomputer.

Operating System Structure

- One of the most important aspect of OS is the ability to multiprogramming. A single user cannot always keep either CPU or I/O devices busy. Multiprogramming increases CPU utilization by organizing jobs so that the CPU always has one for execution.
- The OS always has several jobs in memory simultaneously. This set of jobs can be subset of jobs kept in job pool which contains all jobs that enter into the system since the number of jobs that can be kept simultaneously in memory is usually smaller than the number of jobs that can be kept in job pool.
- The OS picks and begins to execute one of the jobs in memory. In non-multiprogram system, CPU would sit idle. In multiprogram system, OS simply switches to and executes another job. When that job needs to wait, the CPU switches to another job and so on.
- Eventually, the first job finishes waiting and gets the CPU back. As long as at least one job needs to execute, the CPU is never idle.

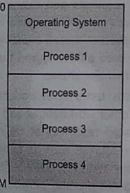


Fig. 1.2 : Memory layout for multiprogramming system

Ex-Lawyer

- As practical example of lawyer, he never works for single client all the time. Lawyer starts the work of another client if the first client is waiting for dates. Multiprogrammed system provides an environment in which the various system resources are utilized effectively but do not provide user interaction with the system.
- Time sharing or multitasking is logical extension of multiprogramming. In time sharing system, CPU executes multiple jobs by switching among them, but switches occur so frequently that the user can interact with each program while it is running.

Objectives of an Operating System

Main Objective of an operating system program is to :

- Make the computing convenient, lowering down complexity of manual tasks.
- Make the computing efficient, saving users valuable time and money.
- Evolve as a general purpose tool that will cover all possible domains.

This can be summarized using simple words as:

- Provide user friendly interface that will make computing a fun job.
- Make tentative arrangement of desired resources playing a role of "Resource Manager".
- Provide appropriate methods for data storage and communication.
- Provide reliable, safe and conductive environment for the execution of user application.
- Support user activities through efficient services like
  - **For Programmers:** Availability of utilities that will be of use while writing computing, executing, testing and debugging programs.
    Example : Debuggers
  - **For End Users:** Availability of various application programs that will make computing a great experience.
    Example : MS-Office

1.2 TYPES OF OPERATING SYSTEMS

1. Batch Processing System

- To improve utilization, the concept of batch operating system was developed. Jobs with similar needs were batched together and were run through the computer as a group. The programmer would leave their program with operator who in turn sort program into batches with similar requirements. The central idea behind the simple batch processing scheme is to use monitor.

- With the use of this type of operating system, the user no longer has direct access to machine. Rather the user submits the job and card to computer operator who batches the job together sequentially and places the entire batch on an I/P device, for use by monitor.

Table 1.1 : Memory Layout for Resident Monitor

Loaders
Device Drivers
Job Sequencing
Control Language Interpreter
User Program Area

Monitor handles scheduling problem. A batch of job is queued up and jobs are executed as rapidly as possible, with no intervening idle time.

Advantages of Batch Processing System :

- Move much of work of operator to the computer.
- Increased performance since it was possible for job to start as soon as the previous job finished.

Disadvantages of Batch Processing System :

- Turn around time can be large from user standpoint.
- More difficult to debug program.
- Due to the lack of protection scheme, one batch job can affect pending jobs.
- A job could corrupt the monitor, thus affecting pending jobs and a job could enter an infinite loop.

2. Uniprogramming System

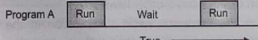


Fig. 1.3 : Uniprogramming

- Above Fig. 1.3 shows uniprogramming system where we have a single program in execution.
- The processor spends a certain amount of time for executing, till requirements of I/O or exceptions are generated. It waits till requirements are fulfilled and then resumes operation.

3. Multiprogrammed Batched System

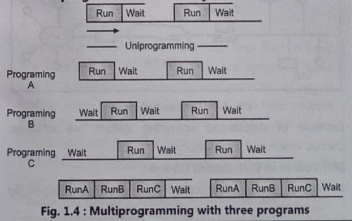


Fig. 1.4 : Multiprogramming with three programs

- Above Fig. 1.4 shows multiprogramming system where we have more than one program in execution.
- As the Fig. 1.4 shows multiprogramming while program A is executing, program B is waiting, processor is switched to B then A goes on waiting and so on. This way, processor always has a program to execute and it never sits idle.

4. Time Sharing System

- With the use of multiprogramming, batch processing can be efficient. However, for many jobs it is desirable to provide a mode in which user interacts directly with the computer. Time sharing is logical extension of multiprogramming. Multiple jobs are executed by the CPU switching between them but, the switches occur so frequently that the user may interact with each program while it is running. Processors time is shared among multiple users. An interactive or hands on computer system provides online communication between the user and the system. The user gives instruction to the operating system or to a program directly and receives an immediate response.
- Batch systems are appropriate for executing large jobs that need little interaction. The user can submit jobs and return later for the results, need not wait while the job is processed. The user submits the command and then waits for the result. The time shared operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of time shared computer.
- Time sharing system provides a mechanism for concurrent execution, which requires CPU scheduling schemes.

Advantages of Time Sharing System :

- Easy to use.
- User friendly.
- Intermediate between all hardware and software of system.
- No need to have any technical lag.

Disadvantages of Time Sharing System :

- If any problems affected in OS, you may lose all the contents which have been stored already.
- Unwanted user can use the system.

5. Clustered System :

- Clustered system is a group of computer systems connected with a high speed communication link.
- Each computer system has its own memory and peripheral devices. Clustering provides high availability.
- Clustering systems are integrated with hardware cluster and software cluster. Hardware cluster supports sharing

of high performance disks, software cluster is in the form of unified control of the computer system in a cluster.

- Cluster nodes contains layers of cluster software and it runs on the node. Each node monitors the network, whether the cluster is working properly or not. If the monitored computer fails, then the monitoring computer takes the control and ownership of its storage and other resources. It restarts the application of failed computer. The user and other clients will get the resources. The failed computer can remain down until it is repaired.

Clustered Systems are Divided into Two Groups:

1. Asymmetric clustering.
2. Symmetric clustering.

1. **Asymmetric Clustering :** In this type of clustering, one machine is always in the hot standby mode and other machines run its application. The hot standby machines only monitor the active server. When active server fails, then hot standby machine takes the control and becomes the active server.
2. **Symmetric Clustering :** In this, more than one host runs the applications, it uses all of the available hardware for operation. Host monitors working of other host for smooth operation of the cluster.

Fig. 1.5 shows Structure of cluster system.

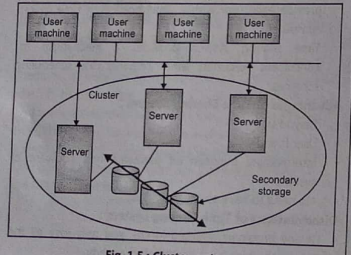


Fig. 1.5 : Cluster system

- Clustered system and symmetric multiprocessor provide a support for high demand applications.
- Clustered system is different then parallel system.
- Other type of clustered system is parallel cluster. It uses shared storage of data. Parallel cluster requires some special purpose software and special type of application. Example of parallel cluster is Oracle Parallel Server.
- Cluster technology changes rapidly. New developments are added to this cluster technology. Nowadays, global cluster concept is used for large organizations.

6. **Distributed Operating Systems :**

- Distributed operating system depends on networking for their operation. Distributed OS runs on and controls the resources of multiple machines. It provides resources sharing across the boundaries of a single computer system. It looks to users like a single machine OS. Distributed OS owns the whole network and makes it look like a virtual uniprocessor or may be a virtual multiprocessor.
- **Definition :** A distributed operating system is one that looks to its users like an ordinary operating system but runs on multiple independent CPUs.

Advantages of Distributed OS:

- **Resource Sharing :** Sharing of software resources such as software libraries, database and hardware resources such as hard disks, printers and CDROM can also be done in a very effective way among all the computers and the users.
- **Higher Reliability :** Reliability refers to the degree of tolerance against errors and component failures. Availability is one important aspect of reliability. Availability refers to the fraction of time for which system is available for user. Availability of hard disk can be increased by having multiple hard disks located at different sites. If one hard disk fails or is unavailable, the program can use some other hard disk.
- **Better Price Performance Ratio :** Reduction in the price microprocessor and increasing the computing power gives good price-performance ratio.
- Shorter responses time and higher throughput.
- **Incremental Growth :** To extend power and functionality of a system by simply additional resources to the system.

Fig. 1.6 shows the distributed system.

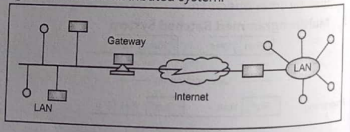


Fig. 1.6 : Distributed system

Example of distributed operating system are amoeba, chorus, mach and v-system.

Difficulties in Distributed OS are :

- There are no current commercially successfully examples.

- Protocol overhead can dominate computation costs.
- Hard to build well.
- Probably impossible to build at the scale of the internet.

7. **Network Operating System**

- Network operating system is designed by using existing operating system.
- In network operating system, user can login to the remote resources and then access that resource. There are multiple machines connected by communication link. User can transfer data from their own machine to remote machine and vice versa.
- Telnet and remote file transfer are the example of network operating system.
- NOS supports multiple user account at the same time and enables concurrent access to shared resource by multiple clients.
- A function of network operating system is distributed over the network computers. It adds new function whenever required. Server machine supports multitasking. Operating system must be capable of executing multiple tasks at the same time.
- User knows the location of the resources before using that resource. Each machine has its own network operating system.

1.3 REAL-TIME OPERATING SYSTEMS

- Time constraints is the key parameter is real-time system. It controls autonomous system such as robots, satellites, air traffic control and hydro-electric dams.
- When user gives an input to the system, it must process within time limit and result is sent back. Real-time system fails if it does not give results within the time limit.
- Real-time system are of two types :
  1. Hard real-time and
  2. Soft real-time.
- Critical task is completed within the time limit in hard real-time system. All the delay in the system is fixed and time bounded. Existing general purpose operating system does not support the hard real-time system functions. Real-time task cannot keep waiting for longer time without allocating kernel.
- Soft real-time system is less restricted type. Soft real-time cannot guarantee that it will be able to meet the deadline under all conditions. Example of soft real-time system is digital telephone and digital audio.
- Real-time operating system uses priority scheduling algorithm to meet the response requirement of a real-time application. General real-time applications with their examples are listed below :
  - **Transportation :** Air traffic control and traffic light system.
  - **Communication :** Digital telephone
  - **Process Control :** Petroleum and paper mill
  - **Detecting :** Burglar system and radar system
  - **Flight Simulation :** Auto pilot shuttle mission simulator
- Memory management in real-time system is less demanding than in other types of multiprogramming operating system. Time critical device management is one of the main characteristics of the real-time system.

1.4 SYSTEM COMPONENTS

1.4.1 System Services

An OS provides an environment for the execution of programs. OS provides certain services to users and programs.

User Services

1. **User Interface (UI)**
   Almost all OS have UI, one is CLI (Command Line Interface) which uses text commands and method for entering them. Another is batch interface, in which commands and directives to control those commands are entered into files and those files are executed. Also, GUI is used which provides direct interface, choose from menu and make selection and keyboard to enter text.
2. **Program Loading and Execution**
   System must be able to load a program into memory and run that program.
3. **I/O Operation**
   Running program may require I/O, which may involve I/O or file. For specific devices, special function may be desired.
4. **File-System Manipulation**
   Program needs to read and write files and directories. They also need to create and delete them by name, search for given file and file information.
5. **Communication**
   There are many circumstances in which one process needs to exchange information with another process. Such communication may occur between processes that are executed on same computer or between the processes that are executing on different computers.

6. Error Detection

The OS needs to be constantly aware of possible errors. Errors may occur in CPU and memory hardware, in I/O devices such as parity error on type, connection failure on network and in user program such as arithmetic overflow, an attempt to access illegal memory location. For such type of error, the OS should take appropriate action to ensure correct and consistent computing.

Another set of OS functions exist for ensuring efficient operation of system itself.

1. Resource Allocation

When there are multiple users or jobs running at same time, resources must be allocated to each of them. Many different types of resources are managed by OS.

2. Accounting

We want to keep track of which users use how much and what kind of computer resources.

3. Protection and Security

When several separate processes are executed concurrently, it should not be possible for one process to intent.

1.4.2 Operating System Structure

1. Simple Structure

- The simple structure OS is MS-DOS. It provides most functionality in less space. It is not divided into modules.
- In MS-DOS, the interfaces and levels of functionality are not well separated. The whole system may crash if user program fails and it was also limited by the hardware.

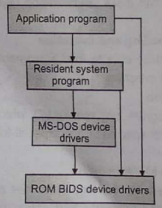


Fig. 1.7

2. Layered Approach

- In this structure, OS can be broken into pieces that are smaller. The OS retains much greater control over the computer and over the applications that make use of that computer. In layered approach, system is broken into a number of layers (levels).

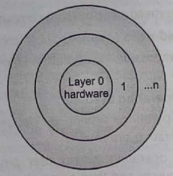


Fig. 1.8

- The bottom layer is hardware (layer 0), the highest layer (layer N) is the user interface. The OS layer is an implementation of abstract object made up of data and the operations that can manipulate that data.
- The main advantage of layered approach is simplicity of construction and debugging. The layers are selected so that each uses functions (operations) and services of only lower level layers. This approach simplifies debugging and system verification.
- The first layer can be debugged without any concern for the rest of systems. It uses only basic hardware. After debugging of first layer, its correct functioning can be assumed while the second layer is debugged. If an error is found during the debugging of a particular layer, the error must be on the layer, because the layers below it are already debugged. Thus, the design and implementation of the system is simplified.
- Each layer is implemented with only those operations provided by lower level layers. A layer does not need to know how these operations are implemented, it needs to know only what these operations do. Hence, each layer hides the existence of certain data structure, operations and hardware from higher level layers.
- The major difficulty with the layered approach involves appropriately defining the various layers. Because a layer can use only lower level layers, careful planning is necessary.

3. Microkernels

- The kernels become large and difficult to manage after UNIX expands. The kernels use microkernel approach. This method structures the OS by removing all non-essential components from the kernel and implementing them as system and user level programs. The result is smaller kernel.
- The main function of the microkernel is to provide a communication facility between the client program and various services that are also running in users space.

- Communication is provided by message passing e.g. if the client program wishes to access a file, it must interact with the file server. The client program and service never interact directly. Rather they communicate indirectly by exchanging messages with the microkernel.

Advantages :

- It is the case of extending the OS.
- All new services are added to user space and consequently do not require modification of the kernel.
- The resulting OS is easier to port from one hardware design to another.
- The microkernel also provides more security and reliability since most services are running rather than kernel processes.
- If a service fails, the rest of the OS systems remain untouched.

Program 1.1 :

Tru 64 UNIX
(digital UNIX)
QNX
(real-time OS)

Disadvantage :

- Microkernel suffers from performance decrease due to increased system function overhead.

(d) Modules

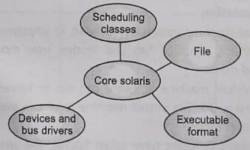


Fig. 1.9

With the OS itself, protection involves ensuring that all access to system resources is controlled. Security of system is also important.

1.4.3 System Calls

System call provides an interface to the services made available by OS.

For Example : Writing a simple program to read data from one file and copy it to another file. The first input that program will need is the name of the two files; input file and output file. These names can be specified in many ways depending on OS system design. One approach is for the program to ask the user for the names of two files. In an interactive system, this approach requires a sequence of system calls first to write message on screen and then to read from the keyboard the characters that define the two files.

- On mouse based and icon based systems, a menu of file names is usually displayed in window. The user can then use the mouse to select the source name and window can be opened for the destination name to be specified. This sequence requires many I/O system calls. Once the two file names are obtained, the program must open the input file and create output file. Each of these operations requires another system call. There are also possible error conditions for each operation. When the program tries to open the input file, it may find that there is no file of that name or that file is protected against access. In these cases, the program should print a message on the console (system call) and terminate abnormally (system call).

Program 1.2 : Example of How System Calls are Used

Ex. System call sequence

```
Acquire I/P file name
Write prompt to screen
Accept I/P
Acquire O/P file name
Write prompt to screen
Accept I/P
Open the I/P File
If file doesn't exist, abort
Create output file
If file exists, abort
Loop read from input file
Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally.
```

- If the input file exists, then we must create a new output file. We may find that there is already an output file with the same name. This situation may cause the

program to abort (system call) or we may delete existing file (system call). Now, both files are set up and we enter a loop that reads from input file (System call) and writes to output file (system call). Each read and write must return status information regarding various possible error conditions.

- Finally, after entire file is copied, the program may close both files (system call), write a message to console (system call) and terminate normally (system call).

1.4.4 System Programs

System program provides convenient environment for program development and execution.

- **File Management :** These programs create, delete, copy, rename, print files.
- **Status Information :** Some programs simply ask the system for date, time, amount of available memory or disk space or number of users.
- **File Modification :** Several text editors may be available to create and modify the contents of files stored on disk.
- **Programming Language Support :** Compilers, assemblers, debuggers for common programming languages (such as C, C++, Java) are provided to the user with OS.
- **Program Loading and Execution :** Program must be loaded into memory for execution. The system may provide absolute loaders, linkage editors.
- **Communication :** These programs provide the mechanism for creating virtual connections among processes, users and computer systems. They allow users to send messages to one another screen.

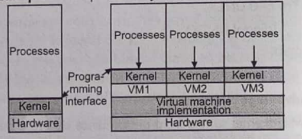
The best current methodology for OS design involves using object oriented programming techniques to create a modular kernel. Here, the kernel has a set of core components and dynamically links in additional services either during boot time or during runtime. Such a strategy uses dynamically loaded modules and is common in modern implementations of UNIX such as solars, Linux, Mac OS X.

Such a design allows the kernel to provide core services yet allows certain features to be implemented dynamically. It is more flexible than layered system in that any module can call any other module. This approach is like microkernel approach in that the primary module has only core functions and knowledge of how to load and communicate with other modules but it is more efficient because modules do not need to involve message passing in order to communicate.

1.5 VIRTUAL MACHINES

- Virtual machine is to abstract the hardware of single computer (CPU, memory, disk drives) into several different execution environments, thereby creating the illusion that each separate execution environment is running is own private computer.
- By using CPU scheduling, virtual memory techniques of an OS can create the illusion that a process has its own processor with its own (virtual) memory. The process has additional features, such as system calls and file systems that are not provided by the bare hardware.
- The virtual machine approach does not provide any such additional functionality but rather provides an interface that is identical to the underlying bare hardware. Each process is provided with a (virtual) copy of the underlying computer.
- Virtual machine is created to share the same hardware yet run several different execution environments concurrently.

For Example : VM operation System, IBM.



(a) Non-virtual machine **(b) Virtual machine**

Fig. 1.10 : System models

Implementation

- Virtual machine concept is difficult to implement. The underlying machine has two modes, user mode and kernel mode.
- The virtual machine software can run in kernel mode since it is OS. The virtual machine itself can execute in only user mode.
- The real I/O might have taken 100ms, the virtual I/O might take less time or more time.

Benefits

- In this environment there is complete protection of various system resources. Each virtual machine is completely isolated from all other virtual machines, so there are no protection problems. It is possible to share a minidisk and thus to share files.
- It is possible to determine network of virtual machine each of which can send information over the virtual communication network.

Examples :

(1) VMware

- VMware is a commercial application that abstracts Intel 80 × 86 hardware into isolated virtual machine. VMware runs as an application on host operating system such as Windows or Linux and allows this host system to concurrently run several different guest operating systems as independent operating system.
- A developer has designed an application and would like to test it on Linux, Free BSD, Windows NT, XP. One option is to obtain four different computers, each running a copy of one of these operating systems.
- Another alternative is to install Linux on a computer system and test the application than to install Free BSD and test and so on. This option allows use of the same physical computer but is time consuming, since we need to install a new operating system for each test. Such testing could be accomplished concurrently on the same physical computer using VMware.
- By this, the programmer could test the application on a host operating system and on three guest operating systems with each system running as a separate virtual machine.
- The architecture of such a system is shown in diagram. Linux is running as the host operating system. BSD, XP, and NT are running as guest operating system. The virtualization layer is the heart of VMware, as it abstracts the physical hardware into isolated virtual machines running as guest operating system. Each virtual machine has its own virtual CPU.

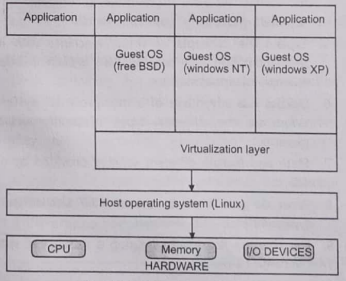


Fig. 1.11 : VMware architecture

- Memory
- Disk Drives
- Network Interfaces

(2) JAVA Virtual Machine

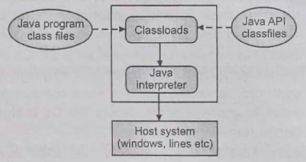


Fig. 1.12

- The JVM is a specification for an abstract computer. It consists of class loader and Java Interpreter that executes the architecture neutral byte codes.
- The class loader loads the compiled class files from both Java program and Java API for execution by Java Interpreter.

1.6 OPERATING SYSTEM DESIGN AND IMPLEMENTATION

Design Goals

- The first problem in designing a system is to define goals and specifications. At highest level, the design of system will be affected by the choice of hardware and type of system batch, time shared single user, multi-user, distributed.
- Requirement can be divided into two groups : User goals and System goals. Users desire certain obvious properties in the system.
- The system should be convenient to use, easy to learn, reliable, fast. The system should be easy to design, create, maintain and operate the system. The system should be flexible, reliable, error free and efficient.

Mechanisms and Policies

- Mechanism determines how to do something, policies determine what will be done. e.g. the timer construct is a mechanism for ensuring CPU protection, but deciding how long the timer is to be set for a particular user is a policy decision.
- Policy decisions are important for all resource allocation. Whenever it is necessary to decide whether or not to allocate resource, a policy decision must be made. Whenever the question is how rather than what, it is mechanism that must be determined.

Implementation

- Once an OS is designed, it must be implemented. Traditionally, OS have been written in assembly language. Now, they are most commonly written in higher level language such as C, C++. The Linux and

windows XP OS are written mostly in C, although there are some small sections of assembly code for device drivers and for saving and restoring the state of registers.

- The advantage of using higher level language is that code can be written faster, is more compact and is easier to understand and debug. The OS is easier to port to some other hardware.

Example: MS-DOS was written in Intel 8088 assembly language. It is available on only Intel family of CPU. The Linux OS, written in C and it is available on number of different CPU. The disadvantages of implementing an OS in a higher level language are reduced speed and increased storage requirements.

1.7 SYSTEM GENERATIONS

- It is possible to design, code, and implement an operating system specifically for one machine at one site. More commonly, however, operating systems are designed to run on any of a class of machines at a variety of sites with a variety of peripheral configurations.
- The system must then be configured or generated for each specific computer site, a process sometimes known as system generation SYSGEN.
- The operating system is normally distributed on disk, on CD-ROM or DVD-ROM, or as an "ISO" image, which is a file in the format of a CD-ROM or DVD-ROM. To generate a system, we use a special program.
- This SYSGEN program reads from a given file, or asks the operator of the system for information concerning the specific configuration of the hardware system, or probes the hardware directly to determine what components are there. The following kinds of information must be determined.
- What CPU is to be used? What options (extended instruction sets, floating-point arithmetic, and so on) are installed? For multiple CPU systems, each CPU may be described.
- How will the boot disk be formatted? How many sections, or "partitions", will it be separated into and what will go into each partition?
- How much memory is available? Some systems will determine this value themselves by referencing memory location after memory location until an "illegal address" fault is generated. This procedure defines the final legal address and hence the amount of available memory.
- What devices are available? The system will need to know how to address each device (the device number), the device interrupt number, the device's type and model, and any special device characteristics.
- Once this information is determined, it can be used in several ways. At one extreme, a system administrator can use it to modify a copy of the source code of the operating system. The operating system then is completely compiled. Data declarations, initializations, and constants, along with conditional compilation, produce an output-object version of the operating system that is tailored to the system described.
- System generation involves simply creating the appropriate tables to describe the system. The major differences among these approaches are the size and generality of the generated system and the ease of modifying it as the hardware configuration changes. Consider the cost of modifying the system to support a newly acquired graphics terminal or another disk drive. Balanced against that cost, of course, is the frequency (or infrequency) of such changes.

EXERCISE

1. What is an operating system? State and explain the basic functions of operating system.
2. What is the purpose of system call and how do system calls relate to operating system?
3. Discuss various architectures of operating system.
4. Explain the following terms :
   (i) System call (ii) Multi-programming
   (iii) Multi-processing (iv) Mode switch
5. Explain the concepts of virtual machines with its implementation and benefits. Also explain in brief, example of virtual machine.
6. Discuss the advantage of a multiprocessor system. What are the different types of multiprocessor systems.
7. State and explain different services provided by an OS.
8. What do you mean by system call? List different types of system calls available.
9. Describe the flow control during a system call with the help of a neat diagram.

✠ ✠ ✠

Unit II

PROCESSES AND CPU SCHEDULING

2.1 PROCESS CONCEPT

- When the OS runs a program, this program is loaded into memory and the control is transferred to this program's first instruction. Then, the program starts to run.
- A process is a program in execution.
- A process is more than a program, because the former has a program counter, stack, data section and so on. Moreover, multiple processes may be associated with one program.
- Components of the process are : Object program, data resources and status of the process execution.
- Object program i.e. code to be executed. Data is used for executing the program. While executing the program, it may require some resources. A last component is used for verifying the status of process execution.
- Program is passive entity, active program becomes process when executable file is loaded into memory.
- Attributes held by a process include :
  1. Hardware state 2. Memory
  3. CPU 4. Progress (executing)
- A process is more than program code, which is sometimes known as the text section. Data section containing global variables and heap containing memory are dynamically allocated during runtime.
- Stack containing temporary data such as function parameters, return addresses, local variables etc.
- Two processes may be associated with same program; they are nevertheless considered two separate execution sequences.
- A program can create multiple processes. For example, Internet Explorer.

2.1.1 Difference between Process and Program

Table 2.1

Sr. No.	Process	Program
1.	Process is active entity.	Program is passive entity.
2.	Process is a sequence of instruction executions.	Program contains the instructions.
3.	Process exists in a limited span of time.	A program exists at single place and continues to exit.
4.	Process is a dynamic entity.	Program is static entity.

...Conti.

2.2 PROCESS STATES

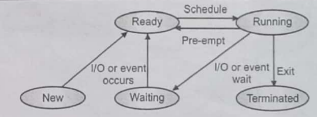


Fig. 2.1 : Process state transition diagram

When process executes, it changes state. Process state is defined as the current activity of the process Fig. 2.1 shows the general form of the process state transition diagram. Process state contains five states. Each process is one of the states. The states are listed below :

- New
- Ready
- Running
- Waiting
- Terminated

- **New :** A process that has just been created.
- **Ready :** Ready processes are waiting to have the processor allocated to them by the operating system so that they can run.
- **Running :** The process that is currently being executed. A running process possesses all the resources needed for its execution, including the processor.
- **Waiting :** A process that cannot execute until some event occurs such as the completion of an I/O operation. The running process may become suspended by invoking an I/O routine.
- **Terminated :** A process that has been released from the pool of executable processes by operating system.

Whenever process changes state, the operating system reacts by placing the process PCB in the list that corresponds to its new state. Only one process can be running on any processor at any instant and many processes may be ready and waiting state.

2.2.1 Suspended Processes

Characteristics of Suspended Process

- Suspended process is not immediately available for execution.
- The process may or may not be waiting on an event.
- For preventing the execution, process is suspended by OS, parent process, process itself and an agent.
- Process may not be removed from the suspended state until the agent orders the removal.
- Fig. 2.2 shows the process state transition diagram with suspended state.

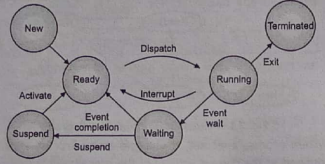


Fig. 2.2 : Suspended state

Swapping is used to move all of a process from main memory to disk. When all the processes in main memory are in blocked state, the OS can suspend one process by putting it in the suspended state and transferring it to disk.

Reasons for Process Suspension :

- Swapping
- Timing
- Interactive user request
- Parent process request

- **Swapping :** OS needs to release required main memory to bring in a process that is ready to execute.
- **Timing :** Process may be suspended while waiting for the next time interval.
- **Interactive User Request :** Process may be suspended for debugging purpose by user.
- **Parent Process Request :** To modify the suspended process or to co–ordinate the activity of various descendants.

Example : Explain whether the following transitions between process states are possible or not. If possible, give the example.

1. Running – Ready
2. Running – Waiting
3. Waiting – Running
4. Running – Terminated

Solution :

1. For changing the state from running to ready is possible. For example, when a process time quantum expires.
2. **Running :** Waiting is also possible. When a process issues an I/O request.
3. **Waiting :** Running is not possible. From waiting state, the process goes to the ready state then running.
4. **Running :** Termination is possible, when a process terminates itself.

2.2.2 Process Control Block (PCB)

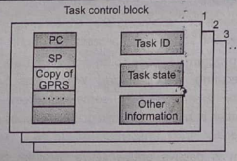


Fig. 2.3 : Process control block

Each process contains the Process Control Block (PCB). PCB is the data structure used by the operating system. Operating system groups all information that needs about particular process. Fig. 2.3 shows the process control block.

- **Pointer :** Pointer points to another process control block pointer is used for maintaining scheduling list.
- **Process State :** Process state may be new, ready, running, waiting and so on.
- **Program Counter :** It indicates the address of next instruction to be executed for this process.
- **Event Information :** For a process in the blocked state this field contains information concerning the event for which the process is waiting.
- **CPU Register :** It includes general purpose register, pointers, index registers and accumulators etc. Number of register and type of register totally depends upon the computer architecture.
- **Memory Management Information :** This information may include the value of base and limit register. This information is useful for deallocating the memory when the process terminates.

- **Accounting Information :** This information includes the amount of CPU and real time used, time limits, job or process number, account numbers etc.

Process control block also includes the information about CPU scheduling, I/O resource management, file management information, priority and so on. The PCB simply serves as the repository for any information that may vary from process to process.

When a process is created, hardware registers and flags are set to the values provided by the loader or linker. Whenever that process is suspended, the contents of a processor register are usually saved on the stack and the pointer to the related stack frame is stored in the PCB. In this way, the hardware state can be restored when the process is scheduled to run again.

2.3 PROCESS SCHEDULING

- Multiprogramming operating system allows more than one process to be loaded into the executable memory at a time and for the loaded process to share the CPU using time multiplexing.
- The scheduling mechanism is the part of the process manager that handles the removal of running process from the CPU and the selection of another process on the basis of a particular strategy.

2.3.1 Scheduling Queues

- When the process enters into the system, they are put into job queue. This queue consists of all processes in the system. The operating system also has other queues.
- Device queue is a queue for which a list of processes is waiting for a particular I/O device. Each device has its own device queue. Fig. 2.4 shows the queuing diagram of process scheduling.
- In the Fig. 2.4, queue is represented by rectangular box. The circles represent the resources that serve the queues. The arrows indicate the flow of processes in the system.

Queues are of Two Types : Ready queue and set of device queues. A newly arrived process is put in the ready queue. Processes are waiting in ready queue for allocating CPU. Once the CPU is assigned to the process, then process will execute. While executing the process, one of the several events could occur :

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

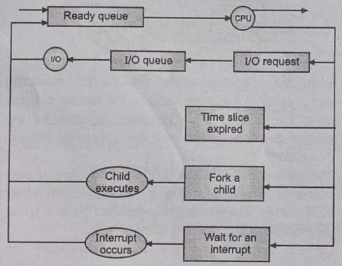


Fig. 2.4 : Queuing diagram

2.3.2 Two State Process Model

Process may be in one of two states :

1. Running 2. Not running

Fig. 2.5 (a) shows state transition diagram for two state process model.

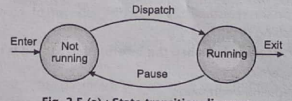


Fig. 2.5 (a) : State transition diagram

When new process is created by OS, that process enters into the system in the

Not Running State :

Fig. 2.5 (b) shows the queuing diagram for two state process model.

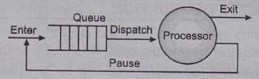


Fig. 2.5 (b) : Queuing diagram for two state process

Processes that are not running are kept in queue, waiting their turn to execute. Each entry in the queue is a printer to a particular process. Queue is implemented by using linked list. Use of dispatcher is as follows. When a process interrupted, that process is transferred in the waiting queue. If the process has completed or aborted, the process is discarded. In either case, the dispatcher then selects a process from the queue to execute.

2.3.3 Context Switch

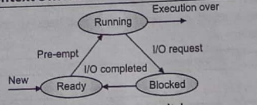


Fig. 2.5 (c) : Context switch

- When the scheduler switches the CPU from executing one process to executing another, the context switcher saves the content of all processor registers for the process being removed from the CPU in its process descriptor.
- The context of a process context switch time is pure overhead context switching. It can significantly affect performance, since modern computers have a lot of general and status registers to be saved.
- Context switch times are highly dependent on hardware support. Context switch requires (n + m) b × k (n + m) b × k time units to save the state of the processor with n general registers and m status registers, assuming store operations are required to save a single register and each store instruction requires k time units.
- Some hardware systems employ two or more sets of processor registers to reduce the amount of context switching time.

When Process is Switched the Information Stored is :

- Program counter
- Scheduling information
- Base and limit register value
- Currently used register
- Changed state
- I/O status
- Accounting

2.4 PROCESS DESCRIPTION

- Operating system schedules and dispatches processes for execution by the processor, allocates resources to processes and responds to requests by user processes for basic services.
- Fig. 2.6 shows the processes and resources.

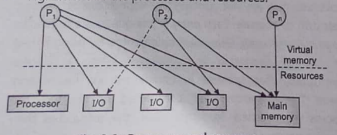


Fig. 2.6 : Processes and resources

Operating System Control Structure

- If the operating system is to manage processes and resources, it must have information about the current status of each process and resource.
- Fig. 2.7 shows general structure of operating system control tables.

Operating system maintains four different tables :

1. Memory
2. I/O
3. File
4. Process

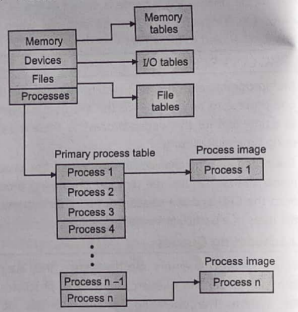


Fig. 2.7 : General structure of OS control tables

1. Memory Tables :

Memory tables are used to keep track of both main and secondary memory. Some of main memory is reserved for operating system.

Memory tables must include the following information :

- Allocation of memory to processes.
- Allocation of secondary memory to processes.
- Any protection attributes of blocks of main or virtual memory.
- Any information needed to manage virtual memory.

2. I/O Tables :

I/O tables are used by operating system, the I/O devices and channels of the computer system.

- At anytime, an I/O device may be available or assigned to a particular process.
- OS needs to know the status of I/O operation.

3. File Tables :

- File table provides information about the existence of files, their location on secondary memory, their current status and other attributes.
- All above information is maintained and used by a file management system.

4. Process Tables :

Operating system uses process tables to manage processes.

2.5 OPERATION ON PROCESS

2.5.1 Modes of Execution

Most processors support at least two modes of execution :

1. **User Mode :** Less privileged mode.
2. **Kernel Mode :** More privileged mode.
   - User programs execute in the user mode.
   - Kernel mode is also called system mode or control mode.
   - Typically, there is a bit in the Program Status Word (PSW) that indicates the mode of execution.
   - Linux operating system uses level 0 for the kernel mode and one other level for user mode.

2.5.2 Process Creation

- Operating system creates a new process with the specified or default attributes and identifier. A process may create several new sub–processes.

  Syntax for creating new process is :

  `CREATE (processID, attributes).`

- Two names are used in the process. They are parent process and child process.
- Parent process is a creating process. Child process is created by the parent process. Child process may create another subprocess. Child process is created by the parent process. So, it forms a tree of processes. Fig. 2.8 shows a tree of processes.
- When operating system issues a CREATE system call, it obtains a new process control block from the pool of free memory. Fills the fields with provided and default parameters and inserts the PCB into the ready list. Thus, it makes the specified process eligible to run the process.

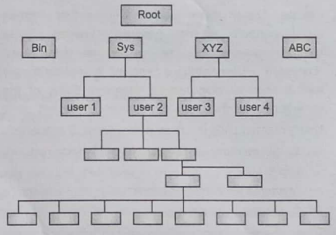


Fig. 2.8 : Tree of process

- When a process is created it requires some parameters. These are priority, level of privilege, requirement of memory, access right, memory protection information etc.
- Process will need certain resources, such as CPU time, memory, files and I/O devices to complete the operation. When process creates a subprocess, that subprocess may obtain its resources directly from the operating system otherwise it uses the resource of parent process.
- When a process creates a new process, two possibilities exist in terms of execution :
  1. The parent continues to execute concurrently with its children.
  2. The parent waits until some or all of its children have terminated.
- For address space, two possibilities occur:
  1. The child process is a duplicate of the parent process.
  2. The child process has program loaded into it.
- A new process is created with the help of system called "fork()". Process given a birth to a child by forking. So, one who has executed fork becomes a parent and a child that in given a birth, becomes child of the parents.
- There are two types of processes :
  1. Processes that are executed in foreground.
  2. Processes that are executed in background.
- Background processes are specially medicated. Specially medicated in the sense, there is a standard process of creating a background process. These processes are isolated from all input as well as output

devices. This is done intentionally so that nobody should disturb normal execution process. These background processes are termed as (sometimes) "Daemons". "Demonizing a process" is another theory that is used to manufacture daemons. Study of the same is beyond the scope of our study at present.

- Newly created child is a copy of its parent. It processes:
  - Same memory area, but own distinct memory space.
  - The same execution environment.
  - Same resources allotted.
- Some of the situations where we can witness process creation are:
  - When user logs in to the system.
  - When user begins execution of program.
  - On service execution by operating system itself, e.g., Web service.
  - When one program kicks off execution of another.

2.5.3 Terminate a Process

- DELETE system call is used for terminating a process. A process may delete itself or by other process. A process can cause the termination of another process via an appropriate system call, The operating system reacts by reclaiming all resources allocated to the specified process, closing files opened by or for process.
- PCB is also removed from its place of residence in the list and is returned to the free pool. The DELETE service is normally invoked as a part of order program termination.
- Following are the reasons for terminating the child process by parent process :
  - The task given to the child is no longer required.
  - Child has exceeded its usage of some of resources that it has been allocated.
  - Operating system does not allow a child to continue if the parent terminates.
- Some of the reasons because of which a process may be terminated are:

➢ **Exit After Normal Execution:**

Most processes terminate because they have done their job.

➢ **Exit Due to Error:**

System halts due to logical or syntax error or any illegal operation. For example, a user program encounters illegal memory reference.

➢ **Exit on Fatal Error:**

An error caused by process due to a bug in program for example, executing an illegal instruction, referring non–existing memory or dividing by zero.

➢ **Exit in Case, Killed by Another Process:**

A process executes a system call telling the operating system to terminate some other process using the process ID as its identity.

2.5.4 Reasons for Process Creation and Termination

1. Reasons for Process Creation

- **New Batch Job :** Operating system is provided with a batch job control stream, usually on tape or disk.
- **Interactive Log On :** A user at a terminal logs onto the system.
- Created by operating system a function on behalf of a user program, without user having to wait.

2. Reasons for Process Termination

- Normal operating is completed by process.
- Process has run longer than the specified total time length.
- Process requires more memory than the system can provide.
- Process tries to access memory location that it is not allowed to access.
- Protection error process attempts to use a resource that it is not allowed to use.
- Invalid instruction process attempts to execute a non–existent instruction.
- **Privileged Instruction :** Process attempts to use an instruction reversed for the operating system.
- **Data Misuse :** A piece of data is of the wrong type or not initialized.
- If deadlock exists, operating system terminates process.
- When parent terminates, operating system terminates the child process.

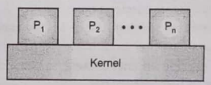
Execution of the Operating System

- The operating system is the same way as ordinary computer software in the sense that the operating system is a set of programs executed by the processor.
- The operating system frequently relinquishes control and depends on the processor to restore control to the operating system.

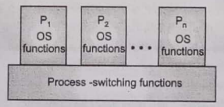
- If the operating system is just a collection of programs and if it is executed by the processor just like any other program, is the operating system a process? If so, how is it controlled? These interesting questions have inspired a number of design approaches. Fig. 2.9 illustrates a range of approaches that are found in various contemporary operating system.

Non-Process Kernel

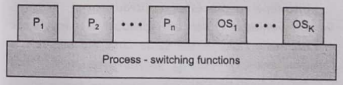
- One traditional approach, common of many older operating systems, is to execute the kernel of the operating system outside of any process. Fig. 2.9 shows this approach, when the currently running process is interrupted or issues a supervisor call, the mode context of this process is saved and control is passed to the kernel.
- The operating system has its own region of memory to use and its own system stack for controlling procedure cause and return. The operating system can perform any desired functions and the context of the interrupted process, which causes execution to resume in interrupted user process.



(a) Separate kernel



(b) Operating system functions execute within user process



(c) Operating system execute processes

Fig. 2.9 : Relationship between operating system and user processes

- Alternatively, the operating system can complete the functions of saving the environment of process and proceed to schedule and dispatch another process. Whether this happens depends on the reason for the interruption and circumstances at the time.
- In any case, the key point here is that the concept of process is considered to apply to user program. The operating system code is executed as a separate entity that operates in privileged mode.

Execution within User Processes

- An alternative that is common with operating system on smaller computers (PCs, work stations) is to execute virtually all operating system software in the context of user process. This view is that the operating system is primarily a collection of routines that the user calls to perform various functions, executed within the environment of the user process.
- This is illustrated in Fig. 2.10, at any given point, the operating system is managing process images. Each image includes not only the regions illustrated in figure but also program data and stack areas for kernel process.

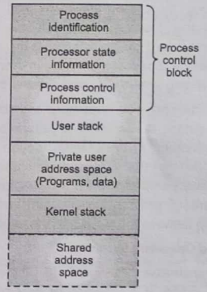


Fig. 2.10 : Process image : Operating system executes within user space

- Operating system code and data are in the shared space and are shared by all user processes.
- When an interrupt, trap or supervisor call occurs, the processor is placed in kernel mode and control is passed to the operating system. To pass control from a user program to the operating system the mode context is saved and a mode switch takes place to an operating system routine. However, execution continues within the same process.
- If the operating system, upon completion of its work determines that the current process should continue to run, then a mode switch resumes the interrupted program within the current processes. This is one of the key advantages of this approach.

- A user program has been interrupted to employ some operating system routine and then resumed, and all of this has occurred without incurring the penalty of two process switches. If however, it is determined that a process switch is to occur rather than returning to the previously executing program then control is passed to a process switching routine. This routine may or may not execute in the current process depending on system design.
- At same point, however, the current process has to be placed in a non–running state and another process designated as the running process. During this place, it is logically most convenient to view execution as taking place outside of all processes.
- In a way, this view of operating system is remarkable. Simply put at certain points in time, a process will save its state information, choose another process to run from among those that are ready and relinquish control to that process. The reason this is not an arbitrary and indeed chaotic situation is that during the critical time, the code that is executed in user process is shared operating system code and not user code.
- Because of the concept of user mode and kernel mode, the user cannot temper with or interfere with the operating system routines, even though they are executing in the user's process environment. This further reminds us that there is a distinction between the concept of process and program and that the relationship between the two is not one to one.

Process-Based Operating System

- Another alternative, illustrated to implement the operating system as a collection of system processes. As in the other options, the software that is part of the kernel executes in a kernel code. In this case, however, major kernel functions are organised as separate processes. Again, there may be a small amount of process switching code that is executed outside of any process.
- This approach has several advantages. It imposes a program design discipline that encourages the use of a modular operating system with minimal, clean interfaces between the modules. In addition, some non-critical operating systems are conveniently implemented as separate processes.
- For example, we mentioned earlier a monitor program that records the level of utilization of various resources and the rate of progress of the user processes in the system. Because this program does not provide a particular service to any active process, it can only be invoked by operating system. As a process, the function can run at an assigned priority level and be interleaved with other processes under dispatcher control.
- Finally, implementing the operating system as a set of processes is useful in a multiprocessor or multicomputer environment, in which some of the operating system services can be shipped out to dedicated processors, improving performance.

2.6 CO-OPERATING PROCESSES

- Processes executing concurrently in the operating system may be either independent processes or cooperating processes. A process is independent if it cannot affect or be affected by the other processes executing in the system.
- A process is cooperating if it can affect or be affected by the other processes executing in the system. Any process that shares data with other processes is a cooperating process.
- There are several reasons for providing an environment that allows process cooperation:
  - **Information Sharing :** Since several users may be interested in the same piece of information, we must provide an environment to allow concurrent access to such information.
  - **Computation Speedup :** If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing cores.
  - **Modularity :** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.
  - **Convenience :** Even an individual user may work on many tasks at the same time. For instance, a user may be editing, listening to music, and compiling in parallel.

2.7 THREADS

The two characteristics of OS, resource ownership and scheduling are independent and could be treated independently by the OS. To distinguish the two characteristics, the unit of dispatching is usually referred to as a thread or lightweight process, while the unit of resource ownership is usually referred to as a process or task.

Multithreading

- Multithreading refers to the ability of an OS to support multiple, concurrent paths of execution within a single process. The traditional approach of a single thread of execution per process, in which the concept of a thread is not recognized, is referred to as a single–threaded approach.
- The two arrangements shown in the left half of Fig. 2.11 are single–threaded approaches. MS-DOS is an example of an OS that supports a single user process and a single thread. Other operating systems, such as some variants of UNIX, support multiple user processes but only support one thread per process.
- The right half of Fig. 2.11 depicts multithreaded approaches. A Java run–time environment is an example of a system of one process with multiple threads. Use of multiple processes, each of which supports multiple threads. This approach is taken in Windows, Solaris, and many modern versions of UNIX, among others.

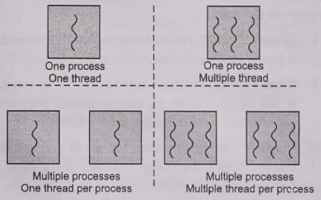
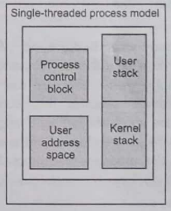
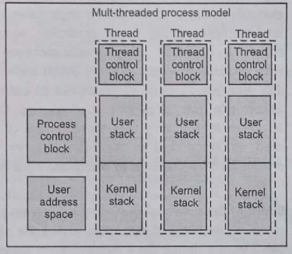


Fig. 2.11 : Threads and processes

- In a multithreaded environment, a process is defined as the unit of resource allocation and a unit of protection. The following are associated with processes :
  - A virtual address space that holds the process image.
  - Protected access to processors, other processes (for inter-process communication), files, and I/O resources (devices and channels).
- Within a process, there may be one or more threads, each with the following :
  - A thread execution state (Running, Ready, etc.).
  - A saved thread context when not running; one way to view a thread is as an independent program counter operating within a process.
  - An execution stack.
  - Some per–thread static storage for local variables.
  - Access to the memory and resources of its process, shared with all other threads in that process.
- Fig. 2.12 illustrates the distinction between threads and processes from the point of view of process management. In a single–threaded process model (i.e., there is no distinct concept of thread), the representation of a process includes its process control block and user address space, as well as user and kernel stacks to manage the call/return behavior of the execution of the process.
- While the process is running, it controls the processor registers. The contents of these registers are saved when the process is not running. In a multithreaded environment, there is still a single process control block and user address space associated with the process, but now there are separate stacks for each thread, as well as a separate control block for each thread containing register values, priority, and other thread–related state information.



(a) Single–threaded process models



(b) Multithreaded process models

Fig. 2.12

- Thus, all of the threads of a process share the state and resources of that process. They reside in the same address space and have access to the same data.
- When one thread alters an item of data in memory, other threads see the results if and when they access that item. If one thread opens a file with read privileges, other threads in the same process can also read from that file.
- An example of an application that could make use of threads is a file server. As each new file request comes in, a new thread can be spawned for the file management program. Because a server will handle many requests, many threads will be created and destroyed in a short period.
- If the server runs on a multiprocessor computer, then multiple threads within the same process can be executing simultaneously on different processors.
- Further, because processes or threads in a file server must share file data and therefore coordinate their actions, it is faster to use threads and shared memory than processes and message passing for this coordination.

➢ **Foreground and Background Work :** For example, in a spreadsheet program, one thread could display menus and read user input, while another thread executes user commands and updates the spreadsheet. This arrangement often increases the perceived speed of the application by allowing the program to prompt for the next command before the previous command is complete.

➢ **Asynchronous Processing :** Asynchronous elements in the program can be implemented as threads. For example, as a protection against power failure, one can design a word processor to write its Random Access Memory (RAM) buffer to disk once every minute. A thread can be created whose sole job is periodic backup and that schedules itself directly with the OS; there is no need for fancy code in the main program to provide for time checks or to coordinate input and output.

➢ **Speed of Execution :** A multithreaded process can compute one batch of data while reading the next batch from a device. On a multiprocessor system, multiple threads from the same process may be able to execute simultaneously. Thus, even though one thread may be blocked for an I/O operation to read in a batch of data, another thread may be executing.

➢ **Modular Program Structure :** Programs that involve a variety of activities or a variety of sources and destinations of input and output may be easier to design and implement using threads.

Thread Functionality

Like processes, threads have execution states and may synchronize with one another. We look at these two aspects of thread functionality in turn.

Thread States : As with processes, the key states for a thread are Running, Ready, and Blocked. Generally, it does not make sense to associate suspend states with threads because such states are process–level concepts. In particular, if a process is swapped out, all of its threads are necessarily swapped out because they all share the address space of the process.

There are four basic thread operations associated with a change in thread state :

1. **Spawn :** Typically, when a new process is spawned, a thread for that process is also spawned.
2. **Block :** When a thread needs to wait for an event, it will block (saving its user registers, program counter, and stack pointers).
3. **Unblock :** When the event for which a thread is blocked occurs, the thread is moved to the Ready queue.
4. **Finish :** When a thread completes, its register context and stacks are deallocated.

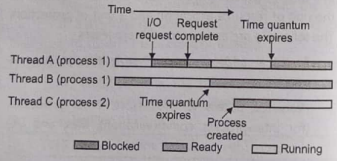


Fig. 2.13 : Multithreading example on a uniprocessor

On a uniprocessor, multiprogramming enables the interleaving of multiple threads within multiple processes. In the example of Fig. 2.13, three threads in two processes are interleaved on the processor. Execution passes from one thread to another either when the currently running thread is blocked or when its time slice is exhausted.

Thread Synchronization : All of the threads of a process share the same address space and other resources, such as open files. Any alteration of a resource by one thread affects the environment of the other threads in the same process. It is therefore necessary to synchronize the activities of the various threads so that they do not interfere with each other or corrupt data structures. For example, if two threads each try to add an element to a doubly linked list at the same time, one element may be lost or the list may end up malformed.

Thread Functionality : Just like process, threads also have execution state like Running, Ready, and Blocked. It is not possible to associate suspended state with thread as it is used with process. All threads related to same process will be swapped out if a process itself is to be swapped out.

Four basis operations undertaken while thread changes its execution state are:

1. **Spawn:** Spawning of new process also leads to spawning of its threads. All such threads are provided with their own data structure as we have studied just sometimes before.
2. **Block:** A thread is waiting for an event to occur can extends opportunity to another related or unrelated thread for execution after saving register, stack and other processing related data of that moment.
3. **Unblock:** Once the event for which thread was waiting is completed thread is moved back to ready queue.
4. **Finish:** After complete execution, thread is moved out of working space and memory acquired is de–allocated.

2.8 TYPES OF THREADS

User-Level and Kernel–Level Threads

There are two broad categories of thread implementation : User-Level Threads (ULTs) and Kernel-Level Threads (KLTs).

1. **User-Level Threads (ULT) :**

In a pure ULT facility, all of the work of thread management is done by the application and the kernel is not aware of the existence of threads. Fig. 2.14 illustrates the pure ULT approach. Any application can be programmed to be multithreaded by using a threads library, which is a package of routines for ULT management. The threads library contains code for creating and destroying threads, for passing messages and data between threads, for scheduling thread execution, and for saving and restoring thread contexts.

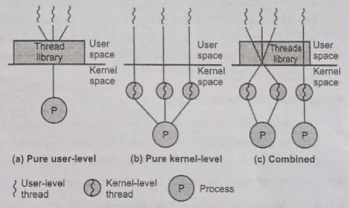


Fig. 2.14 : User-level and Kernel – level threads

By default, an application begins with a single thread and begins running in that thread. This application and its thread are allocated to a single process managed by the kernel. At any time that the application is running (the process is in the Running state), the application may spawn a new thread to run within the same process.

2. **Kernel-Level Threads (KLT) :**

In a pure KLT facility, all of the work of thread management is done by the kernel. There is no thread management code in the application level, simply an Application Programming Interface (API) to the kernel thread facility. e. g. Windows OS.

Fig. 2.14 depicts the pure KLT approach. The kernel maintains context information for the process as a whole and for individual threads within the process. Scheduling by the kernel is done on a thread basis.

Drawbacks

- The kernel can simultaneously schedule multiple threads from the same process on multiple processors.
- If one thread in a process is blocked, the kernel can schedule another thread of the same process.

Combined Approaches

- Some operating systems provide a combined ULT/ KLT facility. In a combined system, thread creation is done completely in user space, as is the bulk of the scheduling and synchronization of threads within an application. The multiple ULTs from a single application are mapped onto some (smaller or equal) number of KLTs.

- In a combined approach, multiple threads within the same application can run in parallel on multiple processors, and a blocking system call need not block the entire process. If properly designed, this approach should combine the advantages of the pure ULT and KLT approaches while minimizing the disadvantages. e.g. Solaris OS

Many-To-Many Relationship

- The idea of having a many–to–many relationship between threads and processes has been explored in the experimental operating system TRIX. In TRIX, there are the concepts of domain and thread.
- A domain is a static entity, consisting of an address space and "ports" through which messages may be sent and received. A thread is a single execution path, with an execution stack, processor state, and scheduling information.

Table 2.2 : Relationship between Threads and Processes

Threads : Processes	Description	Example Systems
One : One	Each thread of execution is a unique process with its own address space and resources.	Traditional UNIX implementations
Many : One	A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.	Windows NT, Solaris, Linux, OS/2, OS/390, MACH
One : Many	A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.	Ra (Clouds), Emerald
Many : Many	Combines attributes of Many : One and One : Many cases	TRIX

One-To-Many Relationship

- In the field of distributed operating systems, there has been interest in the concept of a thread as primarily an entity that can move among address spaces.
- A thread in Clouds is a unit of activity from the user's perspective. A process is a virtual address space with an associated process control block. Upon creation, a thread starts executing in a process by invoking an entry point to a program in that process. Threads may move from one address space to another and actually span computer boundaries.

POSIX Thread Programming in Linux

- Traditional process is a process with single thread of control whereas process in modern operating system is the one with multiple thread control.
- Let us try to create thread using Linux library, basically called a POSIX library. It is abbreviated a portable operating system interface. It is an IEEE standard of library that provides various system calls or for:
  - Create Thread,
  - Schedule thread,
  - Messages of thread.
- POSIX standards involve pthread specification for thread programming. So, pthread by definitions is a set f 'C' language, constructs and procedure calls, realized using pthread.h header fie (a thread library) and a thread library that may be a pert of another library.
- Table 2.3 that gives short description of POSIX pthread system calls used for thread management.

Table 2.3 : Description of POSIX Pthread system

Sr. No.	POSIX Pthread System Calls	Description
1.	Pthread_create	Create a new thread
2.	Pthread_exit	Terminate the thread that is calling
3.	Pthread_join	Waiting for a particular thread exit
4.	Pthread_yeild	Free CPU so that another thread can use
5.	Pthread_attr_init	Create thread and initialize its attributes
6.	Pthread_attr_destroy	Remove thread attribute structure from memory

Example of Thread Creation:

```
int pthread_create
{
    Pthread_t * tid;
    Pthread_attr_*attr;
    ((void*)(*my_routine)(void *))
    Void argument
}
Pthread_t*tid
```

Thread Identifier that is a unique identity provided by operating system to thread.

```
Pthread_attr_*attr;
```


We can create two types of threads

1. Attachable
2. Detachable

We can state the thread type as follows:

- After successfully creation of thread, it executes certain function. Third argument here is nothing but the address of the function/routine that we want to be executed by newly created thread.
- The last argument is the parameter that we need to pass to the function that our thread is to execute.
- So when a call is given to pthread_create function, address of function is passed as an argument to pthread_create, so appropriate library routine start execution function along with continuation of main function. So we will be having two threads. However number of calls to pthread_create, we can have those many number of threads.

```
#include <pthread.h>
int main()
{
    pthread_t tid;
    pthread_create(stid)
    void *mythread(void * att)
    NULL; mythread; NULL;
    if de* /attribute/attachable
void * mythread (void*)
    {
    printf("Hi");
    return (0);
    }
}
```

Steps to Compilation of this Program

gcc	O	Xy	Mythread.c	–l	Pthread
Compile	Output option	Name of o/p	Name of program	Directive to pthread library	Name of library

- One critical problem in the last program is once threads are create i.e.

  Thread 1 –main

  Thread 2 – mythread

Both threads will be executed in parallel. But if Thread 1 completes before Thread 2 then obviously will terminate abnormally. Hence, we need some mechanism to make Thread 1 to wait till Thread 2 completes.

Hence, we will have pthread_ join before the last closing bracket (flower brace) of main program.

```
int pthread_join (pthread_calling thread id tid,
(void *) * valence fund *)
```

- If there is any value return by mythread back to calling function, then it is returned back to main through second argument

```
int pthread_detach
{
    pthread_t dethreadid
}
```

Above syntax is used in case thread does not return anything and hence for main there is no need to wait for threads completion.

Multi–Core Process and Threads

Techniques used to achieve parallelism like pipelining and superscalar processing generated problem like:

(a) Overheating
(b) Complex design
(c) Delays in operation due to overheads of synchronization
(d) Demands air conditioning for temperature control imposes financial burdens.
  - Tremendous flow of multithreaded application
  - Instruction level parallelism brought in limitation since it achieves parallelism at machine instruction level.
  - Pipelining based execution aggressive speculative execution data analysis. This is very costly and cannot be trusted all the times.

Thread–Level Parallelism (TLP)

With the instruction of advanced architectures, single processor based multithreading because a reality. Due to this achievement of parallelism at a granular stage became possible.

- This is parallelism on a more coarser scale.
- Server can serve each client using a separate thread (Web server, database server).
- It is possible to handle complex logic of AI, high definition graphics of application like games using separate threads.
- With multi code architecture, it is possible to get maximum performance by exploiting Thread Level Parallelism which is not possible in case of single–core superscalar processors.

Simultaneous Multithreading

- Trying to achieve parallelism through pipelining has potential problems.
- It can get stalled and a lot of time is wasted in clearing the data for execution of new instruction.
- Lot of time is wasted while waiting for data to be loaded in memory.
- Lot of time is wasted in waiting for result of ongoing computation.
- Many other computational units may remain unused.
- From Fig. 2.15 given below we can see that without SMT we can allow only one thread to execute at one time with processor.

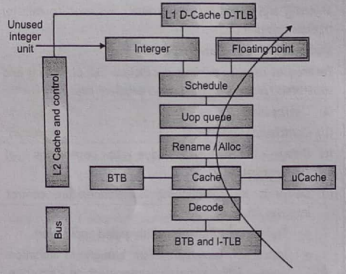


Fig. 2.15

- From Fig. 2.16 given below we can see that with SMT we can allow only one thread to execute at one time with processor.

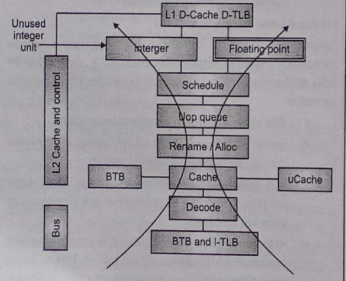


Fig. 2.16

2.9 MULTICORE AND MULTITHREADING

The use of a multicore system to support a single application with multiple threads, such as might occur on a workstation, a video-game console, or a personal computer running a processor-intense application, raises issues of performance and application design.

Performance of Software on Multicore

- The potential performance benefits of a multicore organization depend on the ability to effectively exploit the parallel resources available to the application. Amdahl's law states that:

Speedup

$$= \frac{\text{Time to execute program on a single processor}}{\text{Time to execute program on N parallel processors}}$$

$$= \frac{1}{(1 - f) + \frac{f}{N}}$$

- The law assumes a program in which a fraction (1 – f) of the execution time involves code that is inherently serial and a fraction f that involves code that is infinitely parallelizable with no scheduling overhead.
- This law appears to make the prospect of a multicore organization attractive. Even a small amount of serial code has a noticeable impact. If only 10% of the code is inherently serial (f = 0.9), running the program on a multicore system with eight processors yields a performance gain of only a factor of 4.7.
- Software engineers have been addressing this problem and there are numerous applications in which it is possible to effectively exploit a multicore system.
- Reports on a set of database applications, in which great attention was paid to reducing the serial fraction within hardware architectures, operating systems, middleware, and the database application software.

2.10 INTER-PROCESS COMMUNICATION

- In computing, Inter-Process Communication (IPC) is a set of methods for the exchange of data among multiple threads in one or more processes. Processes may be running on one or more computers connected by a network. IPC methods are divided into methods for message passing, synchronization, shared memory and Remote Procedure Calls (RPC).
- The method of IPC used may vary based on the bandwidth and latency of communication between the threads and the type of data being communicated.

- There are several reasons for providing an environment that allows process cooperation :
  - Information sharing
  - Computational speedup
  - Modularity
  - Convenience
  - Privilege separation
- IPC may also be referred to as inter-thread communication and inter application communication.
- The combination of IPC with the address space concept is the foundation for address space independence / isolation.

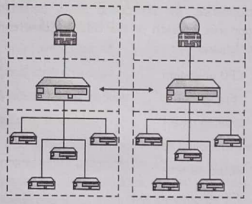


Fig. 2.17 : An example showing a grid computing system connecting many personal computers over the internet using IPC

- IPC enables one application to control another application and for several applications to share the same data without interfering with one another.
- IPC is required in all multiprocessing systems, but it is not generally supported by single process operating systems such as DOS. OS/2 and MS-Windows support an IPC mechanism called DDE.

2.11 MAIN IPC METHOD

Following are the main methods of IPC (Inter Process Communication) :

Table 2.4

Method	Short Description	Provided by
File	A record stored on disk that can be accessed by name by any process	Most operating systems
Signal	A system message sent from one process to another, not usually used to store information but instead give commands	Most operating systems some systems, such as Win NT subsystem, implement signals in only the C run-time library and provide no support for their use as an IPC method. But other subsystems like the POSIX subsystem provided by default until windows 2000. Then available with Interix in XP/2003 then with windows services for UNIX (SFU)
Socket	A data stream sent over a network interface, either to a different process on the same computer or to another computer	Most operating systems
CPipe	A two way data stream interfaced through standard input and output and is read character by character.	All POSIX systems, Windows
Named Pipe	A pipe implemented through a file on the file system instead of standard input and output.	All POSIX systems, Windows
Semaphore	A simple structure that synchronizes threads or processes acting on shared resources.	All POSIX systems, Windows
Shared Memory	Multiple processes given access to the same memory, allowing all to change it read changes made by other processes	All POSIX systems, Windows
Message Passing (Shared Nothing)	Similar to the message queue	Used in MPI paradigm, Java RMI, CORBA, DDS, MSMQ, MailSlots, QNX, others
Memory-Mapped	A file mapped to RAM and can be modified by changing memory	All POSIX systems, Windows

Method	Short Description	Provided by
File	addresses directly instead of outputting to a stream, shares same benefits as a standard file	

2.12 PLATFORM OF IPC

There are several APIs which may be used for IPC. A number of platforms independent of APIs include the following :

- Anonymous pipes and named pipes.
- Common Object Request Broker Architecture (CORBA).
- Freedesktop.org's D-Bus.
- Distributed Computing Environment (DCE).
- Message Bus (Mbus) (specified in RFC 3259).
- MCAPI Multicore Communications API.
- Lightweight Communications and Marshalling (LCM).
- ONC RPC.
- Unix domain sockets.
- XML XML-RPC or SOAP.
- JSON JSON-RPC.

The Following are Platform or Programming Language Specific APIs :

- Apple Computer's Apple events (previously known as Intra Application Communications (IAC)).
- Enea's LINX for Linux (open source) and various DSP and general purpose processors under OSE.
- IPC implementation from CMU.
- Java's Remote Method Invocation (RMI).
- KDE's Desktop Communications Protocol (DCOP) : Now deprecated. D-Bus is used instead.
- Libt2n for C++ under Linux only, handles complex objects and exceptions.
- The Mach kernel's Mach Ports.
- Microsoft's ActiveX, Component Object Model (COM), Microsoft Transaction Server (COM+), Distributed Component Object Model(DCOM), Dynamic Data Exchange (DDE), Object Linking and Embedding (OLE), anonymous pipes, named pipes, Local Procedure Call, *MailSlots*, Message loop, MSRPC, .NET Remoting and *Windows* Communication Foundation (WCF).
- Novell's SPX.
- PHP's sessions.
- POSIX mmap, message queues, semaphores and shared memory.
- RISC OS's messages.
- Solaris Doors.
- System V's message queues, semaphores and shared memory.
- Distributed Ruby.

2.13 SCHEDULING CRITERIA

- Scheduler may use in attempting to maximize system performance. The scheduling policy determines the importance of each of the criteria. Some commonly used criteria are :
  1. CPU utilization
  2. Throughput
  3. Waiting time
  4. Turnaround time
  5. Response time
  6. Priority
  7. Balanced utilization
  8. Fairness

1. **CPU Utilization :** CPU utilization is the average function of time, during which the processor is busy. The load on the system affects the level of utilization that can be achieved. CPU utilization may range from 0% to 100%. On large and expensive system i.e. time shared system, CPU utilization may be the primary consideration.
2. **Throughput :** Throughput refers to the amount of work completed in a unit of time. The number of processes the system can execute in a period of time. The higher the number, the more work is done by the system.
3. **Waiting Time :** The average period of time a process spends waiting. Waiting time may be expressed as turnaround time less the actual execution time.
4. **Turnaround Time :** The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting into the ready queue, executing on the CPU and doing I/O.

5. **Response Time :** Response time is the time from the submission of a request until the first response is produced.
6. **Priority :** Give preferential treatment to processes with higher priorities.
7. **Balanced Utilization :** Utilization of memory, I/O devices and other system resources are also considered. Not only CPU utilization considered for performance.
8. **Fairness :** Avoid the process from the starvation. All the processes must be given equal opportunity to execute.

2.14 TYPES OF SCHEDULER

- Schedulers are of three types.
  1. Long term scheduler
  2. Short term scheduler
  3. Medium term scheduler

1. Long Term Scheduler

- It is also called job scheduler. Long term scheduler determines which programs are admitted to the system for processing. Job scheduler selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduler.
- The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound.
- It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.
- On some systems, the long term scheduler may be absent or minimal. Time–sharing operating systems have no long term scheduler. When process changes the state from new to ready, then there is a long term scheduler.

2. Short Term Scheduler

- It is also called CPU scheduler. Main objective is increasing system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects from among the processes that are ready to execute and allocates the CPU to one of them.
- Short term scheduler also known as dispatcher, execute most frequently and makes the fine grained decision of which process to execute next. Short term scheduler is faster than long term scheduler.

3. Medium Term Scheduler

- Medium term scheduling is part of the swapping function. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium term scheduler is in charge of handling the swapped out–processes.
- Medium term scheduler is shown in the Fig. 2.18.

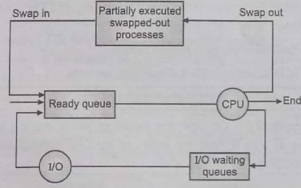


Fig. 2.18 : Queuing diagram with medium term scheduler

- Running process may become suspended by making an I/O request. Suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other process.
- Suspended process is move to the secondary storage device. Saving the image of a suspended process in secondary storage is called swapping, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix.

2.15 SCHEDULING ALGORITHMS

- Scheduling algorithm may be preemptive and non–preemptive. The types of scheduling algorithm are as follows :
  1. First Come First Served Method (FCFS)
  2. Shortest Job First (SJF)
  3. Priority
  4. Round Robin (RR)
  5. Multilevel Feedback Queue (MFQ)
  6. Multilevel Queue.

2.15.1 First Come First Served Method (FCFS)

- FCFS is the simplest scheduling method. CPU is allocated to the process in the order of arrival. The process that requests the CPU first is allocated the CPU first.

- FCFS is also called FIFO. FCFS is non-preemptive scheduling algorithm. Implementation of the FCFS policy is easily managed with a FIFO queue.
- When process enters the ready queue, its process control block is linked onto the tail of the queue.
- When the CPU is free, it is allocated to the process at the head of the queue. The running process is removed from the queue.
- Performance metric is average waiting time. Average waiting for the FCFS is often quite long. FCFS is used in batch systems.
- Real life analogy is buying tickets.
- FCFS is not suitable for real time systems.

SOLVED EXAMPLES

Example 2.1 : *Consider the following set of process that arrive at time 0, with the length of the CPU burst given in milliseconds.*

Process	Burst Time
P_1	24
P_2	3
P_3	3

Calculate the average waiting time when the processes arrive in the following order :

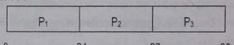
(a) P_1, P_2, P_3

(b) P_2, P_3, P_1

Provide the Gantt chart for the same.

Solution : (a) P_1, P_2, P_3

Gantt Chart :



Such a diagram is called "Gantt charts", showing when each CPU burst uses the CPU. Here, each CPU burst comes from a different thread.

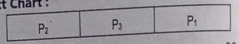
Waiting Time :

Process	Burst Time
P_1	0
P_2	24
P_3	27

Average waiting time = $\frac{0 + 24 + 27}{3} = \frac{51}{3} = 17$ ms

(b) P_2, P_3, P_1

Gantt Chart :



Waiting Time :

Process	Waiting Time
P_1	6
P_2	0
P_3	3

Average Waiting Time = $\frac{6 + 0 + 3}{3} = \frac{9}{3} = 3$ ms

Convoy Effect

- Consider 50 I/O bound processes and 1 CPU bound process is the system. I/O bound processes pass quickly through the ready queue and suspend them waiting for I/O.
- The CPU bound process arrives at the head of the queue and execute the program until completion. I/O bound processes rejoin the ready queue and wait for the CPU bound releasing the CPU.
- I/O devices idle until the CPU bound process completes. A convoy effect happens when a set of processes need to use a resource for short time and one process holds the resources for the long time, blocking all other processes.

Advantages :

- Simple to implement.
- Minimum overhead.

Disadvantages :

- Unpredictable turnaround time.
- Average waiting time is more.

2.15.2 Short Job First Scheduling (SJF)

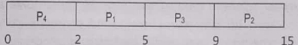
- This algorithm associates with each process the length of the latter's next CPU burst. When the CPU is free, it is assigned to the process of the ready queue which has smallest next CPU burst. If two processes have the same length, FCFS scheduling is used to break the tie.
- SJF scheduling algorithm is used frequently in long term scheduling. SJF algorithm may be either preemptive or non-preemptive.

- A preemptive SJF algorithm will preempt the currently executing process, whereas a non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst. Let us consider the set of process with burst time in milliseconds.

Process	Burst Time
P_1	3
P_2	6
P_3	4
P_4	2

- Arrival time of the process is 0 and processes arrive in the order of P_1, P_2, P_3 and P_4. The Gantt chart, waiting time and turnaround time is given below.

(i) Gantt Chart :



(ii) Waiting Time :

Process	Burst Time
P_1	2
P_2	9
P_3	5
P_4	0

(iii) Average Waiting Time :

Average waiting time = $\frac{2 + 9 + 5 + 0}{4} = \frac{16}{4} = 4$ ms

(iv) Turnaround Time :

It is sum of burst time plus waiting time of each process.

Process	Turnaround Time
P_1	3 + 2 = 5
P_2	6 + 9 = 15
P_3	4 + 5 = 9
P_4	2 + 0 = 2

(v) Average Turnaround Time = $\frac{5 + 15 + 9 + 2}{4}$

$= \frac{31}{4} = 7.75$ ms

SJF algorithm is optimal algorithm. It gives the minimum average waiting time for a given set of processes. SJF algorithm cannot be implemented at the level of short term CPU scheduling. There is no way to know the length of the next CPU burst.

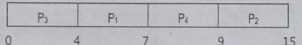
2.15.3 Priority Scheduling

- CPU is allocated to the highest priority of the process from the ready queue. Each process has a priority number. If two or more processes have the same priority, then FCFS algorithm is applied for solving the tie. In our examples, low numbers have the higher priority.
- Priority scheduling is preemptive or non-preemptive. Priority of the process can be defined either internally or externally. Internally defined priority considers the time limits, number of open files, use of memory and use of I/O devices. External priorities are set by using external parameter of the process, like importance of a process, cost of process etc.
- When the process arrives at the ready queue, its priority is compared with the priority of the currently running process. A non-preemptive priority algorithm will simply put the new process at the head of the ready queue.
- A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. In this, current executing process will change the state from running to ready.
- In non-preemptive scheduling algorithm, currently executing process will not change state. Let us consider the set of processes with burst time in milliseconds. Arrival time of the process is 0.

Process	Burst time	Priority
P_1	3	2
P_2	6	4
P_3	4	1
P_4	2	3

- Processes are arrived in the order of P_1, P_2, P_3, P_4. Gantt chart, waiting time and turnaround time for priority scheduling algorithms are given below.

(i) Gantt Chart :



(ii) Waiting Time :

Process	Waiting Time
P_1	4
P_2	9
P_3	0
P_4	7

(iii) Average Waiting Time

$$= \frac{4+9+0+7}{4} = \frac{20}{4} = 5 \text{ ms}$$

(iv) Turnaround Time :

Process	Turnaround
P_1	3 + 4 = 7
P_2	6 + 9 = 15
P_3	4 + 0 = 4
P_4	2 + 7 = 9

(v) Average Turnaround Time $= \frac{7+15+4+9}{4}$

$$= \frac{35}{4} = 8.75 \text{ ms}$$

- A priority scheduling algorithm faces the starvation problem. Starvation problem is solved by using Aging technique. In aging technique, priority of the processes will increase which is waiting for a long time in the ready queue.

2.15.4 Round-Robin Scheduling

- Time sharing system used the round robin algorithm. Use of small time quantum allows round robin to provide good response time. RR scheduling algorithm is a preemptive algorithm. To implement RR scheduling, ready queue is maintained as a FIFO queue of the processes.
- If the time quantum is very short, then short processes will move through the system relatively quickly. It increases the processing overhead involved in handling the clock interrupt and performing the scheduling and dispatch function. Thus, very short time quantum should be avoided.
- Let us consider the set of process with burst time in milliseconds. All the processes are arrived at time 0. We can draw the Gantt chart, calculate waiting time and so on.

Process	Burst Time
P_1	3
P_2	6
P_3	4
P_4	2

The quantum is 2 milliseconds.

(i) Gantt Chart :

P_1	P_2	P_3	P_4	P_1	P_2	P_3	P_2

0 2 4 6 8 9 11 13 15

(ii) Waiting Time :

Process	Waiting Time
P_1	8 – 2 = 6
P_2	13 – 2 – 2 = 8
P_3	11 – 2 = 9
P_4	6

(iii) Average Waiting Time $= \frac{6+9+9+6}{4} = \frac{30}{4}$

$$= 7.5 \text{ ms}$$

2.15.5 Multilevel Queue Scheduling

- Multilevel queues are an extension of priority scheduling whereby all processes of the same priority are placed in a single queue. For example, timesharing systems often support the idea of foreground and background processes. Foreground processes service an interactive user, while background processes are intended to run whenever no foreground process requires the CPU. These two types of processes have different response time requirement, so they require different scheduling algorithms. May be the foreground processes have priority over the background processes.
- Fig. 2.19 shows the multilevel queue scheduling algorithm. It divides the ready queue into the number of separate queues. The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority and process type. Each queue has its own scheduling algorithm.
- One queue may be scheduled by FCFS and another queue scheduled by RR method. Once the processes are assigned to the queue, they cannot change the queue i.e. processes do not move from one queue to the other since processes do not change their foreground or background nature.

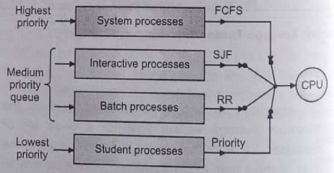


Fig. 2.19 : Multilevel queue scheduling

- Fig. 2.20 shows the ready lists in a multilevel scheduler.

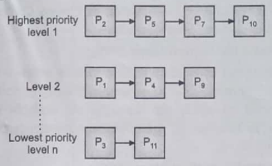


Fig. 2.20 : Multilevel scheduler ready list

- All the processes are arranged in ready list according to the priority. Processes P_1, P_4, P_9 have a larger time slice than processes P_1, P_4 and P_9. So they get a chance to execute only when processes P_2, P_5, P_7 and P_{10} are blocked. Processes P_3 and P_{11} can execute only when all other processes in the system are blocked. They would face starvation if this situation is rare.

2.15.6 Multilevel Feedback Queue Scheduling

- Multilevel Feedback Queue (MFQ) scheduling algorithm overcomes the problem of multilevel queue scheduling algorithm. MFQ allows a process to move between the queues. Fig. 2.21 shows multilevel feedback queue scheduling method. MFQ implements two or more scheduling queues.

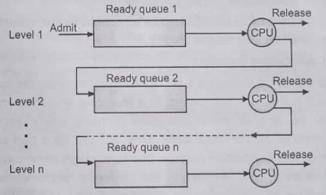


Fig. 2.21 : Multilevel feedback queue

- MFQ idea is to separate processes with different CPU burst time. If a process uses too much CPU time, it will be moved to a lower priority queue.

For Example : Each process may start at the top level queue. If the process is completed within a given time slice, it departs the system. Processes that need more than one time slice may be reassigned by the operating system to a lower priority queue, which gets a lower percentage of the processor time. If the process is still not finished after having run a few times in that queue, it may be moved to yet another lower level queue.

- Multilevel feedback queue scheduler is defined by the following parameters :
  - The number of queues.
  - Scheduling algorithm for each queue.
  - Method used to determine when to demote a process to a lower priority queue.
  - Method used to determine when to upgrade a process to a higher priority queue.
  - Method used to determine which queue a process will enter when that process needs service.

Scheduling In Multiprocessing Systems

- An efficient multiprocessor system must be capable of evenly balancing work between all available CPUs in the system.
- This load balancing is very important for optimum CPU utilization.
- In a multiprocessor system, each processor will have its own scheduler. It is a duty of each of this scheduler to plan optimum utilization of CPU for maximum efficiency.
- There exists one current process per CPU. So, system must maintain information about currently executing and idle processes under each processor at a given time.
- In an SMP system each process's "task struct" structure comprises of attribute that maintain details of number of the processor that it is currently executing on each processor and last processor number on which process was executing earlier.
- This whole computing is flexible in such a way that there doesn't arise a question of process execution on same CPU again and again. Every time process becomes ready it can be allotted to different CPU.
- However Linux OS can restrict a process to one or more processors in the system using the "processor_mask". If bit N of this structure is set, then this process can run on processor N
- When a new processes arrives, scheduler will choose a process that does not have appropriate bit set for the current processors number in its processor_mask.

2.16 THREAD SCHEDULING

Scheduling of threads involves two types of boundary scheduling.

1. Scheduling of user level threads (ULT) to kernel level threads (KLT) via lightweight process (LWP) by the application developer.

2. Scheduling of kernel level threads by the system scheduler to perform different unique os functions.

Lightweight Process (LWP) :

- Light-weight process are threads in the user space that acts as an interface for the ULT to access the physical CPU resources. Thread library schedules which thread of a process to run on which LWP and how long. The number of LWP created by the thread library depends on the type of application. In the case of an I/O bound application, the number of LWP depends on the number of user-level threads.
- This is because when an LWP is blocked on an I/O operation, then to invoke the other ULT the thread library needs to create and schedule another LWP. Thus, in an I/O bound application, the number of LWP is equal to the number of the ULT. In the case of a CPU bound application, it depends only on the application. Each LWP is attached to a separate kernel-level thread.

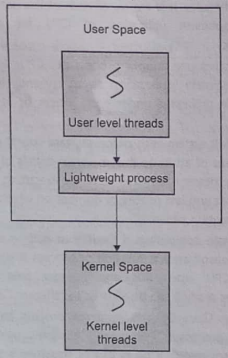


Fig. 2.22

In real-time, the first boundary of thread scheduling is beyond specifying the scheduling policy and the priority. It requires two controls to be specified for the User level threads: (i) Contention scope, (ii) Allocation domain.

(i) Contention Scope :

The word contention here refers to the competition or fight among the User level threads to access the kernel resources. Thus, this control defines the extent to which contention takes place. It is defined by the application developer using the thread library. Depending upon the extent of contention it is classified as Process Contention Scope and System Contention Scope.

(a) Process Contention Scope (PCS) :

The contention takes place among threads within a same process. The thread library schedules the high-prioritized PCS thread to access the resources via available LWPs (priority as specified by the application developer during thread creation).

(b) System Contention Scope (SCS) :

The contention takes place among all threads in the system. In this case, every SCS thread is associated to each LWP by the thread library and are scheduled by the system scheduler to access the kernel resources.

In LINUX and UNIX operating systems, the POSIX Pthread library provides a function Pthread_attr_setscope to define the type of contention scope for a thread during its creation.

```
nt Pthread_attr_setscope(pthread_attr_t *attr, int scope)
```

The first parameter denotes to which thread within the process the scope is defined.

The second parameter defines the scope of contention for the thread pointed. It takes two values.

PTHREAD_SCOPE_SYSTEM
PTHREAD_SCOPE_PROCESS

If the scope value specified is not supported by the system, then the function returns ENOTSUP.

(ii) Allocation Domain :

The allocation domain is a set of one or more resources for which a thread is competing. In a multicore system, there may be one or more allocation domains where each consists of one or more cores. One ULT can be a part of one or more allocation domain. Due to this high complexity in dealing with hardware and software architectural interfaces, this control is not specified. But by default, the multicore system will have an interface that affects the allocation domain of a thread.

Consider a scenario, an operating system with three process P1, P2, P3 and 10 user level threads (T1 to T10) with a single allocation domain. 100% of CPU resources will be distributed among all the three processes. The amount of CPU resources allocated to each process and to each thread depends on the contention scope, scheduling policy and priority of each thread defined by the application developer using thread library and also depends on the system scheduler. These User level threads are of a different contention scope.

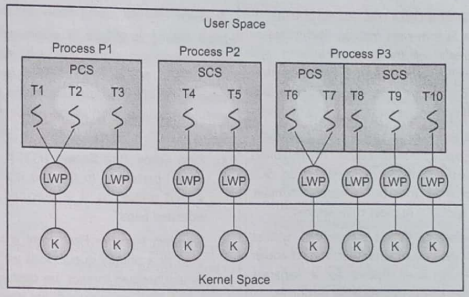


Fig. 2.23

In this case, the contention for allocation domain takes place as follows,

Process P1:

All PCS threads T1, T2, T3 of Process P1 will compete among themselves. The PCS threads of the same process can share one or more LWP. T1 and T2 share an LWP and T3 are allocated to a separate LWP. Between T1 and T2 allocation of kernel resources via LWP is based on preemptive priority scheduling by the thread library. A Thread with a high priority will preempt low priority threads. Whereas, thread T1 of process p1 cannot preempt thread T3 of process p3 even if the priority of T1 is greater than the priority of T3. If the priority is equal, then the allocation of ULT to available LWPs is based on the scheduling policy of threads by the system scheduler(not by thread library, in this case).

Process P2:

Both SCS threads T4 and T5 of process P2 will compete with processes P1 as a whole and with SCS threads T8, T9, T10 of process P3. The system scheduler will schedule the kernel resources among P1, T4, T5, T8, T9, T10, and PCS threads (T6, T7) of process P3 considering each as a separate process. Here, the Thread library has no control of scheduling the ULT to the kernel resources.

Process P3:

Combination of PCS and SCS threads. Consider if the system scheduler allocates 50% of CPU resources to process P3, then 25% of resources is for process scoped threads and the remaining 25% for system scoped threads. The PCS threads T6 and T7 will be allocated to access the 25% resources based on the priority by the thread library. The SCS threads T8, T9, T10 will divide the 25% resources among themselves and access the kernel resources via separate LWP and KLT. The SCS scheduling is by the system scheduler.

Note:

For every system call to access the kernel resources, a Kernel Level thread is created and associated to separate LWP by the system scheduler.

Number of Kernel Level Threads = Total Number of LWP

Total Number of LWP = Number of LWP for SCS + Number of LWP for PCS

Number of LWP for SCS = Number of SCS threads

Number of LWP for PCS = Depends on application developer

Here,

Number of SCS threads = 5

Number of LWP for PCS = 3

Number of LWP for SCS = 5

Total Number of LWP = 8 (=5+3)

Number of Kernel Level Threads = 8

Advantages of PCS over SCS :

If all threads are PCS, then context switching, synchronization, scheduling everything takes place within the userspace. This reduces system calls and achieves better performance.

PCS is Cheaper than SCS :

- PCS threads share one or more available LWPs. For every SCS thread, a separate LWP is associated.For every system call, a separate KLT is created.

- The number of KLT and LWPs created highly depends on the number of SCS threads created. This increases the kernel complexity of handling scheduling and synchronization. Thereby, results in a limitation over SCS thread creation, stating that, the number of SCS threads to be smaller than the number of PCS threads.
- If the system has more than one allocation domain, then scheduling and synchronization of resources becomes more tedious. Issues arise when an SCS thread is a part of more than one allocation domain, the system has to handle n number of interfaces.
- The second boundary of thread scheduling involves CPU scheduling by the system scheduler. The scheduler considers each kernel-level thread as a separate process and provides access to the kernel resources.
- Any application can be implemented as a set of threads which co–operate and execute concurrently in the same logical address space.
- In uniprocessor system threads can be used by making used of "thread switch", since it incurs less cost than that of process switching.
- In case of multiprocessor architecture threads can be effectively implemented to exploit parallelism.

There are four approaches for effective thread scheduling to achieve parallelism :

1. **Load Sharing:**

   In this case, a pool of thread is maintained when processor becomes idle scheduler selects a thread from ready queue for execution. Thus, we can see that rather than process it is a thread that is assigned to processor.

2. **Gang Scheduling:**

   Related threads are chosen and scheduled for execution with same processor. This is possible due to fact that belong to same process share same logical address space.

3. **Dedicated Processor Assignment:**

   Here each program is allotted to numbers of CPUs by distributing equally to number of threads till completion of program execution.

4. **Dynamic Scheduling:**

   By the time execution continues total number of threads in use will vary since some threads may exit the memory after completion. So, based on load, threads are allotted to processor

1. Load Sharing

Load sharing is utilized in uniprocessor system and has many advantages. Load is equally distributed among all process by taking care of load balancing. When the processor is idle scheduler is executed by operating system and idle processor is allotted with next thread. Global queue is used along with different scheduling schemas as:

- **First Come First Served (FCFS):** As jobs arrive each thread pertaining to that job is added to the end of shared queue. So it is executed on last come last executed basis.
- **Smallest Number First:** Here shared ready queue is used as a priority queue: Jobs with minimum number of unscheduled threads are chosen first for execution. Jobs having equal threads are executed based on time of arrival.
- **Pre–Emptive Smallest Number First:** This works in the same way as smallest numbers of thread first method with a small execution. In this case if a job with higher priority arrives than currently executing job (thread of job) will be forcefully suspended from execution making a way of priority job.

Disadvantages of Load Sharing:

- Central queue consumes additional memory and needs to be operating manually exclusive. This itself is an overhead.
- There are very less chances that all suspended processes will again execute under same CPU so caching of data by CPU becomes irrelevant.
- In Load balancing act, selection of each type of thread (CPU and I/O) will not be takes up since, this method works with common pool technique.

2. Gang Scheduling

- The concept of scheduling gang of thread/processes together belongs to same process or all closely related process that belong to same program. Due to this switching will get reduced and hence system performance will increase, scheduling work will get reduced since single instruction of execution will be avoided by gang of processes/threads.
- Scheduling related process tasks is termed as co-scheduling whereas term gang scheduling is used for scheduling of threads of same process. While executing in parallel, threads synchronize with each other reducing possibility of switching.

3. Dedicated Processor Assignment

- In this method, complete application is assigned to a dedicated processor or processors. This may result in poor performance due to possibility that many processes / threads may be getting blocked while waiting for I/O and may waste lot of time.
- This is against the ideology of multiprogramming.
- From the study conducted with vivid execution, it has been observed that performance of system improves till numbers of active threads are equal to number of processors.
- Any further increase in threads leads to degradation of performance since these threads start switching due to preemption and lot of systems energy goes waste in this business.
- Hence both methods, dedicated processor assignment and gang scheduling, try to solve the scheduling problem by addressing problem of processor allocation first.
- Question of how many processors to be assigned to a task depends on number of page frames required to be allocated to an individual task. Data that gives idea of number of pages in "Activity work set" is utilized to decide on processor allocation and problem is addressed.

4. Dynamic Scheduling

- With the help of some modern tools, it is possible to modify the processes on the fly. This helps in improvisation of load (load decrease due to minimum number of threads are created covering all tasks), which finally leads to increase in system performance.
- OS partitions processor with existing jobs.
- Every job is assigned to processor in current partition by assigning tasks to threads.
- It is on individual application to decide on thread scheduling. But this is not true with all applications since some applications are single threaded.
- If application demands one or more processor, following actions are taken:
  - Assign idle processor to complete ongoing requests.
  - If there is a new request, then assign one processor to it by assigning a processor of any other task which is working with more than one processor at that time.
  - If there exists some request that is not served yet than, it can be services either when processor becomes free or when its priority is reached.
  - From the current queue of not satisfied requests, assign one processor to those tasks which are not assigned with any processor since long time.

Real-Time Scheduling

- Real time scheduling depends on timeliness and accuracy of logical result generated using computers. It demands quickness since any delay in decision may lead to hazards.
- Each task will be associated with a "Deadline" specified with respect to either start or end time or sometimes both termed as "Hard real time task", else it is termed "soft real – time task". Tasks that need to meet both start as well as end time is called as "a periodic task" while that is stated as "once per period T" or "exactly T units apart" termed as a "periodic task".

Characteristics of RTOS:

This type of systems has five requirements in common:

1. Determinism
2. Responsiveness
3. User Control
4. Reliability
5. Fail–soft operation

1. Determinism :

Determinism is the ability of system to complete its tasks within predetermined time. This depends on the speed at which system can respond to interrupt and its capability to handle request in estimated time. Determinism is also amount of system delay before acknowledgement of interrupt.

2. Responsiveness :

Responsiveness is another measure used in RTOS in collaboration with determinism. It defines the delay incurred to complete Interrupt Service Subroutine (ISR) after acknowledge.

So, responsiveness can be stated as total required in handling and servicing the interrupt.

- It may get delayed if there is switching of processes in between.
- It will not get much delayed in case switching is in some logical address space.

- This depends on time taken to complete ISR.
- It is H/W specific
- It may get affected in case there exists nested ISRs that will make other process wait finally delaying.

3. **User Bound Control :**

This is perfect control over process priority. This will give power system. It is decide hard and soft tasks and control the operation leading to successful computation.

4. **Reliability :**

Any transient failure may be easily recovered in case of non–real time system. It is manageable in case of multiprocessor system since it hardly leads to poor performance of RTOS. If whole system fails, then it leads to catastrophic failure.

5. **Fail-Soft Operation:**

All RTOS are required to acknowledge all exception generated out of failures. This will be useful in handling this exception to help system to reduce the losses to manageable level. This is done by saving critical data of the system in the memory before system halts. This data further can be used after we resume computation (in recovery).

Important Operation of Real Time OS

- Fast process or thread switch
- Small code size
- Ability to respond to quickly
- Support of multitasking with IPC
- Use of special files that will call data quickly
- Pre–emptive scheduling on priority.
- Isolation/minimization of interrupted which are disabled from service.
- Pre–emption with fixed delay and facility to start/stop tasks
- Special alarm and timeout operations.

RTO Scheduling

- Work with a systematic approach that informs:
  - Whether system have conducted analysis of schedule
  - Whether scheduling is static or dynamic
  - Whether schedule deliver a plan of execution

This hints various types of algorithms such as,

1. **Static Table Driven Approach:**
   - This adheres to arrival, execution and end time limits, priority wise.
   - If user data is obtained from the schedule analysis mode, it tries to meet requirements of all scheduled tasks.
2. **Static Priority Driven Pre–Emptive Scheduling:**
   - Generally, in non RTOS priority, process is determined on the basis of many factors like, in time sharing system input/output bound process gets lower priority.
   - In RTOS priority is based on the time constraints as per process rate monotonic algorithm. We are going to study this further.
3. **Dynamic Planning Based Scheduling:**
   - Existing schedule is updated by accommodating all newly arrived jobs, if it does not have any impact in meeting the deadlines of existing jobs.
4. **Dynamic Best Effort Scheduling:**
   - This schedule is derived at a time when a new job arrives, it is assigned a priority based on its nature and criticality.
   - These schedules are periodic. It is a very simple scheduling technique to implement but the only problem is that scheduler will never know timing statistics of process execution till end.
5. **Deadline Scheduling:**
   - RTOS success strongly depends on: Processing within specified time and not before or later, than of scheduled time.
   - There are some proposals which aim at supporting the system to meet these deadlines. Some of these are:
     - **Ready Time:** This defines the time when, job becomes ready for execution. Operating system can take cognizance of this before actually taking up the task for execution.
     - **Starting Deadline:** This defines the start point of execution of job, when dispatched to CPU.
     - **Completion Deadline:** This defines the complet-ion time of task. Systems never use this data since it can vary due to many factors.

Processing Time:

- This defines the time is required for a task to get executed completely. This data is also of not much use since it may vary due to any operational delay.

Resource Requirement;

- This gives idea about the resource requirement of task for execution.

Priority:

- This factor plays important role since it helps in maintaining "Hard real time task". This will thus help in meeting the deadlines.

Subtask Structure:

- A task may be broken into sub tasks. These subtasks are of two types
  1. "Mandatory task"
  2. "Optional task".
- Only mandatory category tasks are associated with "hard deadlines" and thus given higher priority.

Table 2.5 : Comparison between FCFS and RR Method

Sr. No.	FCFS	Round Robin
1.	FCFS decision made is non–preemptive.	RR decision made is preemptive.
2.	It has minimum overhead.	It has low overhead.
3.	Response time may be high.	Provides good response time for short processes.
4.	It is troublesome for time sharing system.	It is mainly designed for time sharing system.
5.	The workload is simply processed in the order of arrival.	It is similar like FCFS but uses time quantum.
6.	No starvation in FCFS.	No starvation in RR.

2.17 MULTIPLE-PROCESSOR SCHEDULING IN OPERATING SYSTEM

- In multiple-processor scheduling multiple CPU's are available and hence Load Sharing becomes possible. However multiple processor scheduling is more complex as compared to single processor scheduling. In multiple processor scheduling there are cases when the processors are identical i.e. HOMOGENEOUS, in terms of their functionality, we can use any processor available to run any process in the queue.

Approaches to Multiple-Processor Scheduling

- One approach is when all the scheduling decisions and I/O processing are handled by a single processor which is called the Master Server and the other processors executes only the user code. This is simple and reduces the need of data sharing. This entire scenario is called Asymmetric Multiprocessing.
- A second approach uses Symmetric Multiprocessing where each processor is self scheduling. All processes may be in a common ready queue or each processor may have its own private queue for ready processes. The scheduling proceeds further by having the scheduler for each processor examine the ready queue and select a process to execute.

Processor Affinity :

- Processor Affinity means a processes has an affinity for the processor on which it is currently running.
- When a process runs on a specific processor there are certain effects on the cache memory. The data most recently accessed by the process populate the cache for the processor and as a result successive memory access by the process are often satisfied in the cache memory.
- Now if the process migrates to another processor, the contents of the cache memory must be invalidated for the first processor and the cache for the second processor must be repopulated. Because of the high cost of invalidating and repopulating caches, most of the SMP (symmetric multiprocessing) systems try to avoid migration of processes from one processor to another and try to keep a process running on the same processor. This is known as processor affinity.

There are two types of processor affinity:

1. **Soft Affinity :** When an operating system has a policy of attempting to keep a process running on the same processor but not guaranteeing it will do so, this situation is called soft affinity.
2. **Hard Affinity :** Hard Affinity allows a process to specify a subset of processors on which it may run. Some systems such as Linux implements soft affinity but also provide some system calls like sched_setaffinity() that supports hard affinity.

Load Balancing :

- Load Balancing is the phenomena which keeps the workload evenly distributed across all processors in an SMP system. Load balancing is necessary only on systems where each processor has its own private queue of process which are eligible to execute. Load balancing is unnecessary because once a processor becomes idle it immediately extracts a runnable process from the common run queue.
- On SMP(symmetric multiprocessing), it is important to keep the workload balanced among all processors to

fully utilize the benefits of having more than one processor else one or more processor will sit idle while other processors have high workloads along with lists of processors awaiting the CPU.

There are two general approaches to load balancing :

1. **Push Migration :** In push migration a task routinely checks the load on each processor and if it finds an imbalance then it evenly distributes load on each processors by moving the processes from overloaded to idle or less busy processors.
2. **Pull Migration :** Pull Migration occurs when an idle processor pulls a waiting task from a busy processor for its execution.

Multicore Processors :

- In multicore processors multiple processor cores are places on the same physical chip. Each core has a register set to maintain its architectural state and thus appears to the operating system as a separate physical processor. SMP systems that use multicore processors are faster and consume less power than systems in which each processor has its own physical chip.
- However multicore processors may complicate the scheduling problems. When processor accesses memory then it spends a significant amount of time waiting for the data to become available. This situation is called MEMORY STALL. It occurs for various reasons such as cache miss, which is accessing the data that is not in the cache memory. In such cases the processor can spend upto fifty percent of its time waiting for data to become available from the memory. To solve this problem recent hardware designs have implemented multithreaded processor cores in which two or more hardware threads are assigned to each core. Therefore if one thread stalls while waiting for the memory, core can switch to another thread.

There are two ways to multithread a processor :

1. **Coarse-Grained Multithreading :** In coarse grained multithreading a thread executes on a processor until a long latency event such as a memory stall occurs, because of the delay caused by the long latency event, the processor must switch to another thread to begin execution. The cost of switching between threads is high as the instruction pipeline must be terminated before the other thread can begin execution on the processor core. Once this new thread begins execution it begins filling the pipeline with its instructions.
2. **Fine-Grained Multithreading :** This multithreading switches between threads at a much finer level mainly at the boundary of an instruction cycle. The architectural design of fine grained systems include logic for thread switching and as a result the cost of switching between threads is small.

2.18 COMPARISON OF SCHEDULING ALGORITHMS

Table 2.6

Algorithm	Policy Type	Used in	Advantages	Disadvantages
FCFS	Non-preemptive	Batch	• Easy to implement. • Minimum overhead	• Unpredictable turnaround time. • Average waiting is more.
RR	Preemptive	Interactive	• Provides fair CPU allocation. • Provides reasonable response times to interactive users.	• Requires selection of good time slice.
Priority	Non-preemptive	Batch	• Ensures fast completion of important jobs.	• Indefinite postponement of some jobs. • Faces starvation problem.
SJF	Non-preemptive	Batch	• Minimizes average waiting time. • SJF algorithm is optimal.	• Indefinite postponement of some job. • Cannot be implemented at the level of short term scheduling. • Difficulty is knowing the length of the next CPU request.

Algorithm	Policy Type	Used in	Advantages	Disadvantages
Multilevel queues	Preemptive or Non-preemptive	Batch/ Interactive	• Flexible • Gives fair treatment to CPU bound jobs.	• Overhead incurred by monitoring of queues.

Example 2.2 : *Consider following process, with the CPU burst time given in milliseconds.*

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

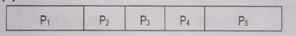
Process are arrived in P_1, P_2, P_3, P_4, P_5 order of all at time 0.

1. *Draw Gantt charts to show execution using FCFS, SJF, non–preemptive priority, (small priority number implies higher priority) and RR (quantum = 1) scheduling.*
2. *Also calculate turnaround time for each scheduling algorithms.*
3. *What is the waiting time of each process for each one of the above scheduling algorithms?*

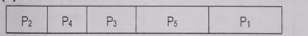
Solution :

1. **Gantt Chart :**

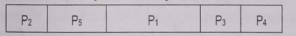
(a) FCFS :

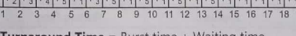


(b) SJF :



(c) Non–Preemptive Priority :





2. **Turnaround Time** = Burst time + Waiting time

(a) For FCFS :

$$= \frac{10 + 11 + 13 + 14 + 19}{5} = \frac{67}{5} = 13.4$$

(b) For SJF :

$$= \frac{1 + 2 + 4 + 9 + 19}{5} = \frac{35}{5} = 7$$

(c) For Non–Preemptive Priority :

$$= \frac{1 + 6 + 16 + 18 + 19}{5} = \frac{60}{5} = 12$$

(d) For RR :

$$= \frac{19 + 2 + 7 + 4 + 14}{5} = \frac{46}{5} = 9.2$$

3. **Waiting Time :**

(a) For FCFS :

Process	Waiting Time
P_1	0
P_2	10
P_3	11
P_4	13
P_5	14

(b) For SJF :

Process	Waiting Time
P_1	9
P_2	0
P_3	2
P_4	1
P_5	4

(c) For Non–Preemptive Priority :

Process	Waiting Time
P_1	6
P_2	0
P_3	16
P_4	18
P_5	1

(d) For RR Method :

Process	Waiting Time
P_1	9
P_2	1
P_3	5
P_4	3
P_5	9

Example 2.3 : *For the following example calculate average turnaround time and average waiting time for the following algorithms.*

(i) FCFS, (ii) Preemptive SJF, (iii) Round Robin (1 time unit).

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

Solution :

(i) For FCCS :

(a) Gantt Chart :

P_1	P_2	P_3	P_4

0 8 12 21 26

(b) Waiting Time :

Process	Waiting Time
P_1	0 – 0 = 0
P_2	8 – 1 = 7
P_3	12 – 2 = 10
P_4	21 – 3 = 18

(c) Average Waiting Time = $\frac{0 + 7 + 10 + 18}{4}$

$= \frac{35}{4} = 8.75$

(d) Turnaround Time : Burst time + Waiting time

Process	Waiting Time
P_1	8 + 0 = 8
P_2	4 + 7 = 11
P_3	9 + 10 = 19
P_4	5 + 18 = 23

(e) Average Turnaround Time = $\frac{8 + 11 + 19 + 23}{4}$

$= \frac{61}{4} = 15.25$

(ii) For Preemptive SJF :

(a) Gantt Chart :

P_1	P_2	P_4	P_1	P_3

0 1 5 10 17 26

(b) Waiting Time :

Process	Waiting Time
P_1	0 + (10 – 1) = 9
P_2	1 – 1 = 0
P_3	17 – 2 = 15
P_4	5 – 3 = 2

(c) Average Waiting Time

$= \frac{9 + 0 + 15 + 2}{4} = \frac{26}{4} = 6.5$

(d) Turnaround Time :

Process	Waiting Time
P_1	8 + 9 = 17
P_2	4 + 0 = 4
P_3	9 + 15 = 24
P_4	5 + 2 = 7

(e) Average Turnaround Time

$= \frac{17 + 4 + 24 + 7}{4} = \frac{52}{4} = 13$

(iii) Round Robin (1 Time Unit) :

(a) Gantt Chart :

P_1	P_2	P_3	P_4	P_1	P_2	P_3	P_4	P_1	P_2	P_3	P_4	P_1	P_2	P_3	P_4

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

P_1	P_2	P_3	P_1	P_2	P_3	P_1	P_2	P_3

17 18 19 20 21 22 23 24 25 26

(b) Waiting Time :

Process	Waiting Time
P_1	16
P_2	9
P_3	15
P_4	11

(c) Average Waiting Time = $\frac{16 + 9 + 15 + 11}{4}$

$= \frac{51}{4} = 12.75$

(d) Turnaround Time :

Process	Waiting Time
P_1	8 + 16 = 24
P_2	4 + 9 = 13
P_3	9 + 15 = 24
P_4	5 + 11 = 16

(e) Average Turnaround Time = $\frac{24 + 13 + 24 + 16}{4}$

$= \frac{77}{4} = 19.25$

Example 2.4 : *For the processes listed below, draw a Gantt chart using priority scheduling. A larger priority number has higher priority.*

(a) Preemptive (b) Non–preemptive

Process	Arrival Time	Burst Time	Priority
P_1	0.0	6	4
P_2	3.0	5	2
P_3	3.0	3	6
P_4	5.0	5	3

Solution : (a) Preemptive Priority :

Gantt Chart :

P_1	P_3	P_1	P_4	P_2

0 3 6 9 14 19

(b) Non–Preemptive Priority :

Gantt Chart :

P_1	P_3	P_4	P_2

0 6 9 14 19

Example 2.5 : *Consider the following set of processes with the length of the CPU burst time given in milliseconds.*

Process	Arrival Time	Burst Time	Priority
P_1	0	8	3
P_2	1	1	1
P_3	2	3	2
P_4	3	2	3
P_5	4	6	4

Draw the Gantt charts illustrating the execution of these processes using FCFS, SJF (preemptive and non–preemptive), priority (preemptive and non–preemptive) scheduling. Calculate turnaround time and waiting time for each process for all five scheduling algorithms.

Solution :

(i) FCFS :

P_1	P_2	P_3	P_4	P_5

0 8 9 12 14 20

Process	Wait Time	Turnaround Time
P_1	0	0 + 8 = 8
P_2	8	8 + 1 = 9
P_3	9	9 + 3 = 12
P_4	12	12 + 2 = 14
P_5	14	14 + 6 = 20

(ii) SJF : Non–Preemptive :

P_2	P_4	P_3	P_5	P_1

0 1 3 6 12 20

Process	Wait Time	Turnaround Time
P_1	0	0 + 1 = 1
P_2	1	1 + 2 = 3
P_3	3	3 + 3 = 6
P_4	6	6 + 6 = 12
P_5	12	12 + 8 = 20

Preemptive

P_1	P_2	P_3	P_4	P_5	P_1	P_3

0 1 2 3 5 11 18 20

(iii) Priority :

P_2	P_3	P_4	P_1	P_5

0 1 4 6 14 20

Process	Wait Time	Turnaround Time
P_1	0	0 + 1 = 1
P_2	1	4
P_3	4	6
P_4	6	14
P_5	14	20

Example 2.6 : *Consider the following set of processes, with the length of the CPU burst time given in milliseconds.*

Process	Arrival Time	Burst Time	Priority
P_1	0	7	3
P_2	1	1	1
P_3	2	3	2
P_4	3	4	4

Draw the Gantt charts illustrating the execution of these processes using round robin (time quantum = 1 millisecond) FCFS, SJF (preemptive and non–preemptive), priority (preemptive and non–preemptive) scheduling. Calculate turnaround time and waiting time for each process for all five scheduling algorithms.

Solution :

(i) FCFS :

P_1	P_2	P_3	P_4

0 7 8 11 15

Process	Arrival Time	Turnaround Time
P_1	0	7
P_2	7	8
P_3	8	11
P_4	11	15

(ii) SJF :

Preemptive

P_1	P_2	P_3	P_4	P_1

0 1 2 5 9 15

Non–Preemptive

P_2	P_3	P_4	P_1

0 1 4 8 15

Process	Arrival Time	Turnaround Time
P_1	0	1
P_2	1	4
P_3	4	8
P_4	8	15

(iii) Priority :

P_2	P_3	P_1	P_4

0 1 4 11 15

Example 2.7 : *Consider the following set of processes, with the length of the CPU burst time given in milliseconds.*

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

Draw the Gantt charts illustrating the execution of these processes using SJF (preemptive and non–preemptive), FCFS and Round robin (assume time slice = 1).

Solution : (i) SJF :

(a) Non–Preemptive :

P_2	P_3	P_4

0 1 5 10

(b) Preemptive :

P_1	P_2	P_3	P_1	P_3

0 2 3 4 6 10

(c) FCFS :

P_1	P_2	P_3

0 4 5 10

(ii) Round Robin :

P_1	P_2	P_3	P_4	P_1	P_5	P_1	P_5

0 2 3 4 5 6 7 8 10

Example 2.8 : *Consider following set of processes.*

Process	Arrival Time	Processing Time
P_0	0	3
P_1	1	5
P_2	3	2
P_3	9	5
P_4	12	5

Draw the Gantt chart and find out the average waiting time and average turnaround time for FCFS and SJF scheduling diagrams.

Solution :

Process	Arrival Time	Processing Time
P_0	0	3
P_1	1	5
P_2	3	2
P_3	9	5
P_4	12	5

(i) FCFS :

P_0	P_1	P_2	P_3	P_3	P_4

0 3 8 10 15 20

Process	Wait Time
P_0	0
P_1	2
P_2	5
P_3	1
P_4	3

Process P_1 arrives at 1 ms and as per Gantt Chart waits to 3 ms.

$\therefore$ Wait time for $P_1 = 3 - 1 = 2$ ms

Similarly wait time for all processes is calculated.

Average wait time $= \frac{0 + 2 + 5 + 1 3}{5} = \frac{11}{5} = 2.2$ ms

(ii) SJF (Non–Preemptive) :

P_2	P_0	P_1	P_3	P_4

0 2 5 10 15 20

Process	Wait Time
P_0	2
P_1	4
P_2	0
P_3	1
P_4	3

Average wait time $= \frac{2 + 4 + 1 + 0 + 3}{5} = \frac{10}{5} = 2$ ms

Example 2.9 : *Consider the following set of processes with the length of the CPU burst time given in milliseconds.*

Process	Arrival Time	Burst Time	Priority
P_1	0	3	3
P_2	1	5	1
P_3	3	2	3
P_4	9	5	2
P_5	12	5	1

Draw the Gantt charts and find out average waiting time and average turnaround time for (i) FCFS (ii) SJF (Preemptive) (iii) SJF (Non Preemptive) (iv) Priority (Preemptive – a smaller priority number implies a higher priority (v) Round Robin (Time Quantum = 1 ms).

Solution :

Process	Arrival Time	Burst Time	Priority
P_1	0	3	3
P_2	1	5	1
P_3	3	2	3
P_4	9	5	2
P_5	12	5	1

(i) FCFS :

P_1	P_2	P_3	P_4	P_5

0 3 8 10 15 20

Process	Arrival	Burst Time
P_1	0	3
P_2	2	7
P_3	5	7
P_4	1	6
P_5	3	8

Average wait time $= \frac{0 + 2 + 5 + 1 + 3}{5} = 2.2$ ms

Average Turnaround time $= \frac{3 + 7 + 7 + 6 + 8}{5}$

$= \frac{31}{5} = 6.2$ ms

Turnaround Time is time of submission of a process to a time of completion of a process.

In the given example P_1 arrives at 0 and completes at 3.

Turnaround time for $P_1 = 3$.

Similarly P_2 gets submitted at 1 ms and gets completed at 8 ms.

$\therefore$ Turnaround time for $P_2 = 7$.

(ii) SJF (Non–Preemptive) :

P_3	P_1	P_2	P_4	P_5

0 2 5 10 15 20

Process	Wait Time	Turnaround Time
P_1	2	5
P_2	4	9
P_3	0	2
P_4	1	6
P_5	3	8

Average wait time = 2 ms.

Average turnaround time = 6 ms.

(iii) SJF (Preemptive) :

P_1	P_3	P_2	P_4	P_5

0 3 5 10 15 20

Process	Wait Time	Turnaround Time
P_1	2	3
P_2	4	9
P_3	0	2
P_4	1	6
P_5	3	8
Average	1.6 ms	5.6 ms

(iv) Priority (Preemptive) :

P1	P2	P1	P3	P4	P5	P4	P3

0 1 6 8 9 12 17 19 20

Process	Wait time	Turnaround Time
P1	6	8
P2	0	5
P3	16	17
P4	8	10
P5	0	5
Average	6 ms	7 ms

(v) Round Robin :

P1	P2	P3	P4	P5	P1	P2	P3	P4	P5	P1	P2	P4	P5	P2	P4	P5	P2	P4	P5

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

Process	Wait Time	Turnaround Time
P1	10	11
P2	17	17
P3	7	5
P4	18	10
P5	19	8
Average	14.2 ms	12.2 ms

Example 2.10 : *Consider the following processes :*

Process	Arrival Time	Burst Time
P1	0.0 ms	6 ms
P2	0.5 ms	4 ms
P3	1.0 ms	2 ms
P4	1.2 ms	1 ms

Find the average turnaround time and average waiting time with respect of FCF, SJF and Round Robin (quantum = 1 ms). Also draw Gantt chart for each algorithm.

Solution :

Process	Arrival Time	Burst Time
P1	0.0 ms	6
P2	0.5 ms	4
P3	1.0 ms	2
P4	1.2 ms	1

(i) FCFS :

P1	P2	P3	P4

0 6 10 12 13

Process	Wait Time	Turnaround Time
P1	0	6
P2	6	9.5
P3	10	11
P4	12	11.8
Average	7 ms	9.07 ms

(ii) SJF :

P1	P4	P3	P2

0 6 7 9 13

Process	Wait Time	Turnaround Time
P1	0	6
P2	8.5	12.5
P3	6	8
P4	4.8	5.8
Average	4.82 ms	8.07 ms

Example 2.11: *Consider the following set of processor with the length of the CPU burst time given in milliseconds.*

Process	Arrival Time	Processing Time
P1	0	7
P2	3	2
P3	4	3
P4	4	1
P5	5	3

Draw a Gantt chart and find out average waiting–time and average turnaround time for (i) FCFS (ii) SJF (iii) RR

Solution :

FCFS

(a) Gantt Chart

P1	P2	P3	P4	P5

0 7 9 12 13 16

(b) Waiting Time

$P_1 = 0 - 0 = 0$

$P_2 = 7 - 3 = 4$

$P_3 = 9 - 4 = 5$

$P_4 = 12 - 4 = 8$

$P_5 = 13 - 5 = 8$

(c) Average Waiting Time

$$= \frac{(0 + 4 + 5 + 8 + 8)}{5}$$

$$= \frac{25}{5} = 5$$

(d) Turnaround Time = Waiting Time + Processing Time

$P_1 = 0 + 7 = 7$

$P_2 = 4 + 2 = 6$

$P_3 = 5 + 3 = 8$

$P_4 = 8 + 1 = 9$

$P_5 = 8 + 3 = 11$

(e) Turnaround Time

$$= \frac{(7 + 6 + 8 + 9 + 11)}{5} = 8.2$$

SJF

(a) Gantt Chart

P1	P2	P3	P4	P5

0 7 8 10 13 16

(b) Waiting Time

$P_1 = 0$

$P_2 = 8 - 3 = 5$

$P_3 = 10 - 4 = 6$

$P_4 = 7 - 4 = 3$

$P_5 = 13 - 5 = 8$

(c) Average Waiting Time

$$= \frac{(0 + 5 + 6 + 3 + 8)}{5}$$

$$= 4.4$$

(d) Turnaround Time

$P_1 = 0 + 7 = 7$

$P_2 = 5 + 2 = 7$

$P_3 = 6 + 3 = 9$

$P_4 = 3 + 1 = 4$

$P_5 = 8 + 3 = 11$

(e) Average Turnaround Time

$$= \frac{(7 + 7 + 9 + 4 + 11)}{5}$$

$$= 7.6$$

RR

Gantt Chart (Time Slice = 1)

P1	P2	P3	P4	P5	P1	P2	P3	P5	P1	P3	P5	P1	P1	P1	P1

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

(a) Waiting Time

$P_1 = 0 + (7 - 3) + (11 - 8) + (14 - 12)$

$= 9$

$P_2 = (3 - 3) + (8 - 4) = 4$

$P_3 = (4 - 4) + (9 - 5) + (12 + 11) = 5$

$P4 = (5 - 4) = 1$

$P_5 = (6 - 5) + (10 - 7) + (13 - 10) = 7$

(b) Average Waiting Time

$$= \frac{(9 + 4 + 5 + 1 + 7)}{5}$$

$$= \frac{26}{5} = 5.2$$

(c) Turnaround Time

$P_1 = 9 + 7 = 16$

$P_2 = 4 + 2 = 6$

$P_3 = 6 + 3 = 9$

$P_4 = 1 + 1 = 2$

$P_5 = 6 + 3 = 9$

(d) Average Turnaround Time

$$= \frac{(16 + 6 + 9 + 2 + 9)}{5} = \frac{52}{5} = 8.40$$

(e) Consider 4 Jobs Arriving in System with Following Details

$C_1 = 20$ $T_1 = 100$

$C_2 = 10$ $T_2 = 150$

$C_3 = 50$ $T_3 = 300$

$C_4 = 70$ $T_4 = 250$

Example 2.12 : *For the following snapshot of process calculate the Turnaround Time (TAT) and Waiting Time (WT)*

Process	Arrival Time	Burst Time	Priority
A	0	3	2
B	1	6	1
C	4	4	3
D	6	2	4

Using FCFS,SJF (Both) RR Tq : 2ms

Solution :

(i) FCFS

Gantt Chart

A	B	C	D

Average Waiting Time:

Step–1 A = 10

B = 3

C = 9

D = 13

Step–2 A = (Actual scheduling Time – Arrival Time)

= 0 – 0 = 0

B = 3 – 1 = 2

C = 9 – 4 = 5

D = 13 – 6 = 7

$$\text{Average} = \frac{(0 + 2 + 5 + 7)}{4} = \frac{14}{4} = 3.5$$

Average Waiting Time = 3.5

Average Turnaround Time

Step–1

A = 0 + 3 = 3

B = 2 + 6 = 8

C = 5 + 4 = 9

D = 7 + 2 = 9

$$\text{Average} = \frac{(3 + 8 + 9 + 9)}{4} = \frac{29}{4} = 7.22$$

Average Turnaround Time = 7.22

(i) SJF : Non Pre–Emptive:

Gantt Chart

A	B	C	D

0 3 9 11 15

A = 0 – 0 = 0

B = 3 – 1 = 2

C = 11 – 4 = 7

D = 9 – 6 = 3

Waiting Time

A = 0

B = 3

C = 11

D = 9

$$\text{Average Waiting Time} = \frac{(0 + 2 + 7 + 3)}{4} = \frac{12}{4} = 3.0$$

Turnaround Time:

0 + 3 = 3

2 + 6 = 8

7 + 4 = 11

3 + 2 = 5

$$\text{Average TAT} = \frac{(3 + 8 + 11 + 5)}{4} = \frac{27}{4} = 6.75$$

(ii) SJF Pre–Emptive:

Consider priority first and Burst next

A	B	A	C	D

Waiting Time:

A = 7 – 1 = 6

B = 1

C = 9

D = 13

Average Waiting Time :

A = 6 – 0 = 6

B = 1 – 1 = 0

C = 9 – 4 = 5

D = 13 – 6 = 7

$$\text{Average Waiting Time:} \frac{(6 + 0 + 5 + 7)}{4} = \frac{18}{4} = 4.5$$

Turnaround Time:

6 + 3 = 9

0 + 6 = 6

5 + 4 = 9

7 + 2 = 9

$$\text{Average Waiting Time} = \frac{(9 + 6 + 9 + 9)}{4} = \frac{33}{4} = 8.2$$

(i) Round Robin (RR)

Gantt Chart:

A	B	C	D	A	B	C	B

0 2 4 6 8 9 11 13 15

A = 6

B = 2 + 5 + 2 = 9

C = 4 + 5 = 9

D = 6

Waiting Time:

A = 6 – 0 = 6

B = 9 – 1 = 8

C = 9 – 4 = 5

D = 6 – 6 = 0

$$\text{Average Waiting Time} = \frac{(6 + 8 + 5 + 0)}{4} = 4.75$$

Turnaround Time:

A = 6 + 3 = 9

B = 9 + 6 = 15

C = 8 + 4 = 12

D = 4 + 2 = 6

$$\text{Average TAT} = \frac{42}{4} = 10.5$$

Example 2.13 : *Consider the following set of processes with the length of CPU burst time given in milliseconds.*

Process	Arrival	Burst Time
P_1	0	6
P_2	1	4
P_3	3	5
P_4	5	3

Draw the Gantt Chart illustrating the execution of these process using SJF (Pre–emptive and non pre–emptive) and FCFS. Calculate average Turnaround Time, average waiting time in each case.

Solution :

(i) FCFS :

Gantt Chart

P_1	P_2	P_3	P_4

0 6 10 15 18

Average Waiting Time:

P_1 = 0 – 0 = 0

P_2 = 6 – 1 = 5

P_3 = 10 – 3 = 7

P_4 = 15 – 5 = 10

$$\text{Average waiting Time} = \frac{(0 + 5 + 7 + 10)}{4} = \frac{22}{4} = 5.5$$

Average Turnaround Time :

P_1 = 0 + 6 = 6

P_2 = 5 + 4 = 9

P_3 = 7 + 5 = 12

P_4 = 10 + 3 = 13

$$\text{Average Turnaround Time} = \frac{(6 + 9 + 12 + 13)}{4} = \frac{40}{4}$$

= 10

(ii) SJF: Non-Pre–Emptive:

Gantt Chart

P_1	P_4	P_2	P_3

0 6 9 13 18

P_1 = 0 – 0 = 0

P_2 = 9 – 1 = 8

P_3 = 13 – 3 = 10

P_4 = 6 – 5 = 1

$$\text{Average Waiting Time} = \frac{(0 + 8 + 10 + 1)}{4} = \frac{19}{4} = 4.75$$

Average TAT:

P_1 = 0 + 6 = 6

P_2 = 8 + 4 = 12

P_3 = 10 + 5 = 15

P_4 = 1 + 3 = 4

$$\text{Average TAT} = \frac{(6 + 12 + 15 + 4)}{4} = \frac{37}{4} = 9.25$$

(i) SJF: Pre–Emptive:

Gantt Chart

P_1	P_2	P_4	P_1	P_3

0 1 5 8 13 18

Waiting Time :

P_1 = 8 – 1 = 7

P_2 = 1

P_3 = 13

P_4 = 5

Average Waiting Time :

P_1 = 7 – 0 = 7

P_2 = 1 – 1 =0

P_3 = 13 – 3 = 10

P_4 = 5 – 5 = 0

$$\text{Average Waiting Time} = \frac{(7 + 0 + 10 + 0)}{4} = \frac{17}{4} = 4.25$$

Average TAT :

$P_1 = 7 + 6 = 13$

$P_2 = 0 + 4 = 4$

$P_3 = 10 + 5 = 15$

$P_4 = 0 + 3 = 3$

$$\text{Average TAT} = \frac{(13 + 4 + 15 + 3)}{4} = \frac{35}{4} = 8.75$$

Example 2.14 : *Draw Gantt chart and calculate turnaround tie, waiting time for following processor using FCFS, SJF non–pre–emptive and Round robin CPU scheduling algorithms. (Round Robin quantum =2).*

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	5
P_3	3	3
P_4	4	1
P_5	6	4

Solution :

(i) FCFS:

Gantt Chart

P_1	P_2	P_3	P_4	P_5

0 8 13 16 17 21

$P_1 = 0$

$P_2 = 8$

$P_3 = 13$

$P_4 = 16$

$P_5 = 17$

Waiting Time :

$P_1 = 0 - 0 = 0$

$P_2 = 8 - 1 = 7$

$P_3 = 13 - 3 = 10$

$P_4 = 16 - 4 = 12$

$P_5 = 17 - 6 = 11$

$$\text{Average Waiting Time} = \frac{(0 + 7 + 10 + 12 + 11)}{5}$$

$$= \frac{40}{5} = 8$$

Average Turnaround Time :

$P_1 = 0 + 8 = 8$

$P_2 = 7 + 5 = 12$

$P_3 = 10 + 3 = 13$

$P_4 = 12 + 1 = 13$

$P_5 = 11 + 4 = 15$

$$\text{Average TAT} = \frac{(8 + 12 + 13 + 13 + 15)}{5} = \frac{61}{5}$$

$$= 12.2$$

(ii) SJF: Non Pre–Emptive:

Gantt Chart

P_1	P_5	P_3	P_4	P_2

0 8 9 12 16 21

Average Waiting Time :

$P_1 = 0 - 0 = 0$

$P_2 = 16 - 1 = 15$

$P_3 = 9 - 3 = 6$

$P_4 = 12 - 4 = 8$

$P_5 = 8 - 6 = 2$

$$\text{Average Waiting Time} = \frac{(0 + 15 + 6 + 8 + 2)}{5}$$

$$= \frac{31}{5} = 6.2$$

Average Turnaround Time :

$P_1 = 0 + 8 = 8$

$P_2 = 15 + 5 = 20$

$P_3 = 6 + 3 = 9$

$P_4 = 8 + 1 = 9$

$P_5 = 2 + 4 = 6$

$$\text{Average TAT} = \frac{8 + 20 + 9 + 9 + 6}{5} = \frac{52}{5} = 10.4$$

(iii) SJF: Pre–Emptive:

Gantt Chart

P_1	P_2	P_4	P_4	P_5	P_1

0 1 6 7 10 14 21

$P_1 = 14 - 1 = 13$

$P_2 = 1$

$P_3 = 7$

$P_4 = 6$

$P_5 = 10$

Average Waiting Time :

$P_1 = 13 - 0 = 13$

$P_2 = 1 - 1 = 0$

$P_3 = 7 - 3 = 4$

$P_4 = 6 - 4 = 4$

$$\text{Average Waiting Time} = \frac{(13 + 0 + 4 + 2 + 4)}{5} = \frac{23}{5} = 4.6$$

Average Turnaround Time :

$P_1 = 13 + 8 = 21$

$P_2 = 0 + 5 = 5$

$P_3 = 4 + 3 = 7$

$P_4 = 2 + 1 = 3$

$P_5 = 4 + 4 = 8$

$$\text{Average TAT} = \frac{(21 + 5 + 7 + 3 + 8)}{5} = \frac{44}{5} = 8.8$$

(iv) Round Robin (RR):

Gantt Chart :

P_1	P_2	P_3	P_4	P_5	P_1	P_2	P_3	P_5	P_1	P_2	P_1

0 2 4 6 7 9 11 13 14 16 18 19 21

$P_1 = 0 + 7 + 7 + 3 = 17$

$P_2 = 2 + 9 + 7 = 18$

$P_3 = 4 + 7 = 11$

$P_4 = 6$

$P_5 = 7 + 5 = 12$

Average Waiting Time :

$P_1 = 17 - 8 = 9$

$P_2 = 18 - 1 = 17$

$P_3 = 11 - 3 = 8$

$P_4 = 6 - 4 = 2$

$P_5 = 12 - 6 = 6$

Average Waiting Time :

$P_1 = 9 + 8 = 17$

$P_2 = 17 + 5 = 22$

$P_3 = 8 + 3 = 11$

$P_4 = 2 + 1 = 3$

$P_5 = 6 + 4 = 10$

$$\text{Average TAT} = \frac{(17 + 22 + 11 + 3 + 10)}{5} = \frac{63}{5} = 12.5$$

EXERCISE

1. Explain the concept of multiprogramming.
2. What are the different process states? Explain the same in detail. Also provide details for the process states in UNIX.
3. What are the difference between the kernel mode and user mode?
4. What are the contents of Thread Control Block?
5. Write a note on user level and kernel level threads.
6. Difference between process and threads.
7. Explain the contents of PCB and also explain the significance of PCB.
8. What is the difference between a process and a thread? What are the contents of Thread Control Block? State the advantages and disadvantages of user level threads.
9. Make the comparison between modern UNIX kernel and the traditional UNIX kernel with neat diagram.
10. Draw and explain the process state transition diagram. Explain the structure of PCB.
11. What is PCB? Draw the schematic showing all fields of a PCB.
12. Explain in detail the steps involved in UNIX process creation
13. Consider the following set of processes with the length of the CPU burst time given in milliseconds.

Process	Arrival Time	Burst Time	Priority
P_1	0	8	3
P_2	1	1	1
P_3	2	3	2
P_4	3	2	3
P_5	4	6	4

Draw the Gantt charts illustrating the execution of these processes using FCFS, SJF (pre–emptive and non pre–emptive), priority scheduling. Calculate turnaround time and waiting time for each process for all 5 scheduling algorithm.

14. With the help of neat diagram explain the concept context switching.
15. Draw process state transition diagram and explain

16. What are the various types of scheduling? Explain in brief.
17. What is thread scheduling?
18. Describe in detail the differences between short–term, medium–term and long–term schedulers with the help of a neat diagram.
19. Explain the design issues for multiprocessor scheduling. State the 4 approaches for multiprocessor thread scheduling and process assignment.
20. List and explain 4 classes of real time scheduling.
21. Consider the following set of processes, with the length of the CPU burst time given in milliseconds.

Process	Arrival Time	Burst Time	Priority
P_1	0	7	3
P_2	1	1	1
P_3	2	3	2
P_4	3	4	4

Draw the Gantt charts illustrating the execution of these processes using Round Robin (Time quantum=1ms), FCFS, SJF, priority (pre–emptive and non pre–emptive). Calculate turnaround time, waiting time.

22. State and explain the scheduling criteria fo uniprocessor scheduling.
23. Describe in detail the difference between short–term medium–term and long–term schedulers with th help of a neat diagram.
24. Explain the term granularity in multiprocesso scheduling. Briefly describe 5 different categories synchronization granularity.
25. Compare FCFS, Round Robin and SJF (pre–emptiv and non pre–emptive) on the basis of followin characteristics :
    (a) Selection function (b) Decision mode
    (c) Response time (d) Effect on processes.
26. What is process? Explain process states with nea diagram.
27. List the information in process control block an explain it.
28. What is thread? Define user level thread and kerne level thread.
29. Explain the following functions with reference to c.
    Pthread_creat()
    Pthread)join()
    Fork().
30. Explain cooperating processes.

✠ ✠ ✠

Unit III

PROCESS SYNCHRONIZATION

3.1 CRITICAL-SECTION PROBLEM

- Each process has a segment of code called as critical section. Critical section is used to avoid race conditions on data items. In critical section, the process may be changing common variables, updating a table, writing a file and so on. At any moment, at most, one process can execute in Critical Section (CS).
- When one process is executing in its section, no other process is to be allowed to execute in its critical section. The execution of critical sections by the process is mutually exclusive in time.
- Following Algorithm 3.1 shows the general structure of process (P). Each process containing three sections. These sections are entry section, exit section and remainder section.

Algorithm 3.1 : General Structure of a Process (P)

```
do
{
    Entry section
        critical section
    Exit section
        remainder system
} while (1);
```

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

1. Mutual exclusion
2. Progress
3. Bounded waiting

1. **Mutual Exclusion :** Suppose one process (P_i) is executing in its critical section, then no other processes are allowed to execute in critical sections.
2. **Progress :** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision on which will enter its critical section reset. This section of the process cannot be postponed indefinitely.
3. **Bounded Waiting :** When a process request access to a critical section, a decision that grants it access may be not delayed indefinitely. A process may not be denied access because of starvation or deadlock.

Solution to the Critical Section Problem is as Follows :

The code implementing an operating system (Kernel code) is subjected to several possible race conditions. For instance

- A kernel data structure that maintain a list of all open files in the system.
- If two processes were to open files simultaneously, the updates to this list could result in a race condition.

Other Kernel Data Structures : Structures for maintaining memory allocation, for maintaining process lists etc.

The kernel developers have to ensure that the operating system is free from each race conditions.

Two approaches, used to handle critical sections in operating system :

1. **Non-Preemptive Kernels :** Does not allow a process running in kernel mode to be pre-empted. If free from race conditions on kernel data structures, as only one process is active in kernel at a time.
2. **Preemptive Kernels :** Allow a process to be pre-empted while it is running in kernel mode. The kernel must be carefully designed to ensure that shared kernel data are free from race conditions.

Why Prefer Pre-emptive Kernel?

- Less risk that a kernel mode process will run for an arbitrarily long period.
- Allow a real-time process to pre-empt a process currently running in kernel.
- Windows XP and Window 2000 are non-preemptive kernels.
- Several commercial versions of UNIX (including solaris and IRIX) and Linux 2.6 Kernel (and the later version) are pre-emptive.

3.1.1 Two Process Solutions

We consider only two processes P_0 and P_1 for solving critical section problem. Following are the algorithms for critical section.

1. Algorithm 3.2 :

- Both the processes P_0 and P_1 share the common integer variable. We assign the name to this variables as turn and is initialized to 0 or 1.

```
if turn = = i
```

then process P_0 is allowed to execute in its critical section.

- Following Algorithm 3.2 shows the structure process P_0 in Algorithm 3.1 Algorithm 3.1 allows only one process to enter into critical section.
- For example, if turn = = 0 and P_1 is ready to enter its critical section, P_1 cannot do so, even though P_0 may be in its remainder section.

Algorithm 3.2 : Structure of Algorithm

```
do
{
    while (turn ! = i)
            critical section
    turn = I;
            remainder section
} while (1);
```

2. Algorithm 3.3 :

- Algorithm 3.1 cannot give sufficient information about the state of each process. It keeps only records of the process which is entered into the critical section. To solve this problem, variable turn is replaced with new variable called flag. It is initialized as:

```
boolean flag [2];
```

- First the elements of array are initialised to false. If Flag[i] is true, then Pi is ready to enter the critical section.

The structure for algorithm 3.2 is given below:

Algorithm 3.3 shows the structure of algorithm 3.2 for process

Algorithm 3.3 : Structure for Algorithm 2

```
do
{
    flag[i] = true;
    while C flag[j];
        critical section
    flag[i] = false
        remainder section
} while(1);
```

- In this algorithm, process P_i first sets flag[i] to be true, then process P_i is ready to enter its critical section. Process P_i also checks for process P_j. If process P_j were ready, the Pi would wait until flag[i] was false. So, process P_i would enter into the critical section.

3. Algorithm 3.4 :

- It gives the correct solution to the critical section problem. Algorithm 3 satisfies all the three requirements of the critical section. The process shares two variables:

```
Boolean Flag[2];
int turn;
Initialize condition is
flag[0] = flag[1] = false
```

Algorithm 3.4 shows the structure of process P_i in algorithm

Algorithm 3.4 : Structure of Process P_i in Algorithm 3

```
do
{
    flag[i] = true;
    turn = j
    while (flag [j] && turn == j);
    critical section
    flag[i] = false
    remainder section
} while(1);
```

- For process, to enter the critical section, first set flag[i] to be true and then set turn to the value j. If both processes try to enter at the same time, turn will set to both i and j at the same time. This algorithm satisfies all the three requirements of critical section.

3.1.2 Multiple Process Solutions

Bakery algorithm is used in multiple processors. It solves the problem of critical section for n processes. Each process requesting entry to critical section is given a numbered token such that the number on the token is larger than a maximum number issued earlier. This algorithm was developed for a distributed environment. The algorithm permits processes to enter the critical section in the order of their token numbers.

The bakery algorithm cannot guarantee that two processes do not receive the same number. In this case, process with the lowest name is served first. If P_i and P_j receive the same number and if i < j, then P_i is served first. Algorithm 3.5 shows the structure of process P_i in the bakery algorithm.

Algorithm 3.5 : Structure of Process P_i in the Bakery Algorithm

```
do
{
    choosing[i] = true;
number[i] = max(number[0], number[1], number[n-1] + 1;
    choosing[i] = false;
    for(j=0; j<n; j++)
{       while(choosing[j]);
        while((number[j]!=0) && number [i,j] <
        number[I,i]))       }
        critical section
        number[i]=0;
    remainder section
}
while(1);
```

3.2 CRITICAL REGIONS

- Critical regions are small and infrequent so that system throughput is largely unaffected by their existence, critical region is a control structure for implementing mutual exclusion over a shared variable.
- The declaration of shared variable is given below :

```
var mutex : shared T;
```

- The variable mutex of type T is to be shared among many processes. The variable mutex can be accessed by only inside the region statement of the following form :

```
region mutex when B do S;
```

- While statement S is being executed, no other process can access the variable mutex. B is the boolean expression that governs the access to the critical region. Critical regions enforce restricted usage of shared variables and prevent potential errors resulting from improper use of ordinary semaphores. Critical region is very convenient for mutual exclusion. However, it is less versatile than a Semaphore.

3.2.1 Conditional Critical Regions

- Conditional critical region allows us to specify synchronization as well as mutual exclusion. It is similar to a critical region. The shared variable is declared in the same way.
- Conditional critical region providers the following features:
  1. Provide mutual exclusion.
  2. It permits a process executing to conditional critical region to block itself until an arbitrary boolean condition becomes true.

Following algorithm gives the idea about conditional critical regions :

Algorithm 3.6 :

```
var x : shared T;
    begin
        repeat
.....
.....
                region X do
                being
                .....
                await condition;
                .....
                end;
```

Variable x is called the conditional critical region variable. The above code allows a process waiting on a condition with a critical region to the suspended in a special queue, pending satisfaction of the related condition.

3.3 PETERSON'S ALGORITHM IN PROCESS SYNCHRONIZATION

Problem: The producer consumer problem (or bounded buffer problem) describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue. Producer produce an item and put it into buffer. If buffer is already full then producer will have to wait for an empty block in buffer. Consumer consume an item from buffer. If buffer is already empty then consumer will have to wait for an item in buffer. Implement Peterson's Algorithm for the two processes using shared memory such that there is mutual exclusion between them. The solution should have free from synchronization problems.

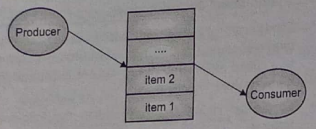


Fig. 3.1

Peterson's Algorithm 3.7 :

```
// code for producer (j)

// producer j is ready
// to produce an item
flag[j] = true;

// but consumer (i) can consume an item
turn = i;

// if consumer is ready to consume an item
// and if its consumer's turn
while(flag[i] == true&& turn == i)

  { // then producer will wait }

  // otherwise producer will produce
  // an item and put it into buffer (critical Section)

  // Now, producer is out of critical section
  flag[j] = false;
  // end of code for producer

  //-------------------------------------------------------
  // code for consumer i

  // consumer i is ready
  // to consume an item
  flag[i] = true;

// but producer (j) can produce an item
  turn = j;

  // if producer is ready to produce an item
  // and if its producer's turn
  while(flag[j] == true&& turn == j)

    { // then consumer will wait }

    // otherwise consumer will consume
    // an item from buffer (critical Section)

    // Now, consumer is out of critical section
    flag[i] = false;
// end of code for consumer
```

Explanation of Peterson's Algorithm

- Peterson's Algorithm is used to synchronize two processes. It uses two variables, a bool array flag of size 2 and an int variable turn to accomplish it.
- In the solution, i represents the Consumer and j represents the Producer. Initially the flags are false. When a process wants to execute it's critical section, it sets it's flag to true and turn as the index of the other process. This means that the process wants to execute but it will allow the other process to run first. The process performs busy waiting until the other process has finished it's own critical section.
- After this the current process enters it's critical section and adds or removes a random number from the shared buffer. After completing the critical section, it sets it's own flag to false, indication it does not wish to execute anymore.
- The program runs for a fixed amount of time before exiting. This time can be changed by changing value of the macro RT.

Program 3.1 : // C Program to Implement Peterson's Algorithm

```
// for producer-consumer problem.
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <sys/types.h>
```



```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdbool.h>
#define _BSD_SOURCE
#include <sys/time.h>
#include <stdio.h>

#define BSIZE 8 // Buffer size
#define PWT 2 // Producer wait time limit
#define CWT 10 // Consumer wait time limit
#define RT 10 // Program run-time in seconds

intshmid1, shmid2, shmid3, shmid4;
key_t k1 = 5491, k2 = 5812, k3 = 4327, k4 = 3213;
bool* SHM1;
int* SHM2;
int* SHM3;

intmyrand(intn) // Returns a random number between 1 and n
{
  time_tt;
  srand((unsigned)time(&t));
  return(rand() % n + 1);
}

intmain()
{
  shmid1 = shmget(k1, sizeof(bool) * 2, IPC_CREAT | 0660); // flag
  shmid2 = shmget(k2, sizeof(int) * 1, IPC_CREAT | 0660); // turn
  shmid3 = shmget(k3, sizeof(int) * BSIZE, IPC_CREAT | 0660); // buffer
  shmid4 = shmget(k4, sizeof(int) * 1, IPC_CREAT | 0660); // time stamp

  if(shmid1 < 0 || shmid2 < 0 || shmid3 < 0 || shmid4 < 0) {
    perror("Main shmget error: ");
    exit(1);
  }
  SHM3 = (int*)shmat(shmid3, NULL, 0);
  intix = 0;
  while(ix < BSIZE) // Initializing buffer
    SHM3[ix++] = 0;

  structtimeval t;
  time_tt1, t2;
  gettimeofday(&t, NULL);
  t1 = t.tv_sec;

  int* state = (int*)shmat(shmid4, NULL, 0);
  *state = 1;
  intwait_time;

  inti = 0; // Consumer
  intj = 1; // Producer

  if(fork() == 0) // Producer code
  {
    SHM1 = (bool*)shmat(shmid1, NULL, 0);
    SHM2 = (int*)shmat(shmid2, NULL, 0);
    SHM3 = (int*)shmat(shmid3, NULL, 0);
    if(SHM1 == (bool*)-1 || SHM2 == (int*)-1 || SHM3 == (int*)-1) {
      perror("Producer shmat error: ");
      exit(1);
    }

    bool* flag = SHM1;
    int* turn = SHM2;
    int* buf = SHM3;
    intindex = 0;

    while(*state == 1) {
      flag[j] = true;
```

```
        printf("Producer is ready now.\n\n");
        *turn = i;
        while(flag[i] == true&& *turn == i)
            ;

        // Critical Section Begin
        index = 0;
        while(index < BSIZE) {
            if(buf[index] == 0) {
                inttempo = myrand(BSIZE * 3);
                printf("Job %d has been produced\n",
tempo);
                buf[index] = tempo;
                break;
            }
            index++;
        }
        if(index == BSIZE)
            printf("Buffer is full, nothing can be
produced!!!\n");
        printf("Buffer: ");
        index = 0;
        while(index < BSIZE)
            printf("%d ", buf[index++]);
        printf("\n");
        // Critical Section End

        flag[j] = false;
        if(*state == 0)
            break;
        wait_time = myrand(PWT);
        printf("Producer will wait for %d seconds\n\n",
wait_time);
        sleep(wait_time);
    }
    exit(0);
}

if(fork() == 0) // Consumer code
{
    SHM1 = (bool*)shmat(shmid1, NULL, 0);
    SHM2 = (int*)shmat(shmid2, NULL, 0);
    SHM3 = (int*)shmat(shmid3, NULL, 0);
    if(SHM1 == (bool*)-1 || SHM2 == (int*)-1 || SHM3
== (int*)-1) {
        perror("Consumer shmat error:");
        exit(1);
    }

    bool* flag = SHM1;
    int* turn = SHM2;
    int* buf = SHM3;
    intindex = 0;
    flag[i] = false;
    sleep(5);
    while(*state == 1) {
        flag[i] = true;
        printf("Consumer is ready now.\n\n");
        *turn = j;
        while(flag[j] == true&& *turn == j)
            ;

        // Critical Section Begin
        if(buf[0] != 0) {
            printf("Job %d has been consumed\n",
buf[0]);
            buf[0] = 0;
            index = 1;
            while(index < BSIZE) // Shifting remaining
jobs forward
            {
                buf[index - 1] = buf[index];
                index++;
            }
            buf[index - 1] = 0;
        } else
            printf("Buffer is empty, nothing can be
consumed!!!\n");
```



```
        printf("Buffer: ");
        index = 0;
        while(index < BSIZE)
            printf("%d ", buf[index++]);
        printf("\n");
        // Critical Section End

        flag[i] = false;
        if(*state == 0)
            break;
        wait_time = myrand(CWT);
        printf("Consumer will sleep for %d
seconds\n\n", wait_time);
        sleep(wait_time);
    }
    exit(0);
}
// Parent process will now for RT seconds before
causing child to terminate
while(1) {
    gettimeofday(&t, NULL);
    t2 = t.tv_sec;
    if(t2 - t1 > RT) // Program will exit after RT
seconds
    {
        *state = 0;
        break;
    }
}
// Waiting for both processes to exit
wait();
wait();
printf("The clock ran out.\n");
return0;
}
```

Output:

```
Producer is ready now.

Job 9 has been produced
Buffer: 9 0 0 0 0 0 0 0
Producer will wait for 1 seconds

Producer is ready now.

Job 8 has been produced
Buffer: 9 8 0 0 0 0 0 0
Producer will wait for 2 seconds

Producer is ready now.

Job 13 has been produced
Buffer: 9 8 13 0 0 0 0 0
Producer will wait for 1 seconds

Producer is ready now.

Job 23 has been produced
Buffer: 9 8 13 23 0 0 0 0
Producer will wait for 1 seconds

Consumer is ready now.

Job 9 has been consumed
Buffer: 8 13 23 0 0 0 0 0
Consumer will sleep for 9 seconds

Producer is ready now.

Job 15 has been produced
Buffer: 8 13 23 15 0 0 0 0
Producer will wait for 1 seconds

Producer is ready now.

Job 13 has been produced
Buffer: 8 13 23 15 13 0 0 0
Producer will wait for 1 seconds
```

Producer is ready now.

Job 11 has been produced

Buffer: 8 13 23 15 13 11 0 0

Producer will wait for 1 seconds

Producer is ready now.

Job 22 has been produced

Buffer: 8 13 23 15 13 11 22 0

Producer will wait for 2 seconds

Producer is ready now.

Job 23 has been produced

Buffer: 8 13 23 15 13 11 22 23

Producer will wait for 1 seconds

The clock ran out.

3.4 SYNCHRONIZATION HARDWARE

- Hardware feature can make the programming task easier and improve system efficiency. Various synchronization mechanisms are available to provide inter process co-ordination and communication. Test and Set instruction is used for critical section problem.
- In must synchronization schemes, a physical entity must be used to represent the resources. This instruction is executed automatically. This test and set instruction is used in multiprocessor environment. If two test and set instructions are executed simultaneously, they will be executed sequentially in some arbitrary order.

Test and set instructions are initially as follows :

```
boolean test and set (boolean and target)
{
    boolean rv = target;
    target = true;
    return rv;
}
```

Test and set instruction is used in implementing mutual exclusion. Data structures for this is given below:

```
do
{
    while (Test and Set (lock));
    critical section
    lock = false;
    remainder section
} while(1)
```

3.5 SEMAPHORES

- The hardware-based solutions are complicated for application programmers to use.
- Semaphore is synchronization tool. It is used to solve critical section problem.
- Semaphore is an integer value. A semaphore is an object that consist of a counter, awaiting list of processes and two methods: signal and wait.
- The definition of wait() is as follows :

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

- After decreasing the counter by 1, if the counter value becomes negative, then add the caller to the waiting list and then block itself.
- The definition of signal() is as follows :

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

- After increasing the counter by 1, if the new counter value is not positive, then remove a process P from the waiting list, resume the execution of process P and return.
- Semaphores are executed automatically. There is no guarantee that no two processes can execute wait and signal operation on the same semaphore at the same time.

- No two processes can execute wait() and signal() operations on the same semaphore at same time.
- wait() and signal() must be executed automatically, otherwise race condition may occur.

Binary Semaphore

- Also known as mutex locks (mutual exclusion).
- Range only between 0 and 1.
- Used to deal with the critical section problem for multiple processes.

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

Counting Semaphore

- Range over an unrestricted domain. Used to control access to a given resource consisting of a finite number of instances.
- The semaphore is intialized to the number of resources available. When the count for the semaphore goes to 0, all resources are being used; processes that with to use a resource will block until the count becomes greater than 0.
- Semaphores can be used to solve synchronisation problems. P_1 with a statement S_1 and P_2 with S_2, we require that S_2 be executed only after S_1 has completed.
- A common semaphore synch, initialised to 0.

S_1;	wait (synch);
signal(synch);	S_2;
P_1	P_2

Properties of Semaphore

- Semaphores are machine independent.
- Semaphores are simple to implement.
- Correctness is easy to determine.
- Semaphore acquire many resources simultaneously.
- Can have many different critical sections with different semaphores.
- For both semaphores, a queue is used to hold processes waiting on the semaphore. FIFO policy is used to remove the process from the queue.
- The process that has been blocked and longest is released from the queue first is called strong semaphore.
- A semaphore that does not specify the order in which processes are removed from the queue is called a weak semaphore.

3.6 CLASSICAL PROBLEM OF SYNCHRONIZATION

Classic problem of synchronization are listed below :

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

3.6.1 Readers and Writers Problem

A database is to be shared among several concurrent processes :

(a) Some processes may want only to read the database-readers.

(b) Others may want to update (to read and write) writers.

- Two groups of processes, readers and writers are accessing a shared resource by the following rules :
  1. Readers can read simultaneously.
  2. Only one writer can write at any time.
  3. When a writer is writing, no reader can read.
  4. If there is any reader reading, all incoming writers must wait. Thus readers have higher priority.
- The reader processes share the following data structure semaphore mutex, wrt;

```
int read count();
```

- Read count is initialized to 0. Keep track of how many processes are currently reading the database.
- Mutex and wrt are initialized to 1. Mutex is used to ensure mutual exclusion when the variable read count is updated, wrt., shared with writers, functions as a mutex-exclusion / semaphore for the writers.
- A writer processes structure :

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

- A reader processes structure

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

Problem Analysis

- We need a semaphore to block readers if a writer is writing.
- When a writer arrives, it must be able to know if there are readers reading, so a reader count is required which must be protected by a lock.
- This reader priority version has a problem. Bounded waiting condition may be violated if readers keep coming, causing the waiting writers no chance to write.
  1. When a reader comes in, it increases the count.
  2. If it is the 1st reader, waits until no writer is writing.
  3. Reads data.
  4. Decrease the counter.
  5. Notifies the writer that no reader is reading if it is the last.
  6. When a writer comes in, it waits until no reader is reading and no writer is writing. Then, it writes data.
  7. Finally, notifies readers and writers that no writer is in.
- The readers-writers problem has several variations all involving priorities, may be the readers having priority or writers having higher priority.
- Can the producer-consumer problem be simply a special case of the readers-writers problem with single writer (the producer) and single reader (the consumer).

  The answer is no. The producer is not just a writer. It must read queue pointers to determine where to write the next item and it must determine if the buffer is full.
- Similarly, the consumer is not just a reader because it must adjust the queue pointers to show that it has removed a unit from the buffer.

3.7 MONITORS SYNCHRONIZATIONS IN SOLARIS

- Monitors are based on abstract data types. A monitor is a programming language construct that provides equivalent functionality to that of semaphores but is easier to control. A monitor consists of procedures that shared object and administrative data.
- **Characteristics of Monitors are as Follows :**
  - Only one process can be active within the monitor at a time.
  - The local data variables are accessible only by the monitor's procedures and not by any external procedure.
  - A process enters the monitors by invoking one of its procedures.
- Monitor provides high level of synchronization. The synchronization of process is accomplished via two specific operations namely wait and signal, which are executed within the monitor's procedures.
- Monitors are high level data abstraction tool combining three features :
  1. Shared data
  2. Operation on data
  3. Synchronization, scheduling
- A monitor is characterised by a set of a programmer defined operators. Monitors were derived to simplifying the complexity of synchronization problems. Every synchronization problem that can be solved with monitors can also be solved with semaphores and vice versa.
- Monitors are based on abstract datatypes. Monitor is an abstract datatype for which only one process may be executing a procedure at any given time. Processes desiring to enter the monitor when it is already in use must wait. This waiting is automatically managed by the monitor.
- Fig. 3.2 shows the schematic view of monitor.

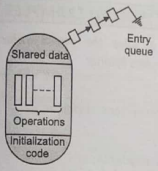


Fig. 3.2

- A monitor is a software module consisting of one or more procedures, an initialization sequence and local data. Below shows the syntax of monitor.

Monitor Syntax

```
Monitor monitor_name
{
    declaration of shared variable
    procedure body P1()
    {
        .....
        .....
    }
    P2()
    {
        procedure body
            .....
            .....
    }
    .
    .
    .
    Pn()
    {
        Procedure body
    }
        {
            initialization code
        }
}
```

The monitor construct has been implemented in a number of programming languages. Since monitors are a language feature, they are implemented with the help of a computer. In response to the keywords monitor, condition, signal, wait and notify, the computer insert little bits of code in the program. The data variables in monitor can be accessed by only one process at a time. A shared data structure can be protected by placing it in a monitor. The data inside the monitor may be either global to all procedures within the monitors or local top a specific procedure.

- A monitor supports synchronization by the use of condition variables that are contained within the monitor and accessible only within the monitor. Two condition variables are:
  1. **x.wait() :** Suspended execution of calling process on condition x. The monitor is now available for use by another process.
  2. **x.signal() :** Resume execution of some process suspended offer a x. Wait on the same condition. This operation resumes exactly one suspended process.
- A condition variable is like a semaphore with two differences:
  1. A semaphore counts the number of excess up operations, but a signal operation on a condition variable has no effect unless some process is waiting, a wait on a condition variable always blocks the calling process.
  2. A wait on a condition variable automatically does an upon the monitor mutex and blocks the caller.

Fig. 3.3 shows monitor with a condition variable.

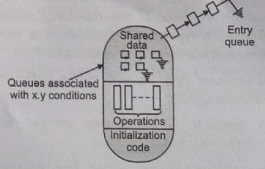


Fig. 3.3 : Monitor with condition variables

```
interface condition
{
    public void x.signal();
    public void x.wait();
}
```

Program 3.2 : Bounded Buffer Problem using Monitors

```
monitor Bounded Buffer
{
    private Buffer b = new Buffer(20);
    private int count = 0;
    private condition nonfull, nonempty();
    public void add (object item)
}
if(count == 20)
    nonfull.wait();
    b.add(item);
    count++;
    non-empty.signal();
}
public object remove()
{
    if(count == 0)
    nonemptyh.wait();
    item result = be removal();
    count = count-1;
    nonfull.signal();
        return result;
}
```

Each condition variable is associated with some logical condition on the state of the monitor. Consider what happen when a consumer is blocked on a non-empty condition variable and producers calls add :

- The producer adds the item to the buffer and calls non-empty.signal().
- The producer is immediately blocked and the consumer is allowed to continue.
- The consumer removes the item from the buffer and leaves the monitor.
- The producer wakes up and since the signal operation wait the last statement in add, leaves the monitor.

Monitor are higher level concept than P and V. They are easier and safer to use but less flexible. Many languages are not supported by the monitor. Java is making monitors much more popular and well known.

SOLVED EXAMPLES

Example 3.1 : *Solve the reader-writer problem using monitor with readers priority.*

Solution :

```
    readers-writers : monitor;
begin
    integer readercount;
    condition ok read, ok write;
    boolean busy;
procedure started;
    begin
    if busy then ok read.wait;
    readercount:=readercount +1;
    okread.signal;
end startread;
procedure end read;
begin
    readercount:=readercount -1;
if
    readercount := then okwrite.signal;
end endread;
procedure start write;
    begin if busy OR read count ≠ 0 the
okwrite.wait
    busy:=true;
    end startwrite;
procedure endwrite;
    begin
    busy := false;
    if ok read.queue then okread.signal
    else okwrite.signal
    end endwrite;
    begin(*Initialization*)
    readercount:=0;
    busy:=false;
    end;
    end readers-writers;
```


Example 3.2 : *Solve Procedure-consumer problem with monitors.*

Solution :

```
monitor procedure-consumer
condition fullempty;
integer count;
procedure insert(item : integer);
begin
    if count = N then wait (full);
    insert-item(item);
    count:=count +1;
    ifcount = 1 then signal (empty)
end;
function remove : integer;
    begin
        if count = 0 then wait (empty);
        remove = remove_item;
        count:=count -1;
        ifcount = N-1 then signal (full);
    end;
    count := 0;
end monitor
procedure producer
        begin
        while true do
    begin
        item=produce-item;
        producer-consumer insert (item)
    end
end;
    procedure consumer;
        begin
        while true do;
            begin
            item = procedure-consumer.remove;
            consume-item(item)
        end
end
```

By making the mutual exclusion of critical regions automatic, monitors make parallel programming much less error prone than with semaphores.

Drawbacks of Monitors :

- Major weakness of monitors is the absence of concurrency if the monitor encapsulates the resource, since only one process can be active within a monitor at a time.
- There is the possibility of deadlocks in the case of nested monitor calls.
- Monitor concept is its lack of implementation most commonly used programming languages.
- Monitors cannot easily be added if they are not natively supported by the language.

3.7.1 Difference between Monitors and Semaphores

- A semaphore is an operating system abstract data type whereas monitors are based on abstract data types.
- Semaphore-level synchronization primitives are difficult to use for complex synchronization situations. Monitors were derived to simplify the complexity of synchronization problems by abstracting away details.
- Semaphores provide a general purpose mechanism for controlling access to critical sections. Their use does not guarantee that access will be mutually exclusive or deadlock will be avoided. A monitor is a programming language construct that guarantees appropriate access to critical sections.
- Monitors uses condition variables. A condition variable is like a semaphore with two differences.
  1. A semaphore counts the number of excess up operations but a signal operation on a condition variable has no effect unless some process is waiting. A wait on a condition variable always blocks the calling process.
  2. A wait on a condition variable automatically does an up on the monitor mutex and blocks the caller.

Example 3.3 : *Implement the reader-writer problem using semaphores and discuss how the critical section requirements are fulfilled.*

Solution : Reader-writer problem using semaphore :

Program readers and writers :

```
var read count : integer;
    x, wsem : semaphore (: = 1)
```

```
procedure reader;
begin
    repeat
    wait(x);
    read count := read count+1;
    if read count = 1 then wait (wsem);
    signal(x);
    READUNIT;
    wait(x);
    read count := read count - 1;
    if read count = 0 then signal (wsem);
    signal(x);
    forever
    end;
Procedure writer;
begin
    repeat
    wait(wsem);
    WRITEUNIT;
    signal(wsem)
    forever
    end;
    begin
        readcount :=0;
    parbegin
        reader;
        writer
        parent
    end
```

In the above solution, reader have priority, the semaphore wsem is used to enforce mutual exclusion. So long as one writer is accessing the shared data area no other writers and no readers may access it. The reader process also makes use of wsem to enforce mutual exclusion. The global variable read count is used to keep track of the number of readers and the semaphore x is used to assume that read count is updated properly.

Example 3.4 : *Implement the producer-consumer problem using monitors and discuss how the critical sections requirements are fulfilled;*

Solution :

```
Monitor procedure-consumer
condition fullempty;
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
        wile true do
    begin
        item = producer-item;
            producer-consumer.insert(item)
        end
end;
    procedure consumer;
        begin
            while true do
            being
            item = procedure_consumer.remove;
            consumer_item(item)
        end
end;
```


Types of Linux Inter-Process Communication and Con-Current Mechanism

1. **Signals:** Is one of the methods used for notification of event by processes or the kernel.
2. **Pipes:** Are a part of shell library that is generally identified by "|" character to direct output of one program as input of another.
3. **FIFOs:** Are also known as Named pipes. They operate on first data in, first data out basis and are used for unrelated process communication.
4. **Message Queues:** Is another from of IPC that allow one or more processes to write messages that can be read by one or more other processes.
5. **Semaphores:** Are shared variables used to have control over resources. These counters are to maintain atomicity of transaction.
6. **Shared Memory:** The mapping of a memory area to be shared by multiple processes.

1. Signals :

- This is one of the methods that notify the occurrence of event to kernel of system or to whom it is intended for.
- This is a privilege facility provided to kernel as well as the super-user/administrator of the system only.
- Process that belongs to normal user can only send signal to processes with the same user ID and Group ID or to processes in the same process group.
- Signal are generated by setting the required signal bit that belongs to structure by name from "task_struct's".
- Signal generated needs to be caught using a module known as signal handler.
- On catching the signal, signal handler can initiate signal action as a part of service routine just like interrupt service routine.
- There are a set of defined signal that the kernel can generate or that can be generated by other processes in system.

Example :

1. Sighup 2. Sigint 3. Sigquit
4. Sigill 5. Sigkill 6. Sigalrm

2. Pipes

- Directing output of one program as input to other is a common feature that is used by most of today's Linux operating system. This is implemented as shown below:

```
$ ps - elf | grep firefox >> abc.txt
```

- In the above example we can see following actions in chronological order.
- Command ps–elf will list out the process details running in the system.
- This output is piped to a command known as "grep" which acts as the filter. It filters out only those records from the previously generated output that comprises of word "firefox".
- In the third stage this filtered output in redirected to a file abc.txt.
- User can execute "$cat abc.txt" and view the final result of the combination of all three commands at one place.
- If available, details of a process generated due to opening of a firefox browser by user will be seen in the file abc.txt
- The pipe exists only inside the kernel and cannot be accessed by processes that created it.

3. Named Pipes/FIFO's

- This is another technique used for IPC. Named pipes are also known as FIFO's which stands for a first In, First out abbreviation.
- First data written into the pipe will be the first data read from the pipe and hence termed "FIFO". FIFO's are created using command mkfifo and are available as entities in the file system.
- FIFO physically exists in file system and is opened and closed by its users.
- It will be the duty of Linux OS to look after synchronization of FIFO. Simultaneously read and write attempts must be taken care of to avoid data inconsistency.
- FIFO's are used in the same way as pipe and they use the same data structure and operations as pipes use.

Example:

Create two named pipes, pipe1 and pipe2 and execute the following commands:

```
$echo -n V | cat - pipe1 > pipe2 &
$cat <pipe2> pipe1
```

As a part of output nothing will appear on the screen but try executing "top" command that show the process status you'll see that both cat programs are running and copying letter V back and forth in an infinite loop.

Sockets

- Another effective way of IPC is the use of sockets. Sockets provide point-to-point, two-way communication between two processes.
- Sockets creates an endpoint of communication to which specific identity is tagged.
- Based on the requirements we can create different types of sockets.
- Socket can be defined using different properties which remain visible to the application. Processes using socket for communication can communicate only with that process which is also using a socket that has same properties.

The Step Involved in Establishing a Socket on the Client Side are as Follows:

- Create a socket using socket() system call.
- Connect the socket to the address provided by server using the connect() system call.
- Send and receive data.
- There are a number of ways to do this, but the simplest is to use the read() and write() in TCP?IP protocol is used for communication or sendto() and receivefrom() system calls in case UDP/IP protocol used.

The Step Involved in Establishing a Socket on the Server Side are as Follows:

- Create a socket using socket() system call.
- Initiate socket address parameters using bind() system call. Address parameter include IP (local loop back adapter address "127.0.0.1" in case of single machine) address and a port number on the host machine.
- Watch for incoming connection request using listen() system call.
- Accept a connection if all parameter are matching using accept() system call, else reject the connection request stating valid reason.
- Once connection is establish, server can read client requested query and write back the result using same socket till client initiates termination of connection.

Sample Client Code:

```
#include <unp.h>
int
    main(int argc, char **argv)
    {
        int sockfd, n;
        char    recvline[mAXLINE + 1];
        st.ruct sockaddrin servaddr;
        if(argc !=2)
            err_quit("usage: a.out <IPaddress>");
        if(sockfd  =  socket(AF:INET,SOCK_STREA
0))
        <---------1
        err sys("socket error");
        bzero(&servaddr, sizeof(servaddr));
        servaddr.sin_faily = AF_INET;
        servaddr.sin_port = htons(13); it daytime
        server  t/
        if(inet_pton(AFINET, argv[1],
        &servaddr.sin addr)<0)<-------2
            err_quitrinet_pon error for :(s, argv[1]);
        if(connect (sockfd, (SA t) &servaddr,
        sizeof (servaddr)))<
            err_sys("connect error");
        while((n = read(sockfd, revlirie, MAXLINE));
        <--------3
        {
            Recvline[n] = 0; it  /*null terminate*/
            If(fputs(recvline, stdout)==EDF)<-------4
            err_sys("fput error");
        }
        if(n < 0)
            err_sys("read error");
        exit(0);
        }
```

From the above client code we can see four system c used chronologically to ask a queue "what is the day, d and time?"

- Socket is created.
- Using credential of created socket (socket connection request is made to server.
- Read the data that is send by server, (this is prefera the day, date and time of query in above example).
- Display the result on screen.

Sample Server Code:

```
#include<time.h>
int
main(int argc, char **argv)
{
    int listenfd, connfd;
    struct sockaddr in servaddr;
    char buff[MAXLINE];
    tine t ticks;
    listenfd = Socket(AF_INET, SOCK_STPEAM,0);
    <---------1
    bzeros(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_IET;<<----------2
    servaddr.sin_addr.s_addr = hton(INDDR_ANY);
    servaddr.sin_port  =  htons(13);   /*daytime server
t*/
    Bind(listenfd,(SA* &servaddr sizeof(servaddr)));
    <---------3
    Listen(listenfd, LISTENQ);<-------4
    For( ; ;)
    {
    Connfd = Accept(listenfd, (SA *) NULL, NULL);
    <---------5
    ticks = time(NULL);
    snprintf(buff, sizeof(buff),
    "%.24s\r\n",ctime(&ticks));
    write(connfd, buff, strlen(buff));<-------6
    Close(conned)<----------7
    }
}
```

Above code clearly explain about the system calls used in server creation

- Create socket
- Initialize address information in socket address structure attributes.
- Associate the socket created with the address initialized.
- Service will start and will be listening "whether there is any client coming in with request for service?". We call socket as listening socket here.
- On arrival of request and completion of parameter negotiation connect request of client is accepted.
- Once request to connect is anticipated by accept, we can say that the connection is establish and client will start asking question which are read by server and the result of computation are written back to same socket.
- After client is done asking all questions it initiates passive close. After replying to all question server activates active close after which connection is physically terminated.

Types of Socket

There are four types of socket:

1. A stream socket.
2. A datagram socket.
3. A sequential packet socket.
4. A raw socket.

1. A Stream Socket:

- Provide full duplex, sequenced and reliable form of communication. It works just like a telephone conversation.
- The socket type is SOCK_STREAM, which, in the Internet domain, uses CP/IP protocol.

2. A Datagram Socket:

- This category socket also support full duplex communication. This is unreliable and connectionless (not wireless) form of communication that uses UDP/IP protocol.
- On receiver side it may collect the data different order from that of order sent by transmitter. This is just like push SMS services.

3. A Sequential Packet Socket:

- This category of socket provides a two-way, sequenced, reliable, connection oriented, for datagram of a fixed maximum length.
- The socket type is SOCK_SEQPACKET. No protocol for this type has been implemented for any protocol family.

4. A Raw Socket:

- Raw socket is an internet socket used for direct sending and receiving of IP packet without any protocol-specific transport layer formatting.
- These socket are usually datagram oriented, but their exact characteristics depend on the interface provided by the protocol.

3.8 DEADLOCKS

- When two trains approach each other on a single track, none of the trains can move as both are waiting for the other one to move on, this scenario depicts a deadlock.
- Definition of deadlock is, in multiprogramming environment, several processes may compete for a finite number of resources. A process requests resource; and if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes.
- In a common area where four roads meet, deadlock can occur as traffic of no road is moved until the other has moved, this situation also leads to deadlock.

3.9 SYSTEM MODEL

- A system consists of a finite number of resources to be distributed among a number of competing processes. The examples of resource types are memory space, CPU cycles, files and I/O devices. If a system has two CPUs, then the resource type CPU has two instances. A process must request a resource before using it and must release the resource after using it.
- The number of resources requested by the process may not exceed the total number of resources available in the system.
- A process may utilize a resource in only the following sequence :
  - **Request :** Process needs to request the resource, if it is available it will be allocated else process needs to wait.
  - **Use :** The process can operate on the resource.
  - **Release :** The process releases the resource.
- A set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set. The important tasks are resource gaining and releasing. The resource may be either physical resources (for example, printers, tape drives, memory space and CPU cycles) or logical resources (for example, files, semaphores and monitors).
- To illustrate a deadlock state, consider a system with three CDRW drives. Suppose each of three processes holds one of these CDRW drives. If each process now requests another driver then three processes will be in a deadlock state. Each is waiting for the event "CDRW is released," which can be caused only by one of the other waiting processes. This example illustrates deadlock involving the same resource type.
- Deadlocks may also involve different resource types. For example, consider a system with one printer and one DVD drive. Suppose that process P_i is holding the DVD and process P_j is holding the printer. If P_i requests the printer P_j requests the DVD drive, a deadlock occurs.

3.10 DEADLOCK CHARACTERIZATION

3.10.1 Necessary Conditions

A deadlock situation can arise if the following four conditions hold simultaneously in a system.

1. **Mutual Exclusion :** One resource must be held in a non sharable mode, only one process at a time can use the resource. If another process requests that resources, the requesting process needs to wait.
2. **Hold and Wait :** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by another processes.
3. **No Pre-Emption :** Resources cannot be pre empted that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
4. **Circular Wait :** A set $\{P_0, P_1, ..., P_n\}$ of waiting processes must exists such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_n, and P_n is waiting for a resource held by P_0.

3.11 RESOURCE-ALLOCATION GRAPH

- Deadlock can be described in terms of a directed graph called a resource-allocation graph, consists of a set of vertices V and a set of edges E. The set of vertices V is divided into two different types of nodes : $P = \{P_1, P_2, ..., P_n\}$, the set consisting of all the active processes in the system, and $R = \{R_1, R_2, ..., R_m\}$ the set consisting of all resource types in the system.

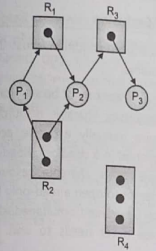


Fig. 3.4 : Resource – allocation graph

- A directed edge from process P_i to resource type R_j is denoted by $P_i \rightarrow R_j$; it signifies that process P_i has requested an instance of resource type R_j and is currently waiting for that resource.
- A directed edge from resource type R_j to process P_i is denoted by $R_j \rightarrow P_i$; it signifies that an instance of resource type R_j has been allocated to process P_i. A directed edge $P_i \rightarrow R_j$ is called a request edge; a directed edge $R_j \rightarrow P_i$ is called assignment edge.
- Pictorially, we represented each process P_i as circle and each resource type R_j as a rectangle. Since resource type R_j may have more than one instance, we represent each such instance as a dot within the rectangle. Note that a request edge points to only the rectangle R_j, where as an assignment edge must also designate one of the dots in the rectangle.
- When process P_i requests an instance of resource type R_j, a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is instantaneously transformed to an assignment edge. When the process no longer needs access to the resource, it releases the resource; as a result, the assignment edge is deleted.

Process States

- Process P_1 is holding an instance of resource type R_2 and is waiting for an instance of resources type R_1.
- Process P_2 is holding an instance of R_1 and an instance of R_2 and is waiting for an instance of R_3.
- If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred.
- If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.
- Suppose that process P_3 requests an instance of resource type R_2. Since no resource instance is currently available, a request edge $P_3 \rightarrow R_2$ is added to the graph (Fig. 3.5) at this point, two minimal cycles exist in the system.

$$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

$$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$$

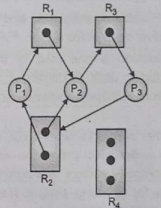


Fig. 3.5 : Resource – allocation graph with a deadlock

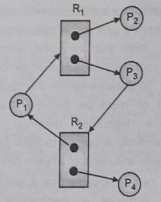


Fig. 3.6 : Resource – allocation graph with a cycle but no deadlock

- Processes P_1, P_2 and P_3 are deadlocked. Process P_2 is waiting for the resource R_3, which held by Process P_3. Process P_3 is waiting for either process P_1 or process P_2 to release resource R_2. In addition, process P_1 is waiting for process P_2 to release resource R_1.
- Now consider the resource-allocation graph in Fig. 3.6. In this example, we also have a cycle.

$$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

- However, there is no deadlock. Observe that process P_4 may release its instance of resource type R_2. That resource can then be allocated to P_3, breaking the cycle.
- In summary, if a resource-allocation graph does not have a cycle, then the system is not in deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state.

3.12 METHODS FOR HANDLING DEADLOCKS

There are three methods to handle deadlock :

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

- The third solution is one used by most operating systems, including UNIX and Windows; it is then up to the application developer to write program that handle deadlocks.
- To ensure that deadlocks never occur, the system can use either a deadlock prevention or a deadlock-avoidance scheme. Deadlock prevention provides a set of methods for ensuring that at least one of the necessary conditions cannot hold.
- Deadlock avoidance requires that the operating system be given in advance additional information concerning which resources a process will request and use during it's lifetime. With this additional knowledge, it can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.
- If a system neither ensures that a deadlock will never occur nor provides a mechanism for deadlock defection and recovery, then we may arrive at a situation where the system is in a deadlocked state yet has no way of recognizing what has happened. Then the system will stop functioning and will need to be restarted manually.

3.13 DEADLOCK PREVENTION

For a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

3.13.1 Mutual Exclusion

- In mutual-exclusion condition must hold for non-sharable resources.
- For example, a printer cannot be simultaneously shared by several processes. Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneously access to the file. A process never needs to wait for a sharable resource.
- In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non sharable.

3.13.2 Hold and Wait

- To ensure that the hold and wait condition never occurs in the system, we must guarantee that whenever a process requests a resource, it does not hold any other resources. One protocol that can be used requires each process to request and be allocated all it's resources before it begins execution.
- An alternative protocol allows a process to request resources only when it has none. A process may request some resources and use them. Before it can request any additional resources, however, it must release all the resources that it is currently allocated.
- To illustrate the difference between these two protocols, we consider a process that copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer. If all resources must be requested at the beginning of the process, then the process must initially request the DVD drive, disk file and printer.

3.13.3 No Pre-Emption

- The third necessary condition for deadlocks in that there be no pre-emption of resources that have already been allocated. To ensure that this condition does not hold, we can use the following protocol.
- If a process is holding some resources and requests another resource that cannot be immediately allocated to it, then all resources currently being held are pre empted. In other words, these resources are implicitly released.

- The pre-empted resources are added to the list of resources, for which the process is waiting. The process will be restarted only when it can regain its old resources, as well as new ones that it is requesting.

3.13.4 Circular Wait

- The fourth and final condition for deadlocks is the circular wait condition. One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.

3.14 DEADLOCK AVOIDANCE

Deadlock-prevention algorithms, prevent deadlocks by restraining how requests can be made. The restraints ensure that at least one of the necessary conditions for deadlocks cannot hold. Low device utilization and low throughput are the disadvantages by this approach.

3.14.1 Safe State

- A state is safe if the system can allocate resources to each process (up to it's maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence.
- A sequence of process $<P_1, P_2, ..., P_n>$ is a safe sequence for the current allocation state if, for each P_i, the resource request that P_i can still make can be satisfied by the currently available resources plus the resources held by all P_j with $j < i$. In this situation if the resources that P_i needs are not immediately available, then P_i can wait until all P_j have finished.
- When they have finished, P_i can obtain all of its needed resources, complete its designated task, return it's allocated resources, and terminate. When P_i terminates, P_i+1 can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be unsafe.
- A safe state is not a deadlocked state conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks, however (Fig. 3.7). An unsafe state may lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe (and deadlocked) states. In an unsafe state, the operating system cannot prevent processes from requesting resources such that a deadlock occurs :

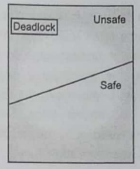


Fig. 3.7 : Safe, unsafe and deadlock state spaces

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

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

- A system can go from a safe state to an unsafe state. Suppose that, at time t_1, process P_2 requests and is allocated one more tape drive. The system is no longer in a safe state. At this point, only process P_1 can be allocated all its tape drives.
- When it returns them, the system will have only 4 available tape drives. Since process P_0 is allocated 5 tape drives but has a maximum of 10, it may request 5 more tape drives. Since they are unavailable, process P_0 must wait. Similarly, process P_2 may request an additional 6 tape drives and have to wait, resulting in a deadlock. Our mistake was in granting, the request from process P_2 for one more tape drive. If we had made P_2 wait until either of the other processes had finished and released it's resources, then we could have avoided the deadlock.

3.14.2 Resource – Allocation – Graph Algorithm

- If we have a resource allocation system with one instance of each resource type, a variant of the resource – allocation graph can be used for deadlock avoidance. In addition to the request and assignment edges already described, we introduce a new type of edge, called claim edge.

- A claim edge $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line. When process P_i requests resource R_j, the claim edge $P_i \rightarrow R_j$ is converted to request edge. Similarly, when a resource R_j is released by P_i, the assignment edge $R_j \rightarrow P_i$ is recovered to a claim edge $P_i \rightarrow R_j$.

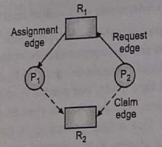


Fig. 3.8 : Resource – allocation graph for: deadlock avoidance

- We note that the resources must be claimed a prior in the system. That is, before process P_i starts executing, all it's claim edges must already appear in the resource – allocation graph. We can relax this condition by allowing a claim edge $P_i \rightarrow R_j$ to be added to the graph only if all the edges associated with process P_i are claim edges.
- Suppose that process, P_i requests resource R_j. The request can be granted only if converting the request edge $P_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow P_i$ does not result in the formation of a cycle in the resource allocation graph.
- If no cycle exists, then the allocation of the resources will leave the system in a safe state., If a cycle is found, then the allocation will put the system in an unsafe state. Therefore, process Pi will have to wait for it's requests to be satisfied.
- To illustrate this algorithm, we consider the resource – allocation graph of Fig. 3.8. Suppose that P_2 requests R_2. Although R_2 is currently free, we cannot allocate it to P_2, since this allocation will create a cycle in the graph (Fig. 3.9). A cycle indicates that the system is in an unsafe state. If P_1 requests R_2, and P_2 requests R_1, then a deadlock will occur.

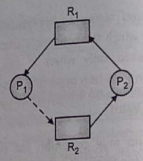


Fig. 3.9 : An unsafe state in a resource-allocation graph

Bankers Algorithm

Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resource types.

- **Available:** Vector of length m. If available [j] = k, there are k instances of resource type R_j available.
- **Max:** n x m matrix. If Max [i,j] = k, then process P_i may request at most k instances of resource type R_j
- **Allocation:** n x m matrix. If Allocation[i,j] = k then P_i is currently allocated k instances of R_j.
- **Need:** n x m matrix. If Need[i,j] = k, then P_i may need k more instances of R_j to complete its task. Need [i,j] = Max[i,j] – Allocation [i,j]

Safety Algorithm

1. Let Work and Finish be vectors of length m and n respectively.

   Initialize

   Work = Available

   Finish [i] = false for i - 1,3, ..., n.
2. Find an i such that both

   (a) Finish [i] = false

   (b) Need[i] <= Work

   If no such i exists, go to step 4.
3. Work = Work + Allocation[i]

   Finish[i] = true

   go to step 2.
4. If Finish [i] == true for all i, then the system is in a safe state

Resource-Request Algorithm

1. If Request[i]<= Need[i] go to step 2.

   Otherwise, raise error condition, since process has exceed to its maximum claim.

2. If Request[i]<= Available, go to step 3. Otherwise P_i must wait, since resources are not available.
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

   Available = Available – Request;

   Allocation[i] = Allocation[i] + Request[i];

   Need[i] = Need[i] – Request[i];

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

Examples on Banker's Algorithm

1. System consists of five processes (P_1, P_2, P_3, P_4, P_5) and three resources (R_1, R_2, R_3). Resource type R_1 has 10 instances, resource type R_2 has 5 instances and R_3 has 7 instances. The following snapshot of the system has been taken :

Process	Allocation			Max			Available		
	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3
P_1	0	1	0	7	5	3	3	3	2
P_2	2	0	0	3	2	2			
P_3	3	0	2	9	0	2			
P_4	2	1	1	2	2	2			
P_5	0	0	2	4	3	3			

Content of the matrix need is calculated as

Need = Max – Allocation

Process	Need		
	R_1	R_2	R_3
P_1	7	4	3
P_2	1	2	2
P_3	6	0	0
P_4	0	1	1
P_5	4	3	1

Currently the system is in safe state

Safe Sequence : Safe sequence is calculated as follows

1. Need of each process is compared with available.

   If $need_i \leq available_i$, then the resources are allocated to that process and process will release the resource.
2. If need is greater than available, next process need is taken for comparison.
3. In the above example, need of process P_1 is (7, 4, 3) and available is (3, 3, 2).

   need ≥ available → false

   So system will move for next process.
4. Need for process P_2 is (1, 2, 2) and available (3, 3, 2) so

   need ≤ available (work)

   (1, 2, 2) ≤ (3, 3, 2) = True

   then finish [i] = True

   Request of P_2 is granted and processes P_2 is release the resource to the system.

   Work : = Work + Allocation

   Work : = (3, 3, 2) + (2, 2, 0)

   : = (5, 3, 2)

   This procedure is continued for all processes.
5. Next process P_3 need (6, 0, 0) is compared with new available (5, 3, 2).

   Need > Available= False

   (6, 0, 0)> (5, 3, 2)
6. Process P_4 need (0, 1, 1) is compared with available (5, 3, 2).

   Need< Available

   (0, 1, 1) < (5, 3, 2)= True

   Available = Available + Allocation

   = (5, 3, 2) + (2, 1, 1)

   = (7, 4, 3) (new available)
7. Then process P_5 need (4, 3, 1) is compared with available (7, 4, 3).

   Need< Available

   (4, 3, 1) < (7, 4, 3)= True

   Available = Available + Allocation

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

   = (7, 4, 5) (New available)
8. One cycle is completed. Again system takes all remaining process in sequence. So process P_1 need (7, 4, 3) with new available (7, 4, 5)

   Need < available= True

   (7, 4, 3) < (7, 4, 5)

   Available = Available + Allocation

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

   = (7, 5, 5) (New available)

9. Process P_3 need is (6, 0, 0) is compared with new available (7, 5, 5).

∴ Need < Available = True

(6, 0, 0) < (7, 5, 5)= True

Available = Available + Allocation

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

= (10, 5, 7)

= (New available)

Safe sequence is < P_2 P_4 P_5 P_1 P_3 >

Example 3.5 : *Consider the following snapshot of a system.*

Process	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	1	5	2	0
P_1	1	0	0	0	1	7	5	0				
P_2	1	3	5	4	2	3	5	6				
P_3	0	6	3	2	0	6	5	2				
P_4	0	0	1	4	0	6	5	6				

Answer the following questions using Banker's Algorithm.

(a) What is the content of the matrix need ?

(b) Is the system in safe state ?

(c) If the request from process P_1 arrives for (0, 4, 2, 0) can the request be granted immediately ?

Solution :

(a) Content of the needed matrix is

Process	A	B	C	D
P_0	0	0	0	0
P_1	0	7	5	0
P_2	1	0	0	2
P_3	0	0	2	0
P_4	0	6	4	2

(b) System is in safe state because resources are available (1, 5, 2, 0).

(c) Request from process P_1 can be granted immediately request is (0, 4, 2, 0) and available resource is (1, 5, 2, 0).

Example 3.6 : *The operating system contains 3 resources, the number of instance of each resource type are 7, 7, 10. The current resource allocation state is as shown below.*

Process	Current Allocation			Maximum Need		
	R_1	R_2	R_3	R_1	R_2	R_3
P_1	2	2	3	3	6	8
P_2	2	0	3	4	3	3
P_3	1	2	4	3	4	4

(a) Is the current allocation in a safe state ?

(b) Can the request made by process P_1 (1, 1, 0) be granted ?

Solution :

(a) First find the available resource in the system

Available = Number of Instance

– Sum of allocation

Process	Current Allocation		
	R_1	R_2	R_3
P_1	2	2	3
P_2	2	0	3
P_3	1	2	4
	5	4	10

⇐ Sum

Available = (7, 7, 10) – (5, 4, 10)

Available resources= (2, 3, 0)

Content of need matrix is

Process	Need		
	R_1	R_2	R_3
P_1	1	4	5
P_2	2	3	0
P_3	2	2	0

Safe sequence < P_3, P_2, P_1 >

The system is in safe state.

(b) Process P_1 request (1, 1, 0) this request is less than need. Need for process P_1 is (1, 4, 5) available resource is (2, 3, 0) and request is (1, 1, 0)

∴ Request < Available

(1, 1, 0) < (2, 3, 0)

After allocating (1, 1, 0) to process P_1 the need become as follows.

Process	Need		
	R_1	R_2	R_3
P_1	0	3	5
P_2	2	3	0
P_3	2	2	0

And available resource is (1, 2, 0)

Need of any process is never satisfied after granting the process P_1 request (1, 1, 0). So the system will be blocked. Therefore request of process P_1 (1, 1, 0) cannot be granted.

Example 3.7 : *Consider following snapshot.*

Process	Allocation		Max		Available	
	R_1	R_2	R_1	R_2	R_1	R_2
P_1	1	2	4	2	1	1
P_2	0	1	1	2		
P_3	1	0	1	3		
P_4	2	0	3	2		

Whether the system is safe or unsafe ?

Solution :

First calculate need

Need = Max – Allocation

Process	Need	
	R_1	R_2
P_1	3	0
P_2	1	1
P_3	0	3
P_4	1	2

System is in safe with safe sequence < P_2, P_4, P_1, P_3 > system will complete it's operation in this sequence.

Example 3.8 : *Consider following snapshot*

Process	Allocation		Max		Available	
	R_1	R_2	R_1	R_2	R_1	R_2
P_1	7	2	9	5	2	1
P_2	1	3	2	6		
P_3	1	1	2	2		
P_4	3	0	5	0		

(a) Calculate content of need matrix

(b) System is safe or unsafe

Solution :

(a) Need = Max – Allocation

Process	Need	
	R_1	R_2
P_1	2	3
P_2	1	3
P_3	1	1
P_4	2	0

(b) System is unsafe sate. Resources are not available for processes P_1 & P_2 to complete their operation.

Weakness of Banker's Algorithm

- It requires that there be a fixed number of resources to allocate.
- The algorithm requires that users state their maximum needs (request) in advance.
- Number of users must remain fixed.
- The algorithm requires that the bankers grant all requests within a finite time.
- Algorithm requires that process returns all resources within a finite time

3.14.3 Advantages of Deadlock Avoidance

- No Pre-emption is necessary.
- Decisions are made dynamically.

3.15 DEADLOCK DETECTION

- If the system is not using any deadlock avoidance and deadlock prevention, then a deadlock does not limit resource access or restrict process actions.
- With deadlock defection request, resources are granted to processes whenever possible. Periodically, the operating system performs an algorithm that allows it to detect the circular wait condition. We discuss the algorithm for deadlock defection.

3.15.1 Single Instance of Each Resource Type

- If all resources have only a single instance, then deadlock detection algorithm that uses a variant of the resource allocation graph, called a wait for graph. Wait for graph is obtained from resource – allocation graph. Nodes of resource is removed and collapsing the appropriate edge. i.e. assignment and request edge.
- Fig. 3.10 shows the resource – allocation graph with corresponding wait for graph. The assumption made in the wait for graph is as follows :

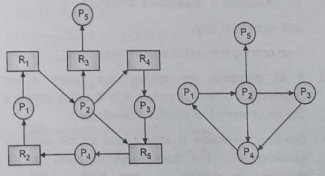


Fig. 3.10 : Resource – allocation graph with corresponding wait-for-graph

(a) An edge from P_i to P_j in a wait-for graph implies that process P_i is waiting for process P_j to release a resource that P_i needs.

(b) An edge $P_i \rightarrow P_j$ exists in a wait for graph if and only if the corresponding resource allocation graph contains two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$ for some resource R_q.

- In the Fig. 3.11 of resource – allocation graph, process P_1 is holding resource R_2 and requesting resource R_1 so this is shown in the following Fig. 3.11. In this process P_1 is request edge with process P_2. It is shown in Fig. 3.11.

Fig. 3.11 : Request edge

- If the wait for graph contains the cycle, then there is a deadlock. To detect deadlock, the system needs to maintain the wait for graph and periodically to invoke an algorithm that searches for a cycle in the graph.
- For detecting a cycle in the graph, algorithm requires (n^2) operation. Where (n) is the number of vertices in the graph. This method is suitable for single instance of resource.

3.15.2 Several Instances of Resource Type

In deadlock detection approaches, the resource allocator simply grants each request for an available resource. For checking for deadlock of the system, the algorithm is as follows.

1. Unmark all active processes from allocation, Max and Available in accordance with the system state.
2. Find an unmarked process i such that

   Maxi ≤ Available

   If found, mark process i, update Available

   Available = Available + Allocation

   and repeat this step.

   If no process is found, then go to next step.
3. If all processes are marked, the system is not deadlocked. Otherwise system is in deadlock state and the set of unmarked process is deadlocked.

- Deadlock detection is only a part of the deadlock handling task. The system applies break to the deadlock to reclaim resources held by blocked processes and to ensure that the affected processes can eventually be completed.

3.15.3 Advantages of Deadlock Detection

- Never delays process initiation.
- Facilitates online handling.

3.15.4 Disadvantages of Deadlock Detection

- Inherent pre-emption losses.
- If algorithm is invoked for every source request, this will incur a considerable overhead in computation time.

3.16 RECOVERY FROM DEADLOCK

Once deadlock has been detected, some strategy is needed for recovery. Following are the solutions to recover the system from deadlock.

1. Process termination.
2. Resource Pre-emption.

3.16.1 Process Termination

All deadlocked processes are aborted most of the operating system use this type of solution.

- Abort all deadlocked processes. This method clearly will break the deadlock cycle, but at greater expense.
- Abort one process at a time until the deadlock cycle is eliminated. This method incurs considerable overhead. The order in which processes are selected for abortion should be on the basic of some criterion of minimum cost.
- Many factors may affect which process is chosen, including
  - ➢ Least amount of processor time consumed so far.
  - ➢ Least amount of output produced so far.
  - ➢ Lowest priority.
  - ➢ Most estimated time remaining.
  - ➢ Least total resources allocated so far.

3.16.2 Resource Pre-Emption

If pre-emption is required to deal with deadlocks then three issues need to be addressed.

1. **Selecting a Victim :** Which resources and which processes are to be pre empted ?
2. **Rollback :** Backup each deadlocked process to some previously defined check point and restart all processes. This requires rollback and restart mechanisms are built in to the system.
3. **Starvation :** How do we ensure that starvation will not occur?

3.17 COMPARISON BETWEEN DEADLOCK DETECTION, PREVENTION AND AVOIDANCE

Parameters	Avoidance	Detection	Prevention
Resource Allocation Policy	Midway between that very liberal.		Conservative under commit resources.
Different Schemes	Manipulate to find at least one safe path.	Invoke periodically to test for deadlock.	Pre-emption resource ordering, requesting all resources at once.
Advantages	No pre-emption necessary.	Never delays process initiation.	No pre-emption necessary.
Disadvantages	Process can be blocked for long period.	Inherent pre-emption losses.	Delays process initiation.

3.18 INTEGRATED DEADLOCK STRATEGY

All the deadlock strategies have their own strengths and weakness. Instead of using one strategy for designing operating system, all the deadlock strategy are used according to the situation. Some of the approaches are :

- Group resources into a number of different resource classes.
- Within a resource class, use the algorithm that is most appropriate for that class.

Resources are classified as follows :

- Swappable space.
- Process resource.
- Main memory.
- Internal resource i. e. I/O channels.

Swappable space means blocks of memory on secondary storage for use in swapping process. Tape drive and files are process resources.

Assignable in pages or segments to processes are the main memory resources.

Example 3.9 : *Consider the following snapshot of a system*

Process	Allocation				Request				Available			
	R_1	R_2	R_3	R_4	R_1	R_2	R_3	R_4	R_1	R_2	R_3	R_4
P_1	0	0	1	2	0	0	1	2	1	5	2	0
P_2	1	0	0	0	1	7	5	0				
P_3	1	3	5	4	2	3	5	6				
P_4	0	6	3	2	0	6	5	2				
P_5	0	0	1	4	0	6	5	6				

Answer the following questions using Banker's algorithm.

(a) What is the content of 'need' matrix ?

(b) Is the system in a safe state ?

(c) If a request from P_2 arrives for (0, 4, 2, 0) can the request be granted immediately ?

Solution :

(a) Content of need matrix is

Process	Allocation			
	R_1	R_2	R_3	R_4
P_1	0	0	0	0
P_2	0	7	5	0
P_3	1	0	0	2
P_4	0	0	2	0
P_5	0	6	4	2

(b) Safe state

(i) Need of P_1 = (0, 0, 0, 0)

Available = (1, 5, 2, 0)

Need < Available

(Require resources are allocated to P_1)

∴ New available= Allocation + Available

= (0, 0, 1, 2) + (1, 5, 2, 0)

= (1, 5, 3, 2)

(ii) For P_2 process

Need = (0, 7, 5, 0)

Available = (1, 5, 3, 2)

Need > Available

(Request is not granted)

New available= Available = (1, 5, 3, 2)

(iii) For P_3 process

Need = (1, 0, 0, 2)

Available = (1, 5, 3, 2)

Need < Available

(Request is granted)

New available= Allocation + Available

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

= (2, 8, 8, 6)

(iv) For P_4 process

Need of P_4 = (0, 0, 2, 0)

Available = (2, 8, 8, 6)

Need < Available

New Available= (2, 14, 11, 8)

(v) Process of P_5

Need of P_5= (0, 6, 4, 2)

Available = (2, 14, 11, 8)

Need < Available

New available= (2, 14, 11, 8) + (0, 0, 1, 4)

= (2, 14, 12, 12)

(vi) Again for P_2 process

Need of P_2 = (0, 7, 5, 0)

Available = (2, 14, 12, 12)

Need < Available

(Request is granted)

New available= (1, 0, 0, 0) + (2, 14, 12, 12)

= (3, 14, 12, 12)

So the safe sequence is P_1, P_3, P_4, P_5, P_2

(c) Request from P_3= (0, 4, 2, 0)

Available = (1, 5, 2, 0)

After granting the request of P_2, available resource is (1, 1, 0, 0)

(i) Need of P_1 = (0, 0, 0, 0)

New available= (1, 1, 0, 0) + (0, 0, 1, 2)

= (1, 1, 1, 2)

(ii) P_2 need is greater than available

(iii) P_3 need is (1, 0, 0, 2)

Need < Available

New available= (1, 1, 1, 2) + (1, 3, 5, 4)

= (2, 4, 6, 6)

(iv) P_4 need is (0, 0, 2, 0)

Need < Available

New available= (2, 4, 6, 6) + (0, 6, 3, 2)

= (2, 10, 9, 8)

(v) P_5 need is (0, 6, 4, 2)

Available = (2, 10, 9, 8)

Need < Available

∴ New available = (2, 10, 9, 8) + (0, 0, 1, 4)

= (2, 10, 10, 12)

(vi) Again P_2 need (0, 7, 5, 0)

Need < Available

If a request from P_2 arrives for (0, 4, 2, 0) then the request is granted immediately.

Example 3.10 : *Apply the deadlock detection algorithm to the following data and show the results*

Available = [2, 1, 0, 0]

Request = 2 0 0 1 *Allocation = 0 0 1 0*

1 0 1 0 *2 0 0 1*

2 1 0 0 *0 1 2 0*

If a request [1, 1, 0, 0] comes from a process, will it be granted ? will the system still remain in same state ?

Solution :

Set W = (2, 1, 0, 0)

The request of process P_2 is less than W, so mark P_2 and set W = W + (2 0 0 1) = (4 1 0 1) Next P_3 process is marked and set

W = W + (0, 1 , 2, 0)

= (4 2 2 1)

Next P_1 process is marked and set

W = W + (2 1 0 0)

= (6 3 2 1).

All the processes are marked, so the processes are not deadlocked.

Example 3.11 : *Write a safety algorithm in the deadlock problem and state it's time complexity. Consider following snapshot of system at time T_0.*

Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	2	3	0
P_1	3	0	2	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			

Calculate the need matrix and state whether system is safe. If yes, justify with stepwise calculations. Give safe sequence.

Solution :

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

For safe sequence :

(a) Need of P_0 (7, 4, 3) is greater than available (2, 3, 0) select next process.

(b) Need of P_1 (0, 2, 0) is less than available (2, 3, 0)

New available= (2, 3, 0) + (3, 0, 2)

= (5, 3, 2)

(c) Need of P_2 (6, 0, 0) is greater than available (5, 3, 2) select next.

(d) Need of P_3 (0, 1, 1) is less than available (5, 3, 2)

New available = (5, 3, 2) + (2, 1, 1)

= (7, 4, 3)

(e) Need of P_4 (4, 3, 1) is less than available (7, 4, 3)

New available= (7, 4, 3) + (0, 0, 2)

= (7, 4, 5)

Again for remaining process i.e. P_0 and P_2, P_0 need is (7, 4, 3) and available is (7, 4, 5) so need is less than available

New available= (7, 4, 5) + (0, 1, 0)

= (7, 5, 5)

Need of P_2 (6, 0, 0) is less than available (7, 5, 5)

So New available = (7, 5, 5) + (3, 0 2) = (10, 5, 7)

The safe sequence is (P_1, P_3, P_4, P_0, P_2)

So system is safe.

Example 3.12 : *Consider following snapshot of system at time T_0.*

Process	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	2	1	0	0
P_1	2	0	0	0	2	7	5	0				
P_2	0	0	3	4	6	6	5	6				
P_3	2	3	5	4	4	3	5	6				
P_4	0	3	3	2	0	6	5	2				

(a) Calculate the need matrix

(b) State whether system is safe ? If yes, give safe sequence.

(c) For the system snapshot at time T_0, if a request from P_2 arrives for (0, 1, 0, 0) can the request be safely granted ?

(d) For the system snapshot at time T_0, If a request from P_3 arrives for (1, 0, 0, 0). Can the request be safely granted ?

Solution :

(a) Need matrix :

Process	A	B	C	D
P_0	0	0	0	0
P_1	0	7	5	0
P_2	6	6	2	2
P_3	2	0	0	2
P_4	0	3	2	0

(b) Safe sequence :

P_0 need (0, 0, 0) < Available (2, 1, 0, 0) so

New available = (2, 1, 0, 0) + (0, 0, 1, 2)

= (2, 1, 1, 2)

P_1 need (0, 7, 5, 0) > available (2, 1, 1, 2) so next process

P_2 need (6, 6, 2, 2) > Available (2, 1, 1, 2)

so next process

P_3 Need (2, 0, 0, 2)< Available (2, 1, 1, 2) so

New available = (2, 1, 1, 2) + (2, 3, 5, 4)

= (4, 4, 6, 6)

P_4 Need (0, 3, 2, 0)< Available (4, 4, 6, 6) so

New available = (4, 4, 6, 6) + (0, 3, 3, 2)

= (4, 7, 9, 8)

P_1 Need (0, 7, 5, 0) < Available (4, 7, 9, 8) so

New available = (4, 7, 9, 8) + (2, 0, 0, 0)

= (6, 7, 9, 8)

P_2 Need (6, 6, 2, 2) < Available (6, 7, 9, 8) so

New available = (6, 7, 9, 8) + (0, 0, 3, 4)

= (6, 7, 12, 12)

Safe sequence = P_0, P_3, P_4, P_1, P_2

(c) We assume that, request for P_2 is granted then

Process	Need Matrix				Available			
	A	B	C	D	A	C	B	D
P_0	0	0	0	0	2	0	0	0
P_1	0	7	5	0				
P_2	6	5	2	2				
P_3	2	0	0	2				
P_4	0	3	2	0				

Calculate safe sequence

Process	Need				Available				New available				T/F
P_0	0	0	0	0	2	0	0	0	2	0	1	2	T
P_1	0	7	5	0	2	0	1	2	2	0	1	2	F
P_2	6	5	2	2	2	0	1	2	2	0	1	2	F
P_3	2	0	0	2	2	0	1	2	4	3	6	6	T
P_4	0	3	2	0	4	3	6	6	4	6	9	8	T

Again for failure process

P_1	0	7	5	0	4	6	9	8	4	6	9	8	F
P_2	6	5	2	2	4	6	9	8	4	6	9	8	F

(d) Assume that, P_3 request is granted then

Process	Need				Available			
	A	B	C	D	A	C	B	D
P_0	0	0	0	0	1	1	0	0
P_1	0	7	5	0				
P_2	6	6	2	2				
P_3	1	0	0	2				
P_4	0	3	2	0				

Calculate safe sequence :

Process	Need				Available				New available				T/F
P_0	0	0	0	0	1	1	0	0	1	1	1	2	T
P_1	0	7	5	0	1	1	1	2	1	1	1	2	F
P_2	6	6	2	2	1	1	1	2	1	1	1	2	F
P_3	1	0	0	2	1	1	1	2	4	4	6	6	T
P_4	0	3	2	0	4	4	6	6	4	7	9	8	T

Again for P_1 and P_2

P_1	0	7	5	0	4	7	9	8	6	7	9	8	T
P_2	6	6	2	2	6	7	9	8	6	7	12	12	T

Safe sequence is $P_0\ P_3\ P_4\ P_1\ P_2$

So request is granted.

Example 3.13 : *Apply the deadlock detection algorithm for following example and show the result available [2 1 0 0].*

Request	Allocation
2001	0010
1010	2001
2100	0120

Solution : Set W = (2, 1, 0, 0)

The request of process P_2 is less than W, so mark P_2 and set

W = W + (2 0 0 1) = (4 1 0 1)

Next P_3 process is marked and set

W = W + (0 1 2 0)

= (4 2 2 1)

Next P_1 process is marked and set

W = W + (2 1 0 0)

= (6 3 2 1)

All the processes are marked, so the processes are not deadlocked.

Example 3.14 : *Consider the following state of the system. Determine if this system is in the safe state or not.*

	R_1	R_2	R_3		R_1	R_2	R_3		R_1	R_2	R_3
P_1	3	2	2	P_1	1	0	0	P_1	2	2	2
P_2	6	1	3	P_2	6	1	2	P_2	0	0	1
P_3	3	1	4	P_3	2	1	1	P_3	1	0	3
P_4	4	2	2	P_4	0	0	2	P_4	4	2	0
Claim Matrix C				Allocation Matrix A				C – A			

R_1	R_2	R_3		R_1	R_2	R_3
9	3	6		0	1	1
Resource Vector				Available Vector		
R				V		
Initial State						

Solution :

Need of P_1 > Than available vector.

Need of P_2 (0, 0, 1) is less available vector

New available = (0, 1, 1) + (6, 1, 2) = (6, 2, 3)

Need of P_3 = (1, 0, 3)

Available = (6, 2, 3)

Need ϕ is less than Available

New available = (6, 2, 3) + (2, 1, 1)

= (8, 3, 4)

Need of P_4 = (4, 2, 0)

Available = (8, 3, 4)

Need is less than Available

New available= (8, 3, 4) + (0, 0, 2) = (8, 3, 6)

Again need of P_1= (2, 3, 2)

Available = (8, 3, 6)

Need is less than vector available

Available vector will be (9, 3, 6) so the system is safe state.

3.19 DINING PHILOSOPHERS PROBLEM

In the dining philosophers problem, five philosophers live in a house, where a table is laid for them. The life of each philosopher consists principally of thinking and eating. Each philosopher needs two forks to eat. The eating arrangements are simple : a round table on which is set a large serving bowl of rice, five plates, one for each philosopher and five forks.

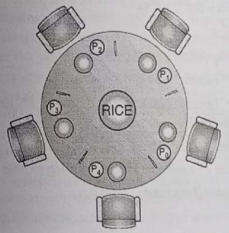


Fig. 3.9 : Dining arrangement for philosophers

- A philosopher wishing to eat goes to his or her assigned place at the table and, using the two forks on either side of the plate, takes and starts eating. The algorithm must satisfy mutual exclusion (no two philosophers can use the same fork at the same time) while avoiding deadlock and starvation.
- This problem may not seem important or relevant in itself. However, it does illustrate basic problems in deadlock and starvation. Furthermore, attempts to develop solutions reveal many of the difficulties in concurrent programming. The solution to this problem is giving by semaphores and monitors.

Solution Using Semaphores

- In this solution, each philosopher picks up first the fork on the left and then the fork on the right. After the philosopher is finished eating the two forks are replaced on the table. This solution, unfortunately leads to lock : If all of the philosophers are hungry at the same time, they all sit down, they all pick up the fork on their left, and they all reach out for the other fork, which is not there. In this situation all philosophers are deadlocked.

A First Solution to the Dining Philosophers Problem

```
/* program diningphilosophers */
semaphore fork [5] = {1};
int i ;
void philosopher (in + 1)
{
    while (true) {
        think ( );
        wait (fork [i] );
        wait (fork [(i + 1) mod s ]);
        eat ( );
        signal (fork [(i + 1) mod s]);
        signal (fork [i]);
    }
}
void main
{
    parbegin (philosopher (0), philosopher (1),
    philosopher (2), philosopher (3), philosopher (4);
    }
```

A Second Solution to Dining Philosophers Problem

```
/* program dinigphilosophers */
    semaphore fork [5] = {1};
    semaphore room = {4};
    int i ;
    void philosopher (int i)
    {
        while (true) {
        thinks ( );
        wait (room);
```

```
        wait (fork [i] );
        wait (fork [(it 1) mod 5]) ;
        eat ( ) ;
        signal (fork [ (it 1) mod 5] ) ;
        signal (fork [i]);
        signal (room)
        }
}
void main ( )
{
    parbegin (philosopher (0), philosopher (1),
    philosopher (2), philosopher (3), philosopher (4);
    }
```

Solution Using a Monitor

- A vector of five condition variables are used to enable a philosopher to wait for the availability of a fork. In addition, there is a Boolean vector that records the availability status of each fork (true means the fork is available). The monitor consists of two procedures. The get-forks procedure is used by a philosopher to take hold of his or her left and right forks.
- If either fork is unavailable, the philosopher process is queued on appropriate condition variable. This enables another philosopher process to enter the monitor. The release forks procedure is used to make two forks available. Monitor solution does not suffer from deadlock, because only one process at a time may be in the monitor.
- For example, the first philosopher process to enter the monitor is guaranteed that it can pick up the right fork after it picks up the left fork before the next philosopher to the right has a chance to hold its left fork, which is this philosophers right fork.

A Solution to the Dining Philosophers Problem Using a Monitor

```
Monitor dining - controller;
cond fork ready [5]; /* condition variable for
synchronization */
Boolean fork [5] = {true} ; /* availability status of each
fork */
void get-forks (int pid) /* pid is the philosopher id
number */
{
    int left = pid ;
    int right = (++ pid) % 5 ;
    /* grant the left fork */
    if (! fork (left) wait (fork ready [left]) ;
    fork (left) = false ;
    /* grant the right fork */
    if (! fork (right)
    (wait (fork ready (right)) ; /* queue on condition
    variable */
    fork(right) = false;
}
void release - forks (int pid)
{
    int left = pid ;
    int right = (++ pid) % 5 ;
    /* release the left fork */
    if (empty (fork ready [left]) /* no one is wait
    for this fork */
    fork (left) = true;
    else
    (signal fork ready [left]) ;
    /* release the right fork */
    if (empty (fork ready [right]) /* no one
    waiting for this fork */
    fork (right) = true ;
    else /* a waken a process waiting on this fork
    single (fork ready [right]) ;
}
void philosopher [k = 0 to 4] /* the five philosopher clie
*/
{
while (true)
{   <think> ;
    get-forks (k) ; /* client request two forks via
    monitor */
    < eat spaghetti >;
    release -forks (k); /* client releases forks via
    the monitor */
}
}
```


3.20 COMBINED APPROACH TO DEADLOCK HANDLING

Methods for Handling Deadlocks :

We can deal with the deadlock problem in one of three ways:

1. We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state.
2. We can allow the system to enter a deadlocked state, detect it, and recover.
3. We can ignore the problem altogether and pretend that deadlocks never occur in the system.

- To ensure that deadlocks never occur, the system can use either a deadlock prevention or a deadlock-avoidance scheme. Deadlock prevention provides a set of methods to ensure that at least one of the necessary conditions cannot hold.
- Deadlock avoidance requires that the operating system be given additional information in advance concerning which resources a process will request and use during its lifetime. With this additional knowledge, the operating system can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.
- If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may arise. In this environment, the system can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred and an algorithm to recover from the deadlock

EXERCISE

1. Explain critical selection problem.
2. Explain critical region with conditions of it.
3. What is synchronization? Explain hardware synchronization.
4. What is semaphores?
5. Explain classical problem of synchronization.
6. What is a monitor? What are the characteristics of it.
7. Differentiate between monitors and semaphores.
8. What is busy waiting with respect to process synchronization? Explain how semaphore reduces severity of this problem. Also define with example:
   (i) General semaphores
   (ii) Binary semaphores
   (iii) Strong semaphores
   (iv) Weak semaphores
9. What are the synchronization primitives used by the message passing system.
10. Write a short note on monitors and explain its structure.
11. Implement the reader-writer problem using semaphores and discuss how the critical section requirements are fulfilled.
12. What is deadlocks?
13. Explain system model.
14. What are the 4 conditions to produce deadlock?
15. Explain resource allocation graph.
16. Explain methods of handling deadlock.
17. Explain in detail deadlock prevention.
18. Write a short note on deadlock avoidance.
19. Describe 2 methods for deadlock recovery
20. Explain deadlock detection.
21. Comparison between deadlock detection, prevention and avoidance.
22. Explain combined approach to deadlock handling.
23. Apply the deadlock detection algorithm to the following data and show the result.

```
Available  = [2,1,0,0]
Request  = 2 0 0 1          Allocation = 0 0 1 0
           1 0 1 0                       2 0
                                         0 1
           2 1 0 0                       0 1 2 0
```

If a request [1 1 0 0] comes from a process will it be granted? Will the system still remain in safe state?

24. Write a safety algorithm in the deadlock problem and state its time complexity, consider following snap shot of system at time T_0

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

Calculate the need matrix and state whether system is safe. If yes, justify with steps calculations. Give safe sequence.

✠ ✠ ✠

Unit IV

MEMORY MANAGEMENT

4.1 BACKGROUND

Memory management deals with managing primary memory. In a multiprogramming system, the user part of memory must be further subdivided to accommodate multiple processes. The task of subdivision is carried out dynamically by the operating system and is known as memory management.

4.1.1 Functions of Memory Management

- To keep track of memory location as to whether the memory location is allocated or free.
- Determining allocation policy for memory.
- Memory allocation technique.
- Deallocation technique and policy. A process may release the allocated memory. Central processing unit fetches instructions from memory according to the value of program counter. These instructions may cause additional loading from and storing to specific memory address.

4.2 MEMORY MANAGEMENT REQUIREMENTS

Memory management requires five parameters :

1. Relocation
2. Protection
3. Sharing
4. Logical organization
5. Physical organization

1. Relocation

- If user knows an address of the process, user can fetch its contents. For a process residing in the main memory, set the program counter to an absolute address of it's first instruction and thus initiates its run.
- Process can load only with absolute address for instructions and data, only when those specific addresses are free in main memory. It is not possible for the user to know in advance which other programs will be resident in main memory at the time of execution of a program.
- For instance, user cannot load a process, if some other processes are currently occupying that area which is needed by process. This may happen even though there is enough space in the memory. To avoid such situation, processes and generated to be relocatable.
- Fig. 4.1 shows the relocation concept process which occupies a continuous region of main memory. The operating system will need to know the location of process control information and of the execution stack as well as the entry point to begin execution of the program for this process.
- All the addresses in the process are relative to the start address. So, user can allocate any area in the memory to load this process. Process instruction, data, PCB and any other data structure required by the process can be accessed easily if the addresses are relative.

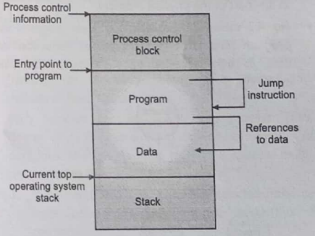


Fig. 4.1 : Relocation

2. Protection

- Memory protection is used to avoid interference between programs existing in memory. Normally, a user process cannot access any portion of the operating system, either program or data.
- A program in one process usually cannot branch to an instruction in another process. Implementation of memory protection requires support from a machine architecture in the form of memory bound registers. This is because operating system cannot anticipate all the memory references that a program will make.

3. Sharing

- Protection mechanism must allow several processes to access the same portion of main memory. Processes those are co-operating to same data structure.
- The memory management system must therefore allow controlled access to shared areas of memory without compromising essential protection.

4.2.1 Address Binding

- Programs and data stores on a secondary storage disk as a binary executable file. For executing the program, it is brought into the memory and placed within a process.
- The collection of processes on the disk that is waiting to be brought into memory for execution forms the input queue.
- Input queue is used for selecting one of the process for execution and loads into memory. As a process is executed, it's access data from memory and process will terminate. The memory space become free. Most of the operating system allows the user process to reside in any part of the physical memory.
- Fig. 4.2 shows the processing of user program. User program consists of various types of instructions and data. It also accesses various library files while executing.

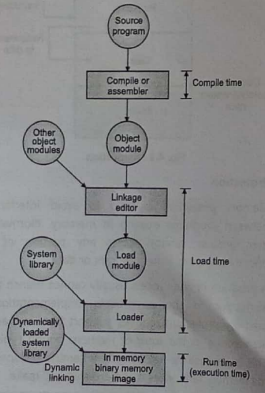


Fig. 4.2 : Different steps for processing user program

- Source program uses symbolic address. A compiler will bind these symbolic addresses to relocatable addresses.
- Loader will in turn bind these relocatable addresses to absolute addresses. Each binding is a mapping from one address space to another.
- Binding of instruction or data memory addresses can be done at any step along the way.
  - ➢ Compile time
  - ➢ Load time
  - ➢ Execution time

1. Compile Time

Process will reside in memory at compile time. It generates absolute code at compile time binding MS-DOS, com format programs are absolute code bound at compile time. After compiling of the process, if the memory location is changed, then it is necessary that the code must be recompiled.

2. Load Time

Compiler must generate relocatable code after compile time. In this situation, final binding is delayed until load time. If the starting address changes, it needs only to reload the user code to incorporate this changed value.

3. Execution Time

Process may be moved from one memory segment to another memory segment in execution then binding must be delayed until run time. Special hardware must be available for this scheme to work. Most general purpose operating system uses execution time binding method.

4.2.2 Logical v/s Physical Address Space

- Logical address is generated by CPU, physical address is the address of main memory and it is loaded into the memory address register.
- Compile-time and load-time address binding methods generate some logical address and physical address.
- Execution time address-binding scheme results in different logical and physical address.
- Logical address space set of all logical addresses generated by program is a logical address space.

- Physical address space is a set of all physical addresses corresponding to these logical addresses.
- Memory management unit mapping at run-time from virtual to physical address is done by a hardware device. This hardware device is MMU.
- Relocation register is also called base register.
- Value of the relocation register is added to every address generated by a user process at the time it is sent to memory.

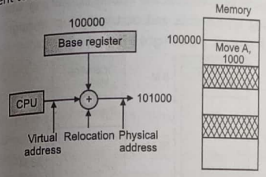


Fig. 4.3 : Dynamic relocation

- Fig. 4.3 shows the dynamic relocation. Dynamic relocation implies that mapping from the virtual address space to the physical address space is performed at run-time, usually with some network assistance. MS-DOS operating system running on the Intel 80 × 86 family of processors use four relocation register 8 when loading and running processes.
- Relocation is performed by hardware and is invisible to user. Dynamic relocation makes it possible to move a particularly executed process from one area of memory into another without affected.

4.3 CONTIGUOUS MEMORY ALLOCATION

- The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate the part of the main memory in the most efficient way possible. This section explains one common method, contiguous memory allocation.
- The memory is usually divided into two partitions : one for the resident operating system and one for the user processes. We can place the operating system in either low memory or high memory. The major factor affecting this decision is the location of the interrupt vector.
- Since the interrupt vector is often in low memory, programmers usually place the operating system in low memory as well. Thus, in this text, we discuss only the situation where the operating system resides in low memory. The development of the other situation is similar.

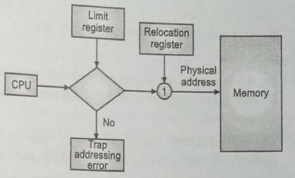


Fig. 4.4 : Hardware support for relocation and limit registers

- We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory. In this contiguous memory allocation, each process is contained in a single contiguous section of memory.

4.4 MEMORY PARTITIONING

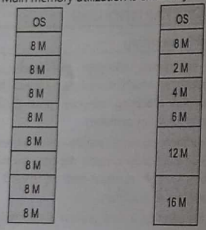
4.4.1 Fixed Partitioning

- Single contiguous allocation is simple memory management scheme that requires no special hardware features. For allocating memory, it is divided into number of fixed size partitions.
- Each partition contains exactly one process. Thus, degree of multi-programming depends on number of partitions. Multiple partition method is used in multi-programming environment.
  1. If the partition is free, process is selected from the input queue and is loaded into the free partition of memory.
  2. When the process terminates, the memory partitions become available for another process.
- This type of memory allocation is no longer in use. Batch operating system uses the fixed-size partition scheme. The operating system keeps the record of memory allocation and deallocation in the form of table. Initially, all the memory is available for user processes.
- When the process arrives in the system and needs memory, operating system searches for large hole for this process. Hole is one large block of free available memory. If any free hole is found, process is allocated to the free hole of memory as is needed.

- After allocating number of holes for the processes, a set of various size holes is scattered throughout memory at any given time when a process arrives and searches for (memory) set of holes. But the holes must be large enough to accommodate the process.

Partition Sizes

- Fig. 4.5 (a) and (b) shows examples of two alternatives for fixed partitioning. Possibility is to make use of equal size partitions.
- Any process whose size is less than or equal to the partition size can be loaded into any available partition.
- If all memory partitions are full and no process is in the ready or running state, the operating system can swap a process out of any of the partitions and load in another process.
- There are two difficulties with the use of equal size fixed partitions :
  1. A program may be too big to fit into a partition.
  2. Main memory utilization is extremely inefficient.



(a) Equal size partitions (b) Unequal size partitions

Fig. 4.5 : Example of fixed partitioning of 64 M byte memory

Advantages :

- Simple to implement
- Does not require expertise to understand and use such system.
- Less overhead

Disadvantages :

- Memory is not fully utilized.
- Poor utilization of processors.
- Users process (job) being limited to the size of available main memory.

4.4.2 Dynamic Partitioning

Memory partitions are of variable length and number. When process is brought into main memory, it is allocated exactly as much memory as it requires and no more.

Fig. 4.6 (a) to (d) shows the effect of dynamic memory partitioning.

Initially, main memory is empty, except for the operating system. (Refer Fig. 4.6 (a)).

The first three processes are loaded in, starting where the operating system ends and occupying just enough space for each process. (Refer Fig. 4.6 (b), (c), (d))

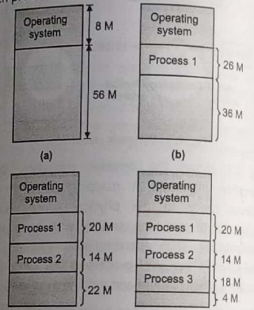


Fig. 4.6

The operating system swaps out process 2 (Refer Fig. 4.6 (a)), which leaves sufficient room to load a new process process 4 (Refer Fig. 4.6 (b)). Process 4 is smaller than process 2, another small hole is created. A point is reached at which none of the processes in main memory is ready but process 2, in the ready suspend state, is available.

Because there is insufficient room in memory for process 2 the operating system swaps process 1 out (refer Fig. 4.6 (c)) and swaps process 2 back in (refer Fig. 4.6 (d)).

4.5 FRAGMENTATION

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

Fragmentation is of Two Types :

S.N.	Fragmentation	Description
1.	Internal fragmentation	Memory block assigned to process is bigger. Some portion of memory is left unused as it cannot be used by another process.
2.	External fragmentation	Total memory space is enough to satisfy a request or to reside a process in it, but it is not continuous so it cannot be used.

External fragmentation can be reduced by compaction or shuffle memory contents to place all free memory together in one large block. To make compaction feasible, relocation should be dynamic.

Internal Fragmentation

- Internal fragmentation occurs when the memory allocator leaves extra space empty inside of a block of memory that has been allocated for a client. This usually happens because the processor's design stipulates that memory must be cut into blocks of certain sizes for example, blocks may be required to be evenly be divided by four, eight or 16 bytes.
- When this occurs, a client that needs 57 bytes of memory, for example, may be allocated a block that contains 60 bytes, or even 64.
- The extra bytes that the client doesn't need go to waste and over time these tiny chunks of unused memory can build up and create large quantities of memory that can't be put to use by the allocator. Because all of these useless bytes are inside larger memory blocks, the fragmentation is considered internal.

External Fragmentation

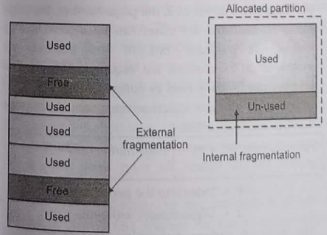


Fig. 4.7 : External and internal fragmentation

- External fragmentation happens when the memory allocator leaves sections of unused memory blocks between portions of allocated memory. For example, if several memory blocks are allocated in a continuous line but one of the middle blocks in the line is freed (perhaps because the process that was using that block of memory stopped running), the free block is fragmented. The block is still available for use by the allocator later if there's a need for memory that fits in that block, but the block is now unusable for larger memory needs.
- It cannot be lumped back in with the total free memory available to the system, as total memory must be continuous for it to be useable for larger tasks. In this way, entire sections of free memory can end up isolated from the whole that are often too small for significant use, which creates an overall reduction of free memory that over time can lead to a lack of available memory for key tasks. Both the fragmentation can be shown by above Fig. 4.7.

4.6 COMPACTION

- One technique for overcoming external fragmentation is compaction. From time to time, the operating system shifts the processes so that they are continuous and all the free memory is together in one block.
- For example, in Fig. 4.8, compaction results in a block of free (hole) memory of length 660 K. This may well be sufficient to load in an additional process. Three holes of size look, 300 K and 260 K can be compacted into hole of size 660 K. Compaction is not always possible. The difficulty with compaction is that it is a time consuming procedure, wasteful of processor time.

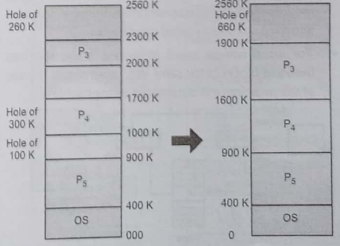


Fig. 4.8 : Compaction

- Compaction implies the need for a dynamic relocation capability. If relocation is static and is done at assembly or load time, compaction cannot be done. Relocation is done at execution time, then compaction is possible. Cost is major factor for compaction.
- The simplest algorithm of compaction is, the processes move towards one side of memory and holes moves in the other direction of memory. It produces one large hole of available memory. But cost of this scheme can be more.
- Fig. 4.9 shows the other possible combination of compaction. Swapping can also be combined with compaction. The compaction process is completed by updated free list. If swapping is already part of the system, the additional code for compaction may be minimal.

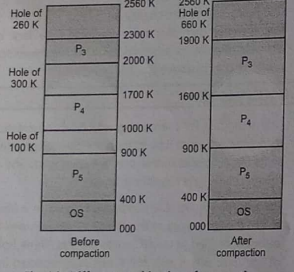


Fig. 4.9 : Different combination of compaction

4.7 PAGING

- Paging is a memory management scheme. Paging reduces the external fragmentation. Memory is divided into fixed size blocks called page frames.
- The virtual address space of process is also split into fixed size blocks of the same size, called pages. The size of the page frame is determined by the hardware.

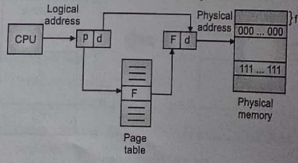


Fig. 4.10 : Paging hardware

- When a process is loaded into memory, the operating system loads each page into an unused page frame. The page frames used need not be continuous. Operating system maintains the page table for each process. The page table shows the frame location for each page of the process.
- CPU generates the logical address and it consists of two parts : a page number (P) and page offset (d). In case of simple partition, a logical address is the location of a word relative to the beginning of the program, the processor that into a physical address.
- Page table stores the number of the page frame allocated for each page. The page number is used as index into a page table. 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. Paging model of memory is shown in Fig. 4.11.

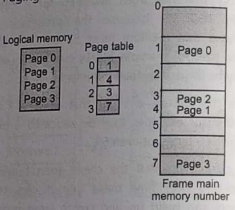


Fig. 4.11 : Paging model

- The size of page is typically a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture. Page size is always a power of 2 because all addresses are binary and are divided into page or page frame number and offset. By making the page size a power of 2, the page number, the page frame number and the offset can be determined by looking at particular bits in the address, no mathematical calculations are required. The physical address can be generated by concatenating the offset to the frame number logical address consists of page number and page offset.

Page number (P) = m – n	Page number (d) = n

P = Index into the page table.

d = Displacement within the page.

- The logic of address transaction process in paged system is shown in the following Fig. 4.12. For example,

the virtual address is 30200H. This virtual address is split by hardware into the page number and offset within that page.

- High order 12 bit is used as page number i.e. 003H and lower order 12 bit is used for the offset. i.e. (200H) page number is used to index the page table and to obtain the corresponding physical frame number (FFFH). This value is then concatenated with the offset to produce the physical address (FFF200H) which is used to reference the target item in memory.

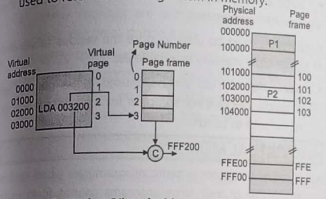


Fig. 4.12 : Paging (Virtual address to physical address)

- The operating system keeps track of the status of each page frame by means of a physical memory map that may be structured as a static table. The logical address space may be same size as the physical address space or it may be smaller. If logical address is small, it prevents one process.
- From monolizing all of the memory. If the logical address space is larger than the physical address space, all pages could not be resident in physical memory. Simple paging has no capability for dealing with reference to pages that are not in memory.

4.8 PAGE ALLOCATION: HARDWARE SUPPORT FOR PAGING

A hardware mechanism is needed to perform the mapping from each instruction's effective address to the appropriate physical memory location. There must be separate register for each page. These may be actual high speed registers or a special area of physical memory.

4.8.1 Translation Look A Side Buffer

- Problem with paging is that, extra memory reference to access translation table can slow programs down by a factor of two or three. Too many entries in translation table to keep team all loaded in fast processor memory.
- The standard solution to this problem is to use a special, small fast lookup hardware cache, called translation look a side buffer. The TLB is associate, high-speed memory. A Translation buffer is used to store a few of the translation table entries. It is very fast only for a small number of entries on each memory references.

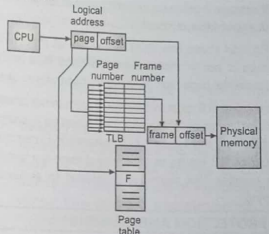


Fig. 4.13 : Paging with TLB

1. First ask TLB if it knows about the page. If so, the reference proceeds fast.
2. If TLB has no information for page, must go through page and segment table to get information. Reference takes a long time, but gives the info for this page to TLB so it will know it for next reference.

Fig. 4.13 shows the TLB. The greatest performance improvement is achieved when the associative memory is significantly faster than the normal page table lookup and the hit ratio is high. The hit ratio is the ratio between accesses that find a match in the associative memory and those that do not.

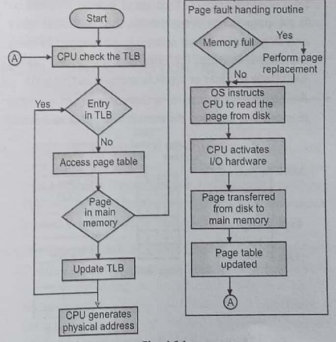


Fig. 4.14

Disadvantages of TLB :

- If two pages use the same entry of the memory, only one of them can be remembered at once.
- If process is referencing both pages at same time, TLB does not work very well.
- Fig. 4.14 shows flow chart for use of TLB. If the desired page is not in main memory, a page fault interrupt causes the page fault handling routine to be invoked.

Each entry in TLB must include the page number as well as the complete page table entry. The processor is equipped with hardware that allows it to simultaneously interrogate a number of TLB entries to determine if there is a match on page number. This technique is referred to as associate mapping.

4.9 PROTECTION AND SHARING

- Protection bits are used with each page frame for protecting memory in paging, for example, protection bits (i.e. access bits) may allow read only, execute-only, or other restricted forms of access.
- Every reference physical address is being computed, the protection bits can be checked to verify that no writes are being able to read-only page.
- Specification of access rights in paging system is useful for pages shared by several processes, but it is of much less value inside the boundaries of a given address space one more bit is generally attached to each entry in the page table, a valid-invalid bit.
- When valid bit is set, then the page is in the process logical address space. Thus, page is legal. If invalid bit is set, page is not in the process logical address space and illegal page. Illegal addresses are mapped by using the valid-invalid bit. The operating system sets bits for each page to allow or disallow accesses to that page.
- A single physical copy of a shared page can be easily mapped into as many distinct address space as desired.

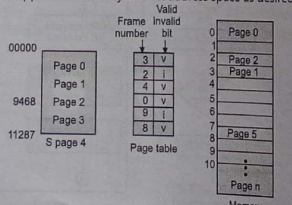


Fig. 4.15 : Valid or invalid bit in a page table

4.9.1 Advantages of Paging

- Paging eliminates fragmentation.
- Support higher degree of multiprogramming.
- Paging increases memory and processor utilization.
- Compaction overhead required for the relocated partition scheme is also eliminated.

4.9.2 Disadvantages of Paging

- Page address mapping hardware usually increases the cost of the computer.
- Memory must be used to store the various tables like page table, memory map table etc.
- Some memory will still be unused if the number of available blocks is not sufficient for the address spaces of the jobs to be run.

4.10 VIRTUAL MEMORY

- In many computer systems, programmers often realize that some of their large programs cannot fit in main memory for execution. Even if there is enough main memory for one program, the main memory may be shared with other users, causing any one program to occupy some fraction of memory which may not be sufficient for the program to execute. The usual solution is to introduce management schemes that intelligently allocate portions of memory to users as necessary for the efficient running of their programs. The use of virtual memory is to achieve this goal.
- Virtual memory allows execution of partially loaded process. The ability to execute a partially loaded process is also advantageous from the operating systems point of view. Fig. 4.16 shows the virtual memory with relation to physical memory. In fact, an examination of real programs shows us that, in many cases, the entire program is not needed.

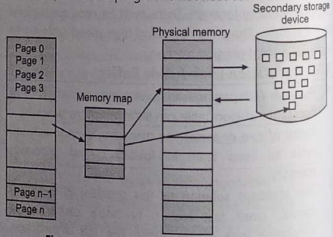


Fig. 4.16 : Virtual memory with other memory

- Following are the situations, when entire program is not required to load fully.
  - User written error handling routines are used only when an error occurs in the data or computation.
  - Certain options and features of a program may be used rarely.
  - Many tables are assigned a fixed amount of address space even though only a small amount of the table is actually used.
  - Many routines are commonly used at mutually exclusive times during a run.
- The ability to execute a program that is only partially in memory would confer many benefits.
  - Less number of I/O would be needed to load or swap each user program into memory.
  - A program would no longer be constrained by the amount of physical memory that is available.
  - Each user program could take less physical memory, more programs could be run at the same time, with a corresponding increase in CPU utilization and through put.
- Virtual memory makes the task of programming much easier. Virtual memory is commonly implemented by demand paging. Virtual memory mechanism bridges the size and speed gaps between the main memory and secondary storage and is usually implemented in part by software technique. Fig. 4.17 shows virtual memory organization.

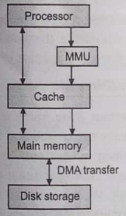


Fig. 4.17 : Virtual memory organization

Benefit of Virtual Memory

- For execution of a program only part of the program needs to be in memory. This is how we can project logical address space to be much larger than actual address space.
- Inter process sharing is possible since read only pages can be shared without any hindrance.
- Usage of data structure like arrays may lead to loss of memory since it may be underutilized. There may exist a program which is of seldom use, such occupants can be swapped out easily making a space for other critical processes maintaining smooth functioning of the system.

Working of Virtual Memory in Brief :

- We can find two categories of program in working memory.
  - **Category 1:** Program which manages the operations of the system. To be more precise, these are programs that belongs to OS itself (Kernel). We cannot imagine of a working system without these programs. Operating system brings life into the dead machine.
  - **Category 2:** Programs which belong to the user. As we have discussed, any application/program that user wants to execute has to brought in main memory for execution.
- Whenever data address required for further execution, an interrupt is generated (exception is terms of software) and process gets blocked in waiting for input of desired data(page).
- Using page replacement algorithm, desired page is replaced by another page which is of no use at that moment or which is not required to be in memory for some more time, in the main memory.
- Replaced page is made available to the blocked process and thus process will resume its operation further.

4.11 HARDWARE AND CONTROL STRUCTURES

4.11.1 Locality of Reference

- To prevent Thrashing, we must provide a process with as many frames as it needs. But how do we know how many frames it "needs"? There are several techniques. The working-set strategy starts by looking at how a process is actually using frames.
- This approach defines the locality of process execution. The locality model states that, as a process executes, it moves from locality to locality. A locality is a set of pages that are actively used together. A program is generally composed of several different localities, which may overlap.

- For example, when a function is called, it defines a new locality. In this locality, memory references are made to the instructions of the function call, its local variables, and a subset of the global variables. When we exit the function, the process leaves this locality, since the local variables and instructions of the function are no longer in active use.
- We may return to this locality later. Thus, we see that localities are defined by the program structure and its data structures. The locality model states that all programs will exhibit this basic memory reference structure. Suppose we allocate enough frames to a process to accommodate its current locality.
- It will fault for the pages in its locality until all these pages are in memory; then, it will not fault again until it changes localities. If we do not allocate enough frames to accommodate the size of the current locality, the process will thrash, since it cannot keep in memory all the pages that it is actively using.

4.11.2 Page Fault

Page fault is a type of exception raised by computer hardware when a running program accesses a memory page that is not currently mapped by the Memory Management Unit (MMU) into the virtual address space of a process.

Steps in Handling a Page Fault : The Fig. 4.18 shows the steps in handling the page fault.

1. Check an internal table for this process, to determine whether the reference was a valid or invalid memory access.
2. If the reference was invalid, terminate the process if it was valid, but not yet brought in that page, now page it in.
3. Find tree frame
4. Schedule a disk operating to read the desired page into the newly allocated frame.
5. When disk read is complete, modify the internal table kept with process and page table to indicate that the page is now in memory.
6. Restart the instruction that was interrupted by; illegal address trap. The process can now access the page as though it had always been in memory.

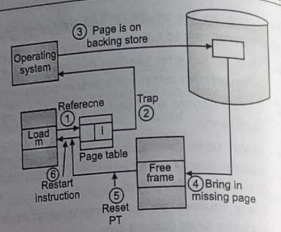


Fig. 4.18

4.11.3 Working Set

- The working set model is based on the assumption of locality. This model uses a parameter Δ to define the working set window. The idea is to examine the most recent page references. The set of pages in the most recent page references is the working set.
- If a page is in active use, it will be in the working set. If it is no longer being used, it will drop from the working set time units after its last reference. Thus, the working set is an approximation of the program's locality. For example, given the sequence of memory references, if Δ = 10 memory references, then the working set at time t_1 is {1, 2, 5, 6, 7}.
- By time t_2, the working set has changed to {3, 4}. The accuracy of the working set depends on the selection of Δ. If is too small, it will not encompass the entire locality; if is too large, it may overlap several localities. In the extreme, if is infinite, the working set is the set of pages touched during the process execution.

4.11.4 Dirty Page/Dirty Bit

- In page replacement, if no frame is free, we find one that is not currently being used and free it. We can free a frame by writing its contents to swap space and changing the page table to indicate that the page is no longer in memory. We can now use the freed frame to hold the page for which the process faulted.
- If no frames are free, two page transfers (one out and one in) are required. This situation effectively doubles the page-fault service time and increases the effective access time accordingly. We can reduce this overhead by using a modify bit or dirty bit.
- When this scheme is used, each page or frame has a modify bit associated with it in the hardware. The modify bit for a page is set by the hardware whenever any word or byte in the page is written into, indicating that the page has been modified.

- When we select a page for replacement, we examine it's modify bit. If the bit is set, we know that the page has been modified since it was read in from the disk. In this case, we must write the page to the disk. If the modify bit is not set, however, the page has not been modified since it was read into memory.
- In this case, we need not write the memory page to the disk: it is already there. This technique also applies to read-only pages (for example, pages of binary code).
- Such pages cannot be modified; thus, they may be discarded when desired. This scheme can significantly reduce the time required to service a page fault, since it reduces I/O time by one-half if the page has not been modified.

4.12 DEMAND PAGING

- A demand paging is similar to a paging system with swapping. With demand paging, a page is brought into main memory only when a reference is made to a location on that page. Lazy swapper concept is used in demand paging. A lazy swapper never swaps a page into a memory unless that page will be needed.
- Demand paging combines the features of simple paging and overlaying to implement virtual memory. In this, such page of program is stored continuously in the paging swap space on a secondary storage as locations continuously in the paging swap space as secondary storage.
- As locations in pages are referenced, the pages are copied into memory page frame. One the page is in the memory, it is accessed as in simple paging. Fig. 4.19 shows transfer of page from secondary storage to main memory.

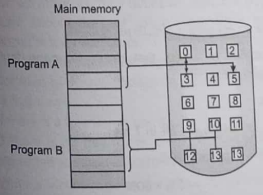


Fig. 4.19 : Transfer of page

- Some form of hardware support is required to distinguish between those pages that are in memory and those pages that are on the disk. The valid-invalid bit scheme can be used for this purpose. Each entry in the page table has at a minimum two fields. Page frame and valid-invalid bit.
- When valid bit is set, then the associated page is legal and present in the memory. Whenever a virtual address is generated, the memory management hardware extracts the page number from the address and appropriate entry in the page table is accessed. The valid-invalid bit is checked. If page is not in memory, a page fault occurs, transferring control to the page fault routine in the operating series.

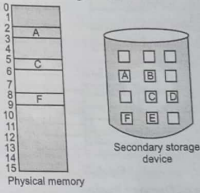


Fig. 4.20

- When the running process experiences a page fault, it must be suspended until the missing page is brought into main memory. The disk address of faulted page is usually provided in the File Map Table (FMT). This table is parallel to the page map table. Thus, when processing a page fault, the operating system uses the virtual page number provided by mapping hardware to index the FMT and to obtain the related disk address
- For convenience and mapping speed some systems place the disk address of each out of memory page into the corresponding PMT entry. This is possible because such entries are otherwise invalid for main memory mapping purpose.

Hardware Support

The hardware to support demand paging is the same as hardware for paging and swapping.

- **Page Table :** Page table has the ability to mark an entry invalid through a valid-invalid bit.
- **Secondary Memory :** This memory holds those pages that are not present in main memory. It is usually high speed disk.

Software Algorithm

- Demand page memory management provides tremendous flexibility for operating system. It must interact with information management to access and store copies of jobs address spaces as secondary storage. File map table is used to store the information regarding a file. FMT is not used by hardware. It is usually accessed by software. A possible format and use of file map table is shown in Fig. 4.21.

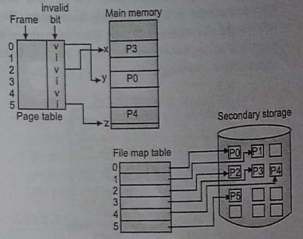


Fig. 4.21

Performance of Demand Paging

Demand paging can have a significant effect on the performance of a computer system. Let us calculate effective access time for a demand paged memory. Let P be the probability of c page fault. ($0 \le P \le 1$)

Effective access time = $(1 - P) \times ma + P \times$ Page fault time

where,

P = Page fault

ma = Memory access time

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

Advantages of Demand Paging

- Large virtual memory
- More efficient use of memory
- Unconstrained multiprogramming. There is no limit on degree of multiprogramming.

Disadvantages of Demand Paging

- Number of tables and amount of processor over head for handling page interrupts are greater than in the case of the simple paged management techniques.
- Due to lack of an explicit constraints on a jobs address space size.

Table 4.1 : Comparisons of Demand Paging with Segmentation

Sr. No.	Segmentation	Demand Paging
1.	Segments may of different size.	Pages are of same size.
2.	Segment can be shared.	Pages cannot be shared.
3.	It allows dynamic growth of segments.	Page size is fixed.
4.	Segment map table indicates the address of each segment in memory.	Page map table keeps track of pages in memory.
5.	Segments are allocated to the program while compilation.	Pages are loaded in memory on demand.
6.	Provides virtual memory.	Also provide virtual memory.

SOLVED EXAMPLES

Example 4.1 : *In demand paged memory, 200 ms are required to satisfy memory request if the page is in memory. If a page, is not in memory, the request takes 7 ms if a free frame is available. It takes 15 ms if the page to be swapped out has been modified. What is effective access time if the page fault rate is 5% and 60% of the time the page to be replaced has been modified. Assume the system is only running a single process & the CPU is idle during page swaps.*

Solution :

1. Percentage of access satisfied in 200 ms = 95%
2. Effective access time = $(1 - P) \times ma + P \times$ Page fault time

$= 0.95 \times 0.2 + 0.02 \times 7000 + 0.03 \times 15000$

Effective access time = 590.19 µs

(a) Here, 5% of access that results in page fault, 40% require 7 ms.

thus,

$5\% \times 40\% =$ 2% in 7 ms

(b) For 5% page fault 60%

$= 5\% \times 60\%$

= 3% in 15 ms

Example 4.2 : *If the average page fault service time of 25 milli-seconds and a memory access time of 100 nano-seconds. Calculate the effective access time.*

Solution :

Effective access time = $(1 - P) \times ma + P \times$ Page fault time

$= (1 - P) \times 100 + P \times 25000000$

$= 100 - 100\,P + 25000000 \times P$

$= 100 + 24999900\,P$

4.13 PAGE REPLACEMENT ALGORITHMS

- Page replacement policy deals with the selection of page in memory to be replaced when a new page must be brought in. While a user process is executing, a page fault occurs. The hardware traps to the operating system, which checks its internal tables, to see that this page fault is genuine one rather than an illegal memory access. The operating system determines where the desired page is residing on the disk, but then finds that there are no free frames on the free frame list. All memory is in use. This is shown in Fig. 4.22 when all frames in main memory are occupied and it is necessary to bring a new page satisfy a page fault, replacement policy is concerned with selecting a page currently in memory to be replaced.

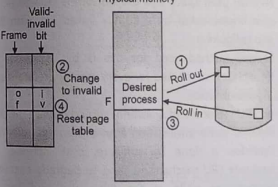


Fig. 4.22 : Page replacement

- All policies have as their objective that the page is removed should be the page least likely to be referenced in the near future.

Working of Page Replacement Algorithm

1. Find location of described page on the disk.
2. Find a free frame
   a. If there is a free frame, use it.
   b. If there is no free frame, use a page replacement algorithm to select a victim frame.
   c. Write a victim page to the disk, change the page and frame tables accordingly.
3. Read desired page into free frame, change the page and frame tables.
4. Restart user process.

Fig. 4.22 shows page replacement.

Memory Reference String

The string of memory references is called reference string. A succession of memory references made by a program executing on a computer with 1 MB of memory is given below in hex notation.

..., 14489, 11488, 14494, 14496, A1F8, 14497, 144999, 1449, ...

When analyzing page replacement algorithms, we are interested only in the pages being referenced. Assuming a 256 byte page size the referenced pages are obtained simply by omitting the two least significant hex digits.

...144, 144, 144, 144, A1, 144, 144, 263, 144, ...

The pattern of page reference above can be compared into a reference string for page replacement analysis as follows.

..., 144, A1, 144, 263, 144, ...

A reference string obtained in this way is used to illustrate most of the following replacement algorithms

Replacement Algorithms are of Following Types :

1. First In First Out (FIFO)
2. Least Recently Used (LRU)
3. Optimal
4. Second Chance
5. Not Recently Used (NRU)

4.13.1 FIFO Page Replacement Algorithm

- FIFO is one of the simplest methods. FIFO page replacement algorithm selects the page that has been in memory the longest. When a page must be replaced, the oldest page is chosen. To implement FIFO page replacement algorithm, the memory manager must keep track of relative order of loading of pages into memory. One way to accomplish this is to maintain a FIFO queue of pages. When a page is brought into memory, it is inserted at the tail of the queue.
- FIFO page replacement algorithm is easy to understand and program, it's performance is not always good. FIFO is not the first choice of operating system designers for page replacement algorithm.
- Let us consider the reference string.

0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5, 6, 7

Page frame is 3. Initially, all three frames are empty. The first three reference (0, 1, 2) cause page faults and are brought into these empty frames. The next reference 3 is replaced page 0, because page 0 was brought in first.

This process is as shown below :

0 1 2 3 0 1 2 3 0 1 2 3 4 5 6 7

Reference String

	1	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
Frame	0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
0	0	0	0	3	3	3	2	2	2	1	1	1	4	4	4	7
1		1	1	1	0	0	0	3	3	3	2	2	2	5	5	5
2			2	2	2	1	1	1	0	0	0	3	3	3	6	6
Page fault	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*

* Page fault

This example incurs 16 page faults in FIFO algorithm; FIFO algorithm with four page frames.

Frame	0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4
1		1	1	1	1	1	1	1	1	1	1	1	1	1	5	5
2			2	2	2	2	2	2	2	2	2	2	2	2	6	6
3				3	3	3	3	3	3	3	3	3	3	3	3	7
Page fault	*	*	*	*									*	*	*	*

Number of page fault = 8.

Belady's Anomaly :

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

4.13.2 LRU Page Replacement Algorithm

- Least recently used policy replaces the page in memory that has not been referenced for longest time. The LRU algorithm performs better than FIFO. LRU algorithm belongs to a larger class of stack replaced algorithms. When more real memory is made available to the executing program, stack algorithm therefore does not suffer from Belady's anomaly. Let us apply LRU algorithm to the reference string with frame is 3.

0 1 2 3 0 1 2 3 0 1 2 3 4 5 6 7

Reference String

Frame	0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
0	0	0	0	3	3	3	2	2	2	1	1	1	4	4	4	7
1			1	1	1	0	0	0	3	3	3	2	2	2	5	5
2				2	2	2	1	1	1	0	0	0	3	3	3	6
Page fault	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*

Number of page fault = 16

Consider same reference string with 4 frames

Frame	0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
1		1	1	1	1	1	1	1	1	1	1	1	1	5	5	5
2			2	2	2	2	2	2	2	2	2	2	2	2	6	6
3				3	3	3	3	3	3	3	3	3	3	3	3	7
Page fault	*	*	*	*									*	*	*	*

Number of page fault = 8.

- Implementation of LRU algorithm imposes too much overhead to be handed by softwares alone. Thus implementation of pure LRU replacement algorithm requires extensive and dedicated hardware support for desired stack operation. Stack is one of the solutions for implementing LRU algorithm. Whenever, a page is referenced, it is removed from stack and put on the top. In this way the top of stack is always the most recently used space and the bottom is LRU page. It is best implemented by a doubly linked list, with head and tail pointer.
- Second method used for this is by using counter. Counter is incremented for every memory reference.

LRU Approximation

- An LRU replacement algorithm should update the page removal status information after every page reference. Updation is done by software, cost increase, but hardware LRU mechanisms tend to degrade execution performance while at the time, they substantially increase cost. For this reason, simple and efficient algorithms that approximate LRU have been developed. With hardware support, reference bit is automatically set to 1 by hardware whenever that page is referenced. The single reference bit per clock can be used to approximate LRU removal.
- Page removed software periodically rests the reference bit to 0, while the execution of users job causes some reference bits to be set to 1. At any particular time, if the reference bit is 0 then that page has been referenced since the last time the reference bit was reset to 0.

4.13.3 Optimal Page Replacement Algorithm

- Optimal policy selects for replacement that page for which the time to next reference is longest. An optimal page replacement algorithm has lowest page fault rate of all algorithms and will never suffer from Belady's algorithm (anomaly). This algorithm is impossible to implement system to have perfect knowledge of future events.
- Let us consider reference string with frame equal to 3.

0123 0123 01234567

Frame	0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0	0	0	1	1	1	4	4	7
1		1	1	1	1	1	2	2	2	2	2	2	2	2	5	5
2			2	2	2	2	2	2	2	2	2	2	2	2	6	6
3			2	3	3	3	3	3	3	3	3	3	3	3	3	6
Page fault	*	*	*	*			*			*				*	*	*

Number of page faults = 10

With page frame = 4 for same references string

Frame	0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
1		1	1	1	1	1	1	1	1	1	1	1	1	5	5	5
2			2	2	2	2	2	2	2	2	2	2	2	2	6	6
3				3	3	3	3	3	3	3	3	3	3	3	3	7
Page fault	*	*	*	*									*	*	*	*

Number of page fault = 8

Example 4.3 : *Consider the following page reference string :*

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

How many page faults occur for the following replacement algorithms, assuming three frames i) FIFO ii) LRU iii) Optimal.

Solution : (i) FIFO

Frame	2	3	2	1	5	2	4	5	3	2	5	2
0	2	2	2	2	5	5	5	5	3	3	3	3
1		3	3	3	3	2	2	2	2	2	5	5
2				1	1	1	4	4	4	4	4	2
Page fault	①	②		③	④	⑤	⑥		⑦		⑧	⑨

Number of page fault = 9

(ii) LRU

Frame	2	3	2	1	5	2	4	5	3	2	5	2
0	2	2	2	2	2	2	4	4	4	2	2	2
1		3	3	3	3	2	2	2	2	2	5	5
2				1	5	5	5	5	5	5	5	5
Page fault	①	②		③	④		⑤		⑥			

Number of page fault = 6

(iii) Optimal

Frame	2	3	2	1	5	2	4	5	3	2	5	2
0	2	2	2	2	2	2	4	4	4	2	2	2
1		3	3	3	3	3	3	3	3	3	3	3
2				1	5	5	5	5	5	5	5	5
Page fault	①	②		③	④		⑤			⑥		

Number of page fault = 6

Example 4.4 : *Consider following page reference. Indicate page faults and calculate total number of page faults for optimal and LRU. The total number of available frames are 4.*

1 2 3 2 5 6 3 4 6 3 7 3 1 5 3 6 3 4 2 4 3 4 5 1

Solution :

(1) Optimal (frame = 4)

Reference String	0	1	2	3	Page Fault
1	1	–	–	–	*
2	1	2	–	–	*
3	1	2	3	–	*
2	1	2	3	–	
5	1	2	3	5	*
6	1	6	3	5	*
3	1	6	3	5	
4	1	6	3	4	*
6	1	6	3	4	
3	1	6	3	4	
7	1	6	3	7	*
3	1	6	3	7	
1	1	6	3	7	
5	1	6	3	5	*
3	1	6	3	5	

...Conti.

6	1	6	3	5	
3	1	6	3	5	
4	1	4	3	5	*
2	2	4	3	5	*
4	2	4	3	5	
3	2	4	3	5	
4	2	4	3	5	
5	2	4	3	5	
1	1	4	3	5	*

Number of page fault = 11

(2) LRU

Reference String	0	1	2	3	Page Fault
1	1	–	–	–	*
2	1	2	–	–	*
3	1	2	3	–	*
2	1	2	3	–	
5	1	2	3	5	*
6	6	2	3	5	*
3	6	2	3	5	
4	6	4	3	5	*
6	6	4	3	5	
3	6	4	3	5	
7	6	4	3	5	*
3	6	4	3	7	
1	6	1	3	7	*
5	5	1	3	7	*
3	5	1	3	7	
6	5	1	3	6	*
3	5	1	3	6	*
4	5	4	3	6	*
2	2	4	3	6	*
4	2	4	3	6	
3	2	4	3	6	
4	2	4	3	6	
5	2	4	3	5	*
1	1	4	3	5	*

Number of page fault = 14

Counting Based Page Replacement

Page replacement algorithm depends on the counter of the number of references.

Least Frequently Used (LFU)

The least frequently used algorithm selects a page for replacement if the page has not been used often in the post. LFU tends to react slowly to changes in locality. LFU used frequency counts from the beginning of the page reference stream. Memory reference string 0 1 2 3 0 1 2 3 0 1 2 3 4 5 6 7 with page frame is 3. LFU replacement is shown below.

Frame	0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0	0	0	0	3	3	3	3	7
1		1	1	1	1	1	1	3	3	1	1	1	1	1	1	1
2			2	3	3	3	2	2	2	2	2	2	4	5	6	7
Page fault	*	*	*	*			*	*		*		*	*	*	*	*

Example 4.5 : *Paging system consist of physical memory 224 bytes pages of logical address space is 256. Page size of 210 bytes, how many bits are in a logical address.*

Solution :

Logical address space = $256 = 2^8$

Page size = 2^{10} bytes

So, Total logical address space = $2^8 \times 2^{10} = 2^{18}$ bytes

For 2^{18} byte address space bit address is required.

Example 4.6 : *On a system using simple segmentation, compare the physical address for each of the logical address, logical address is given in the following segment table. If the address generates a segment fault, indicate so*

Segment	Base	Length
0	330	124
1	876	211
2	111	99
3	498	302

(a) 0, 99 (b) 2, 78 (c) 1, 265 (d) 3,222 (e) 0, 111

Solution :

(a) 0, 99

Offset = 99

Segment length= 124

Segment = 0

- Offset 99 is less than segment length 124.
- Starting location of segment 0 is start from 330.
- Physical address= offset + segment base
  = 99 + 330 = 429

(b) 2, 78

Segment = 2

Offset = 78

Segment length= 99

- Offset 78 is less than segment length 99
- Starting location of segment 2 is 111
- Physical address= segment base + offset
  = 111 + 78 = 189

(c) 1, 265

Segment = 1

Offset = 265

Segment length= 211

- Offset 256 is greater than segment length 211

This address results in a segment fault.

(d) 3, 222

Segment = 3

Offset = 222

Segment length= 302

- Offset 222 is less than segment length 302
- Starting location of segment 3 is 498
- Physical address= segment base + offset
  = 498 + 222
  = 720

(e) 0, 111

Segment = 0

Offset = 111

Segment length= 124

- Offset 111 is less than segment length 124
- Starting location of segment 0 is 330
- Physical address= segment base + offset
  = 330 + 111 = 441

Example 4.7 : *System using a paging and segmentation, the virtual address space consists of upto 8 segments where each segment can be up to 229 byte long. The hardware pages each segment 256 bytes pages. How many bits in the virtual address specify the*

(1) Segment number (2) Page number

(3) Offset within page (4) Entire virtual address

Solution :

(1) Virtual address space consists of upto 8 segments.

So $8 = 2^3$

$\therefore$ 3 bits are needed to specify segment number.

(2) Page number : Hardware pages each segment into 256 byte pages 30

$256 = 2^8$ byte pages.

Size of segment is 2^{29} bytes

$\therefore$ $2^{29}/2^8 = 2^{29-8} = 21$ pages

$\therefore$ 21 bits are required to specify the page number

(3) Offset within the page : For 2^8 byte page, 8 bits are needed.

(4) Entire virtual address :

= Segment number + page number + offset = 3 + 21 + 8 = 32

4.13.4 Second Chance (SC) Page Replacement Algorithm

- The basic algorithm of second chance replacement is a FIFO replacement algorithm. When a page has been selected, however, we inspect its reference bit. If the value is 0, we proceed to replace this page; but if the reference bit is set to 1, we give the page a second chance and move on to select the next FIFO page. When a page gets a second chance, its reference bit is cleared, and its arrival time is reset to the current time.
- Thus, a page that is given a second chance will not be replaced until all other pages have been replaced (or given second chances). In addition, if a page is used often enough to keep its reference bit set, it will never be replaced.
- One way to implement the second-chance algorithm is as a circular queue. A pointer indicates which page is to be replaced next. When a frame is needed, the pointer advances until it finds a page with a 0 reference bit.
- As it advances, it clears the reference bits. Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position. Notice that, in the worst case, when all bits are set, the pointer cycles through the whole queue, giving each page a second chance.
- It clears all the reference bits before selecting the next page for replacement. Second chance replacement degenerates to FIFO replacement if all bits are set.

Second Chance (or Clock) Page Replacement Policy

- Apart from LRU, OPT and FIFO page replacement policies, we also have the second chance/clock page replacement policy. In the Second Chance page replacement policy, the candidate pages for removal are considered in a round robin matter, and a page that has been accessed between consecutive considerations will not be replaced. The page replaced is the one that, when considered in a round robin matter, has not been accessed since its last consideration.
- It can be implemented by adding a "second chance" bit to each memory frame-every time the frame is considered (due to a reference made to the page inside it), this bit is set to 1, which gives the page a second chance, as when we consider the candidate page for replacement, we replace the first one with this bit set to 0 (while zeroing out bits of the other pages we see in the process). Thus, a page with the "second chance" bit set to 1 is never replaced during the first consideration and will only be replaced if all the other pages deserve a second chance too!.

Example :

Let's say the reference string is 0 4 1 4 2 4 3 4 2 4 0 4 1 4 2 4 3 4 and we have 3 frames. Let's see how the algorithm proceeds by tracking the second chance bit and the pointer.

Initially, all frames are empty so after first 3 passes they will be filled with {0, 4, 1} and the second chance array will be {0, 0, 0} as none has been referenced yet. Also, the pointer will cycle back to 0.

Pass 4: Frame={0, 4, 1}, second_chance = {0, 1, 0} [4 will get a second chance], pointer = 0 (No page needed to be updated so the candidate is still page in frame 0), pf = 3 (No increase in page fault number).

Pass-5: Frame={2, 4, 1}, second_chance= {0, 1, 0} [0 replaced; it's second chance bit was 0, so it didn't get a second chance], pointer=1 (updated), pf=4

Pass-6: Frame={2, 4, 1}, second_chance={0, 1, 0}, pointer=1, pf=4 (No change)

Pass-7: Frame={2, 4, 3}, second_chance= {0, 0, 0} [4 survived but it's second chance bit became 0], pointer=0 (as element at index 2 was finally replaced), pf=5

Pass-8: Frame={2, 4, 3}, second_chance= {0, 1, 0} [4 referenced again], pointer=0, pf=5

Pass-9: Frame={2, 4, 3}, second_chance= {1, 1, 0} [2 referenced again], pointer=0, pf=5

Pass-10: Frame={2, 4, 3}, second_chance= {1, 1, 0}, pointer=0, pf=5 (no change)

Pass-11: Frame={2, 4, 0}, second_chance= {0, 0, 0}, pointer=0, pf=6 (2 and 4 got second chances)

Pass-12: Frame={2, 4, 0}, second_chance= {0, 1, 0}, pointer=0, pf=6 (4 will again get a second chance)

Pass-13: Frame={1, 4, 0}, second_chance= {0, 1, 0}, pointer=1, pf=7 (pointer updated, pf updated)

Page-14: Frame={1, 4, 0}, second_chance= {0, 1, 0}, pointer=1, pf=7 (No change)

Page-15: Frame={1, 4, 2}, second_chance= {0, 0, 0}, pointer=0, pf=8 (4 survived again due to 2nd chance!)

Page-16: Frame={1, 4, 2}, second_chance= {0, 1, 0}, pointer=0, pf=8 (2nd chance updated)

Page-17: Frame={3, 4, 2}, second_chance= {0, 1, 0}, pointer=1, pf=9 (pointer, pf updated)

Page-18: Frame={3, 4, 2}, second_chance= {0, 1, 0}, pointer=1, pf=9 (No change)

In this example, second chance algorithm does as well as the LRU method, which is much more expensive to implement in hardware.

More Examples :

Input: 2 5 10 1 2 2 6 9 1 2 10 2 6 1 2 1 6 9 5 1 3

Output: 14

Input: 2 5 10 1 2 2 6 9 1 2 10 2 6 1 2 1 6 9 5 1 4

Output: 11

Algorithm :

Create an array frames to track the pages currently in memory and another Boolean array second_chance to track whether that page has been accessed since it's last replacement (that is if it deserves a second chance or not) and a variable pointer to track the target for replacement.

Start traversing the array arr. If the page already exists, simply set its corresponding element in second_chance to true and return.

If the page doesn't exist, check whether the space pointed to by pointer is empty (indicating cache isn't full yet) – if so, we will put the element there and return, else we'll traverse the array arr one by one (cyclically using the value of pointer), marking all corresponding second_chance elements as false, till we find a one that's already false. That is the most suitable page for replacement, so we do so and return.

Finally, we report the page fault count.

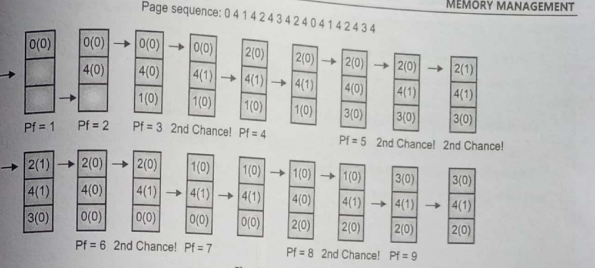


Fig. 4.23

Program 4.1 :

```
// CPP program to find largest in an array
// without conditional/bitwise/ternary/ operators
// and without library functions.
#include<iostream>
#include<cstring>
#include<sstream>
usingnamespacestd;

// If page found, updates the second chance bit to true
staticboolfindAndUpdate(intx,intarr[],
          boolsecond_chance[],intframes)

{
  inti;

  for(i = 0; i< frames; i++)
  {

    if(arr[i] == x)
    {
      // Mark that the page deserves a second chance
      second_chance[i] = true;

      // Return 'true', that is there was a hit
      // and so there's no need to replace any page
      returntrue;
    }
  }

  // Return 'false' so that a page for replacement is selected
  // as he reuested page doesn't exist in memory
  returnfalse;

}

// Updates the page in memory and returns the pointer
staticintreplaceAndUpdate(intx,intarr[],
        boolsecond_chance[],intframes,intpointer)
{
  while(true)
  {

    // We found the page to replace
    if(!second_chance[pointer])
    {
      // Replace with new page
      arr[pointer] = x;

      // Return updated pointer
      return(pointer + 1) % frames;
    }
```

```
    // Mark it 'false' as it got one chance
    // and will be replaced next time unless accessed
again
    second_chance[pointer] = false;

    //Pointer is updated in round robin manner
    pointer = (pointer + 1) % frames;
  }
}

staticvoidprintHitsAndFaults(string reference_string,
                             intframes)
{
  intpointer, i, l=0, x, pf;

  //initially we consider frame 0 is to be replaced
  pointer = 0;

  //number of page faults
  pf = 0;

  // Create a array to hold page numbers
  intarr[frames];

  // No pages initially in frame,
  // which is indicated by -1
  memset(arr, -1, sizeof(arr));

  // Create second chance array.
  // Can also be a byte array for optimizing memory
  boolsecond_chance[frames];

  // Split the string into tokens,
  // that is page numbers, based on space

  string str[100];
  string word = "";
  for(autox :reference_string)
  {
    if(x == ' ')
    {
      str[l]=word;
      word = "";
      l++;
    }
    else
    {
      word = word + x;
    }
  }
  str[l] = word;
  l++;
  // l=the length of array

  for(i = 0; i< l; i++)
  {
    x = stoi(str[i]);

    // Finds if there exists a need to replace
    // any page at all
    if(!findAndUpdate(x,arr,second_chance,frames))
    {
      // Selects and updates a victim page
      pointer = replaceAndUpdate(x,arr,
          second_chance,frames,pointer);

      // Update page faults
      pf++;
    }
  }
  cout<<"Total page faults were "<< pf <<"\n";
}

// Driver code
intmain()
```



```
{
  string reference_string = "";
  intframes = 0;

  // Test 1:
  reference_string = "0 4 1 4 2 4 3 4 2 4 0 4 1 4 2 4 3
4";
  frames = 3;

  // Output is 9
  printHitsAndFaults(reference_string,frames);

  // Test 2:
  reference_string = "2 5 10 1 2 2 6 9 1 2 10 2 6 1 2 1
6 9 5 1";
  frames = 4;

  // Output is 11
  PrintHitsAndFaults(reference_string,frames);
  return0;
}
```

Output:

Total page faults were 9

Total page faults were 11

Note:

The arrays arr and second_chance can be replaced and combined together via a hashmap (with element as key, true/false as value) to speed up search.

Time complexity of this method is O (Number_of_frames * reference_string_length) or O (mn) but since number of frames will be a constant in an Operating System (as main memory size is fixed), it is simply O(n) (Same as hashmap approach, but that will have lower constants).

Second chance algorithm may suffer from Belady's Anomaly.

4.13.5 Not Recently Used (NRU) Page Replacement Algorithm

- It is a page replacement algorithm. This algorithm removes a page at random from the lowest numbered non-empty class. Implicit in this algorithm is that it is better to remove a modified page that has not been referenced in atleast one clock tick than a clean page that is in heavy use.
- It is easy to understand, moderately efficient to implement and gives a performance that while certainly not optimal, may be adequate. When page is modified, a modified bit is set. When a page needs to be replaced, the Operating System divides pages into 4 classes.

  1 : Not Referenced, Not Modified

  2 : Not Referenced, Modified

  3 : Referenced, Not Modified

  4 : Referenced, Modified
- Out of above 4 categories, NRU will replace a Not Referenced, Not Modified page, if such page exists. Note that this algorithm implies that a Modified but Not Referenced is less important than a Not Modified and Referenced.

Example :

Page	Referenced	Modified
0	1	0
1	0	1
2	0	0
3	1	1

Case-1 implies to Not Referenced and Modified.

Case-2 implies to Not Referenced and Not Modified.

Case-3 implies to Referenced and Modified.

Case-0 implies to Referenced and Not Modified.

Algorithm :

From the given reference string NRU will remove a page at random from the lowest numbered nonempty class. Implicit in this algorithm is that it is better to remove a modified page that has not been referenced in at least one clock tick (typically 20 msec) than a clean page that is in heavy use.

Example :

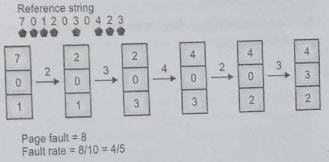


Fig. 4.24

Memory States :

- **Available :** Currently this address is not used by this process.
- **Reserved :** Used strictly by VMM for process.
- **Committed :** It is used by process to access virtual pages. These pages may reside either on disk or physical memory. In case of disk it will be in files while in case of physical memory. In case of disk it will be in files while in case of physical memory it will be in paging file.
- The resident work set uses variable allocation.
- On creation, every process is assigned with working set.
- Required amount of page are brought in to physical memory where Memory Management (MM) keeps track of the number allocated process per process.
- The best thing in case of work set management is that, in case if process page faults, then additional page is allotted to active process to grow.
- This is done without swap of any of older page. This growth in work set of active process helps in controlling the page miss in further processing.
- When no much load is incurred, VMM claims whole memory by moving out less frequently used pages from memory of active process work set.

EXERCISE

1. What are the requirements for management of memory?
2. What is relocation?
3. What is the difference between page and frames?
4. What advantage does the segmentation system pose over paging system?
5. Explain problem of fragmentation.
6. Explain address translation in paging.
7. Explain with a diagram the concept of a translation look aside buffer.
8. Explain with a neat diagram memory management in windows 2000.
9. Describe the following memory allocation strategies:
   (i) First fit (ii) Best fit (iii) Worst fit
10. Why is paging necessary? Explain the paging hardware with TLB with the help of suitable diagram.
11. Describe how address translation takes place with the help of neat diagrams in:
    (i) Paging (ii) Segmentation
12. Explain the concept of paging with neat diagram.
13. Explain internal and external fragmentation.
14. Differentiate the contiguous and non-contiguous memory allocation.
15. Explain in detail various partitioning memory management.
16. A process references page in the following order 5, 6, 7, 8, 5, 6, 9, 5, 6, 7, 8, 9 .Use the FIFO and LRU page replacement algorithm to find out the number of page faults for this reference string using 3 page frames. Repeat for 4 page frames. Analyze the difference in the result.
17. Describe the following:
    (i) Principle of locality (ii) Thrashing
18. Write short note on virtual memory management.
19. What is page fault rate? Explain with example.
20. Given the following page reference string: 1, 2, 3, 2, 5, 6, 3, 4, 6, 3, 7, 3, 1, 6, 3, 4, 5, 3, 2, 4, 3, 4, 5,1 Number of page frames are 4. Show the page trace and calculate the number of page frames for the following page replacement policies
    (i) LRU (ii) Optimal (iii) FIFO
    Also explain Betady's anomaly
21. Draw schematic of virtual memory.
22. What is working set model and how it is used in page replacement?
23. Discuss the concept "Locality of reference".

✠ ✠ ✠

Unit V

FILE MANAGEMENT

5.1 INPUT/OUTPUT MANAGEMENT

5.1.1 I/O Devices

I/O devices can be divided into block device and character device :

1. **A Block Device**
   Here the information is stored in fixed size block with its own address.
2. **A Character Device**
   - This type of device delivers or accepts a stream of characters and individual characters are not addressable e.g. keyboard, pointers.
   - The essential property of a block device is that it is possible to read or write each block independently of all other ones. Seek operation is possible in case of block devices.
     For Example : in case of a disk, no matter where the arm currently is, it is always possible to seek another cylinder.
   - While in case of character device, it is not possible to have any seek operation.
   - Since the number of device registers and the nature of device instructions vary from device to device, a device driver operating system component hides the complexity of an I/O device so that operating system can access various devices in a relatively uniform manner.

I/O Devices can be Characterised into Three Sections:

1. **Human Readable:**
   Suitable for communicating with the computer user. These devices have important role of notifying every action and reaction to computer as well as external world. It includes printer, terminals, video display, keyboard, mouse etc.
2. **Machine Readable:**
   Suitable for communicating with electronic equipment. These devices are used to establish communication with system itself. For Example: disk driver, USB keys, sensors, controllers, and actuators.
3. **Communication:**
   Suitable for communicating with remote devices. These devices play role of communication with other devices place at different location.
   For Example : Digital line drivers and modems etc.
   All these devices have great differences with respect to their functioning.

Key Differences Among I/O Devices :

1. **Data Rate:**
   - There are different types of device having different data carrying capacity at data transfer rate.
   - Data rate is a capacity of a device to read and returned required data in unit time.
2. **Application:**
   - The way these devices are used has great impact on the software and policies used by the management software and supporting utilities.
   - Example, to connect file transfer operation to and from disk support of file management software is required. Many memory operations are used in bringing required data writing back newly produces or modified data. A strong virtual memory management software is required that will actually coordinate these operations successfully.
   - There also exist many important applications that manage process synchronization, process scheduling, and disk scheduling using suitable algorithms.
3. **Complexity of Control:**
   - If we will consider any input or output device, its functionality depends on the nature of hardware used. To make these devices work, device drivers play vital role,
   - When operating system needs to work with these devices, it uses these drivers that help in conduction of various tasks from device.
   - Operating system, thus works with these devices by hiding internal complexities of the functioning by making a computing a pleasant experience to the user of system.
4. **Unit of Transfer:**
   - It defines the rate at which data may be transferred as a stream of bytes or characters to and from computer or device. (e.g., terminal I/O) or in larger block (e.g., disk I/O).

5. Data Representation:

- Data may be read and returned by device using different data encoding schemes as per requirement, including differences in character code and parity conventions.

6. Error Conditions:

- Every device will have its own way of working as well as reporting errors, every error have its own consequences. There are different ways these errors can be handled. This will vary from device to device.
- This diversity makes it an important task to have a uniform and consistent approach towards I/O.

5.1.2 Organization of the I/O Function

There are three techniques for performing I/O :

1. Programmed I/O
2. Interrupt driven I/O
3. Direct Memory Access (DMA)

1. Programmed I/O

- The processor issues an I/O command on behalf of a process, to an I/O module. That process then waits for the operations to be completed.
- In this technique, the processor is responsible for extracting data from main memory for output and storing data in main memory for input.

2. Interrupt Driven I/O

- The problem with programmed I/O is that the processor has to wait a long time for the I/O module of concern to be ready for either reception or transmission of more data. The processor while waiting must repeatedly interrogate the status of the I/O module which degrades the performance of the entire system.
- The alternative for the processor is to issue an I/O command on behalf of a process, continues to execute subsequent institutions and interrupted by the I/O module, when latter has completed its task.

3. Direct Memory Access (DMA)

- The I/O functions have evolved as the computer system is evolving.
- The processor directly controls a peripheral device. A controller is added. The processor uses programmed I/O without interrupt.
- Same configuration as in step b is used, but interrupts are involved. The processor need not spend time waiting for an I/O operation to be performed.
- The I/O module is given direct control of memory via DMA. It can move a block of data to or from memory without involving processor, except at the beginning and end of a transfer.
- A DMA is a device that participates in condition of bulk data transfer activities without wasting processors valuable time. The processor renders the request of block data transfer first to DAM module and is intimated back only the entire block has been transfer.

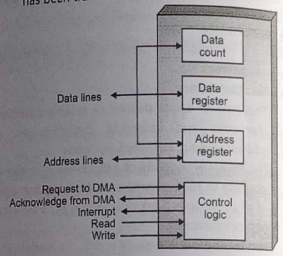


Fig. 5.1 : Typical DMA block diagram

A DMA module is seen in the Fig. 5.1. DMA units is capable of simulating processors memory transfer operations efficiently. This is possible by taking over control of the system bus in the same way like how processor does. It needs to do this to transfer of data to and from memory or devices using system bus.

DMA Operation Works on:

On receiving instruction for bulk memory transfer processor issues a command to the DMA along with some important details regarding the transfer that is to be made:

Whether data is to be read or written. The address of the I/O device involved, communication on the data lines.

- The sharing location in memory to or from read or writes operation is to be conducted. This is made available through DMA address register.
- The number of words to be read or written, this is conveyed through DMA count register.

Processor relinquishes the bus control to DMA and continues with other work.

DMA Performs this Task Using Different Approaches Like:

1. Burst Mode:

Bus control is given back any after completion of work.

2. Cycle Stealing Mode:

Bus control is acquired in between cycles when bus is idle.

3. Bus Interleaved Mode:

Bus control is obtained interrupt (request and release strategy) in between. The DMA module transfer the entire block of data, one word at a time, directly to or from memory, without disturbing processor (In case of burst mode). When the transfer is complete, the DMA module sends an interrupt signal to the processor. The type of architecture also influences the DMA operation.

Some of the Different Ways are:

1. Single-Bus, Detached DMA Controllers:

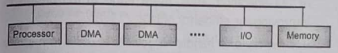


Fig. 5.2 : Single bus, detached DMA controller

Each Transfer Uses Bus Twice.

- I/O to DMA then DMA to memory
- CPU is suspended twice
- Single Bus, Integrated DMA controller
- Controller may support > 1 device
- Each transfer uses bus once
- DMA to memory
- CPU is suspended once

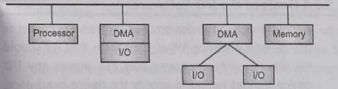


Fig. 5.3: Single-bus, integrated DMA-I/O

2. Single-Bus Integrated DMA-I/O

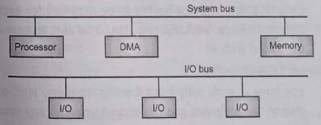


Fig. 5.4: I/O bus

- Separate I/O Bus
- Bus support all DMA enabled devices
- Each transfer uses bus once
- DMA to memory
- CPU suspended once

Various Innovations that Have Contributed in Efficient I/O Operation :

- In simple microprocessor-controlled devices processor directly control the I/O device.
- With the help of a special hardware known as "controller" (operates through drivers) or I/O module, Processor executes programmed I/O instead of interrupts and accomplish the task.
- Another approach is to raise the interrupt. The processor need not spend time waiting for an I/O operation rather interrupts service subroutine is executed, thus increasing efficiency.
- I/O module directly hands over memory control to DMA. It can now move a block of data to or from memory without involving the processor, except at the beginning and end of the transfer.
- System can have a separate I/O processor. This special processor will be equipped with its own instruction set to perform I/O. The CPU directs the I/O processor to execute an I/O program in main memory. The I/O processor fetches and executes these instructions without disturbing processor.
- I/O also may be equipped with its own local memory. With this, management of large set of I/O devices becomes possible with minimal processor involvement. Such type of architecture is commonly seen in communication devices. The I/O processor takes care of most of the tasks involved in controlling the terminals.
- Thus, execution of multiple I/O operations becomes possible without putting burden on processor. With the involvement of memory, program based I/O becomes possible.

5.1.3 Life Cycle of an I/O Request

The diagram can be explained in various steps as :

- A process issues a blocking read system call to a file descriptor of file that has been opened previously.
- The system call code in the kernel checks the parameters for correctness. In case of input, if data are already available in buffer cache, the data is returned to the process and I/O request is completed.
- Otherwise, a physical I/O needs to be performed, so the process is removed from run queue and is placed in wait queue for the device and I/O request is scheduled. Eventually, the I/O subsystem sends the request to device driver.

- The device driver allocates kernel buffer space to receive the data and schedules the I/O. The driver then sends the commands to the device controller by writing into device control registers.
- The driver may poll for status and data or it may have set up a DMA transfer into kernel memory. We assume that the transfer is managed by a DMA controller, which generates an interrupt when the transfer completes.
- The correct interrupt handler receives the interrupt via interrupt vector table, stores any necessary data and signals the device driver then returns from the interrupt.
- The device driver receives the signal, determines which I/O request completed, determine the requests status and signal the kernel I/O subsystem that the request has been completed.
- The kernel transfers data or return code to the address space of the requesting process and move, process from the wait queue back to ready queue.
- Making the process to the ready queue unblocks the processes when the scheduler assigns the process to the CPU the processes resumes execution at the completion of the system call.

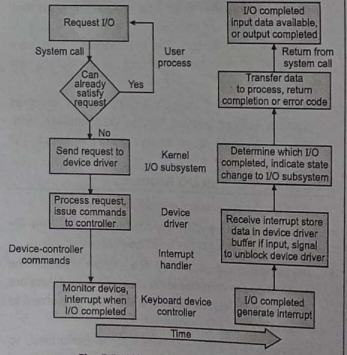


Fig. 5.5 : Life cycle of an I/O request

5.1.4 Operating System Design Issues

There are two main objectives in designing the I/O facility which needs to be fulfilled

1. Efficiency
2. Generality

- Most I/O devices are extremely slow compared with main memory and processor. There is a way to handle this problem is the use of multiprogramming which allows some processes to be waiting on I/O operations while another process is executing. Now-a-days, we have vast size of main memory. Inspite of having a main memory which is huge in size it will still often be the case that the I/O is not keeping pace with the activities of processor.
- Swapping is used to bring additional ready processes to keep processor busy, but this activity is also a kind of I/O operation. Thus, major effort in I/O design in improving the efficiency of I/O.
- It is desirable to handle all devices in uniform manner and it applies to both the way in which processes view I/O devices and the way in which the operating system manages the I/O devices and operations. To achieve this generality, use a hierarchical modular approach to the design of I/O function. This approach would hide most of the details of device I/O in lower level so that user processes and upper level of OS see devices in terms of general functions such as read, write, open, close etc.

5.2 PRINCIPLES OF I/O SOFTWARE

5.2.1 Goals of the I/O Software

- A key concept in the design of I/O software is known as 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, a program that reads a file as input should be able to read a file on a hard disk, a DVD, or on a USB stick without having to be modified for each different device. Similarly, one should be able to type a command such as

```
sort <input >output
```

and have it work with input coming from any kind of disk or the keyboard and the output going to any kind of disk or the screen. It is up to the operating system to take care of the problems caused by the fact that these devices really are different and require very different command sequences to read or write.

Programmed I/O :

There are three fundamentally different ways that I/O can be performed. programmed I/O, interrupt-driven I/O and

I/O using DMA. The simplest form of I/O is to have the CPU do all the work. This method is called programmed I/O.

Interrupt-Driven I/O :

- Now let us consider the case of printing on a printer that does not buffer characters but prints each one as it arrives. If the printer can print, say 100 characters/sec, each character takes 10 msec to print. This means that after every character is written to the printer's data register, the CPU will sit in an idle loop for 10 msec waiting to be allowed to output the next character.
- This is more than enough time to do a context switch and run some other process for the 10 msec that would otherwise be wasted. The way to allow the CPU to do something else while waiting for the printer to become ready is to use interrupts.
- When the system call to print the string is made, the buffer is copied to kernel space, as we showed earlier, and the first character is copied to the printer as soon as it is willing to accept a character. At that point the CPU calls the scheduler and some other process is run. The process that asked for the string to be printed is blocked until the entire string has printed.
- When the printer has printed the character and is prepared to accept the next one, it generates an interrupt. This interrupt stops the current process and saves its state. Then the printer interrupt-service procedure is run. If there are no more characters to print, the interrupt handler takes some action to unblock the user. Otherwise, it outputs the next character, acknowledges the interrupt, and returns to the process that was running just before the interrupt, which continues from where it left off.

5.2.2 Interrupt Handlers

- In modern computer hardware, three features are provided by the CPU and by the interrupt-controller hardware.
  1. We need the ability to defer interrupt handling during critical processing.
  2. We need an efficient way to dispatch to the proper interrupt handler for a device without first polling all the devices to see which one raised the interrupt.
  3. We need multilevel interrupts, so that the operating system can distinguish between high- and low-priority interrupts and can respond with the appropriate degree of urgency.
- Most CPUs have two interrupt request lines. One is the non maskable interrupt, which is reserved for events such as unrecoverable memory errors. The second interrupt line is maskable: it can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted. The maskable interrupt is used by device controllers to request service. The interrupt mechanism accepts an address-a number that selects a specific interrupt-handling routine from a small set. In most architectures, this address is an offset in a table called the interrupt vector.
- This vector contains the memory addresses of specialized interrupt handlers. The purpose of a vectored interrupt mechanism is to reduce the need for a single interrupt handler to search all possible sources of interrupts to determine which one needs service. In practice, however, computers have more devices (and, hence, interrupt handlers) than they have address elements in the interrupt vector.
- A common way to solve this problem is to use interrupt chaining, in which each element in the interrupt vector points to the head of a list of interrupt handlers. When an interrupt is raised, the handlers on the corresponding list are called one by one, until one is found that can service the request. This structure is a compromise between the overhead of a huge interrupt table and the inefficiency of dispatching to a single interrupt handler.

5.3 DEVICE DRIVERS

- For device hardware manufacturers, each type of operating system has its own standards for the device-driver interface. A given device may ship with multiple device drivers-for instance, drivers for Windows, Linux, AIX, and MacOS X.
- Modern operating systems gain significant flexibility from the multiple stages of lookup tables in the path between a request and a physical device controller. The mechanisms that pass requests between applications and drivers are general. Thus, we can introduce new devices and drivers into a computer without recompiling the kernel. In fact, some operating systems have the ability to load device drivers on demand.
- At boot time, the system first probes the hardware buses to determine what devices are present. It then loads in the necessary drivers, either immediately or when first required by an I/O request.

5.4 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 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.

5.5 DISK SCHEDULING

- In multiprogramming systems, many processes are generating requests for reading and writing disk records. Because these processes often make requests faster than they can be serviced by the moving head disks, waiting queues are built for each device.
- In order to stop unbounded increase in the queue length, these pending requests must be examined and serviced in an efficient manner.
- Thus, the total delay seen by the process has several components.
- The overhead of getting into and out of the OS and the time the OS spends fiddling with queues etc.
- The queuing time spent waiting for the disk to become available.
- The latency spent waiting for the disk to get the right track and sector.
- The transfer time spent actually reading or writing the data.

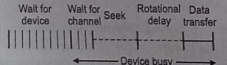


Fig. 5.6 : Timing of a disk I/O transfer

- Disk scheduling involves a careful examination of pending requests to determine the most efficient ways to service the waiting requests.
- Although there is a queue of requests, there is no reason why the requests have to be satisfied first come first served.
- In fact, that is a very bad way to schedule disk requests.
- Since requests from different processes may be scattered all over the disk, satisfying them in the order they arrive would entail an awful lot of jumping around on the disk, resulting in excessive rotational latency and seek time, both for individual requests and for the system as a whole.
- Fortunately, better algorithms are not hard to devise.
- For each I/O request, first head is selected. It is then moved over the destination track.
- The disk is then rotated to position the desired sector under the head and finally the read write operation is performed.

There are Two Objectives of any Disk Scheduling Algorithm:

1. **Minimize the Throughput :** The average number of requests satisfied per unit time.
2. **Maximize the Response Time :** The average time a request must wait before it is satisfied.

Disk Scheduling Algorithms : Various types of disk scheduling algorithms are as follows :

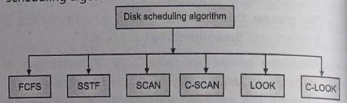


Fig. 5.7

1. FCFS (First Come First Served) Scheduling

- Requests are served in the order of their arrival.
- Consider an e.g. a disk queue with request for I/O to blocks on cylinders.

  98, 183, 37, 122, 14, 124, 65, 67 in that order of the disk head is initially at cylinder 53, it will move from 53 to 98 and then 183, 37 so on.
- Total head movement of 640 cylinder.

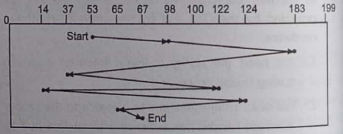


Fig. 5.8 : FCFS Scheduling

Advantages :

- All requests are treated equally, i.e. average response time will be same for all requests.
- It is not possible for a request to wait forever.

Disadvantages :

- There is a lot of unnecessary seeking. In the previous example, we saw wild movement of head from cylinder 122 to 14 and then back to cylinder 124.

2. Shortest Seek Time First (SSTF) Scheduling

- We can minimize the head movement by servicing the request closest to the current position of head.
- Consider an e.g. same as the above one 98, 183, 37, 122, 14, 124, 65, 67.

  Current head position at 53.
- So closest request to initial head position 53 is at cylinder 65. Once we are at cylinder 65, next closest request is at cylinder 67, then request 37 as compared to 98.

The requests are served as shown in the Fig. 5.9 below :

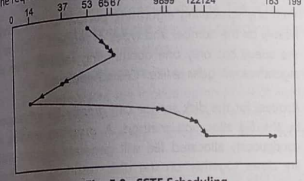


Fig. 5.9 : SSTF Scheduling

Problem with this Approach :

- Some request can wait much longer time than others. This happens if the disk head stays in one part of the disk servicing request and does not service request from remote cylinders. For example, we have two requests in a queue for cylinders 14 and 186, while servicing the request from 14, a new request near 14 arrives.
- This new service request will be served making request at 186 to wait.
- While this request is being serviced, another request close to 14 could arrive. It may continue causing request for cylinder 186 to wait indefinitely.

SSTF with Aging

- The age of a request is the time it has been waiting to be serviced. As the time goes on, a request that has not been satisfied will get older.
- The priority that determines the request which is chosen next is the combination of priority of the algorithm we are adding aging to and the age of the request.
- The priority is figured that OS a request gets older its priority increases and after a certain age it will be guaranteed to have the highest priority.

3. SCAN Scheduling

- The two scheduling algorithms which we have discussed serves the request in one direction.
- The disk arm starts at one end and moves towards the other end of the disk. Servicing request as it reaches each cylinder until it gets to the other end of the disk. At the other end, the direction of head movement is reversed and servicing continues.
- Before applying SCAN to schedule request, direction of head movement is necessary to be known

  e.g. 98, 183, 37, 122, 14, 124, 65, 67.

  Current head position at cylinder 53.
- Direction of head movement is towards 0. The head will service the requests at 37 and 14. At cylinder 0, the arm will reverse and move towards other end of the disk servicing request at 65, 67, 98, 122, 124 and 183.

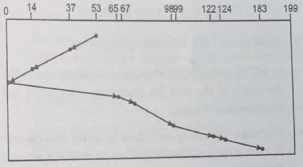


Fig. 5.10 : SCAN Scheduling

- This algorithm is also called as an elevator algorithm because the disk arm behaves like an elevator in a building, first servicing all the request going up and reversing to service request the other way.
- If a request arrives in the queue just in front of the head, it will be serviced immediately, a request arriving just behind the head will have to wait until the arm moves to the ends of the disk, reverses direction and comes back.

4. C-SCAN Scheduling

- In SCAN scheduling, if there are few requests in front of the head, these requests will be served immediately. The heaviest density of request is at the other end of the disk. These requests have waited longer. So why not start from there?

- This scheduling algorithm provides more uniform wait time. C-SCAN moves the head from one end of the disk to the other, servicing request along the way. When the head reaches the other end, it immediately returns to the beginning of the disk without servicing any request on the return trip.
- Consider an e.g.

  Queue = 98, 183, 37, 122, 14, 124, 65, 67.

  Head starts at 53.

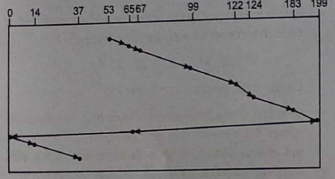


Fig. 5.11 : C-SCAN scheduling

5. Look and C Look Scheduling Algorithm

- In SCAN and C-SCAN the disk arm moves across the full width of the disk. But practically, algorithm is not implemented this way.
- Practically, the arm goes only as far as the final request in each direction. Then it reverses its direction without first going all the way to the end of the disk.
- Consider an e.g. Queue : 98, 183, 37, 122, 14, 124, 65, 67.

  Head starts at 53. In look scheduling algorithm, the head is at 53, so it will serve the request 53, 37, 14 and then reverse its direction would serve the request 65, 67 and so on without going all the way to the end of the disk.

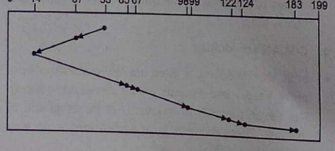


Fig. 5.12 (a) : Look scheduling

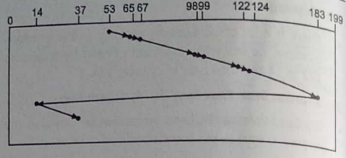


Fig. 5.12 (b) : C-look scheduling

5.5.1 Selection Criteria for Disk Scheduling Algorithm

- SSTF is very common and has a natural appeal. SCAN and C-SCAN perform better for systems that place heavy load on the disk because they are less likely to have starvation problem.
- With any scheduling algorithm, performance depends heavily on the number and types of request. Suppose if the queue has only one outstanding request, then all algorithms will behave like FCFS scheduling.
- Request for the disk service can greatly be influenced by the file allocation method. A program reading a contiguously allocated file will generate request that are closer together on the disk, resulting in limited head movement.
- A linked file on the other hand may include blocks that are widely scattered on the disk resulting in greater head movement.
- Location of directories and index block is also important since every file must be opened to be used and opening a file requires searching the directory structure.
- Suppose the directory entry is on the first cylinder and files data are on final cylinder, then disk head has to move entire width of the disk.
- Caching the directories and index blocks in main memory can also help to reduce the disk arm movement particularly for read request.
- Along with seek time for modern disks, the rotational latency should be considered, which could be as large as average seek time. But it is difficult for the OS to schedule for improved rotational latency because modern disks do not disclose the physical location of logical blocks.

5.6 FILE MANAGEMENT

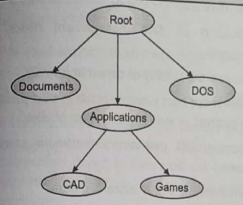


Fig. 5.13

The above Fig. 5.13 shows the structure of typical file system.

- File system is an internal part of the operating system. This is a traditional way of data collection and is mostly associated with secondary storage. File is used as the medium of giving input as well as collecting the output for most of the application.
- Our computer works on stored program concept and hence programs are looked up on as set of instruction grouped together in file. It serves as a snapshot of data abstraction, typically associated with secondary storage.
- The file system permits users to create data collections, called files, with desirable properties.

Some of the Properties of the Files:

1. **Long Term Existence:**

   Files are permanently stored on disk or any other secondary storage like USB drivers. Data stored in such devices is non volatile. It can be referred after many days.

2. **Sharable Between Processes:**

   Each file has its own names and access right assigned that decide on permissions associated with respect owner, group and others. Owner can share the file content with other users by granting privileges.

3. **Structure**

   Depending on the file system, a file can have its own internal structure. As per the demand of application, files can be organized into hierarchical or more complex structure to reflect the relationships among files.

   File is used to store data in organized manner. It also supports set of functions that are to be used to manage the file operations. Typical operations include the following:

(a) Create:

A new file created, named and positioned within the directory structure of files.

(b) Delete:

A file is moved out of memory file structure and is permanently destroyed.

(c) Open:

Files available in the directory structure can be opened by process allowing the process to perform required functions on the files.

(d) Close:

The file is closed with respect to a process, so that the process no longer may perform function on the file, until the process opens the files again.

(e) Read:

A process can read all or required file data using this function.

(f) Write:

We can update the content of the file by adding new data to file. This will increase the file like owner name, time of creation; time last modified access right...Etc.

5.6.1 File Structure

Every file is associated with four terms.

i. Field. ii. Record.

iii. File. iv. Database

i. Field:

Field is basic entity of data. It consists of a single value. Each field will have its own length and data type. Field can be of fixed length or variable length depends upon application requirement,

ii. Record:

A group of fields contribute to a record (different data type). Accounts table record can be account number, amount, account type, date of creation, Name of account owner etc.

Each record will have unique identity field that will help to differentiate among other records.

iii. File:

This is collection of related records. A file has its own set of attributes ex- name, Owner's name, data of last

update, size, type, access permission etc. It is a unit on which an application program may be dependant for reading data as input or to store data as output or may be to refer the data in decision making process. Files can be used dedicatedly as a private file by restricting the access. It can also be shared among multiple users or application.

iv. Database:

- It is viewed as a collection of data generally in table.
- A database may contain multiple files in the term of tables.
- Record is a group of related attributes or fields.
- Each field in term has its own characteristics, we have already seen.

Typically application that uses the data through files must support some minimum set of operation such as.

- Retrieve all : When application needs to refer to all records and in turn all fields of records, these functions is used. All the records will be retrieved in sequence.
- Retrieve – one: This function is used to select one particular record.
- Retrieve – next: This is used when records are to be retrieved in some chronological order one by one, till some condition is fulfilled.
- Retrieve – previous: This is exactly same as retrieve – next, only small difference is this function will demand all records which are already accessed.
- Insert – one: Used to insert one record to database like entry of new account created.
- Delete – one: Used to delete old and unwanted records permanently.
- Update – one: This is used to update the data pertaining to one specific record of file.
- Retrieve – few: This is used to retrieve more than one record at once based on some selection criteria.
- **Example :** From x date to y date

File Management System (FMS)

- File management system provide service to every application that user executes. Any application under the sun needs to co-operate with the file management system to access required file.

As a part of co-operation, FMS performs some set of operation. They are:

- To support all data management needs of user application.
- To assume the validity of data that is in file.
- To deliver the best possible result in best possible time from systems as well as users point of view.
- Support al I/O operation's maximum possible I/O device.
- To eliminate the use of discarded data.
- To provide all I/O routine support.
- To provide I/O support to multiple users in all multi user system.

Most General Purpose Minimal Set of Necessary Require-ments are :

- Must be able to do all operations on a file by add, delete, create, read, write etc.
- Provide security to files from unauthorized access.
- One must have all facilities to restructure the file as per the requirement.
- Movement of data form one file to other must be possible.
- Every file operation must be supported by backup and other operation like:
  - Recovery, if file is deleted.
  - Repairing, if file is damaged.
  - Owner must be able to access file by name rather numeric.

Overview :

- All computer applications need to store and retrieve information. While a process is running, it can store a limited amount of information within its address space. For some applications the size is adequate, but for most of the applications like banking, corporate record keeping, it is very small.
- Apart from the above hitch there are other problems with keeping information within a process address space. These are :
  - When the process terminates, the information is lost, while the database applications require that the information need to be retained.
  - Another issue is that it is frequently necessary for multiple processes to access the information at the same time, but if the information is stored inside the address space of a process, only that process can access that information.

- The usual solution to all these problems is to store information on disks and other external media is called file.
- A file is a named collection of related information that is recorded on the secondary storage. Information stored in files must be persistent i.e. not to be affected by process creation or termination.
- Files are managed by operating system. How they are structured, named, accessed, used, protected and implemented are major topics in O.S. design. As a whole, that part of the operating system dealing with files is known as file system.
- If we look at it from users point of view then important aspects will be how files appear to them. i.e. What constitutes file, how files are named and protected, what operations are allowed on files and so on.

5.6.2 File Naming

- Files are an abstraction mechanism. They provide a way to store information and read it back. This must be done in such a way as to shield user from details of how and where the information is stored.
- Most important characteristic of any abstraction mechanism is the way objects being managed are named 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.
- Mostly all operating systems allow strings of one to eight letters as legal file names. Frequently, digits and special characters are also permitted. Say urgent !
- Some systems distinguish between upper case and lower case letters e.g. UNIX, while others do not e.g. MS-DOS. Thus, in UNIX, files with names URGent, urgent, URGENT are treated differently.
- Many OS support two part file names, separated by a period '. '. The part following the period is called file extension. In MS-DOS, file names are 1-8 characters, plus an optional extension of 1 to 3 characters.

Let us go through some file extensions and their meanings :

Table 5.1

Extension	Meaning
File.back	Backup file
File.C	Source program
File.gif	Compuserve graphical inter change format image
File.html	Worldwide web Hyper Text Markup Language document
File.mpg	Movie encoded with MPEG format
File.txt	General text file

In some cases, file extensions are just conventions and are not enforced. On the other hand, a C compiler may actually insist that the files it is to compile end in "C" and it may refuse to compile if they do not.

5.6.3 File Attributes

- Every file has a name and its data. In addition, all OS associate other information with each file i.e. date and time the file was created and files size. These extra items are called files attributes.
- Attributes may vary from one OS to other, but most common are :
  - **Name :** The symbolic file name only information kept in human readable form.
  - **Type :** This information is useful for systems, which supports different readable form.
  - **Location :** It is a pointer to a device and to the location of the file on that device.
  - **Size :** The current size of the file and maximum allowed sizes are included in this attribute.
  - **Protection :** Access control information controls who can do reading, writing, executing and so on.
  - **Time, Date and User Identification :** This information may be kept for (a) creation, (b) last modification, (c) last use. These data can be useful for protection, security and usage monitoring.

The information about all files is kept in the directory structure which resides on secondary storage. It may take 16 to 800 bytes to record this information.

5.6.4 File Operations

Files exists, to store information and allow it to be retrieved later. Different systems provide different operations to allow storage and retrieval.

Let us discuss most common system calls relating to files :

- **CREATE :** 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.
- **OPEN :** Before using a file, a process must open it. The purpose of this 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 processes 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 is read from file. Byte comes from current position. The caller must specify how much data is needed and must also provide a buffer to put them in.
- **WRITE :** Data is written to file at current position. If the current position is the end of the file, the files size increases. If the current position is in the middle of the file, existing data is overwritten and lost forever.
- **APPEND :** It is a restricted from of WRITE. It can only add data to the end of file.
- **SEEK :** For random access files, a method is needed to specify from where to take the data. SEEK, System call repositions, the pointer to the current position to a specific place in the file.
- **GET ATTRIBUTES :** Processes often need to read file attributes to do their work.
- **SET ATTRIBUTES :** Some of the attributes are user settable and can be changed after the file has been created. This system call makes it possible.
- **RENAME :** It frequently happens that a user needs to change the name of an existing file. This system call makes it possible.

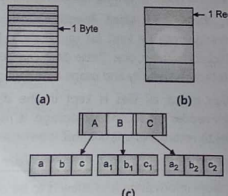


Fig. 5.14

- Fig. 5.14 (a) is an unstructured sequence. The OS does not know what is in the file. All it sees are bytes. UNIX and MS-DOS use this approach. Having the OS regard files as nothing more than byte sequences provides maximum flexibility.
- User programs can put anything they want in files and name them any way that is convenient.
- As shown in Fig. 5.14 (b) 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 read operation requires one record and write operation overwrites or appends one record.
- An old system that viewed files as a sequence of fixed length record was CPIM. Now-a-days, the idea of a file as a sequence of fixed length records is pretty much gone.
- Third kind of file structure consists of a tree of records, not necessary all of them of 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.

5.6.5 File Types

- Many OS support several types of files. UNIX and MS-DOS have regular files and directories. UNIX also has character and block special files.
- Regular files are the ones that contain user information. Directories are system files for maintaining the structure of the file.
- Character special files are related to input/output and used to model serial I/O devices as printers.
- Block special files are used to model disks.
- Regular files are generally either ASCII files or binary files.
  - ASCII files consists of line of text. In some systems, each line is terminated by a carriage return and in others, lime feed character is sued. Advantage of ASCII files is that they can be displayed and printed as it is and they can be edited with ordinary text editor.
  - Other files are binary files. Listing them on a printer gives an incomprehensible listing full of what is apparently random junk. Usually they have some internal structure. Second example of a binary file is an archive It consist of collection of library procedures compiled but not linked.

- All OS must recognize one file type, their own executable file, but some recognize more. E.g. when a WINDOWS user double clicks on a file name an appropriate program is launched with the file as a parameter. The OS determines which program to run based on the file extension.
- Having strongly typed files like this causes problems whenever user does anything that the system designer did not expect.
- E.g. a system in which program output files have type dat (data fiels). If a user writes a program formatter that reads a pas file, transforms it and then write transformed file as output. The output file will be of type dat.
- If the user gives this file to Pascal compiler to compile it, the system will refuse because it has the wrong extension.

5.6.6 File System Organisation

- The term file organisation is referred to the logical structuring of the records determined by the way in which they are accessed. It should not be confused with the physical storage of the file in some types of storage media. The physical organisation of the file on secondary storage depends on blocking and file allocation strategy.
- It is a high level decision to specify a system of file organisation for a computer software program or a computer system designed for a particular purpose. Performance is high on the list of priorities for this design process, depending on how the file is being used.
- The design of the file organisation usually depends mainly on the system environment.
- The instance, factors such as whether the file is going used for transaction oriented processes like OLTP or data warehousing or whether the file is shared among various processes like those found in a typical distributed system or standalone.
- It must also be asked whether the file is on a network and used by a number of users and whether it may be accessed internally or remotely and how often it is accessed.
- However, all things considered the most important considerations might be :
  - Rapid access to a record or a number of records which are related to each other.
  - The adding, modification or deletion of records.
  - Efficiency of storage and retrieval of records.
  - Redundancy, being the method of ensuring data integrity.
- To provide an efficient and convenient access to the disk, the OS imposes a file system to allow data to be stored, located and retrieved easily.
- File system is composed of many levels. Each level in the design uses the features of lower levels to create new features for use by higher levels.
- The lowest level I/O control consists of device drivers and interrupt handlers to transfer information between the memory and disk system.
- The basic file system needs to issue generic commands to the appropriate device driver to read and write physical blocks on disks.
- File organisation module knows about files and their logical block as well as physical blocks. This module also includes the free space manager, which tracks unallocated blocks and provides these blocks to file organisation module when requested.
- Logical file system uses directory structure to provide the file organisation module with the information it needs, given a symbolic file name. This module is also responsible for protection and security.
- To create a new file, an application program calls the logical file system. The logical file system knows the format of the directory structures. To create a new file, it reads the appropriate directory into memory, updates it with the new entry and writes it back to the disk.

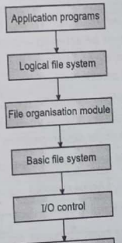


Fig. 5.15 : Layered file system

- When a file has been created, it can be used for I/O for each I/O operation, the directory structure can be searched to find the file, its parameter checked, its data blocks located finally operations on those data blocks are performed. Each operation involves lots of overhead.
- Before the file can be used for I/O it must be opened. Directory structure is searched for desired file entry.
- Once the file is found associated information as size, owner, access, permissions, etc. are copied into a table in memory known as open file table containing information about currently opened files as shown in the Table 5.2 below :

Table 5.2

Index	File Name	Permissions	Access Dates	Pointer to Disk Block
0	Program C	rw rw rw	--	--
1	Mail.txt	rw	--	
2	.	.	.	.
	.	.	.	.

- The index into the open file table is returned to user program whenever a file is referenced. All further references are made through the index rather than with the symbolic name. The name given to index varies. UNIX systems refer to it as a file descriptor and windows /NT as a file handle.
- When the file is closed by all users that have opened it, the updated file information is copied back to the disk based directory structure.

5.6.7 File Access

- Files store information. When it is used, this information must be accessed and read into computer memory. There are several ways that the information in the file can be accessed.
- There are various methods supported by various systems, choosing the right one for particular application is a major design problem.
- Five types of file access methods are possible.

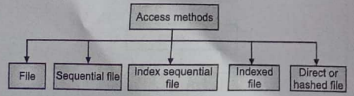


Fig. 5.16

1. File

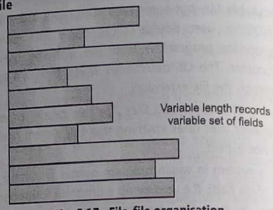


Fig. 5.17 : File-file organisation

- Least complicated form of file organisation. Data are collected in the order in which they arrive. Each record consists of one burst of data. Records may have different fields or similar filed in different orders.
- Each field should be self-describing including field name as well as a value. Because there is a no structure to the pile file, record access is by exhaustive search.
- If we wish to find a record that contains a particular field with a particular value, it is necessary to examine each record in a file until the desired record is found.

2. Sequential Files (Access)

- In this type of file, a fixed format is used for all records. All records are of same length, consisting of the same number of fixed length fields in a particular order.
- The key field uniquely identifies the record. Sequential files are typically used in batch applications.
- Access requires the sequential search of the file for a key match. If the entire file or a large portion of the file can be brought into main memory at one time, more efficient search techniques are possible.
- Considerable processing and delay are encountered to access a record in a large sequential file.
- This mode of access is most common for example editors and compilers usually access fields in this fashion.
- Sequential access is based on a tape model of a file and works as well on sequential access devices as it does on random access ones.

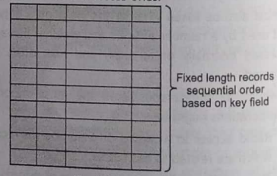


Fig. 5.18 : Sequential file

3. Index Sequential

- This approach overcomes the limitations of sequential file. Like sequential file, in this approach records are organised in sequence based on a key field.
- Two new features are added to this approach :
  1. An index to the file to support random access.
  2. An overflow file.
- The index provides look up capability which searching of record easier while the overflow is similar to the log file used with a sequential file but is integrated so that records in the overflow file are located by following a pointer from their predecessor record.
- In the simplest form, the structure of indexed sequential file would have a single level of indexing. This index is a simple sequential file. Each record in the index file consists of two fields : a key field and a pointer to the main file.
- This is how a key search is performed : the search key is compared with the index keys to find the highest index key coming in front of the search key, while a linear search is performed from the record that the index key points to, until the search key is matched or until the record pointed to the next index entry is reached.
- Regardless of double file access (index data) required by this sort of search, the access time reduction is significant compared with sequential file searches.
- Lets examine, for sake of example, a simple linear search on a 1000 record sequentially organised file An average of 500 key comparisons are needed (and this assumes the search key are uniformly distributed among the data keys).
- However, using an index evenly spaced with 80 entries, the total number of comparisons is reduced to 50 in the index file plus 50 in the data file : a five to one reduction in the operations count !
- The index sequential file greatly reduces the time required to access a single record without sacrificing the sequential nature of the file.

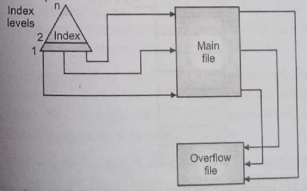


Fig. 5.19

4. Index File

- In this type of access, index for the file is constructed. The index contains pointer to various blocks.
- To find an entry in the file, we first search the index and then use the pointer to access the file directory and to find the desired entry.
- E.g. a retail price file lists the Universal Product Codes (UPC) for items, with associated prices. Each entry consists of :
  - 8 digit UPC
  - Six digit price for 16 byte entry.
- If our disk has 824 bytes, we can store 64 entries per block, a file of 120,000 entries would occupy about 2000 blocks. By keeping the file sorted by UPC, we can define an index consisting of first UPC in each block.
- This index would have 2000 entries of 8 digits each. To find the price of particular item, we can search the index.
- This structure allows us to search large file doing little i/o with large files, the index file itself may become too large to be kept in memory one solution is to create an index for the index file.
- The primary index file would contain pointers to secondary index files which would point to actual data items.
- To find a particular item, we first make a binary search of the master index which provides the block number of secondary index. This block is read in and again a binary search is used to find the block containing the desired record. Finally this block is read sequentially.
- This is implemented in IBM's indexed sequential access method.
- Fig. 5.20 shows structure implemented by VMS index and relative files.

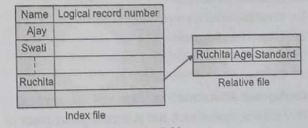


Fig. 5.20

5. Direct Access

- A file is made up of fixed length, logical records that allow programs to read and write records rapidly in no particular order.

- This method is based on a disk model of a file, since disk allows random access to any file block.
- For direct access, a file is viewed as a numbered sequence of blocks or records. A direct access file allows arbitrary blocks to be read or written.
- Direct access files are of great use of immediate access to large amount of information. Databases are often of these types.
- The block number provided by the user to the OS is normally a relative block number. A relative block number is an index relative to the beginning of the file. The use of relative block number allows the OS to decide where the file should be placed.

5.6.8 Implementing Files

- File system is implemented on the disk and the memory. How to implement the file system, it writes according to the operating system and file system.
- If the file system is implemented on the disk, it contains following information :
  - Boot Block Control :
    - (a) OS requires some information while system boot up.
    - (b) If the disk is divided into number of partitions, the OS is scored in the first partition of the disk.
  - Partition Block Control :
    - (a) It contains detailed information about the partition.
    - (b) It includes block size, free block pointers, free block count and free PCB count.
    - (c) In UNIX file system, it is called as superblock, in NTFs, it is master file table.

5.6.9 File Implementation Methods

Most important issue in implementing file storage is keeping track of which disk blocks goes with which file. Various methods are used in different operating systems.

1. *Contiguous allocation.*
2. *Linked list allocation.*
3. *Linked list allocation using index.*
4. *I-nodes.*

1. *Contiguous Allocation*

- *This scheme stores each file as a contiguous block of data on the disk. The advantages of this scheme are as follows:*
  - *It is simple to implement because keeping track of where a files blocks are is reduced to remembering just address of first block.*
  - Good performance because entire file can be read from the disk in a single operation.
- But everything doesn't come for free and so this scheme also has its share of disadvantages :
  - It is not feasible unless the maximum file size is known at the time the file created, without this information the OS doesn't know how much disk space to reverse.
  - Another disadvantage is the fragmentation of the disk space is wasted which otherwise would how been used.

2. Linked List Allocation

- Another method to store files is to keep each one as a linked list of disk blocks.

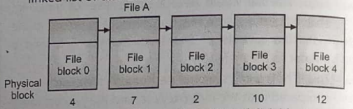


Fig. 5.21 : Storing file as a linked list of disk blocks

The first word of each block is used as a pointer to next one. The rest of the block is for data.

- Every disk block can be used in this method. No space is lost to disk fragmentation. It is sufficient for directory entry to just store the disk address of the first block and then rest can be found from there.
- On the other hand, reading a file sequentially is faster than the random access. In addition, some amount of space is wasted in storing pointers.

3. Linked List Allocation Using Index

- The disadvantage of linked list allocation can be eliminated by taking the pointer word from each disk block and patting it in a table or index in memory. The previous Fig. 5.21 looks like as shown in this Fig. 5.22.

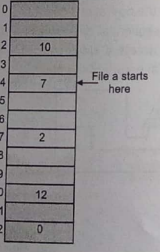


Fig. 5.22 : Linked list allocation using a table in main memory

- File A uses disk blocks 4, 7, 2, 10 and 12.
- Using this organisation, the entire block is available for data. Random access is much easier. It is sufficient for the directory entry to keep the starting block number and still be able to locate all the blocks.
- Main disadvantage of this method is that entire table must be in memory all the time, to make it work.

4. I-Nodes

- Another method to keep track of which block belong to which file is to associate with each file a little table called i-node. It lists the attributes and disk addresses of the files block.

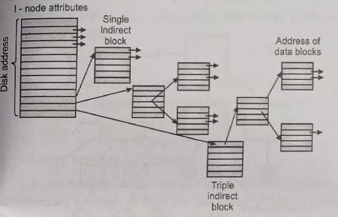


Fig. 5.23

- First few disk addresses are stored in the i-node itself so for small files all the necessary information is in the i-node. For larger files, one of the addresses in the i-node, called double indirect block, contains the address of a block that contains a list of single indirect block. Each of these single indirect blocks points to a few hundred data blocks. If this is not enough triple indirect blocks can be used.

5.7 FILE DIRECTORIES

- The file systems of computers can be extensive. Some systems store thousands of files on hundreds of GB of disk. Managing such a huge data may become cumbersome without proper organisation of data. So, we need some mechanism to organise the data.
- To keep track of files, file systems normally have directories, which in many systems are themselves files. Now let us study their structure, properties and possible operations on directories.
- The information about all file (s) is kept in the directory structure which also resides on secondary storage.
- Normally, directory entry consists of the files names and its unique identifier.
- Identifier in turn locates the other file attributes.
- It may take more than a kilobyte to record this information for each file.
- In a system with many files, the size of the directory itself may be megabytes.
- The directory is itself a file, owned by the OS and accessible by various file management routines.
- Generally, directory is a set of logically associated files and other directories of files (which are known as sub-directory).

Hence, it is a systematic organisation of files to ease their use.

5.7.1 Directory Structure

There are various layers in which directories can be arranged, depending on the amount of data. It can be :

1. Single level directory structure.
2. Two level directory structure.
3. Tree structured directory.

1. Single Level Directory Structure

- This is the simplest directory structure. A directory typically contains a number of entries, one per file.

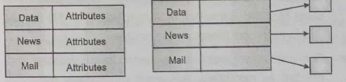


Fig. 5.24 : Single level directory structure

- This type of structure has various entries : Each entry contains the file name, the file attributes and the disk addresses where the data is stored or where directory holds file name and pointer to other data structure where attributes and disk addresses are found. When a file is opened, the OS searches its directory until it finds the name of the file to be opened.
- It then extracts the attributes and disk addresses directly from the entry and puts them in a table in main memory.
- But, this structure has its limitations, when the number of files increase or when there is more than one user. Since all files are in same directory, they must have unique names. If multiple users are calling their data

file by same name i.e. say "test", then unique name rule is violated. Even with single user, as the number of users increases, it becomes difficult to remember the names of all the files, in order to create files with unique name.

2. Two Level Directory Structure

- In order to overcome the disadvantage of single level structure, another idea is to have a single directory for all files in the entire system is to have one directory per user.
- In this two level directory, each user has his/her own User File Directory (UFD). When a user job starts, the systems root directory is searched. The root directory is indexed by user name which points to user directory.

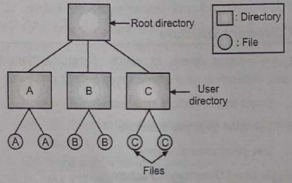


Fig. 5.25 : Two level directory structure

- When user refers to a particular file only his own user directory is searched. Thus, different users may have same files with same name, as long as all the file names within each user directory are unique.
- The user directories themselves must be created and deleted as necessary. A special program is run with appropriate user name and account information. The program creates a new user file directory and adds an entry for it to the master file directory.
- This structure has its own limitations. This structure isolates one user from other. This isolation is advantageous if users are completely independent, but has a disadvantage when the users want to co-operate on some task and to access one anothers file.
- A two level directory structure can be thought of as a tree or an inverted tree of height 2. The root of the tree is root directory.

3. Tree Structured Directory

- Two level directory structure although eliminates name conflicts among users but is not satisfactory for users with large number of files. It is quite common for users to group their files together in logical ways.
- Consider an example, a student has a collection of files that together forms entire book of system programming, second collection of files which together forms the programs of system programming and so on. Same way is needed to group these files together in flexible ways chosen by the user.
- What is needed is general hierarchy with this approach, each user can have as many directories as are needed so that files can have grouped together in natural ways. A directory contains a set of files or subdirectories. A directory is simply another file, but treated in a special way. All directories have the same internal name one bit in each directory entry defines the entry as a file (0) or as sub-directory (1).

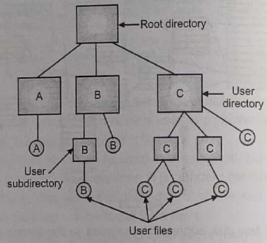


Fig. 5.26 : Tree structured directory

- An interesting, policy decision in a tree structured directory is how to handle the deletion of a directory. If a directory is empty, its entry in its containing directory can be deleted. But if the directory to be deleted is not empty, but contains several files or sub-directories.

There are two approaches possible :

1. One approach is as employed is MS-DOS, is not to delete a directory unless it is empty. Thus, to delete a directory, all files in that directory must be deleted. If there are any sub-directories, this procedure must be applied recursively to them.
2. Another approach, which is used in UNIX, is to provide option that, when a request is made to delete a directory, all that directories files and sub-directories be deleted.

With a tree structured directory system, users can access, in addition to their files, the files of other users.

5.7.2 Path Names

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

1. Absolute Path Name.
2. Relative Path Name.

1. Absolute Path Name

- Consists of the path from the root directory to the file e.g. the path \user\ast\maibox means that the root directory contains the sub-directory user, which in turn contains the sub-directory ast, which contains the file mailbox.
- Absolute path names always start at the root directory and are unique.

2. Relative Path Name

- This is used along with the concept of working directory or current directory. A user can designate one directory as the current working directory in which case all path names not beginning at root directory are taken to the working directory.
- E.g. in relative path name if the current working directory is \user\ast then the file whose absolute path is \user \ast \mailbox can be referred as mailbox.
- Some programs need to access a specific file without regard to what the working directory is. In that case they should always use absolute path names. E.g. a spelling checker might need to read \usr\lib\dictionary to do its work. It should use the full absolute path name because it does not know what the working directory will be when it is called.
- But by explicity, changing the working directory, it knows for sure where it is in the directory tree, so that it can use relative paths.
- In many systems, each process has its own working directory, so when a process changes its working directory and later exists, no other processes are affected and no traces of change are left behind in the file system. So it is convenient for a process to change its working directory.
- On the other hand, if a library procedure changes the working directory and does not change back to where it was when it is finished, the rest of the program may not work, since its assumption about where it is, may be invalid. So library procedures rarely changes its working directory.
- Most OS that support a hierarchical directory system have two special entries in every directory. "." and ".." pronounced as dot and dot dot.

  Dot refers to : current directory.

  dot dot refers to : its parent directory.

5.7.3 Directory Operations

The system calls for managing directories has more variations from system to system than system calls for files.

CREATE : A directory is created. It is empty except for dot and dot dot, which are put there automatically by the system.

DELETE : A directory is deleted. Only an empty directory can be deleted.

OPENDIR : Directories can be read. Before a directory can be read it must be opened. E.g. to list all the files in a directory, a listing program opens the directory to read out the names of the files it contains.

CLOSEDIR : When a directory has been read, it should be closed to free up internal table space.

READDIR : This call returns the next entry in an open directory. Formerly, it was possible to read directories using usual READ system call, but this approach forces the programmer to know with internal structure of directory. But READDIR, returns one entry in a standard format.

RENAME : In many respects, directories are same like files. So directories can also be renamed like files.

LINK : It is a technique that allows file to appear in more than one directory. This system call specifies an existing file and a path name and creates a link from the existing file to the name specified by the path.

UNLINK : A directory entry is removed. If the file is being unlinked is only present in one directory, it is removed from the file system.

5.8 SECONDARY STORAGE MANAGEMENT

A file is a collection of blocks. The OS or file system is responsible for allocating blocks to files, it raises two issues :

(i) Space on secondary storage must be allocated to files.

(ii) It is necessary to keep the space available for allocation.

5.8.1 File Allocation

- When a new file is created, the space allocation for it is done in two ways : (a) Pre-allocation Allocation, (b) Dynamic Allocation.

- Space is allocated to files as one or more contiguous units, which is referred as portions. The size of portion ranges from a single block to the entire file.
- One question arises in our mind is what size of portion should be used for file allocation ?
- **Portion Size :**
  - ➢ At one extreme, a portion large enough to hold the entire file is allocated. At other extreme, space on the disk is allocated one block at a time.
  - ➢ In choosing a portion size, there is a trade off between efficiency from the point of view of a single file versus overall efficiency.
- There are two alternatives possible variable, large contiguous portions blocks.
  1. **Variable Large Contiguous Portions :** It provides better performance. The variable size avoids waste and file allocation tables are small but the space is hard to reuse.
  2. **Blocks :** Small fined or portions provide greater flexibility. They may require large tables or complex structures for their allocation.
- Either option is compatible with pre-allcoation or dynamic allocation. In the case of variable, large contiguous portions of a file is preallcoated one contiguous group of blocks.
- In case of blocks, all of the portions required are allocated at one time.
- With variable size portions use need to be concerned with the fragmentation of free space.

5.8.2 Pre-allocation v/s Dynamic Allocation

- A pre-allocation policy requires that the maximum size of a file be declared at the time of file creation request.
- But in many applications it is difficult to estimate reliably the maximum potential size of a file. In those cases, users and programmers tend to overestimate file size, so as not to run out of space.
- But this wastes lots of secondary storage space. Therefore, there are advantages to the use of dynamic allocation, which allocates space to a file in portions as needed.

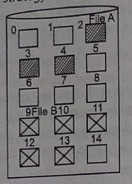
Now Let us Study File Allocation Methods :

These methods are given as

1. Contiguous Allocation.
2. Chained Allocation.
3. Index Allocation.

1. Contiguous Allocation

- Single contiguous set of blocks is allocated to a file at the time of file creation. Thus it is pre-allocation strategy using variable size block.



File allocation table

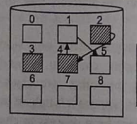
File name	Start block	Length
File A	2	3
File B	9	5

Fig. 5.27

- The FAT needs only single entry for each file showing starting block and length of the file. In this method, it is easy to retrieve single block. E.g. if file starts at block b and i^{th} block of the file is wanted, its location on secondary storage is (b + i – 1).
- External fragmentation makes it difficult to find contiguous blocks of space of sufficient length. From time to time, it will be necessary to perform a compaction algorithm to free up additional space on disk.
- Since this method uses pre-allcoation strategy, it is necessary to declare the size of the file at the time of creation.

2. Chained Allocation

- Allocation is on an individual block basis. Each block contains a pointer to the next block in the chain. Even the FAT needs only a single entry for each file, showing starting block and length of the file.



File allocation table

File name	Start block	Length
File A	2	3
---	---	---
---	---	---

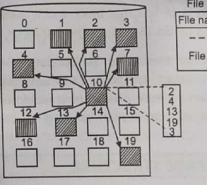
Fig. 5.28

- Any free block can be added to the chain, no external fragmentation. But if use have to bring in several blocks of a file at a time, as in sequential processing, then a series of accesses to different parts of the disk are required. So one consequence of chaining is there is no accommodation of the principle of locality.

3. Index Allocation

- This scheme addresses man of the problems of contiguous and chained allocation. In this case FAT contains a separate one level index for each file, the index has one entry for each portion allocated to the file.
- File indexes are not physically stored as part of the FAT. The file index for a file is kept in a separate block and the entry for the file in the fat points to the block.
- Allocation may be on the basis of either fixed size blocks.
- Allocation by fixed size blocks eliminates external fragmentation whereas allocation with variable size blocks improves locality.

File consolidation may be done from time to time. File consolidation reduces the size of index in variable size blocks. Index allocation supports both sequential and direct access to the file.

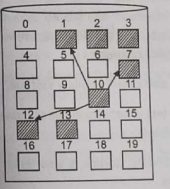


File allocation table

File name	Index block
---	---
File B	10

2
4
13
19
3

Fig. 5.29 : Index allocation with fixed size blocks



File allocation table

File name	Index block
---	---
File B	10

Start block	Length
1	3
12	2
7	1

Fig. 5.30 : Index allocation with variable size blocks

File Allocation Methods : Let us summarise them from the secondary storage point of view :

Table 5.3

	Continuous	Chained	Indexed	
Preallocation	Necessary	Possible	Possible	
Fixed or variable size portions ?	Variable	Fixed blocks	Fixed blocks	Variable
Portion size	Large	Small	Small	Medium
Allocation Frequency	Once	Low to high	High	Low
Time to allocate	Medium	Long	Short	Medium
File allocation table size	One entry	One entry	Large	Medium

...Conti.

Another point to consider in secondary storage management is free space management.

5.8.3 Free Space Management

- Files are normally stored on disk. So management of disk space is a major concern for the file system designers. Since there is a limited amount of disk space, it is necessary to reuse the space from deleted files for new files, if possible.
- To keep track of free disk space, the system maintains a free space list. The free space list records all disk blocks that are free i.e. blocks are not allocated to same file or directory.
- To create a file, we search the free space list for the required amount of space and allocate that space to the new file. This space is then removed from the free space list.
- Storing a file as a contiguous sequence of bytes, has obvious problem of grouping file in size. For this reason, all file systems chop files up into fixed size blocks that need not be adjacent. Once we decide to store files in fixed size blocks, usual question arises in our mind is how big the block should be ? We all know that disks are organised into sector, track and cylinder. So these are obvious candidates for the unit of allocation.
- Having large unit of allocation as cylinder means that every file, even a 1 byte file, ties up an entire cylinder. E.g. medium file size in UNIX environment is about 1k, so allocating a 32 k cylinder for each file would waste 97% of the total disk space.
- On the other hand, using small allocation unit means that each file will consist of many blocks. Reading each block requires a seek and a rotational delay. So reading a file consisting of many blocks will be slow. The usual compromise is to choose blocks i.e. of 512, 1K or 2K bytes. If a 1k block size is chosen on a disk with a 512 byte sector size then file system will always read or write two consecutive sectors and treat them as a single indivisible unit.

- Whatever decision is made, it should probably be re-evaluated periodically. Once a block size is chosen, next issue is how to keep track of free blocks. For this stuff, lets get back to the free list concept, when a file is deleted its disk space is added to the free space list.

The Free List can be Implemented in Many Ways :

1. Bit Vector

- Usually free space list is implemented as a bit vector. Each block is represented by 1 bit. If the block is free, the bit is 1; if the block is allocated, the bit is 0.
- E.g. consider a disk where blocks 2, 3, 5, 8 are free and rest are allocated. The free space bit would be as follows : 00188081.
- This approach is relatively simple and efficient to find the first free block or n consecutive free blocks on the disk. Apple Macintosh OS uses the bit vector method to allocate disk space.
- Bit vector are inefficient unless the entire vector is kept in main memory. Keeping it in memory is possible for smaller disk, but not for larger ones.

2. Linked List

- Another approach is to link together all the free disk blocks, keeping a pointer to the first free block in a special location. On the disk and coching it in memory. First block contains a pointer to the next free disk block and so on.
- With our previous example i.e. 2, 3, 5, 8, 8 are free blocks and rest are allocated we would keep a pointer to block 2 as the first free blocks. Block 2 would contain pointer to block 3 and so on.

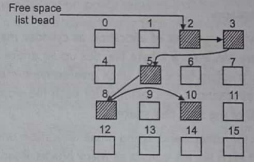


Fig. 5.31 : Link list

- In this approach, we need to traverse list and to traverse the list, we must read each block which requires substantial I/O time.
- But traversing the free list is not a frequent task. The OS simply needs a free block so that it can allocate that block to a file. So the first block in the free list is used.

3. Grouping

- Another approach is to store addresses of n free blocks in the first free block. The first n-1 of these blocks are actually free. The last block contains the addresses of another n free blocks and so on.
- Advantage of this approach is that the addresses of large number of free blocks can be found quickly.

4. Counting

- Last approach is to take advantage of the fact that, generally many contiguous blocks may be allocated or freed simultaneously, particularly when space is allocated with contiguous allocation algorithm or through clustering.
- Rather than keeping a list of n free disk addresses, we can keep address of first free block and n number of free contiguous blocks that follow the first block.
- Each entry in the free space list consists of a disk address and a count. Although each entry requires more space than a simple disk address would require, the overall list will be shorter, as long as count is greater than 1.

5.9 MASS STORAGE STRUCTURE

Secondary storage devices are those devices whose memory is non volatile, meaning, the stored data will be intact even if the system is turned off. Here are a few things worth noting about secondary storage.

- Secondary storage is also called auxiliary storage.
- Secondary storage is less expensive when compared to primary memory like RAMs.
- The speed of the secondary storage is also lesser than that of primary storage.
- Hence, the data which is less frequently accessed is kept in the secondary storage.
- A few examples are magnetic disks, magnetic tapes, removable thumb drives etc.

Magnetic Disk Structure

In modern computers, most of the secondary storage is in the form of magnetic disks. Hence, knowing the structure of a magnetic disk is necessary to understand how the data in the disk is accessed by the computer.

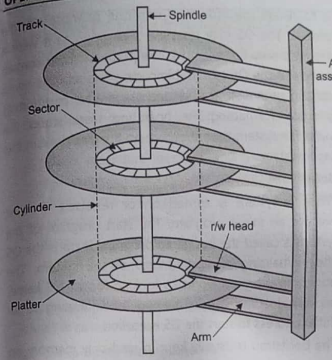


Fig. 5.32

Structure of a Magnetic Disk

- A magnetic disk contains several platters. Each platter is divided into circular shaped tracks. The length of the tracks near the centre is less than the length of the tracks farther from the centre. Each track is further divided into sectors, as shown in the figure.
- Tracks of the same distance from centre form a cylinder. A read-write head is used to read data from a sector of the magnetic disk.

The speed of the disk is measured as two parts:

1. **Transfer Rate:** This is the rate at which the data moves from disk to the computer.
2. **Random Access Time:** It is the sum of the seek time and rotational latency.

- Seek time is the time taken by the arm to move to the required track. Rotational latency is defined as the time taken by the arm to reach the required sector in the track.
- Even though the disk is arranged as sectors and tracks physically, the data is logically arranged and addressed as an array of blocks of fixed size. The size of a block can be 512 or 1024 bytes. Each logical block is mapped with a sector on the disk, sequentially. In this way, each sector in the disk will have a logical address.

5.10 DISK ATTACHMENT IN OPERATING SYSTEM

Disk Attachment

There are two ways for computers to access disc storage. One method is to use I/O ports, also known as host-attached storage, on small systems. Another method is through a remote host in a distributed file system, which is known as network-attached storage.

Host-Attached Storage

- Host-attached storage (HAS) is storage accessed via local I/O ports. These ports make use of various technologies. IDE or ATA is the I/O bus architecture used by most desktop PCs. This architecture allows for up to two drivers per I/O bus. SATA is a new related standard that has simplified cabling. High-end workstations and servers typically use more advanced I/O architectures such as Small Computer System Interface (SCSI) and fibre channel (FC).
- SCSI (Small Computer System Interface) is bus architecture. Its physical medium is typically a ribbon wire with many conductors. A maximum of 16 devices may be attached to the bus using the SCSI protocol. The devices typically comprise of one controller card (SCSI initiator) on the host and up to 15 storage devices (SCSI targets). A SCSI disk is a typical SCSI target. Although, the protocol permits every SCSI target to address up to 8 logical units. Logical unit addressing is widely used to direct commands to the RATD array or portable media library components.
- A Fibre Channel is a high-speed serial architecture that may use optical fibre or a four-conductor copper wire. It comes in two varieties. One type of fabric is a big switched fabric with a 24-bit address space. This variant is projected to take the lead in the future and will serve as the foundation for storage-area networks (SANs). Many hosts and storage devices may be connected to the fibre due to the large address space and switched nature of the communication that provides better flexibility in I/O communication. Another PC version is an arbitrated loop (FC-AL), which may address up to 126 devices, including drives and controllers.
- As host-attached storage, a wide range of storage devices are suitable. Hard disc devices, RAID arrays, CDs, DVDs, and tape drives are among them. The I/O commands that start data transfer to a host-attached storage device can read and write logical data blocks routed to specially designated storage units.

5.11 DISK MANAGEMENT IN OPERATING SYSTEM

- The operating system is responsible for various operations of disk management. Modern operating systems are constantly growing their range of services

and add-ons, and all operating systems implement four essential operating system administration functions. These functions are as follows:

- ➢ Process Management
- ➢ Memory Management
- ➢ File and Disk Management
- ➢ I/O System Management

- Most systems include secondary storage devices (magnetic disks). It is a low-cost, non-volatile storage method for data and programs. The user data and programs are stored on different storage devices known as files. The OS is responsible for allocating space to files on secondary storage devices as required.
- It doesn't ensure that files are saved on physical disk drives in contiguous locations. It is highly dependent on the provided space. New files are mostly stored in various locations if the disk drive is full. On the other hand, the OS example file hides that the file is divided into several parts.
- The OS requires tracking the position of the disk drive for each section of every file on the disk. It may include tracking many files and file segments on a physical disk drive in some circumstances. Furthermore, the OS must be able to identify each file and conduct read and writes operations on it according to the requirements. As a result, the OS is mainly responsible for setting the file system, assuring the security and reliability of reading and writing activities to secondary storage, and keeping access times consistent.

Disk Management of the OS includes the various aspects, such as:

1. Disk Formatting

- A new magnetic disk is mainly a blank slate. It is platters of the magnetic recording material. Before a disk may hold data, it must be partitioned into sectors that may be read and written by the disk controller. It is known as physical formatting and low-level formatting.
- Low-level formatting creates a unique data structure for every sector on the drive. A data structure for a sector is made up of a header, a data region, and a trailer. The disk controller uses the header and trailer to store information like an error-correcting code (ECC) and a sector number.
- The OS must require recording its own data structures on the disk drive to utilize it as a storage medium for files. It accomplishes this in two phases. The initial step is to divide the disk drive into one or more cylinder groups. The OS may treat every partition as it were a separate disk. For example, one partition could contain a copy of the OS executable code, while another could contain user files. The second stage after partitioning is logical formatting. The operating store stores the initial file system data structure on the disk drive in this second stage.

2. Boot Block

- When a system is turned on or restarted, it must execute an initial program. The start program of the system is called the bootstrap program. It starts the OS after initializing all components of the system. The bootstrap program works by looking for the OS kernel on disk, loading it into memory, and jumping to an initial address to start the OS execution.
- The bootstrap is usually kept in read-only memory on most computer systems. It is useful since read-only memory does not require initialization and is at a fixed location where the CPU may begin executing whether powered on or reset. Furthermore, it may not be affected by a computer system virus because ROM is read-only. The issue is that updating this bootstrap code needs replacing the ROM hardware chips.
- As a result, most computer systems include small bootstrap loader software in the boot ROM, whose primary function is to load a full bootstrap program from a disk drive. The entire bootstrap program can be modified easily, and the disk is rewritten with a fresh version. The bootstrap program is stored in a partition and is referred to as the boot block. A boot disk or system disk is a type of disk that contains a boot partition.

3. Bad Blocks

- Disks are prone to failure due to their moving parts and tight tolerances. When a disk drive fails, it must be replaced and the contents transferred to the replacement disk using backup media. For some time, one or more sectors become faulty. Most disks also come from the company with bad blocks. These blocks are handled in various ways, depending on the use of disk and controller.
- On the disk, the controller keeps a list of bad blocks. The list is initialized during the factory's low-level format and updated during the disk's life. Each bad sector may be replaced with one of the spare sectors by directing the controller. This process is referred to as sector sparing.

5.12 SWAP-SPACE MANAGEMENT IN OPERATING SYSTEM

- Swapping is a memory management technique used in multi-programming to increase the number of processes sharing the CPU. It is a technique of removing a process from the main memory and storing it into secondary memory, and then bringing it back into the main memory for continued execution. This action of moving a process out from main memory to secondary memory is called Swap Out and the action of moving a process out from secondary memory to main memory is called Swap In.

Swap-Space :

The area on the disk where the swapped-out processes are stored is called swap space.

1. Swap-Space Management :

- Swap-Swap management is another low-level task of the operating system. Disk space is used as an extension of main memory by the virtual memory. As we know the fact that disk access is much slower than memory access, in the swap-space management we are using disk space, so it will significantly decreases system performance.
- Basically, in all our systems we require the best throughput, so the goal of this swap-space implementation is to provide the virtual memory the best throughput. In these article, we are going to discuss how swap space is used, where swap space is located on disk, and how swap space is managed.

2. Swap-Space Use :

- Swap-space is used by the different operating-system in various ways. The systems which are implementing swapping may use swap space to hold the entire process which may include image, code and data segments. Paging systems may simply store pages that have been pushed out of the main memory. The need of swap space on a system can vary from a megabytes to gigabytes but it also depends on the amount of physical memory, the virtual memory it is backing and the way in which it is using the virtual memory.
- It is safer to overestimate than to underestimate the amount of swap space required, because if a system runs out of swap space it may be forced to abort the processes or may crash entirely. Overestimation wastes disk space that could otherwise be used for files, but it does not harm other.

Following table shows different system using amount of swap space:

Table 5.4

System	Swap Space
1. Solaris	Equal amount of physical memory
2. Linux	Double the amount of physical memory

Figure Different systems using amount of swap-space

Explanation of Above Table :

- Solaris, setting swap space equal to the amount by which virtual memory exceeds page-able physical memory. In the past Linux has suggested setting swap space to double the amount of physical memory. Today, this limitation is gone, and most Linux systems use considerably less swap space.
- Including Linux, some operating systems; allow the use of multiple swap spaces, including both files and dedicated swap partitions. The swap spaces are placed on the disk so the load which is on the I/O by the paging and swapping will spread over the system's bandwidth.

Swap-Space Location :

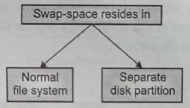


Fig. 5.33 : Location of swap-space

A swap space can reside in one of the two places –

1. Normal file system
2. Separate disk partition

- Let, if the swap-space is simply a large file within the file system. To create it, name it and allocate its space normal file-system routines can be used. This approach, through easy to implement, is inefficient. Navigating the directory structures and the disk-allocation data structures takes time and extra disk access. During reading or writing of a process image, external fragmentation can greatly increase swapping times by forcing multiple seeks.
- There is also an alternate to create the swap space which is in a separate raw partition. There is no presence of any file system in this place. Rather, a swap space storage manager is used to allocate and de-allocate the blocks. from the raw partition. It uses the

algorithms for speed rather than storage efficiency, because we know the access time of swap space is shorter than the file system. By this Internal fragmentation increases, but it is acceptable, because the life span of the swap space is shorter than the files in the file system. Raw partition approach creates fixed amount of swap space in case of the disk partitioning.

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

Swap-Space Management: An Example

- The traditional UNIX kernel started with an implementation of swapping that copied entire process between contiguous disk regions and memory. UNIX later evolve to a combination of swapping and paging as paging hardware became available.
- In Solaris, the designers changed standard UNIX methods to improve efficiency. More changes were made in later versions of Solaris, to improve the efficiency.
- Linux is almost similar to Solaris system. In both the systems the swap space is used only for anonymous memory, it is that kind of memory which is not backed by any file. In the Linux system, one or more swap areas are allowed to be established. A swap area may be in either in a swap file on a regular file system or a dedicated file partition.

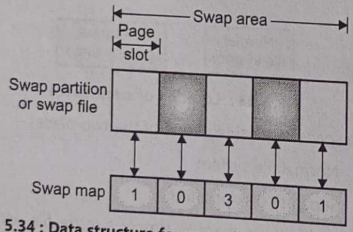


Fig. 5.34 : Data structure for swapping on Linux system

- Each swap area consists of 4-KB page slots, which are used to hold the swapped pages. Associated with each swap area is a swap-map- an array of integers counters, each corresponding to a page slot in the swap area.
- If the value of the counter is 0 it means page slot is occupied by swapped page. The value of counter indicates the number of mappings to the swapped page. For example, a value 3 indicates that the swapped page is mapped to the 3 different processes.

EXERCISE

1. Discuss the evolution of the I/O function in detail.
2. Discuss types of files with suitable examples.
3. Describe I/O buffering in detail.
4. Why I/O buffering is necessary? State and explain different I/O buffering techniques
5. Write a short note on secondary storage management.
6. Describe the following :
   (i) File sharing
   (ii) Record blocking.
7. Describe methods of record blocking with the help of neat diagrams.
8. Describe any 4 types of file organization.
9. Explain key features of windows file system.
10. List and explain various access methods used by file system to retrieve information.
11. Explain disk free space management techniques.
12. A disk drive has 640 cylinders, numbered 0-639. The drive is currently serving the request at cylinder 68. The queue of pending requests in FIFO order is: 84, 153, 32, 128, 10, 133, 61, 69. Starting from the current head position, what is the total distance that the disk arms moves to satisfy all the pending requests for the following disk scheduling algorithms :
    (i) FCFS
    (ii) C-SCAN
    (iii) SCAN
    (iv) SSTF
13. A disk drive has 640 cylinders, numbered 0-639. The drive is currently serving the request at cylinder 200. The queue of pending requests in FIFO order is: 184, 232, 128, 25, 533, 161, 169. Starting from the current head position, what is the total distance that the disk arms moves to satisfy all the pending requests for the following disk scheduling algorithms :
    (i) FCFS
    (ii) C-SCAN
    (iii) SCAN
    (iv) SSTF