Operating systems-PCCS404
UNIT-1

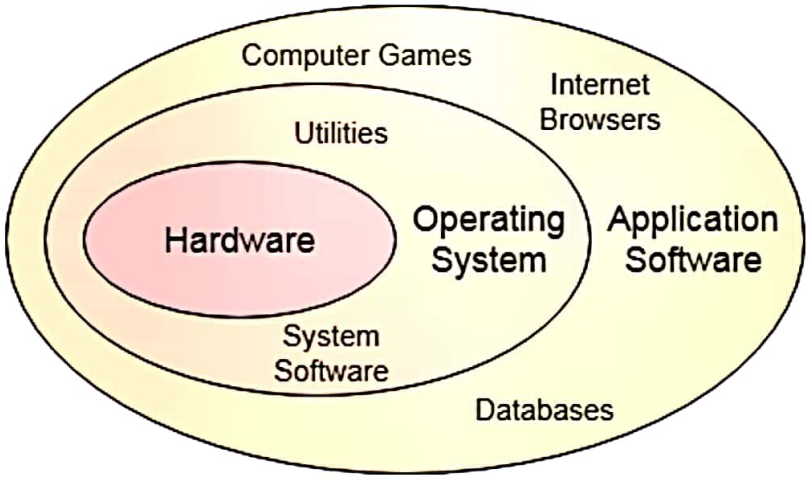
UNIT – I INTRODUCTION

Introduction: Concept of Operating Systems, Generations of Operating systems, Types of Operating Systems, OS Services, System Calls, Structure of an OS - Layered, Monolithic, Microkernel Operating Systems, Concept of Virtual Machine. Case study on UNIX and WINDOWS Operating System.

Operating System Definition and Function

In the Computer System (comprises of Hardware and software), Hardware can only understand machine code (in the form of 0 and 1) which doesn't make any sense to a naive user.

We need a system which can act as an intermediary and manage all the processes and resources present in the system.



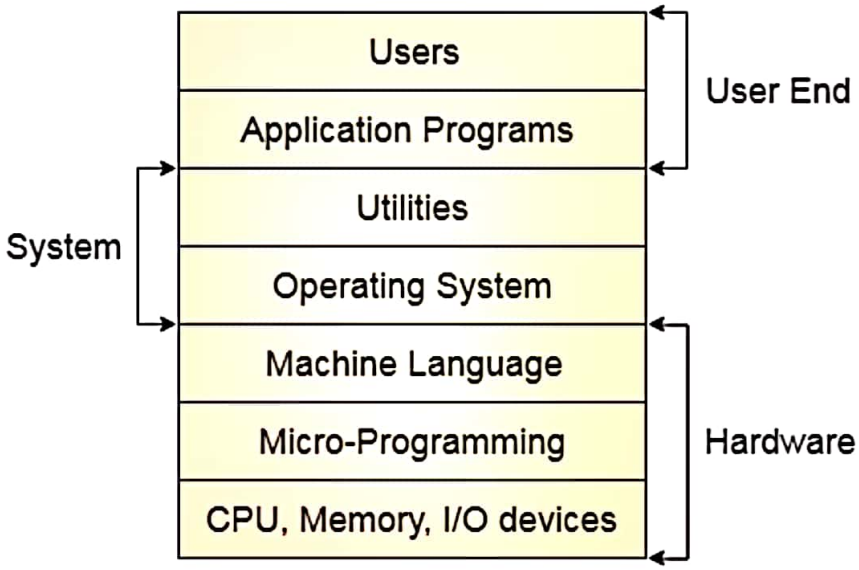
An **Operating System** can be defined as an **interface between user and hardware**. It is responsible for the execution of all the processes, Resource Allocation, CPU management, File Management and many other tasks.

The purpose of an operating system is to provide an environment in which a user can execute programs in convenient and efficient manner.

Structure of a Computer System

A Computer System consists of:

- Users (people who are using the computer)
- Application Programs (Compilers, Databases, Games, Video player, Browsers, etc.)
- System Programs (Shells, Editors, Compilers, etc.)
- Operating System (A special program which acts as an interface between user and hardware)
- Hardware (CPU, Disks, Memory, etc)



What does an Operating system do?

1. Process Management
2. Process Synchronization
3. Memory Management
4. CPU Scheduling
5. File Management
6. Security

OS GENERATION:

❖ The 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 Computer 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, the special program called SYSGEN program reads from a given file, or asks the operator of the system for information concerning the specific configuration of the hardware system.

❖ The following kinds of information must be determined.

o What CPU is to be used? o How will the boot disk be formatted? o How much memory is available? o What devices are available? o What operating-system options are desired, or what parameter values

are to be used? ❖ A system administrator can use this information to modify a copy of the source code of the operating system. The operating system then is completely compiled.

- ❖ The system description can lead to the creation of tables and the selection of modules from a precompiled library. These modules are linked together to form the generated operating system
- ❖ It is also possible to construct a system that is completely table driven. All the code is always part of the system, and selection occurs at execution time, rather than at compile or link time.

Types of OS

There are many types of operating system exists in the current scenario:

Time-sharing operating systems

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

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

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

The operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time. Computer systems that were designed primarily as batch systems have been modified to time-sharing systems.

Advantages of Timesharing operating systems are as follows –

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

Disadvantages of Time-sharing operating systems are as follows –

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

- Problem of data communication.

Distributed operating System

Distributed systems use multiple central processors to serve multiple real-time applications and multiple users. Data processing jobs are distributed among the processors accordingly.

The processors communicate with one another through various communication lines (such as high-speed buses or telephone lines). These are referred as **loosely coupled systems** or distributed systems. Processors in a distributed system may vary in size and function. These processors are referred as sites, nodes, computers, and so on.

The advantages of distributed systems are as follows –

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

Network operating System

A Network Operating System runs on a server and provides the server the capability to manage data, users, groups, security, applications, and other networking functions. The primary purpose of the network operating system is to allow shared file and printer access among multiple computers in a network, typically a local area network (LAN), a private network or to other networks.

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

The advantages of network operating systems are as follows –

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

The disadvantages of network operating systems are as follows –

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

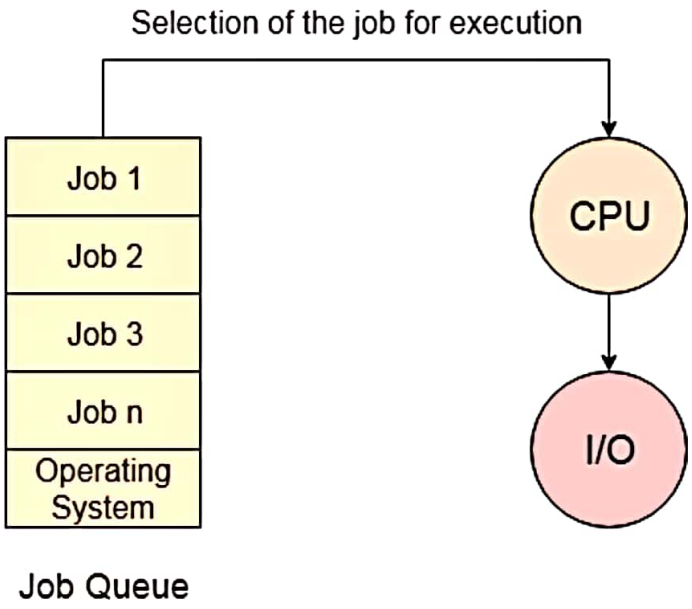
Operating systems-PCCS404
UNIT-1

Batch Operating System

In the era of 1970s, the Batch processing was very popular. The Jobs were executed in batches. People were used to have a single computer which was called mainframe.

In Batch operating system, access is given to more than one person; they submit their respective jobs to the system for the execution.

The system put all of the jobs in a queue on the basis of first come first serve and then executes the jobs one by one. The users collect their respective output when all the jobs get executed.



Disadvantages of Batch OS

1. Starvation

Batch processing suffers from starvation. If there are five jobs J1, J2, J3, J4, J4 and J5 present in the batch. If the execution time of J1 is very high then other four jobs will never be going to get executed or they will have to wait for a very high time. Hence the other processes get starved.

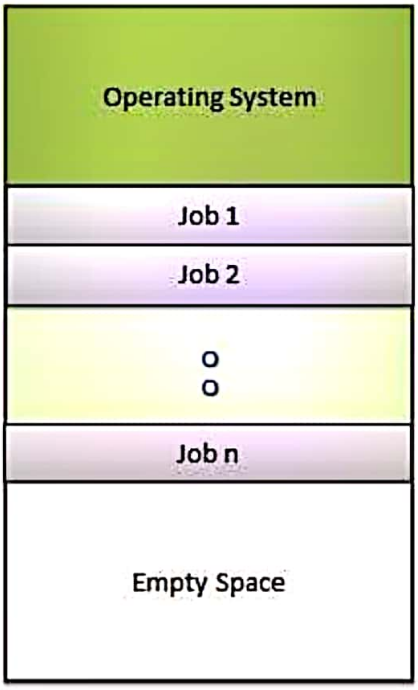
2. Not Interactive

Batch Processing is not suitable for the jobs which are dependent on the user's input. If a job requires the input of two numbers from the console then it will never be going to get it in the batch processing scenario since the user is not present at the time of execution.

Multiprogramming Operating System

Operating systems-PCCS404
UNIT-1

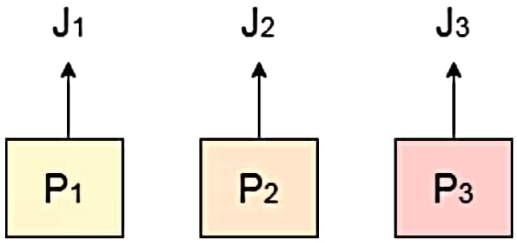
Multiprogramming is an extension to the batch processing where the CPU is kept always busy. Each process needs two types of system time: CPU time and IO time.



In multiprogramming environment, for the time a process does its I/O, The CPU can start the execution of other processes. Therefore, multiprogramming improves the efficiency of the system.

Multiprocessing Operating System

In Multiprocessing, Parallel computing is achieved. There are more than one processors present in the system which can execute more than one process at the same time. This will increase the throughput of the system.



Multi Processing

Real Time Operating System

Operating systems-PCCS404
UNIT-1

In Real Time systems, each job carries a certain deadline within which the Job is supposed to be completed, otherwise the huge loss will be there or even if the result is produced then it will be completely useless.

The Application of a Real Time system exists in the case of military applications, if you want to drop a missile then the missile is supposed to be dropped with certain precision.

Operating System - Services

An Operating System provides services to both the users and to the programs.

- It provides programs an environment to execute.
- It provides users the services to execute the programs in a convenient manner.

Following are a few common services provided by an operating system −

- Program execution
- I/O operations
- File System manipulation
- Communication
- Error Detection
- Resource Allocation
- Protection

Program execution

Operating systems handle many kinds of activities from user programs to system programs like printer spooler, name servers, file server, etc. Each of these activities is encapsulated as a process.

A process includes the complete execution context (code to execute, data to manipulate, registers, OS resources in use). Following are the major activities of an operating system with respect to program management −

- Loads a program into memory.
- Executes the program.
- Handles program's execution.
- Provides a mechanism for process synchronization.
- Provides a mechanism for process communication.
- Provides a mechanism for deadlock handling.

I/O Operation

An I/O subsystem comprises of I/O devices and their corresponding driver software. Drivers hide the peculiarities of specific hardware devices from the users.

An Operating System manages the communication between user and device drivers.

- I/O operation means read or write operation with any file or any specific I/O device.
- Operating system provides the access to the required I/O device when required.

File system manipulation

A file represents a collection of related information. Computers can store files on the disk (secondary storage), for long-term storage purpose. Examples of storage media include magnetic tape, magnetic disk and optical disk drives like CD, DVD. Each of these media has its own properties like speed, capacity, data transfer rate and data access methods.

A file system is normally organized into directories for easy navigation and usage. These directories may contain files and other directions. Following are the major activities of an operating system with respect to file management −

- Program needs to read a file or write a file.
- The operating system gives the permission to the program for operation on file.
- Permission varies from read-only, read-write, denied and so on.
- Operating System provides an interface to the user to create/delete files.
- Operating System provides an interface to the user to create/delete directories.
- Operating System provides an interface to create the backup of file system.

Communication

In case of distributed systems which are a collection of processors that do not share memory, peripheral devices, or a clock, the operating system manages communications between all the processes. Multiple processes communicate with one another through communication lines in the network.

The OS handles routing and connection strategies, and the problems of contention and security. Following are the major activities of an operating system with respect to communication −

- Two processes often require data to be transferred between them
- Both the processes can be on one computer or on different computers, but are connected through a computer network.
- Communication may be implemented by two methods, either by Shared Memory or by Message Passing.

Error handling

Errors can occur anytime and anywhere. An error may occur in CPU, in I/O devices or in the memory hardware. Following are the major activities of an operating system with respect to error handling −

- The OS constantly checks for possible errors.
- The OS takes an appropriate action to ensure correct and consistent computing.

Resource Management

Operating systems-PCCS404
UNIT-1

In case of multi-user or multi-tasking environment, resources such as main memory, CPU cycles and files storage are to be allocated to each user or job. Following are the major activities of an operating system with respect to resource management −

- The OS manages all kinds of resources using schedulers.
- CPU scheduling algorithms are used for better utilization of CPU.

Protection

Considering a computer system having multiple users and concurrent execution of multiple processes, the various processes must be protected from each other's activities.

Protection refers to a mechanism or a way to control the access of programs, processes, or users to the resources defined by a computer system. Following are the major activities of an operating system with respect to protection −

- The OS ensures that all access to system resources is controlled.
- The OS ensures that external I/O devices are protected from invalid access attempts.
- The OS provides authentication features for each user by means of passwords.

SYSTEM CALLS:

❖ The system call provides an interface to the operating system services.

❖ Application developers often do not have direct access to the system calls, but can access them through an application programming interface (API). The functions that are included in the API invoke the actual system calls.

❖ Systems execute thousands of system calls per second. Application developers design programs according to an **application programming interface (API)**.

❖ For most programming languages, the Application Program Interface provides a **system call interface** that serves as the link to system calls made available by the operating system. ❖ The system-call interface intercepts function calls in the API and invokes the necessary system calls within the Operating system.

❖ **Example: System calls for writing a simple program to read data from one file and copy them to another file**

❖ The caller of the system call need know nothing about how the system call is implemented or what it does during execution.

❖ The caller need only obey the API and understand what the operating system will do as a result of The execution of that system call.

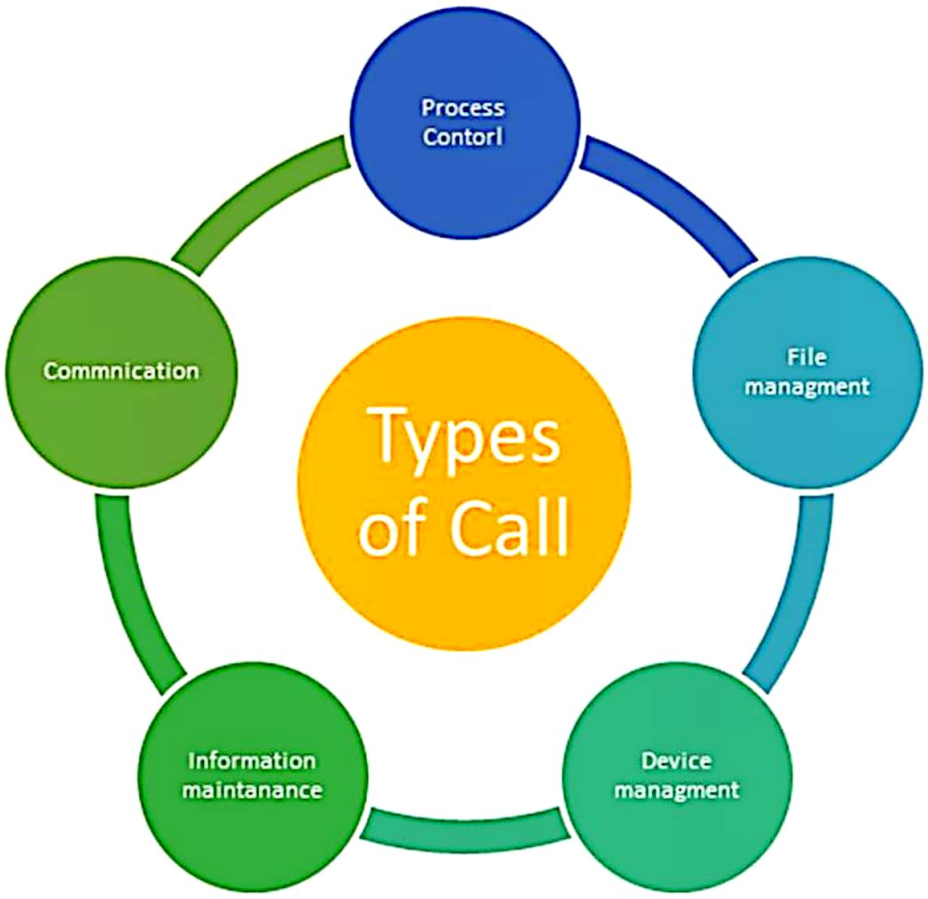
❖ Three general methods are used to pass parameters to the operating

system o pass the parameters in registers o parameters are generally stored in a block, or table, in memory, and the address of the block is passed as a parameter in a register o Parameters also can be placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system.

Types of System Calls:

❖ System calls can be grouped roughly into six major categories

1.Processcontrol,
2.File manipulation,
3.Device manipulation,
4.Information maintenance,
5.Communications,
6.Protection



1.Process Control

This system calls perform the task of process creation, process termination, etc.

Functions:

- End and Abort
- Load and Execute
- Create Process and Terminate Process
- Wait and Signed Event
- Allocate and free memory

2.File Management or manipulation

File management system calls handle file manipulation jobs like creating a file, reading, and writing, etc.

Functions:

- Create a file
- Delete file
- Open and close file
- Read, write, and reposition
- Get and set file attributes

3.Device Management or manipulation

Device management does the job of device manipulation like reading from device buffers, writing into device buffers, etc.

Functions

- Request and release device
- Logically attach/ detach devices
- Get and Set device attributes

4.Information Maintenance

It handles information and its transfer between the OS and the user program.

Functions:

- Get or set time and date
- Get process and device attributes

5.Communication:

These types of system calls are specially used for interprocess communications.

Functions:

- Create, delete communications connections
- Send, receive message
- Help OS to transfer status information
- Attach or detach remote devices

OPERATING SYSTEM STRUCTURE:

❖ The operating systems are large and complex. A common approach is to partition the task into small components, or modules, rather than have one **monolithic** system.

❖ The structure of an operating system can be defined the following structures.

← Simple structure ← Layered approach ← Microkernels ← Modules ← Hybrid systems

Simple structure:

❖ The Simple structured operating systems do not have a well defined structure. These systems will be simple, small and limited systems.

Example: MS-DOS.

❖ In MS-DOS, the interfaces and levels of functionality are not well separated. ❖ In MS-DOS application programs are able to access the basic I/O routines. This causes the entire systems to be crashed when user programs fail.

Example: Traditional UNIX OS

❖ It consists of two separable parts: the kernel and the system programs. ❖ The kernel is further separated into a series of interfaces and device driver

❖ The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls.

❖ This monolithic structure was difficult to implement and maintain.

Layered approach:

❖ A system can be made modular in many ways. One method is the **layered approach**, in which the operating system is broken into a number of layers (levels). The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface.

❖ An operating-system layer is an implementation of an abstract object made up of data and the operations that can manipulate those data.

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

❖ Each layer is implemented only with 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.

❖ The major difficulty with the layered approach involves appropriately defining the various layers

because a layer can use only lower-level layers.

❖ A problem with layered implementations is that they tend to be less efficient than other types.

Microkernels:

❖ In the mid-1980s, researchers at Carnegie Mellon University developed an operating system called **Mach** that modularized the kernel using the **microkernel** approach

❖ This method structures the operating system by removing all nonessential components from the kernel and implementing them as system and user-level programs.

❖ Microkernel provide minimal process and memory management, in addition to a communication facility.

❖ The main function of the microkernel is to provide communication between the client program and the various services that are also running in user space.

❖ The client program and service never interact directly. Rather, they communicate indirectly by exchanging messages with the microkernel.

❖ One benefit of the microkernel approach is that it makes extending the operating system easier. All new services are added to user space and consequently do not require modification of the kernel.

❖ The performance of microkernel can suffer due to increased system-function overhead.

Modules:

❖ The best current methodology for operating-system design involves using **loadable kernel Modules**

❖ The kernel has a set of core components and links in additional services via modules, either at boot time or during run time.

❖ The kernel provides core services while other services are implemented dynamically, as the kernel is running

❖ Linking services dynamically is more comfortable than adding new features directly to the kernel, which would require recompiling the kernel every time a change was made.

Example: Solaris OS

❖ The Solaris operating system structure is organized around a core kernel with seven types of loadable kernel modules:

← Scheduling classes ← File systems ← Loadable system calls ← Executable formats ← STREAMS modules ← Miscellaneous ← Device and bus drivers

HYBRID SYSTEMS

❖ The Operating System combines different structures, resulting in hybrid systems that address

performance, security, and usability issues.

- ❖ They are monolithic, because having the operating system in a single address space provides very efficient performance
- ❖ However, they are also modular, so that new functionality can be dynamically added to the kernel.
- ❖ Example: Linux and Solaris are monolithic (simple) and also modular, IOS.
- ❖ Apple IOS Structure

TOPICS LET:
1.Concept of virtual machine
2.case study on unix and windows os

2 MARK QUESTIONS

1.What is operating system?

A. An operating system (OS) is a collection of software that manages computer hardware resources and provides common services for computer programs. The operating system is a vital component of the system software in a computer system

2.What is time sharing operating system?

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

3.What is a system call?

A. A **system call** is a mechanism that provides the interface between a process and the operating system. It is a programmatic method in which a computer program requests a service from the kernel of the OS.

4.what is real-time operating system?

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

5.List the services provided by the operating system?

A. Following are a few common services provided by an operating system −

- Program execution
- I/O operations
- File System manipulation
- Communication

- Error Detection
- Resource Allocation
- Protection

6.what is virtual machine?

A. Virtual Machine abstracts the hardware of our personal computer such as CPU, disk drives, memory, NIC (Network Interface Card) etc, into many different execution environments as per our requirements, hence giving us a feel that each execution environment is a single computer. For example: VirtualBox.

7.what are the advantages of multi processing operating system?

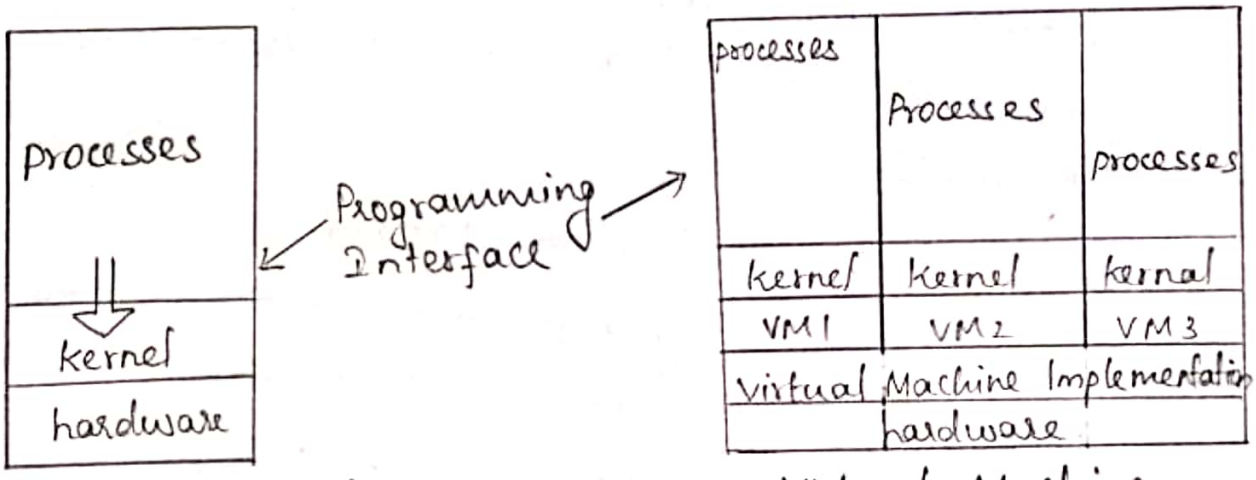
- The biggest advantage of a multiprocessor system is that it helps you to get more work done in a shorter period.
- The code is usually straightforward.
- Takes advantage of multiple CPU & cores
- Helps you to avoid GIL limitations for CPython
- Remove synchronization primitives unless if you use shared memory.
- Child processes are mostly interruptible/killable
- It helps you to get work done in a shorter period.
- These types of systems should be used when very high speed is required to process a large volume of data.
- Multiprocessing systems save money compared to single processor systems as processors can share peripherals and power supplies.

* Virtual Machines:

The fundamental idea behind a virtual machine is to abstract the hardware of a single computer into several different execution environments thereby creating the illusion that each seperate execution environment is running its own private computer.

By using CPU scheduling and virtual Memory techniques an operating system can create the illusion that a process has its own processor with its own (virtual) Mfy.

The virtual Machine approach does not provide any such additional functionality but rather provides an interface that is identical to the bare hardware. Each process is provided with a (virtual) copy of the computer.



Non virtual Machine

Virtual Machine.

There are several reasons for creating a virtual Machine, all of which are fundamentally related to being able to share the same hlw yet run seoveral different execution environment.

Minidisks:-

The physical Machine cannot allocate a disk drive to each virtual machine because the virtual machine software itself will need substantial disk space to provide virtual mly and spooling. Hence we need virtual disk which is known as minidisks. The sum of the sizes of all the minidisks must be smaller

than the size of the physical disk space available. Users are given their own Virtual machines.

Benefits:-

* Complete protection of the various s/m resources.
* Sharing of disk.
* Communication.

Examples:-

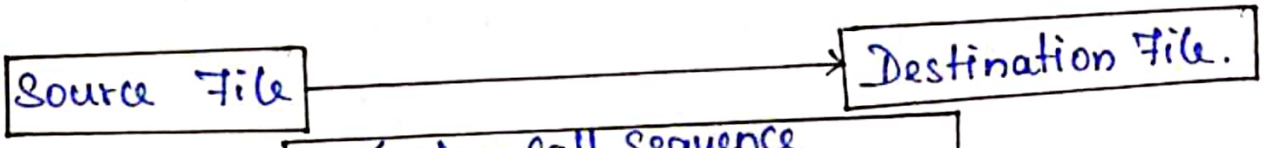
* VMware
* The Java virtual Machine.

* System Calls:-

System calls provides an interface to the services made available by an operating system. These are generally available as routines written in C & C++.

How system calls are used:-

To read data from one file and copy it in another file.



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 exist; abort

Loop
  Read from i/p file
  Write to output file
Until read fails
close the output file

* Write completion msg to screen
* Terminate Normally.

The run-time support system for most programming languages provides a system-call interface that servers as the link to system calls made available by the operating system.

The system call interface intercepts function calls in the API and invokes the necessary system call within the operating system.

A number is associated with each system calls and the system call interface maintains a table indexed according to these numbers.

The system call interface invokes the intended sim call in the OS kernel & returns the status of the sim call and any return values.

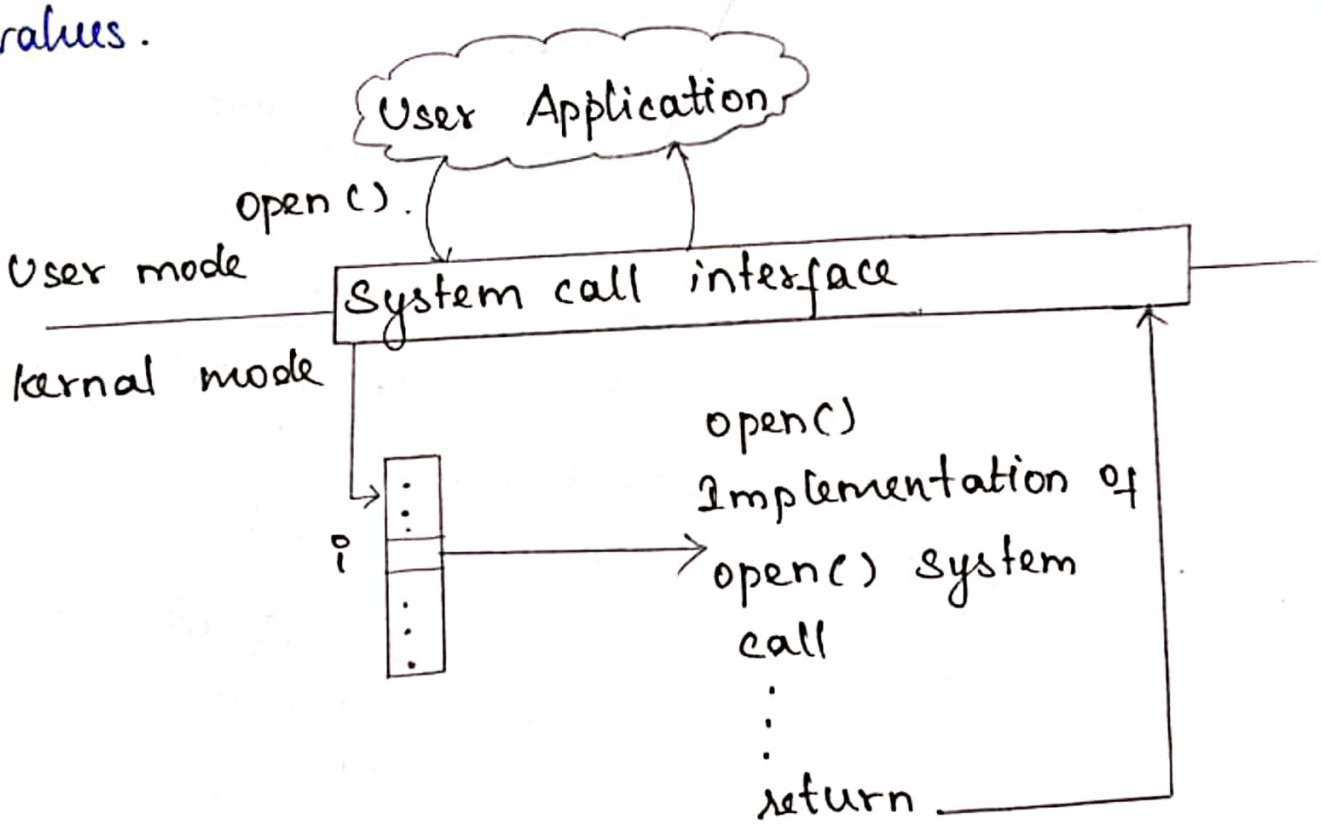


fig:- Handling of a user application invoking the open() system call.

System calls occur in different ways depending on the computer in use. Often more information is required than simply the identity of desired slm call.

Passing parameters to the operating system:.

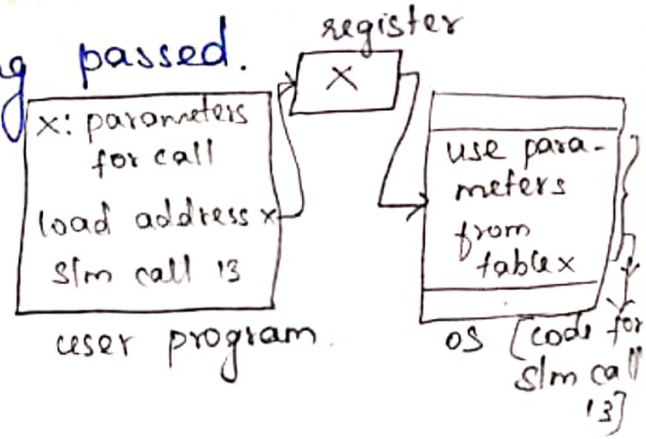
Three general Methods are there to pass parameters to the operating system.

1. To pass parameters in registers.

2. Parameters are stored in a block, table, memory and the address of the block is passed as a parameter in a register.

2. Parameters also can be placed or pushed onto the stack by the program and popped off the stack by the OS.

3. Some OS prefer the block / stack because those approaches do not limit the no or length of parameters being passed.



Types of system calls:.

* Process control
  - End, abort
  - Load, execute
  - Create process, terminate process
  - Get process attributes, set process attributes
  - Wait for time

- wait event, Signal event
- Allocate and free memory.

* File Management.
  - Create file, delete file
  - Open, close
  - Read, write, reposition.
  - Get file attributes, set file attributes.

* Device Management
  - Request device, Release device.
  - read, write, reposition
  - Get device attributes, set device attributes
  - logically attach or detach devices.

* Information Maintenance
  - Get time or date, set time or date
  - Get sim data, set s/m data.
  - get process, file or device attributes
  - set. " " "

* Communications.
  - Create, delete communication connection
  - send, receive msg
  - Transfer status info
  - Attach or detach remote devices.

UNIT – II PROCESSES

Processes: Definition, Process Relationship, Different states of a Process, Process State transitions, Process Control Block (PCB), Context switching

Thread: Definition, Various states, Benefits of threads, Types of threads, Concept of multithreads,

Process Scheduling: Foundation and Scheduling objectives, Types of Schedulers, Scheduling criteria: CPU utilization, Throughput, Turnaround Time, Waiting Time, Response Time; Scheduling algorithms: Pre-emptive and Non pre-emptive, FCFS, SJF, RR; Multiprocessor scheduling: Real Time scheduling: RM and EDF.

PROCESSES:

A Process is defined as a program in execution. A Program does nothing unless its instructions are executed by a CPU. A program in execution is called a process. In order to accomplish its task, process needs the computer resources.

There may exist more than one process in the system which may require the same resource at the same time. Therefore, the operating system has to manage all the processes and the resources in a convenient and efficient way.

Some resources may need to be executed by one process at one time to maintain the consistency otherwise the system can become inconsistent and deadlock may occur.

The operating system is responsible for the following activities in connection with Process Management:

1. Scheduling processes and threads on the CPUs.
2. Creating and deleting both user and system processes.
3. Suspending and resuming processes.
4. Providing mechanisms for process synchronization.

Process States:

❖ As a process executes, it changes **state**. The state of a process is defined in part by the current activity of that process

A process may be in one of the following states:

New. The process is being created.

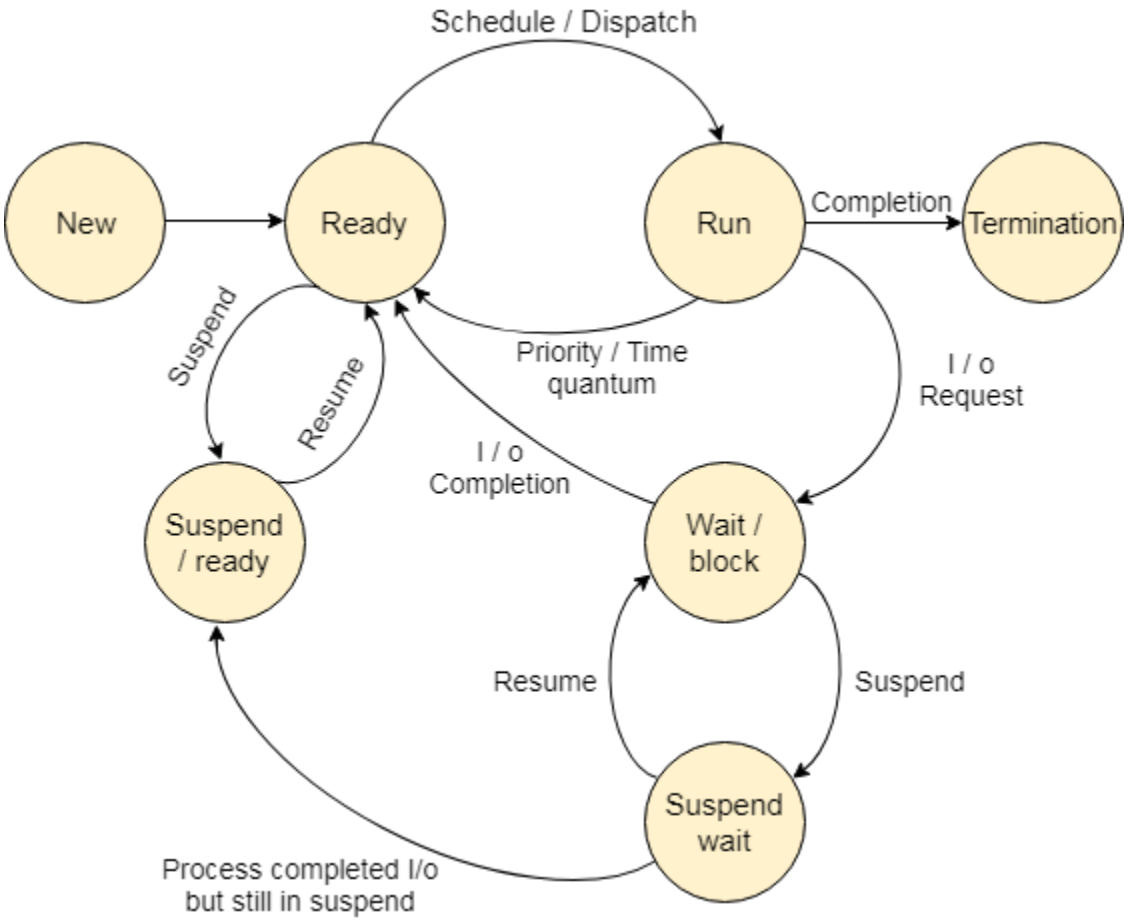
Running. Instructions are being executed.

Waiting. The process is waiting for some event to occur

Ready. The process is waiting to be assigned to a processor.

Terminated. The process has finished execution.

State Diagram



The process, from its creation to completion, passes through various states. The minimum number of states is five.

The names of the states are not standardized although the process may be in one of the following states during execution.

1. New

A program which is going to be picked up by the OS into the main memory is called a new process.

2. Ready

Whenever a process is created, it directly enters in the ready state, in which, it waits for the CPU to be assigned. The OS picks the new processes from the secondary memory and put all of them in the main memory.

The processes which are ready for the execution and reside in the main memory are called ready state processes. There can be many processes present in the ready state.

3. Running

One of the processes from the ready state will be chosen by the OS depending upon the scheduling algorithm. Hence, if we have only one CPU in our system, the number of running processes for a particular time will always be one. If we have n processors in the system then we can have n processes running simultaneously.

4. Block or wait

From the Running state, a process can make the transition to the block or wait state depending upon the scheduling algorithm or the intrinsic behavior of the process.

When a process waits for a certain resource to be assigned or for the input from the user then the OS move this process to the block or wait state and assigns the CPU to the other processes.

5. Completion or termination

When a process finishes its execution, it comes in the termination state. All the context of the process (Process Control Block) will also be deleted the process will be terminated by the Operating system.

6. Suspend ready

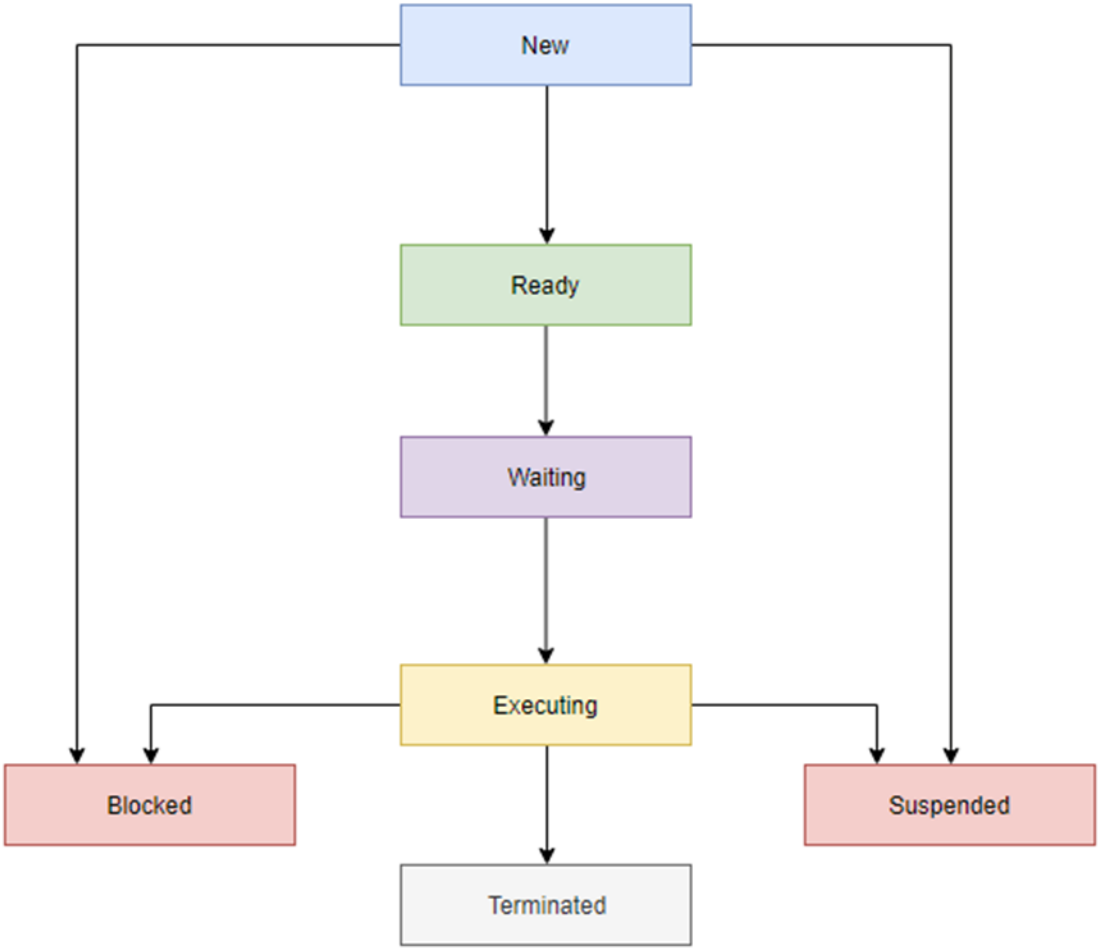
A process in the ready state, which is moved to secondary memory from the main memory due to lack of the resources (mainly primary memory) is called in the suspend ready state.

If the main memory is full and a higher priority process comes for the execution then the OS have to make the room for the process in the main memory by throwing the lower priority process out into the secondary memory. The suspend ready processes remain in the secondary memory until the main memory gets available.

7. Suspend wait

Instead of removing the process from the ready queue, it's better to remove the blocked process which is waiting for some resources in the main memory. Since it is already waiting for some resource to get available hence it is better if it waits in the secondary memory and make room for the higher priority process. These processes complete their execution once the main memory gets available and their wait is finished.

PROCESSES
OPERATING SYSTEMS -UNIT II



Process Control Block:

❖ Each process is represented in the operating system by a **process control block (PCB)**—also called a **task control block**.
❖ It contains many pieces of information associated with a specific process

❖ **Process state**. The state may be new, ready, running, waiting, halted, and so on.
❖ **Program counter**. The counter indicates the address of the next instruction to be executed for this process.
❖ **CPU registers**. The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers

- **CPU-scheduling information.** This information includes a process priority, pointers to scheduling queues
- **Memory-management information.** This information may include such items as the value of the base and limit registers and the page tables
- **Accounting information.** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers,
- **I/O status information.** This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

Various Times related to the Process

1. Arrival Time

The time at which the process enters into the ready queue is called the arrival time.

2. Burst Time

The total amount of time required by the CPU to execute the whole process is called the Burst Time. This does not include the waiting time. It is confusing to calculate the execution time for a process even before executing it hence the scheduling problems based on the burst time cannot be implemented in reality.

3. Completion Time

The Time at which the process enters into the completion state or the time at which the process completes its execution, is called completion time.

4. Turnaround time

The total amount of time spent by the process from its arrival to its completion, is called Turnaround time.

5. Waiting Time

The Total amount of time for which the process waits for the CPU to be assigned is called waiting time.

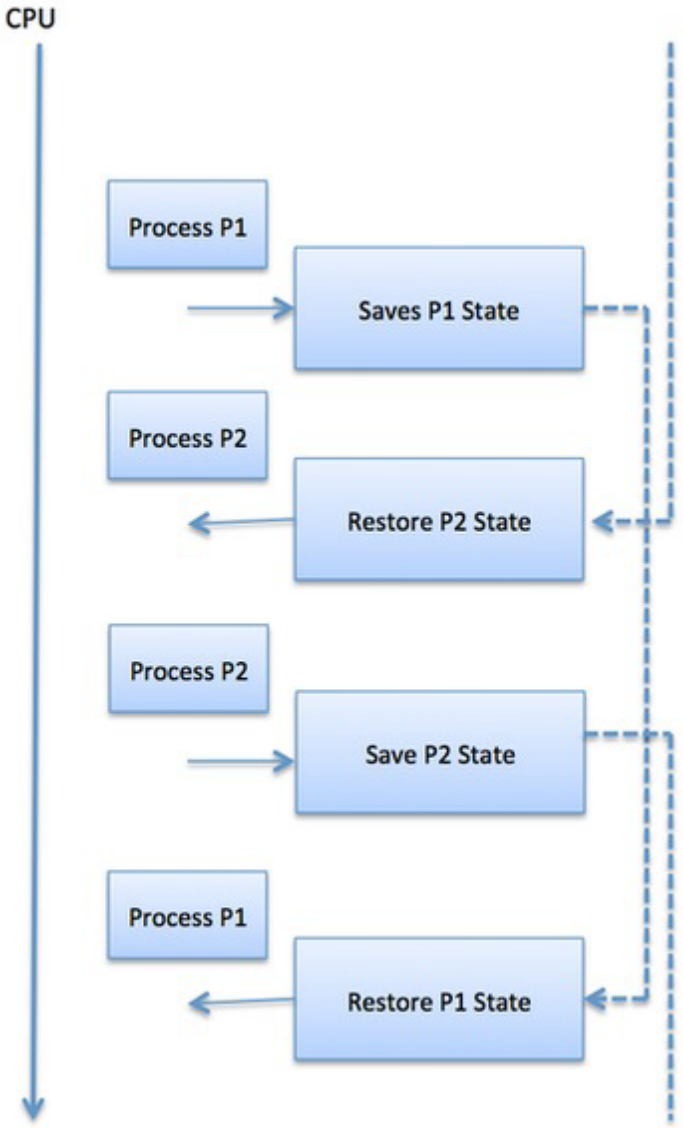
6. Response Time

The difference between the arrival time and the time at which the process first gets the CPU is called Response Time.

CONTEXT SWITCHING

A context switch is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time. Using this technique, a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features.

When the scheduler switches the CPU from executing one process to execute another, the state from the current running process is stored into the process control block. After this, the state for the process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can start executing.



Context switches are computationally intensive since register and memory state must be saved and restored. To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers. When the process is switched, the following information is stored for later use.

- Program Counter
- Scheduling information
- Base and limit register value

- Currently used register
- Changed State
- I/O State information
- Accounting information

THREADS

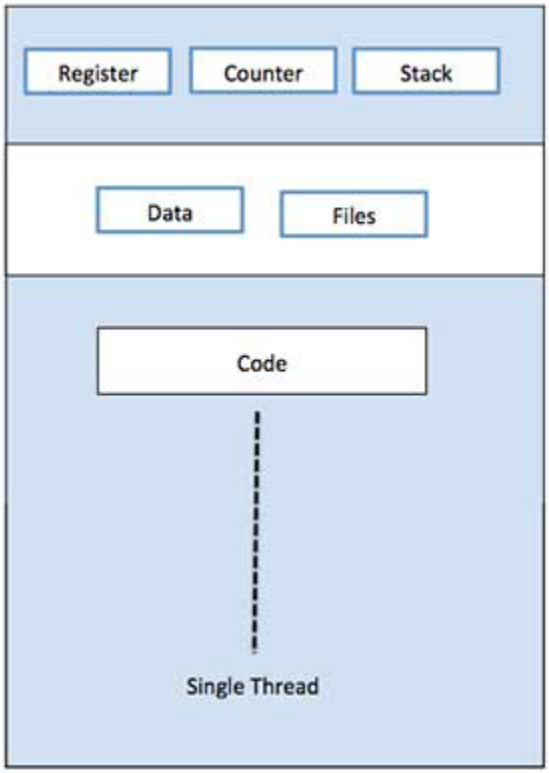
What is Thread?

A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.

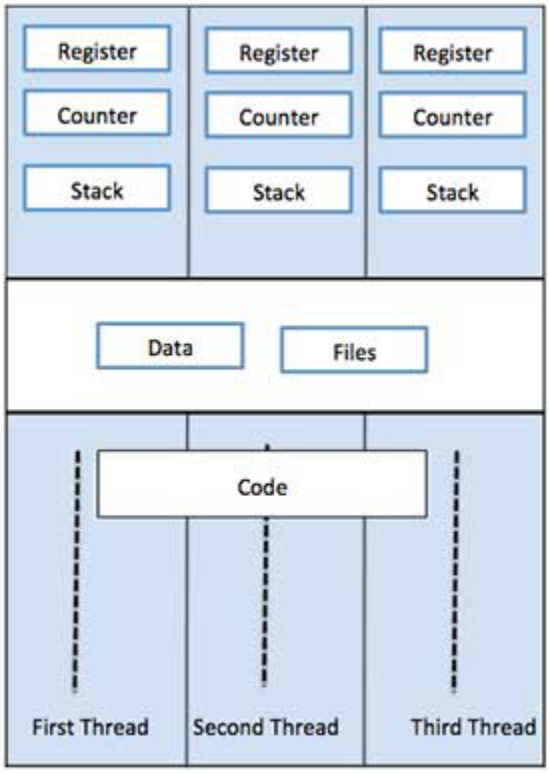
A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

A thread is also called a **lightweight process**. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.

Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control. Threads have been successfully used in implementing network servers and web server. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors. The following figure shows the working of a single-threaded and a multithreaded process.



Single Process P with single thread



Single Process P with three threads

Advantages of Thread

- Threads minimize the context switching time.
- Use of threads provides concurrency within a process.
- Efficient communication.
- It is more economical to create and context switch threads.
- Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

Types of Thread

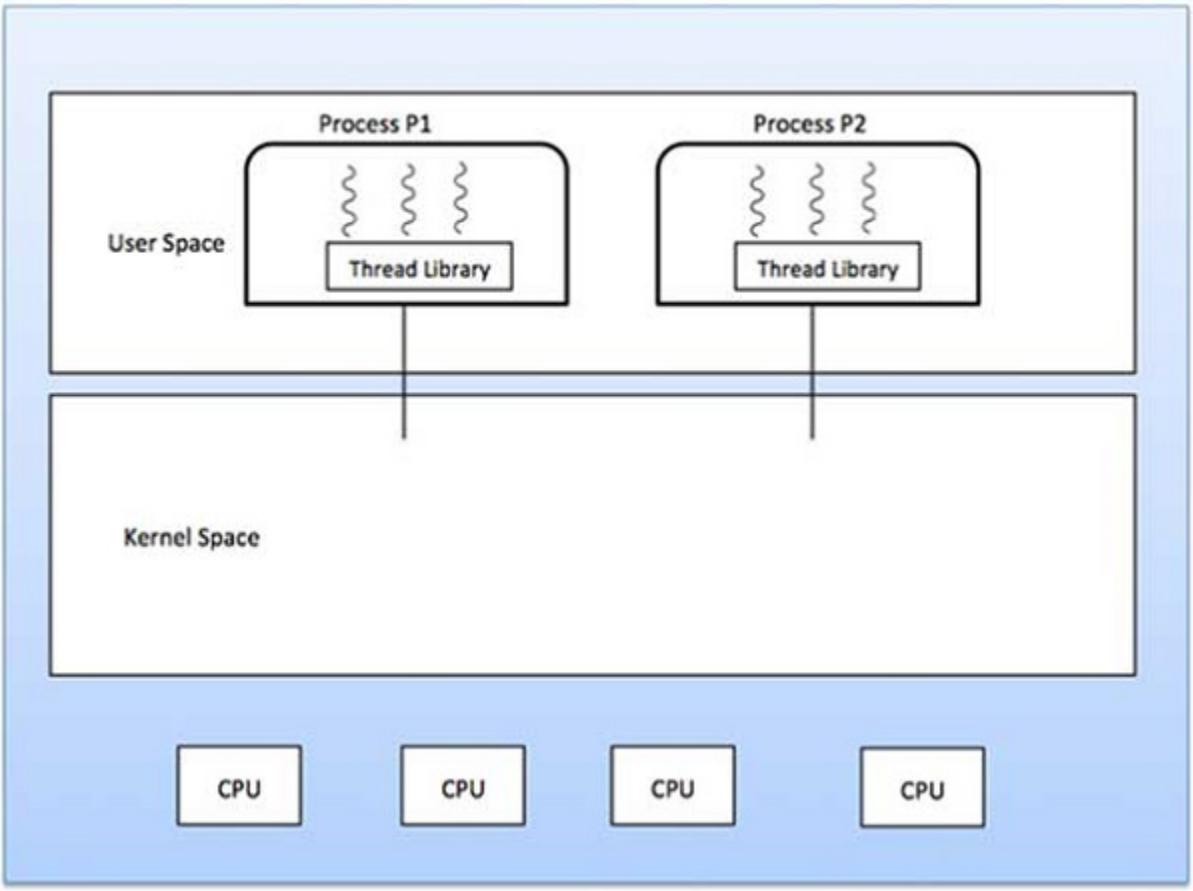
Threads are implemented in following two ways −

- **User Level Threads** − User managed threads.
- **Kernel Level Threads** − Operating System managed threads acting on kernel, an operating system core.

User Level Threads

In this case, the thread management kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between

threads, for scheduling thread execution and for saving and restoring thread contexts. The application starts with a single thread.



Advantages

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

Disadvantages

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

Kernel Level Threads

In this case, thread management is done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can

be programmed to be multithreaded. All of the threads within an application are supported within a single process.

The Kernel maintains context information for the process as a whole and for individuals threads within the process. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

Advantages

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

Disadvantages

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

Multithreading Models

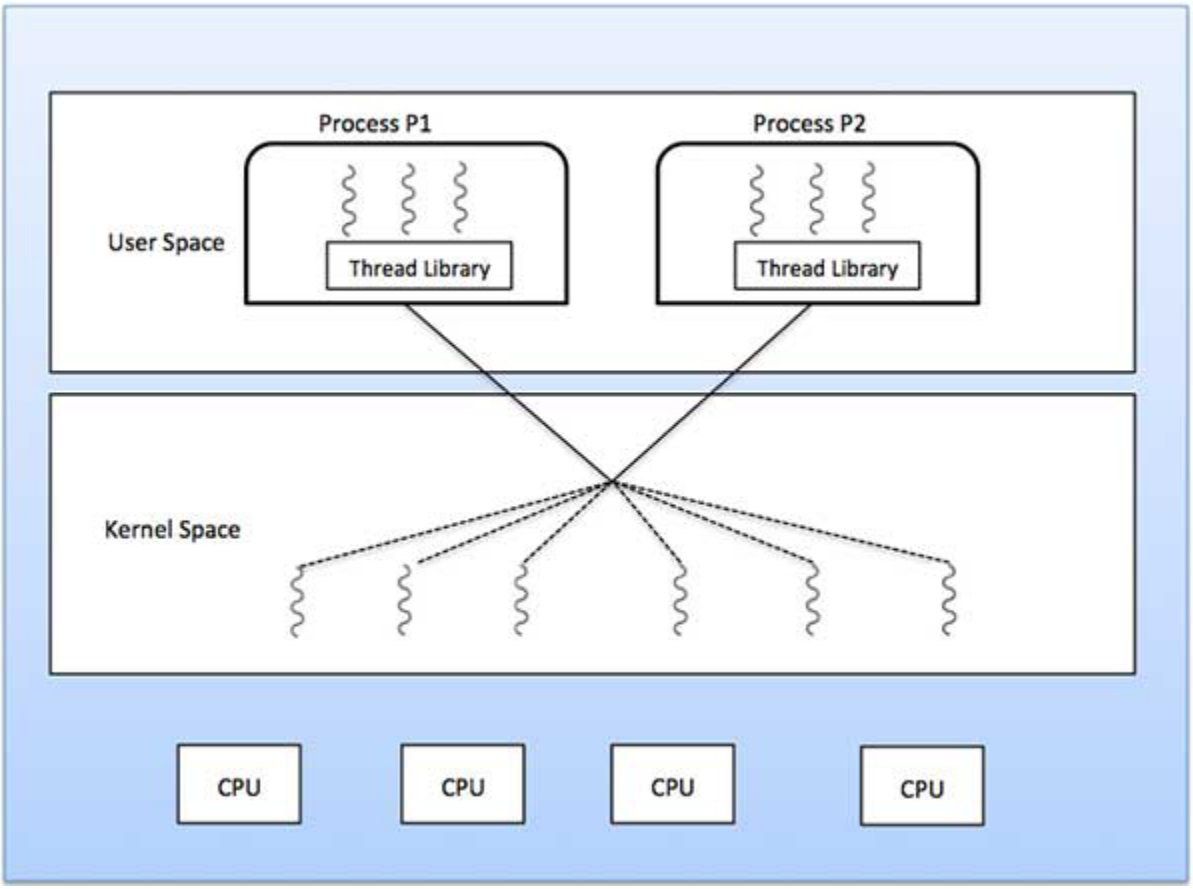
Some operating system provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the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

- Many to many relationship.
- Many to one relationship.
- One to one relationship.

Many to Many Model

The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads.

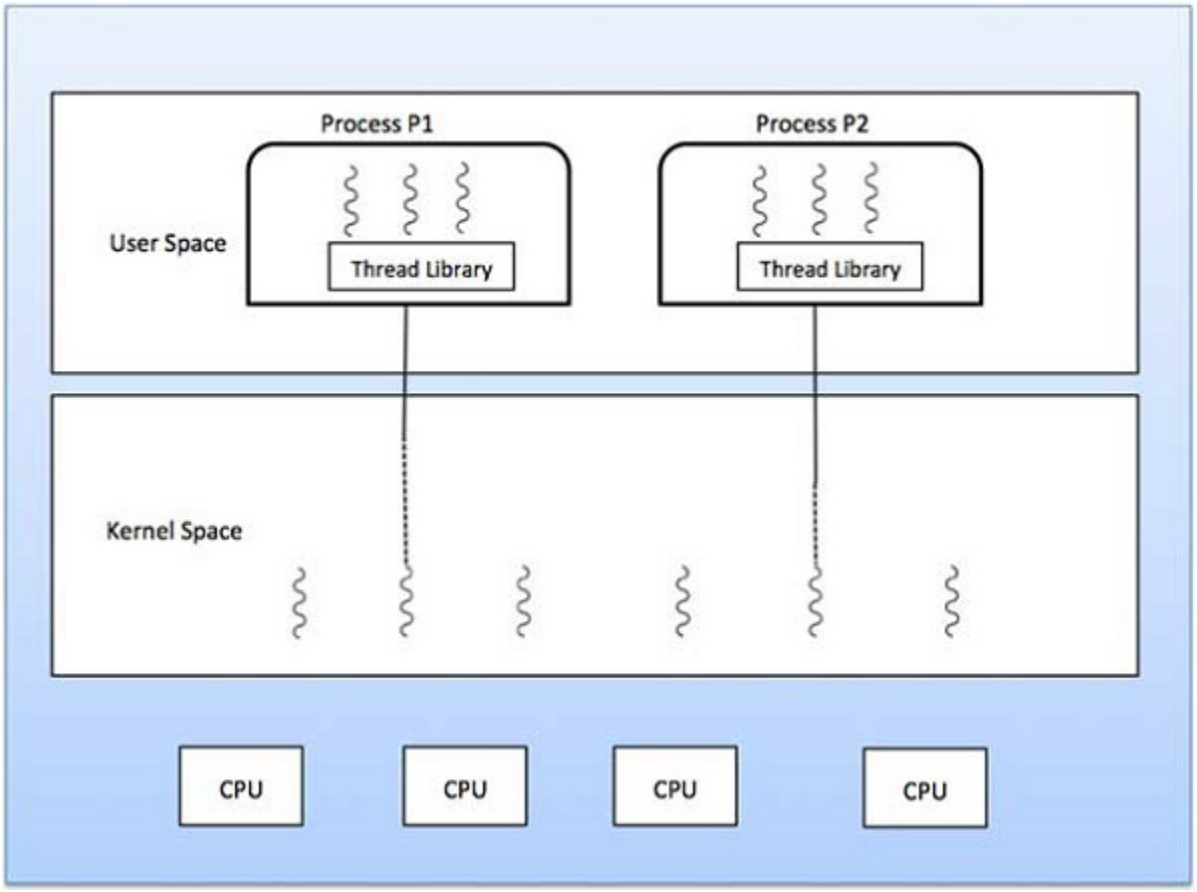
The following diagram shows the many-to-many threading model where 6 user level threads are multiplexing with 6 kernel level threads. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine. This model provides the best accuracy on concurrency and when a thread performs a blocking system call, the kernel can schedule another thread for execution.



Many to One Model

Many-to-one model maps many user level threads to one Kernel-level thread. Thread management is done in user space by the thread library. When thread makes a blocking system call, the entire process will be blocked. Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.

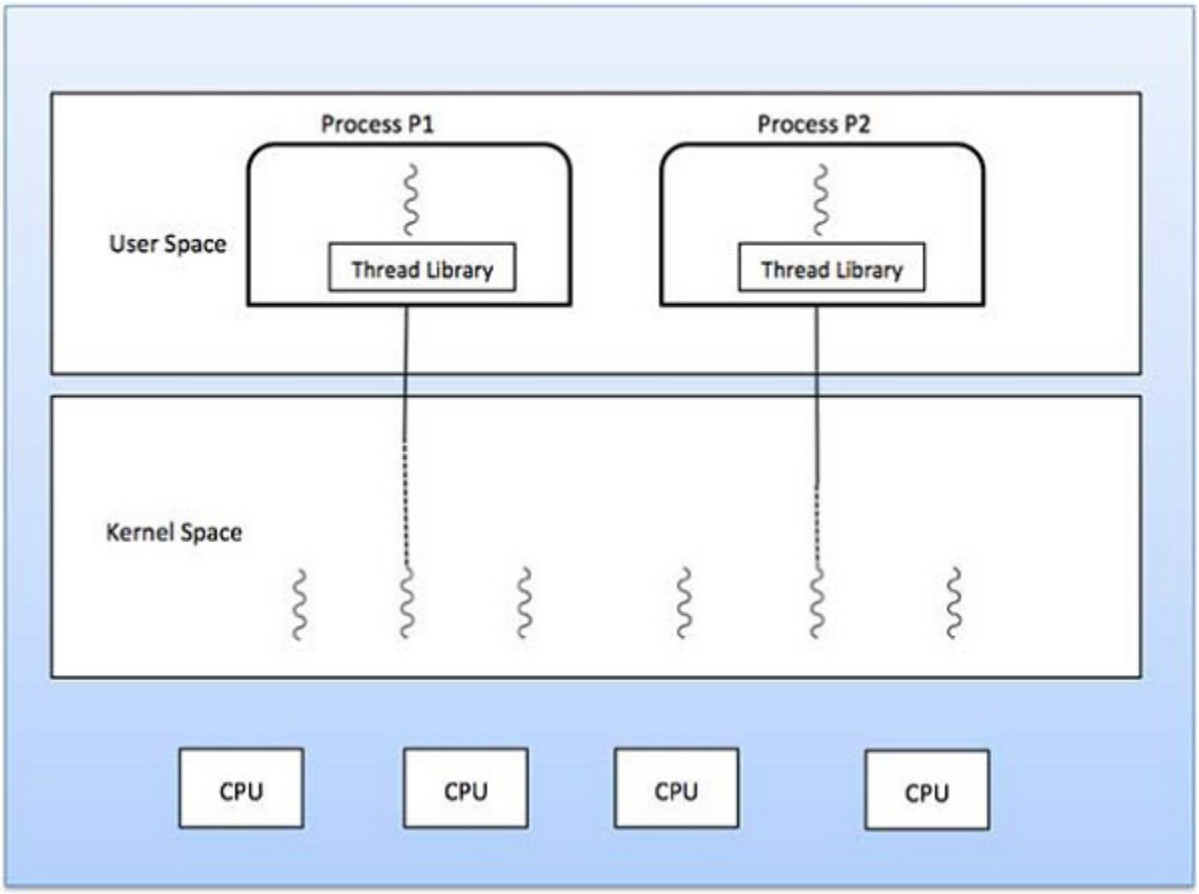
If the user-level thread libraries are implemented in the operating system in such a way that the system does not support them, then the Kernel threads use the many-to-one relationship modes.



One to One Model

There is one-to-one relationship of user-level thread to the kernel-level thread. This model provides more concurrency than the many-to-one model. It also allows another thread to run when a thread makes a blocking system call. It supports multiple threads to execute in parallel on microprocessors.

Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, windows NT and windows 2000 use one to one relationship model.



Operating System - Process Scheduling

Definition

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

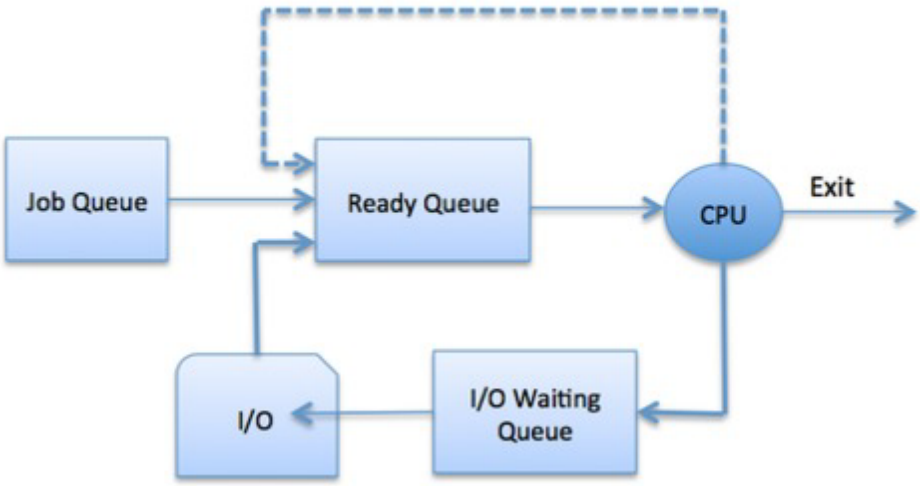
Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

Process Scheduling Queues

The OS maintains all PCBs in Process Scheduling Queues. The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

The Operating System maintains the following important process scheduling queues −

- **Job queue** − This queue keeps all the processes in the system.
- **Ready queue** − This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
- **Device queues** − The processes which are blocked due to unavailability of an I/O device constitute this queue.



The OS can use different policies to manage each queue (FIFO, Round Robin, Priority, etc.). The OS scheduler determines how to move processes between the ready and run queues which can only have one entry per processor core on the system; in the above diagram, it has been merged with the CPU.

Two-State Process Model

Two-state process model refers to running and non-running states which are described below −

S.N.	State & Description
1	**Running** When a new process is created, it enters into the system as in the running state.
2	**Not Running** Processes that are not running are kept in queue, waiting for their turn to execute. Each entry in the queue is a pointer to a particular process. Queue is implemented by using linked list. Use of dispatcher is as follows. When a process is 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.

Schedulers

Schedulers are special system software which handle process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run. Schedulers are of three types −

- Long-Term Scheduler
- Short-Term Scheduler
- Medium-Term Scheduler

Long Term Scheduler

It is also called a **job scheduler**. A long-term scheduler determines which programs are admitted to the system for processing. It selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduling.

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 not be available or minimal. Time-sharing operating systems have no long term scheduler. When a process changes the state from new to ready, then there is use of long-term scheduler.

Short Term Scheduler

It is also called as **CPU scheduler**. Its main objective is to increase 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 a process among the processes that are ready to execute and allocates CPU to one of them.

Short-term schedulers, also known as dispatchers, make the decision of which process to execute next. Short-term schedulers are faster than long-term schedulers.

Medium Term Scheduler

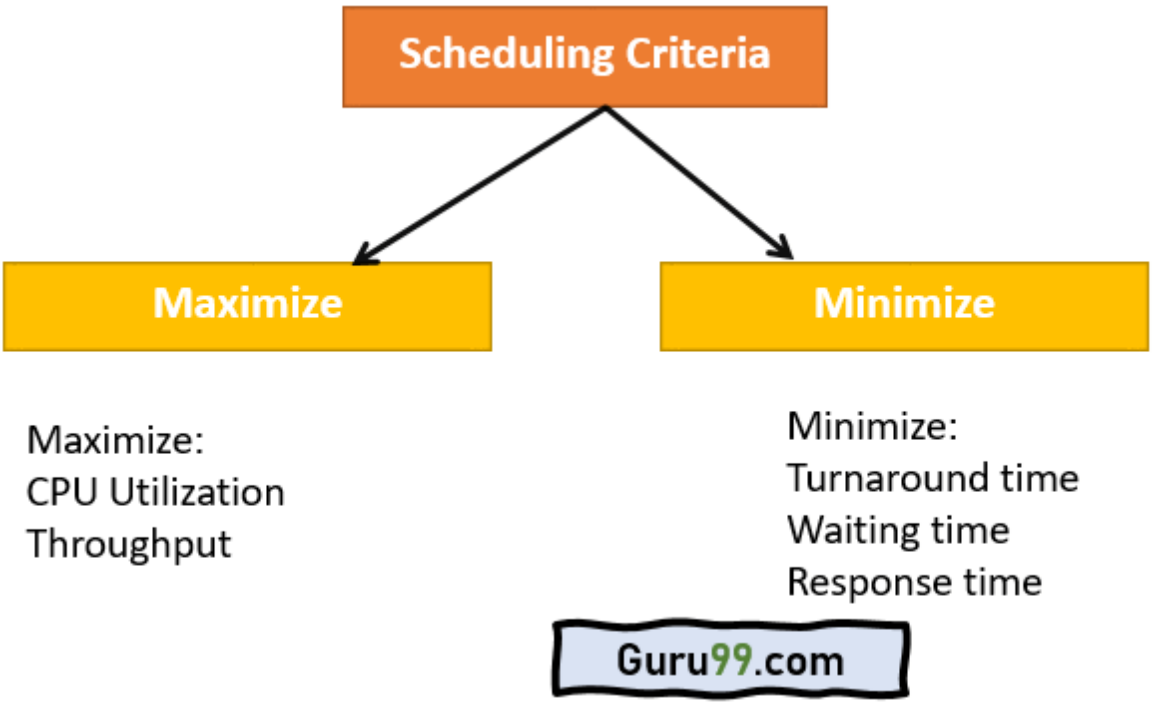
Medium-term scheduling is a part of **swapping**. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium-term scheduler is in-charge of handling the swapped out-processes.

A running process may become suspended if it makes an I/O request. A suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other processes, the suspended process is moved to the secondary

storage. This process is called **swapping**, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix.

CPU Scheduling Criteria

A CPU scheduling algorithm tries to maximize and minimize the following:



Maximize:

CPU utilization: CPU utilization is the main task in which the operating system needs to make sure that CPU remains as busy as possible. It can range from 0 to 100 percent. However, for the RTOS, it can be range from 40 percent for low-level and 90 percent for the high-level system.

Throughput: The number of processes that finish their execution per unit time is known Throughput. So, when the CPU is busy executing the process, at that time, work is being done, and the work completed per unit time is called Throughput.

Minimize:

Waiting time: Waiting time is an amount that specific process needs to wait in the ready queue.

Response time: It is an amount to time in which the request was submitted until the first response is produced.

Turnaround Time: Turnaround time is an amount of time to execute a specific process. It is the calculation of the total time spent waiting to get into the memory, waiting in the queue and, executing on the CPU. The period between the time of process submission to the completion time is the turnaround time.

Interval Timer

Timer interruption is a method that is closely related to preemption. When a certain process gets the CPU allocation, a timer may be set to a specified interval. Both timer interruption and preemption force a process to return the CPU before its CPU burst is complete.

Most of the multi-programmed operating system uses some form of a timer to prevent a process from tying up the system forever.

What is Dispatcher?

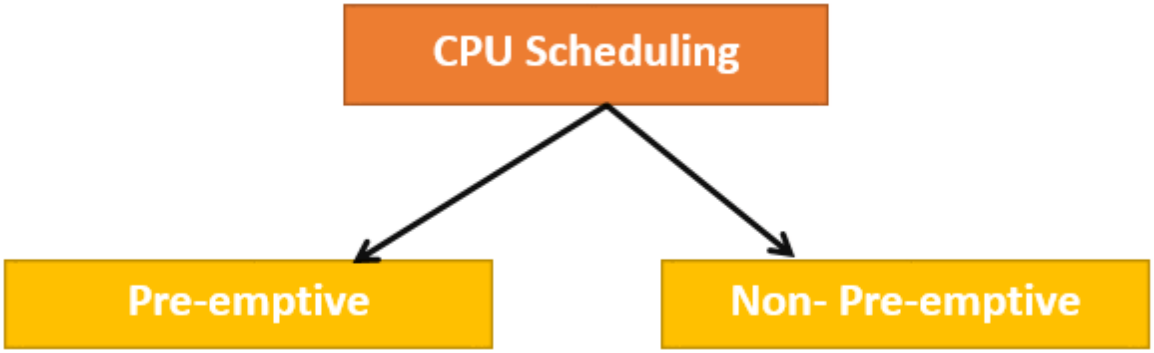
It is a module that provides control of the CPU to the process. The Dispatcher should be fast so that it can run on every context switch. Dispatch latency is the amount of time needed by the CPU scheduler to stop one process and start another.

Functions performed by Dispatcher:

- Context Switching
- Switching to user mode
- Moving to the correct location in the newly loaded program.

Types of CPU Scheduling

Here are two kinds of Scheduling methods:



Preemptive Scheduling

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

Non-Preemptive Scheduling

In this type of scheduling method, the CPU has been allocated to a specific process. The process that keeps the CPU busy will release the CPU either by switching context or terminating. It is the only method that can be used for various hardware platforms. That's because it doesn't need special hardware (for example, a timer) like preemptive scheduling.

When scheduling is Preemptive or Non-Preemptive?

To determine if scheduling is preemptive or non-preemptive, consider these four parameters:

1. A process switches from the running to the waiting state.
2. Specific process switches from the running state to the ready state.
3. Specific process switches from the waiting state to the ready state.
4. Process finished its execution and terminated.

Only conditions 1 and 4 apply, the scheduling is called non- preemptive.

All other scheduling are preemptive.

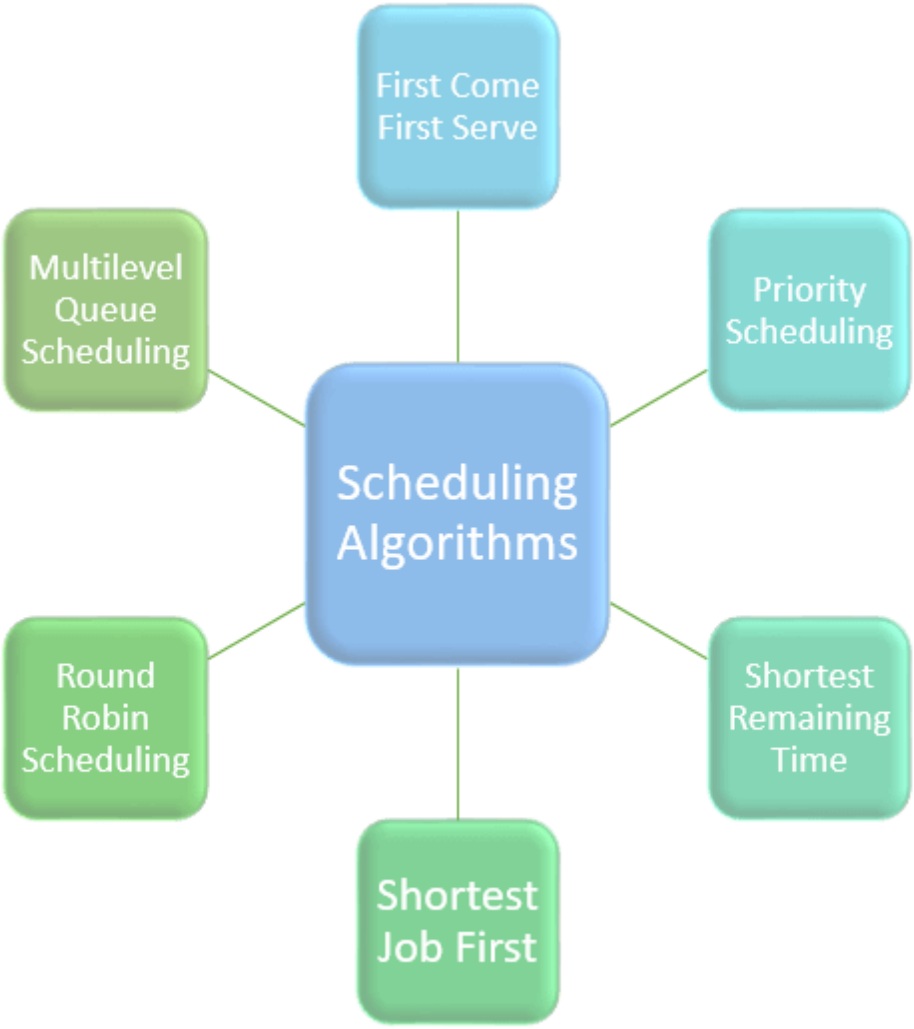
Important CPU scheduling Terminologies

- **Burst Time/Execution Time:** It is a time required by the process to complete execution. It is also called running time.
- **Arrival Time:** when a process enters in a ready state
- **Finish Time:** when process complete and exit from a system
- **Multiprogramming:** A number of programs which can be present in memory at the same time.
- **Jobs:** It is a type of program without any kind of user interaction.
- **User:** It is a kind of program having user interaction.
- **Process:** It is the reference that is used for both job and user.
- **CPU/IO burst cycle:** Characterizes process execution, which alternates between CPU and I/O activity. CPU times are usually shorter than the time of I/O.

Types of CPU scheduling Algorithm

There are mainly six types of process scheduling algorithms

1. First Come First Serve (FCFS)
2. Round Robin Scheduling
3. Shortest-Job-First (SJF) Scheduling
4. Shortest Remaining Time
5. Priority Scheduling
6. Multilevel Queue Scheduling



Scheduling Algorithms

here are various algorithms which are used by the Operating System to schedule the processes on the processor in an efficient way.

There are the following algorithms which can be used to schedule the jobs.

1. First Come First Serve

It is the simplest algorithm to implement. The process with the minimal arrival time will get the CPU first. The lesser the arrival time, the sooner will the process gets the CPU. It is the non-preemptive type of scheduling.

2. Round Robin

In the Round Robin scheduling algorithm, the OS defines a time quantum (slice). All the processes will get executed in the cyclic way. Each of the process will get the CPU for a small amount of time (called time quantum) and then get back to the ready queue to wait for its next turn. It is a preemptive type of scheduling.

3. Shortest Job First

The job with the shortest burst time will get the CPU first. The lesser the burst time, the sooner will the process get the CPU. It is the non-preemptive type of scheduling.

4. Shortest remaining time first

It is the preemptive form of SJF. In this algorithm, the OS schedules the Job according to the remaining time of the execution.

5. Priority based scheduling

In this algorithm, the priority will be assigned to each of the processes. The higher the priority, the sooner will the process get the CPU. If the priority of the two processes is same then they will be scheduled according to their arrival time.

6. Highest Response Ratio Next or multi level queue scheduling

In this scheduling Algorithm, the process with highest response ratio will be scheduled next. This reduces the starvation in the system.

FCFS Scheduling

First come first serve (FCFS) scheduling algorithm simply schedules the jobs according to their arrival time. The job which comes first in the ready queue will get the CPU first. The lesser the arrival time of the job, the sooner will the job get the CPU. FCFS scheduling may cause the problem of starvation if the burst time of the first process is the longest among all the jobs.

Advantages of FCFS

- Simple
- Easy
- First come, First serv

Disadvantages of FCFS

1. The scheduling method is non preemptive, the process will run to the completion.
2. Due to the non-preemptive nature of the algorithm, the problem of starvation may occur.

3. Although it is easy to implement, but it is poor in performance since the average waiting time is higher as compare to other scheduling algorithms.

Example

Let's take an example of The FCFS scheduling algorithm. In the Following schedule, there are 5 processes with process ID **P0, P1, P2, P3 and P4**. P0 arrives at time 0, P1 at time 1, P2 at time 2, P3 arrives at time 3 and Process P4 arrives at time 4 in the ready queue. The processes and their respective Arrival and Burst time are given in the following table.

The Turnaround time and the waiting time are calculated by using the following formula.

1. Turn Around Time = Completion Time - Arrival Time
2. Waiting Time = Turnaround time - Burst Time

The average waiting Time is determined by summing the respective waiting time of all the processes and divided the sum by the total number of processes.

Process ID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Wai
0	0	2	2	2	0
1	1	6	8	7	1
2	2	4	12	8	4
3	3	9	21	18	9
4	4	12	33	29	17

Avg Waiting Time=31/5

(Gantt chart)

Shortest Job First (SJF) Scheduling

Till now, we were scheduling the processes according to their arrival time (in FCFS scheduling). However, SJF scheduling algorithm, schedules the processes according to their burst time.

In SJF scheduling, the process with the lowest burst time, among the list of available processes in the ready queue, is going to be scheduled next.

However, it is very difficult to predict the burst time needed for a process hence this algorithm is very difficult to implement in the system.

Advantages of SJF

1. Maximum throughput
2. Minimum average waiting and turnaround time

Disadvantages of SJF

1. May suffer with the problem of starvation
2. It is not implementable because the exact Burst time for a process can't be known in advance.

There are different techniques available by which, the CPU burst time of the process can be determined. We will discuss them later in detail.

Example

In the following example, there are five jobs named as P1, P2, P3, P4 and P5. Their arrival time and burst time are given in the table below.

PID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting T
1	1	7	8	7	0
2	3	3	13	10	7
3	6	2	10	4	2
4	7	10	31	24	14
5	9	8	21	12	4

Since, No Process arrives at time 0 hence; there will be an empty slot in the **Gantt chart** from time 0 to 1 (the time at which the first process arrives).

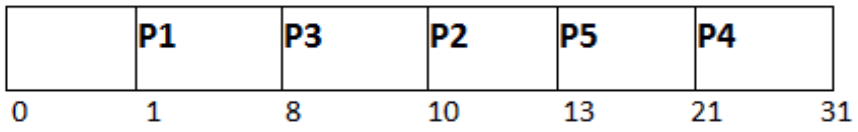
According to the algorithm, the OS schedules the process which is having the lowest burst time among the available processes in the ready queue.

Till now, we have only one process in the ready queue hence the scheduler will schedule this to the processor no matter what is its burst time.

This will be executed till 8 units of time. Till then we have three more processes arrived in the ready queue hence the scheduler will choose the process with the lowest burst time.

Among the processes given in the table, P3 will be executed next since it is having the lowest burst time among all the available processes.

So that's how the procedure will go on in **shortest job first (SJF)** scheduling algorithm.



Avg Waiting Time = 27/5

Round Robin Scheduling Algorithm

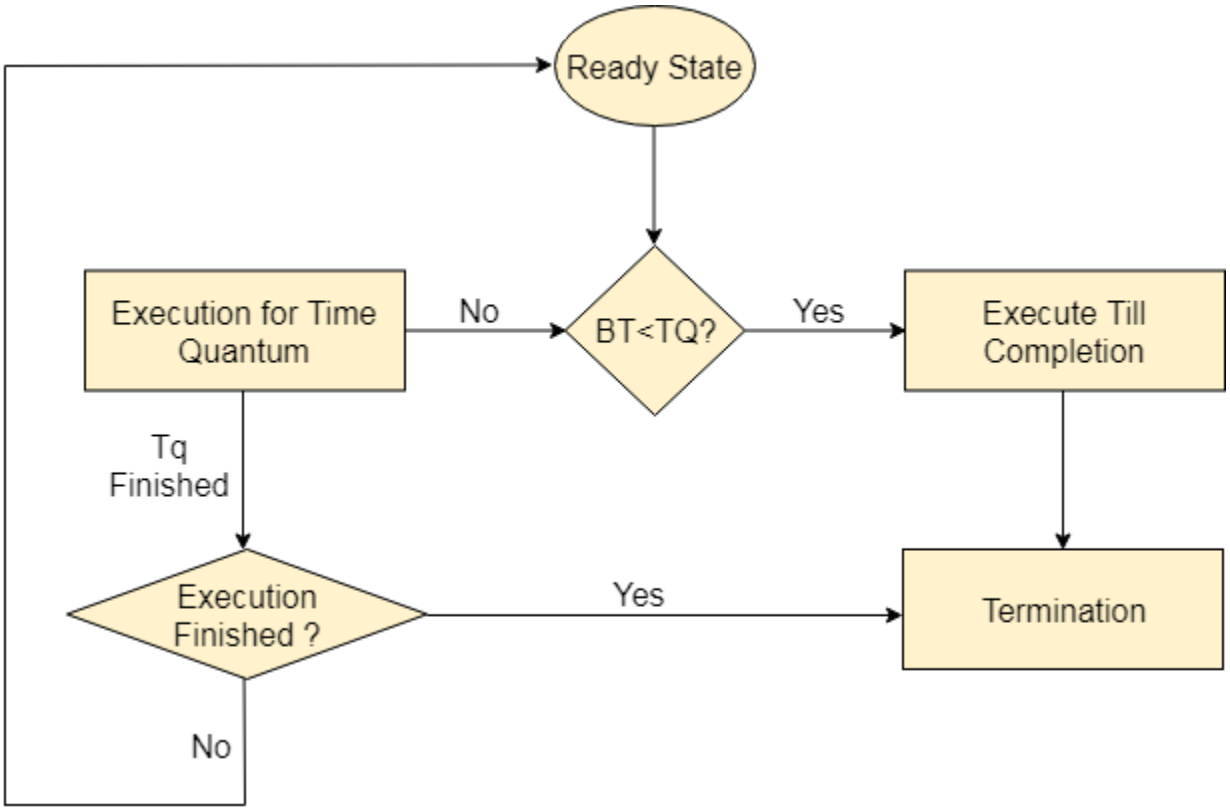
Round Robin scheduling algorithm is one of the most popular scheduling algorithm which can actually be implemented in most of the operating systems. This is the **preemptive version** of first come first serve scheduling. The Algorithm focuses on Time Sharing. In this algorithm, every process gets executed in a **cyclic way**. A certain time slice is defined in the system which is called time **quantum**. Each process present in the ready queue is assigned the CPU for that time quantum, if the execution of the process is completed during that time then the process will **terminate** else the process will go back to the **ready queue** and waits for the next turn to complete the execution

Advantages

1. It can be actually implementable in the system because it is not depending on the burst time.
2. It doesn't suffer from the problem of starvation or convoy effect.
3. All the jobs get a fare allocation of CPU.

. Disadvantages

1. The higher the time quantum, the higher the response time in the system.
2. The lower the time quantum, the higher the context switching overhead in the system.
3. Deciding a perfect time quantum is really a very difficult task in the system.



RR Scheduling Example

In the following example, there are six processes named as P1, P2, P3, P4, P5 and P6. Their arrival time and burst time are given below in the table. The time quantum of the system is 4 units.

Process ID	Arrival Time	Burst Time
1	0	5
2	1	6
3	2	3
4	3	1
5	4	5
6	6	4

According to the algorithm, we have to maintain the ready queue and the Gantt chart. The structure of both the data structures will be changed after every scheduling.

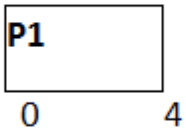
Ready Queue:

Initially, at time 0, process P1 arrives which will be scheduled for the time slice 4 units. Hence in the ready queue, there will be only one process P1 at starting with CPU burst time 5 units.

P1
5

GANTT chart

The P1 will be executed for 4 units first.



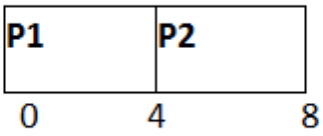
Ready Queue

Meanwhile the execution of P1, four more processes P2, P3, P4 and P5 arrives in the ready queue. P1 has not completed yet, it needs another 1 unit of time hence it will also be added back to the ready queue.

P2	P3	P4	P5	P1
6	3	1	5	1

GANTT chart

After P1, P2 will be executed for 4 units of time which is shown in the Gantt chart.



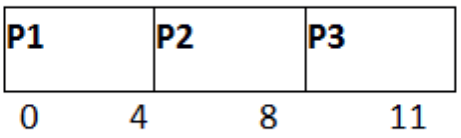
Ready Queue

During the execution of P2, one more process P6 is arrived in the ready queue. Since P2 has not completed yet hence, P2 will also be added back to the ready queue with the remaining burst time 2 units.

P3	P4	P5	P1	P6	P2
3	1	5	1	4	2

GANTT chart

After P1 and P2, P3 will get executed for 3 units of time since its CPU burst time is only 3 seconds.



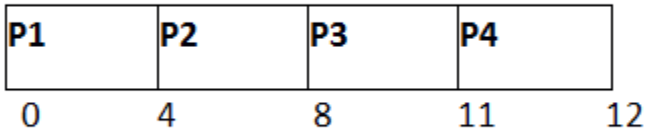
Ready Queue

Since P3 has been completed, hence it will be terminated and not be added to the ready queue. The next process will be executed is P4.

P4	P5	P1	P6	P2
1	5	1	4	2

GANTT chart

After, P1, P2 and P3, P4 will get executed. Its burst time is only 1 unit which is lesser then the time quantum hence it will be completed.



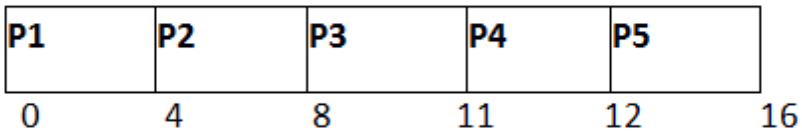
Ready Queue

The next process in the ready queue is P5 with 5 units of burst time. Since P4 is completed hence it will not be added back to the queue.

P5	P1	P6	P2
5	1	4	2

GANTT chart

P5 will be executed for the whole time slice because it requires 5 units of burst time which is higher than the time slice.



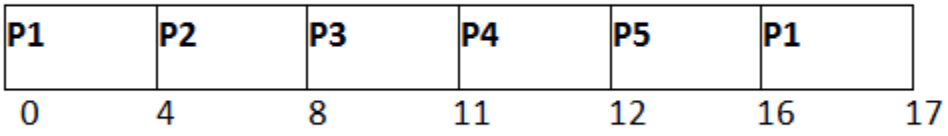
Ready Queue

P5 has not been completed yet; it will be added back to the queue with the remaining burst time of 1 unit.

P1	P6	P2	P5
1	4	2	1

GANTT Chart

The process P1 will be given the next turn to complete its execution. Since it only requires 1 unit of burst time hence it will be completed.



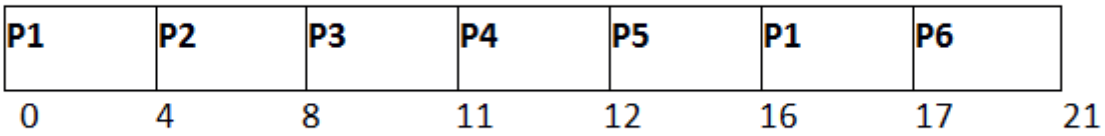
Ready Queue

P1 is completed and will not be added back to the ready queue. The next process P6 requires only 4 units of burst time and it will be executed next.

P6	P2	P5
4	2	1

GANTT chart

P6 will be executed for 4 units of time till completion.



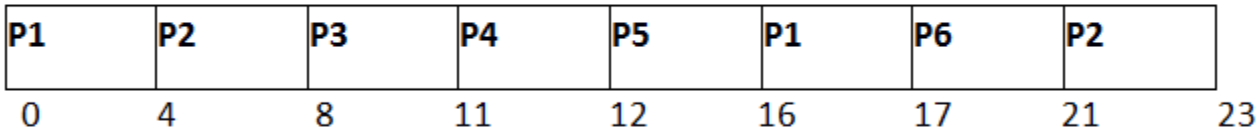
Ready Queue

Since P6 is completed, hence it will not be added again to the queue. There are only two processes present in the ready queue. The Next process P2 requires only 2 units of time.

P2	P5
2	1

GANTT Chart

P2 will get executed again, since it only requires only 2 units of time hence this will be completed.



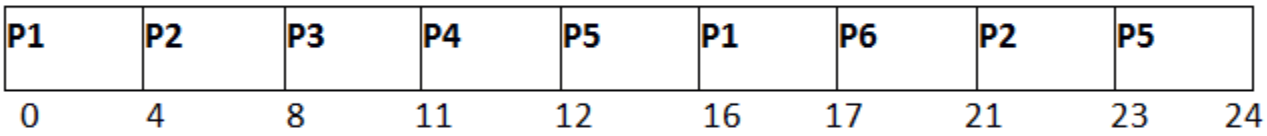
Ready Queue

Now, the only available process in the queue is P5 which requires 1 unit of burst time. Since the time slice is of 4 units hence it will be completed in the next burst.

P5
1

GANTT chart

P5 will get executed till completion.



The completion time, Turnaround time and waiting time will be calculated as shown in the table below.

As, we know,

1. Turn Around Time = Completion Time - Arrival Time
2. Waiting Time = Turn Around Time - Burst Time

rocess ID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time

1	0	5	17	17	12
2	1	6	23	22	16
3	2	3	11	9	6
4	3	1	12	9	8
5	4	5	24	20	15
6	6	4	21	15	11

Avg Waiting Time = (12+16+6+8+15+11)/6 = 76/6 units

Highest Response Ratio Next (HRRN) Scheduling

Highest Response Ratio Next (HRNN) is one of the most optimal scheduling algorithms. This is a non-preemptive algorithm in which, the scheduling is done on the basis of an extra parameter called Response Ratio. A Response Ratio is calculated for each of the available jobs and the Job with the highest response ratio is given priority over the others.

Response Ratio is calculated by the given formula.

Response Ratio = (W+S)/S

Where,

W → Waiting Time
S → Service Time or Burst Time

If we look at the formula, we will notice that the job with the shorter burst time will be given priority but it is also including an extra factor called waiting time. Since,

HRNN W
HRNN (1/S)

Hence,

This algorithm not only favors shorter job but it also concern the waiting time of the longer jobs.

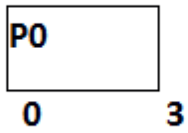
Its mode is non preemptive hence context switching is minimal in this algorithm.

HRNN Example

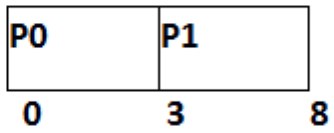
In the following example, there are 5 processes given. Their arrival time and Burst Time are given in the table.

Process ID	Arrival Time	Burst Time
0	0	3
1	2	5
2	4	4
3	6	1
4	8	2

At time 0, The Process P0 arrives with the CPU burst time of 3 units. Since it is the only process arrived till now hence this will get sche immediately.



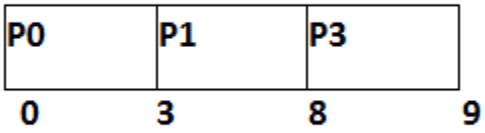
P0 is executed for 3 units, meanwhile, only one process P1 arrives at time 3. This will get scheduled immediately since the OS doesn't h



P1 is executed for 5 units. Meanwhile, all the processes get available. We have to calculate the Response Ratio for all the remaining jobs.

RR (P2) = ((8-4) +4)/4 = 2
RR (P3) = (2+1)/1 = 3
RR (P4) = (0+2)/2 = 1

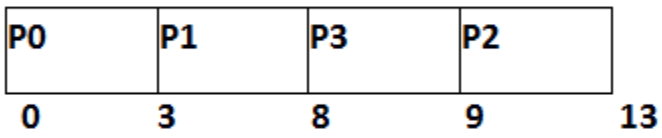
Since, the Response ratio of P3 is higher hence P3 will be scheduled first.



P3 is scheduled for 1 unit. The next available processes are P2 and P4. Let's calculate their Response ratio.

RR (P2) = (5+4)/4 = 2.25
RR (P4) = (1+2)/2 = 1.5

The response ratio of P2 is higher hence P2 will be scheduled.



Now, the only available process is P4 with the burst time of 2 units, since there is no other process available hence this will be scheduled.

Process ID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting T
0	0	3	3	3	0
1	2	5	8	6	1
2	4	4	13	9	5
3	6	1	9	3	2

4	8	2	15	7	5

Average Waiting Time = 13/5

TOPICS LEFT IN UNIT-II

1. Real Time scheduling:
2. RM and
3. EDF.

ALL THE BEST

OPERATING SYSTEMS -U18PCCS404

UNIT-III

UNIT – III IPC AND DEADLOCK **9**

Inter-process Communication: Critical Section, Race Conditions, Mutual Exclusion, Hardware Solution, Strict Alternation, Peterson's Solution, The Producer Consumer Problem, Semaphores, Event Counters, Monitors, Message Passing, Classical IPC Problems: Reader's & Writer Problem, Dinning Philosopher Problem etc.

Deadlocks: Definition, Necessary and sufficient conditions for Deadlock, Deadlock Prevention, Deadlock Avoidance: Banker's algorithm, Deadlock detection and Recovery.

What is Inter Process Communication?

Inter process communication (IPC) is used for exchanging data between multiple threads in one or more processes or programs. The Processes may be running on single or multiple computers connected by a network. The full form of IPC is Inter-process communication.

It is a set of programming interface which allow a programmer to coordinate activities among various program processes which can run concurrently in an operating system. This allows a specific program to handle many user requests at the same time.

Since every single user request may result in multiple processes running in the operating system, the process may require to communicate with each other. Each IPC protocol approach has its own advantage and limitation, so it is not unusual for a single program to use all of the IPC methods

The Critical Section Problem

Critical Section is the part of a program which tries to access shared resources. That resource may be any resource in a computer like a memory location, Data structure, CPU or any IO device.

The critical section cannot be executed by more than one process at the same time; operating system faces the difficulties in allowing and disallowing the processes from entering the critical section.

The critical section problem is used to design a set of protocols which can ensure that the Race condition among the processes will never arise.

In order to synchronize the cooperative processes, our main task is to solve the critical section problem. We need to provide a solution in such a way that the following conditions can be satisfied.

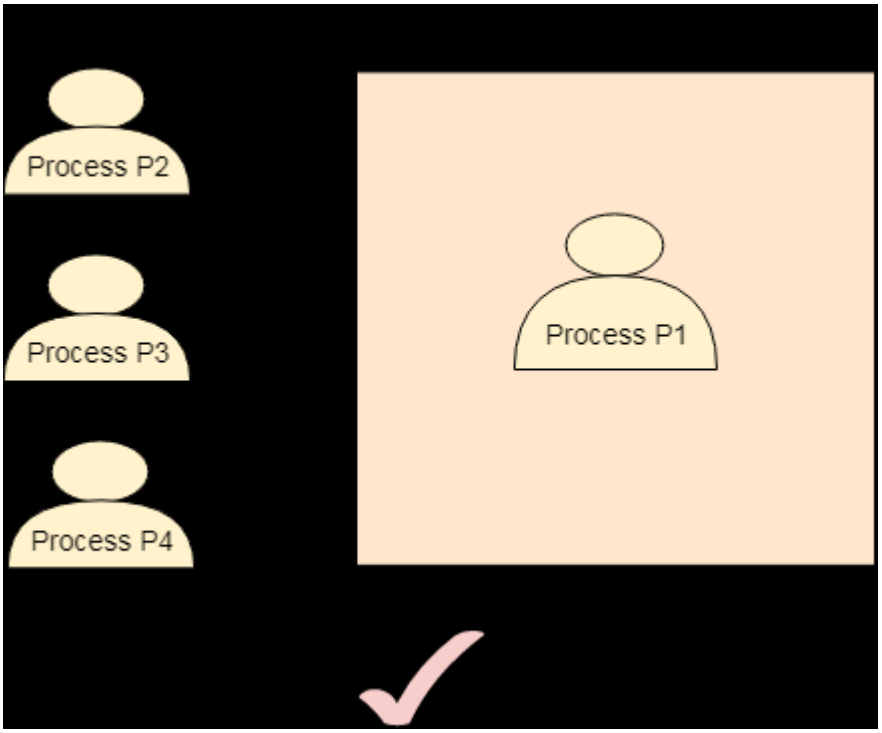
Requirements of Synchronization mechanisms

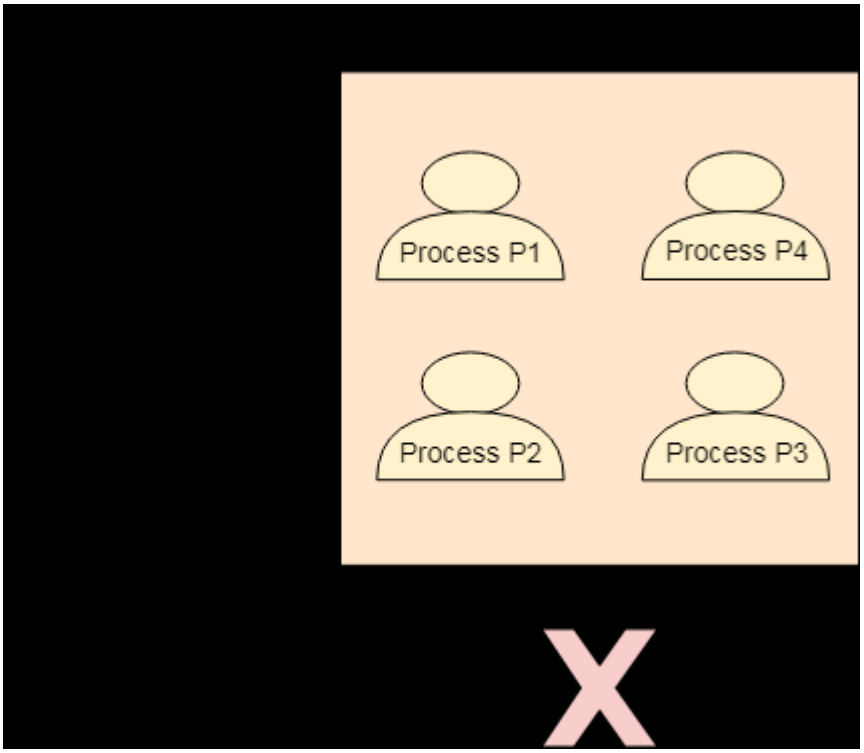
OPERATING SYSTEMS -U18PCCS404
UNIT-III

Primary

1. **Mutual Exclusion**

   Our solution must provide mutual exclusion. By Mutual Exclusion, we mean that if one process is executing inside critical section then the other process must not enter in the critical section.





2. **Progress**

   Progress means that if one process doesn't need to execute into critical section then it should not stop other processes to get into the critical section.

Secondary

1. **Bounded Waiting**

   We should be able to predict the waiting time for every process to get into the critical section. The process must not be endlessly waiting for getting into the critical section.

2. **Architectural Neutrality**

   Our mechanism must be architectural natural. It means that if our solution is working fine on one architecture then it should also run on the other ones as well.

Strict Alternation Approach

Turn Variable or Strict Alternation Approach is the software mechanism implemented at user mode. It is a busy waiting solution which can be implemented only for two processes. In this approach, A turn variable is used which is actually a lock.

This approach can only be used for only two processes. In general, let the two processes be Pi and Pj. They share a variable called turn variable. The pseudo code of the program can be given as following.

For Process Pi

```
Non - CS
while (turn ! = i);
Critical Section
turn = j;
Non - CS
```

For Process Pj

```
Non - CS
while (turn ! = j);
Critical Section
turn = i ;
Non - CS
```

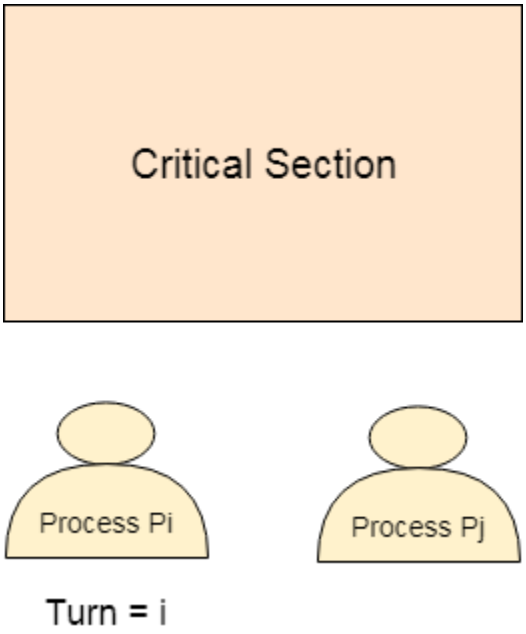
The actual problem of the lock variable approach was the fact that the process was entering in the critical section only when the lock variable is 1. More than one process could see the lock variable as 1 at the same time hence the mutual exclusion was not guaranteed there.

This problem is addressed in the turn variable approach. Now, A process can enter in the critical section only in the case when the value of the turn variable equal to the PID of the process.

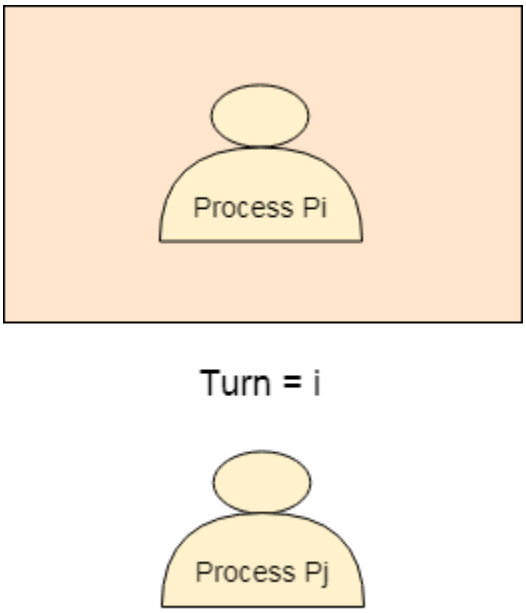
There are only two values possible for turn variable, i or j. if its value is not i then it will definitely be j or vice versa.

In the entry section, in general, the process Pi will not enter in the critical section until its value is j or the process Pj will not enter in the critical section until its value is i.

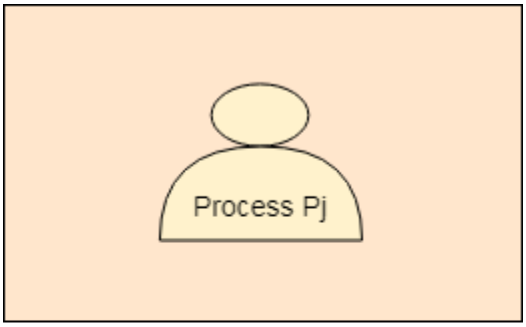
Initially, two processes Pi and Pj are available and want to execute into critical section.



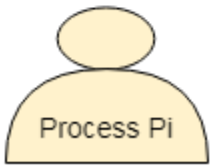
The turn variable is equal to i hence Pi will get the chance to enter into the critical section. The value of Pi remains I until Pi finishes critical section.



Pi finishes its critical section and assigns j to turn variable. Pj will get the chance to enter into the critical section. The value of turn remains j until Pj finishes its critical section.



Turn = j



Analysis of Strict Alternation approach

Let's analyze Strict Alternation approach on the basis of four requirements.

Mutual Exclusion

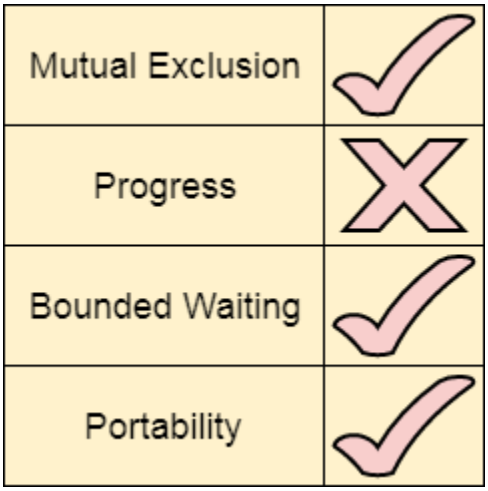
The strict alternation approach provides mutual exclusion in every case. This procedure works only for two processes. The pseudo code is different for both of the processes. The process will only enter when it sees that the turn variable is equal to its Process ID otherwise not Hence No process can enter in the critical section regardless of its turn.

Progress

Progress is not guaranteed in this mechanism. If Pi doesn't want to get enter into the critical section on its turn then Pj got blocked for infinite time. Pj has to wait for so long for its turn since the turn variable will remain 0 until Pi assigns it to j.

Portability

The solution provides portability. It is a pure software mechanism implemented at user mode and doesn't need any special instruction from the Operating System.



Paterson Solution

This is a software mechanism implemented at user mode. It is a busy waiting solution can be implemented for only two processes. It uses two variables that are turn variable and interested variable.

The Code of the solution is given below

```
# define N 2
# define TRUE 1
# define FALSE 0
int interested[N] = FALSE;
int turn;
voidEntry_Section (int process)
{
   int other;
   other = 1-process;
   interested[process] = TRUE;
   turn = process;
   while (interested [other] =True && TURN=process);
}
voidExit_Section (int process)
{
   interested [process] = FALSE;
}
```

Till now, each of our solution is affected by one or the other problem. However, the Peterson solution provides you all the necessary requirements such as Mutual Exclusion, Progress, Bounded Waiting and Portability.

Analysis of Peterson Solution

```
voidEntry_Section (int process)
{
   1. int other;
   2. other = 1-process;
   3. interested[process] = TRUE;
   4. turn = process;
   5. while (interested [other] =True && TURN=process);
}

Critical Section

voidExit_Section (int process)
{
   6. interested [process] = FALSE;
}
```

This is a two process solution. Let us consider two cooperative processes P1 and P2. The entry section and exit section are shown below. Initially, the value of interested variables and turn variable is 0.

Initially process P1 arrives and wants to enter into the critical section. It sets its interested variable to True (instruction line 3) and also sets turn to 1 (line number 4). Since the condition given in line number 5 is completely satisfied by P1 therefore it will enter in the critical section.

P1 → 1 2 3 4 5 CS

Meanwhile, Process P1 got preempted and process P2 got scheduled. P2 also wants to enter in the critical section and executes instructions 1, 2, 3 and 4 of entry section. On instruction 5, it got stuck since it doesn't satisfy the condition (value of other interested variable is still true). Therefore it gets into the busy waiting.

P2 → 1 2 3 4 5

P1 again got scheduled and finish the critical section by executing the instruction no. 6 (setting interested variable to false). Now if P2 checks then it are going to satisfy the condition since other process's interested variable becomes false. P2 will also get enter the critical section.

P1 → 6
P2 → 5 CS

Any of the process may enter in the critical section for multiple numbers of times. Hence the procedure occurs in the cyclic order.

Mutual Exclusion

The method provides mutual exclusion for sure. In entry section, the while condition involves the criteria for two variables therefore a process cannot enter in the critical section until the other process is interested and the process is the last one to update turn variable.

Progress

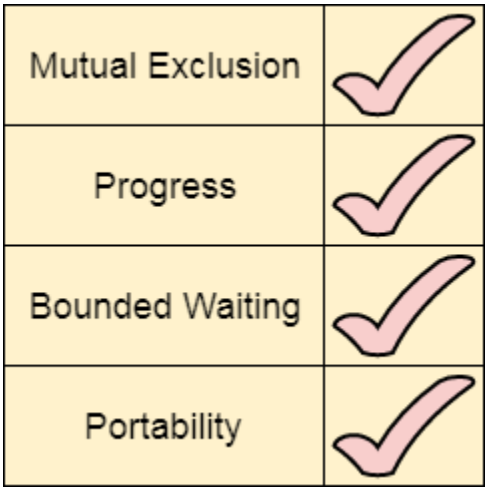
An uninterested process will never stop the other interested process from entering in the critical section. If the other process is also interested then the process will wait.

Bounded waiting

The interested variable mechanism failed because it was not providing bounded waiting. However, in Peterson solution, A deadlock can never happen because the process which first sets the turn variable will enter in the critical section for sure. Therefore, if a process is preempted after executing line number 4 of the entry section then it will definitely get into the critical section in its next chance.

Portability

This is the complete software solution and therefore it is portable on every hardware.



Mutual exclusion

A **mutual exclusion** (mutex) is a program object that prevents simultaneous access to a shared resource. This concept is used in concurrent programming with a critical section, a piece of code in which processes or threads access a shared resource.

Race Condition

A race condition is a situation that may occur inside a critical section. This happens when the result of multiple thread execution in critical section differs according to the order in which the threads execute.

Race conditions in critical sections can be avoided if the critical section is treated as an atomic instruction. Also, proper thread synchronization using locks or atomic variables can prevent race conditions.

Producer Consumer problem

Let's examine the basic model that is sleep and wake. Assume that we have two system calls as **sleep** and **wake**. The process which calls sleep will get blocked while the process which calls will get waked up.

There is a popular example called **producer consumer problem** which is the most popular problem simulating **sleep and wake** mechanism.

The concept of sleep and wake is very simple. If the critical section is not empty then the process will go and sleep. It will be waked up by the other process which is currently executing inside the critical section so that the process can get inside the critical section.

In producer consumer problem, let us say there are two processes, one process writes something while the other process reads that. The process which is writing something is called **producer** while the process which is reading is called **consumer**.

In order to read and write, both of them are usinga buffer. The code that simulates the sleep and wake mechanism in terms of providing the solution to producer consumer problem is shown below.

```
#define N 100 //maximum slots in buffer
#define count=0 //items in the buffer
void producer (void)
{
   int item;
   while(True)
   {
      item = produce_item(); //producer produces an item
      if(count == N) //if the buffer is full then the producer will sleep
         Sleep();
      insert_item (item); //the item is inserted into buffer
      countcount=count+1;
      if(count==1) //The producer will wake up the
      //consumer if there is at least 1 item in the buffer
      wake-up(consumer);
   }
}

void consumer (void)
{
   int item;
   while(True)
   {
      {
         if(count == 0) //The consumer will sleep if the buffer is empty.
         sleep();
         item = remove_item();
         countcount = count - 1;
         if(count == N-1) //if there is at least one slot available in the buffer
         //then the consumer will wake up producer
         wake-up(producer);
         consume_item(item); //the item is read by consumer.
      }
   }
}
```

The producer produces the item and inserts it into the buffer. The value of the global variable count got increased at each insertion. If the buffer is filled completely and no slot is available then the producer will sleep, otherwise it keep inserting.

On the consumer's end, the value of count got decreased by 1 at each consumption. If the buffer is empty at any point of time then the consumer will sleep otherwise, it keeps consuming the items and decreasing the value of count by 1.

The consumer will be waked up by the producer if there is at least 1 item available in the buffer which is to be consumed. The producer will be waked up by the consumer if there is at least one slot available in the buffer so that the producer can write that.

Well, the problem arises in the case when the consumer got preempted just before it was about to sleep. Now the consumer is neither sleeping nor consuming. Since the producer is not aware of the fact that consumer is not actually sleeping therefore it keep waking the consumer while the consumer is not responding since it is not sleeping.

This leads to the wastage of system calls. When the consumer get scheduled again, it will sleep because it was about to sleep when it was preempted.

The producer keep writing in the buffer and it got filled after some time. The producer will also sleep at that time keeping in the mind that the consumer will wake him up when there is a slot available in the buffer.

The consumer is also sleeping and not aware with the fact that the producer will wake him up.

This is a kind of deadlock where neither producer nor consumer is active and waiting for each other to wake them up. This is a serious problem which needs to be addressed.

Using a flag bit to get rid of this problem

A flag bit can be used in order to get rid of this problem. The producer can set the bit when it calls wake-up on the first time. When the consumer got scheduled, it checks the bit.

The consumer will now get to know that the producer tried to wake him and therefore it will not sleep and get into the ready state to consume whatever produced by the producer.

This solution works for only one pair of producer and consumer, what if there are n producers and n consumers. In that case, there is a need to maintain an integer which can record how many wake-up calls have been made and how many consumers need not sleep**. This integer variable is called semaphore.**

We will discuss more about semaphore later in detail.

Introduction to semaphore

To get rid of the problem of wasting the wake-up signals, Dijkstra proposed an approach which involves storing all the wake-up calls. Dijkstra states that, instead of giving the wake-up calls directly to the consumer, producer can store the wake-up call in a variable. Any of the consumers can read it whenever it needs to do so.

Semaphore is the variables which stores the entire wake up calls that are being transferred from producer to consumer. It is a variable on which read, modify and update happens automatically in kernel mode.

Semaphore cannot be implemented in the user mode because race condition may always arise when two or more processes try to access the variable simultaneously. It always needs support from the operating system to be implemented.

According to the demand of the situation, Semaphore can be divided into two categories.

1. Counting Semaphore
2. Binary Semaphore or Mutex

We will discuss each one in detail.

Counting Semaphore

There are the scenarios in which more than one processes need to execute in critical section simultaneously. However, counting semaphore can be used when we need to have more than one process in the critical section at the same time.

The programming code of semaphore implementation is shown below which includes the structure of semaphore and the logic using which the entry and the exit can be performed in the critical section.

In this mechanism, the entry and exit in the critical section are performed on the basis of the value of counting semaphore. The value of counting semaphore at any point of time indicates the maximum number of processes that can enter in the critical section at the same time.

A process which wants to enter in the critical section first decrease the semaphore value by 1 and then check whether it gets negative or not. If it gets negative then the process is pushed in the list of blocked processes (i.e. q) otherwise it gets enter in the critical section.

When a process exits from the critical section, it increases the counting semaphore by 1 and then checks whether it is negative or zero. If it is negative then that means that at least one process is waiting in the blocked state hence, to ensure bounded waiting, the first process among the list of blocked processes will wake up and gets enter in the critical section.

The processes in the blocked list will get waked in the order in which they slept. If the value of counting semaphore is negative then it states the number of processes in the blocked state while if it is positive then it states the number of slots available in the critical section.

```
struct Semaphore
{
   int value; // processes that can enter in the critical section simultaneously.
   queue type L; // L contains set of processes which get blocked
}
Down (Semaphore S)
{
   SS.value = S.value - 1; //semaphore's value will get decreased when a new
   //process enter in the critical section
   if (S.value< 0)
   {
      put_process(PCB) in L; //if the value is negative then
      //the process will get into the blocked state.
      Sleep();
   }
   else
      return;
}
up (Semaphore s)
{
   SS.value = S.value+1; //semaphore value will get increased when
   //it makes an exit from the critical section.
   if(S.value<=0)
   {
      select a process from L; //if the value of semaphore is positive
      //then wake one of the processes in the blocked queue.
      wake-up();
   }
   }
}
```

Problem on Counting Semaphore

.Generally the questions are very simple that contains only subtraction and addition.

Wait → Decre → Down → P
Signal → Inc → Up → V

A Counting Semaphore was initialized to 12. then 10P (wait) and 4V (Signal) operations were computed on this semaphore. What is the result?

S = 12 (initial)
10 p (wait) :
SS = S -10 = 12 - 10 = 2
then 4 V :
SS = S + 4 =2 + 4 = 6

Hence, the final value of counting semaphore is 6.

Binary Semaphore or Mutex

In counting semaphore, Mutual exclusion was not provided because we has the set of processes which required to execute in the critical section simultaneously.

However, Binary Semaphore strictly provides mutual exclusion. Here, instead of having more than 1 slots available in the critical section, we can only have at most 1 process in the critical section. The semaphore can have only two values, 0 or 1.

Let's see the programming implementation of Binary Semaphore.

```
StructBsemaphore

{

   enum Value(0,1); //value is enumerated data type which can only have two values 0 or 1.

   Queue type L;

}

/* L contains all PCBs corresponding to process

Blocked while processing down operation unsuccessfully.

*/

Down (Bsemaphore S)

{

   if (s.value == 1) // if a slot is available in the

   //critical section then let the process enter in the queue.
```

```
    {
        S.value = 0; // initialize the value to 0 so that no other process can read it as 1.
    }
    else
    {
        put the process (PCB) in S.L; //if no slot is available
        //then let the process wait in the blocked queue.
        sleep();
    }
}
Up (Bsemaphore S)
{
    if (S.L is empty) //an empty blocked processes list implies that no process
    //has ever tried to get enter in the critical section.
    {
        S.Value =1;
    }
    else
    {
        Select a process from S.L;
        Wakeup(); // if it is not empty then wake the first process of the blocked queue.
    }
}
```

Message Passing:

It is a mechanism for a process to communicate and synchronize. Using message passing, the process communicates with each other without resorting to shared variables.

IPC mechanism provides two operations:

- Send (message)- message size fixed or variable
- Received (message)

What is Readers Writer Problem?

Readers writer problem is another example of a classic synchronization problem. There are many variants of this problem, one of which is examined below.

The Problem Statement

There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context. They are **reader** and **writer**. Any number of **readers** can read from the shared resource simultaneously, but only one **writer** can write to the shared resource. When a **writer** is writing data to the resource, no other process can access the resource. A **writer** cannot write to the resource if there are non zero number of readers accessing the resource at that time.

The Solution

From the above problem statement, it is evident that readers have higher priority than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource.

Here, we use one **mutex** `m` and a **semaphore** `w`. An integer variable `read_count` is used to maintain the number of readers currently accessing the resource. The variable `read_count` is initialized to `0`. A value of `1` is given initially to `m` and `w`.

Instead of having the process to acquire lock on the shared resource, we use the mutex `m` to make the process to acquire and release lock whenever it is updating the `read_count` variable.

The code for the **writer** process looks like this:

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

   /* perform the write operation */
```

```
   signal(w);
}
```

And, the code for the **reader** process looks like this:

```
while(TRUE)
{
    //acquire lock
    wait(m);
    read_count++;
    if(read_count == 1)
        wait(w);

    //release lock
    signal(m);

    /* perform the reading operation */

    // acquire lock
    wait(m);
    read_count--;
    if(read_count == 0)
        signal(w);

    // release lock
    signal(m);
}
```

Here is the Code uncoded(explained)

- As seen above in the code for the writer, the writer just waits on the **w** semaphore until it gets a chance to write to the resource.
- After performing the write operation, it increments **w** so that the next writer can access the resource.

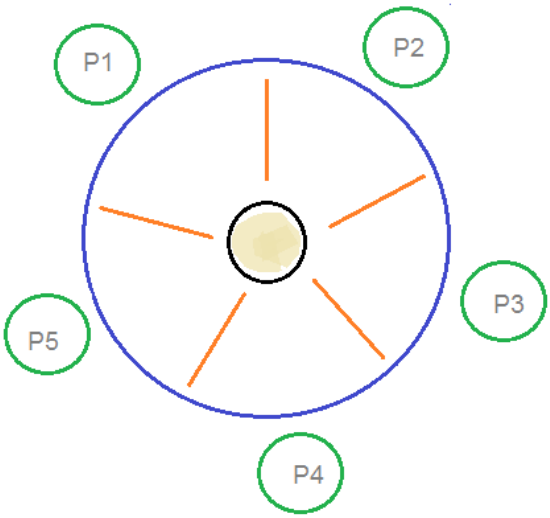
- On the other hand, in the code for the reader, the lock is acquired whenever the **read_count** is updated by a process.
- When a reader wants to access the resource, first it increments the **read_count** value, then accesses the resource and then decrements the **read_count** value.
- The semaphore **w** is used by the first reader which enters the critical section and the last reader which exits the critical section.
- The reason for this is, when the first readers enters the critical section, the writer is blocked from the resource. Only new readers can access the resource now.
- Similarly, when the last reader exits the critical section, it signals the writer using the **w** semaphore because there are zero readers now and a writer can have the chance to access the resource.

Dining Philosophers Problem

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

What is the Problem Statement?

Consider there are five philosophers sitting around a circular dining table. The dining table has five chopsticks and a bowl of rice in the middle as shown in the below figure.



Dining Philosophers Problem

At any instant, a philosopher is either eating or thinking. When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.

Here's the Solution

From the problem statement, it is clear that a philosopher can think for an indefinite amount of time. But when a philosopher starts eating, he has to stop at some point of time. The philosopher is in an endless cycle of thinking and eating.

An array of five semaphores, `stick[5]`, for each of the five chopsticks.

The code for each philosopher looks like:

```
while(TRUE)
{
    wait(stick[i]);
    /*
        mod is used because if i=5, next
        chopstick is 1 (dining table is circular)
    */
    wait(stick[(i+1) % 5]);

    /* eat */
    signal(stick[i]);

    signal(stick[(i+1) % 5]);
    /* think */
}
```

When a philosopher wants to eat the rice, he will wait for the chopstick at his left and picks up that chopstick. Then he waits for the right chopstick to be available, and then picks it too. After eating, he puts both the chopsticks down.

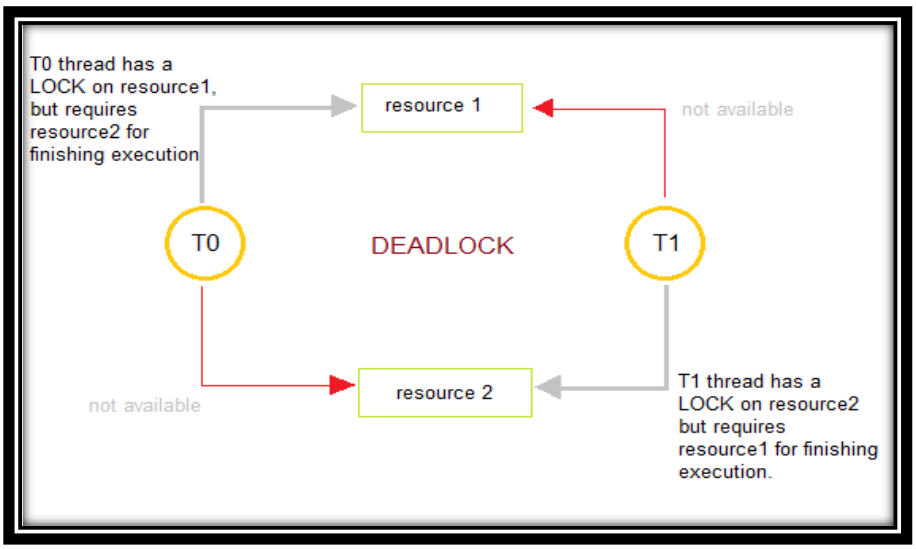
But if all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever. The possible solutions for this are:

- A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.

- Allow only four philosophers to sit at the table. That way, if all the four philosophers pick up four chopsticks, there will be one chopstick left on the table. So, one philosopher can start eating and eventually, two chopsticks will be available. In this way, deadlocks can be avoided.

Deadlock?

Deadlocks are a set of blocked processes each holding a resource and waiting to acquire a resource held by another process.



Every process needs some resources to complete its execution. However, the resource is granted in a sequential order.

1. The process requests for some resource.
2. OS grant the resource if it is available otherwise let the process waits.
3. The process uses it and release on the completion.

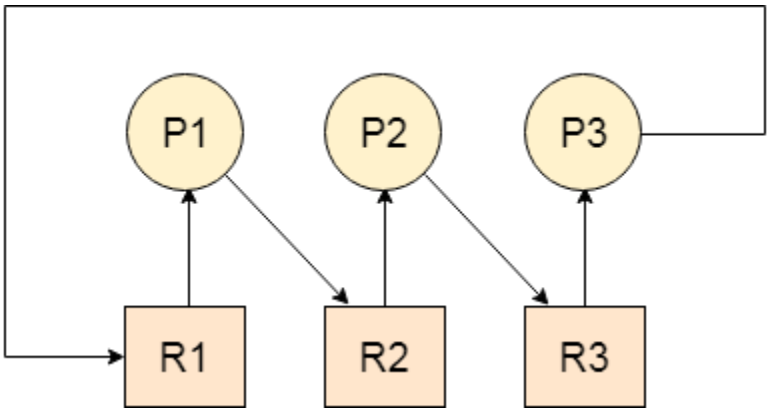
A Deadlock is a situation where each of the computer process waits for a resource which is being assigned to some another process. In this situation, none of the process gets executed since the resource it needs, is held by some other process which is also waiting for some other resource to be released.

Let us assume that there are three processes P1, P2 and P3. There are three different resources R1, R2 and R3. R1 is assigned to P1, R2 is assigned to P2 and R3 is assigned to P3.

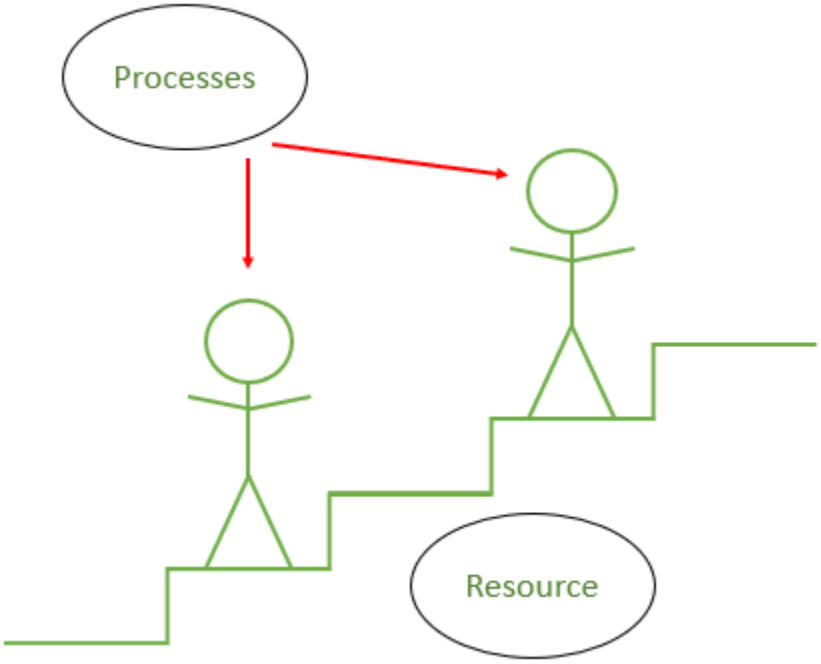
After some time, P1 demands for R1 which is being used by P2. P1 halts its execution since it can't complete without R2. P2 also demands for R3 which is being used by P3. P2 also stops its execution

because it can't continue without R3. P3 also demands for R1 which is being used by P1 therefore P3 also stops its execution.

In this scenario, a cycle is being formed among the three processes. None of the process is progressing and they are all waiting. The computer becomes unresponsive since all the processes got blocked.



Necessary and sufficient conditions for Deadlock (or)



Deadlock prevention

Deadlock happens only when Mutual Exclusion, hold and wait, No preemption and circular wait holds simultaneously. If it is possible to violate one of the four conditions at any time then the deadlock can never occur in the system.

The idea behind the approach is very simple that we have to fail one of the four conditions but there can be a big argument on its physical implementation in the system.

If we simulate deadlock with a table which is standing on its four legs then we can also simulate four legs with the four conditions which when occurs simultaneously, cause the deadlock.

However, if we break one of the legs of the table then the table will fall definitely. The same happens with deadlock, if we can be able to violate one of the four necessary conditions and don't let them occur together then we can prevent the deadlock.

Let's see how we can prevent each of the conditions.

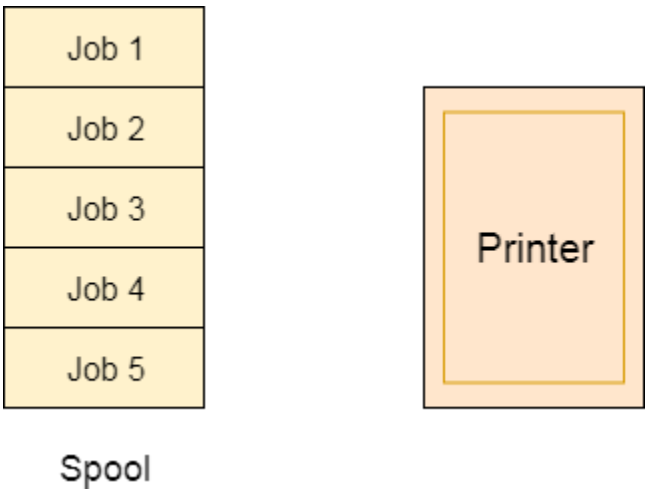
1. Mutual Exclusion

Mutual section from the resource point of view is the fact that a resource can never be used by more than one process simultaneously which is fair enough but that is the main reason behind the deadlock. If a resource could have been used by more than one process at the same time then the process would have never been waiting for any resource.

However, if we can be able to violate resources behaving in the mutually exclusive manner then the deadlock can be prevented.

Spooling

For a device like printer, spooling can work. There is a memory associated with the printer which stores jobs from each of the process into it. Later, Printer collects all the jobs and print each one of them according to FCFS. By using this mechanism, the process doesn't have to wait for the printer and it can continue whatever it was doing. Later, it collects the output when it is produced.



Although, Spooling can be an effective approach to violate mutual exclusion but it suffers from two kinds of problems.

1. This cannot be applied to every resource.
2. After some point of time, there may arise a race condition between the processes to get space in that spool.

We cannot force a resource to be used by more than one process at the same time since it will not be fair enough and some serious problems may arise in the performance. Therefore, we cannot violate mutual exclusion for a process practically.

2. Hold and Wait

Hold and wait condition lies when a process holds a resource and waiting for some other resource to complete its task. Deadlock occurs because there can be more than one process which are holding one resource and waiting for other in the cyclic order.

However, we have to find out some mechanism by which a process either doesn't hold any resource or doesn't wait. That means, a process must be assigned all the necessary resources before the execution starts. A process must not wait for any resource once the execution has been started.

!(Hold and wait) = !hold or !wait (negation of hold and wait is, either you don't hold or you don't wait)

This can be implemented practically if a process declares all the resources initially. However, this sounds very practical but can't be done in the computer system because a process can't determine necessary resources initially.

Process is the set of instructions which are executed by the CPU. Each of the instruction may demand multiple resources at the multiple times. The need cannot be fixed by the OS.

The problem with the approach is:

1. Practically not possible.
2. Possibility of getting starved will be increases due to the fact that some process may hold a resource for a very long time.

3. No Preemption

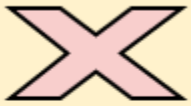
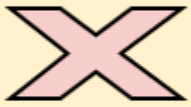
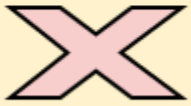
Deadlock arises due to the fact that a process can't be stopped once it starts. However, if we take the resource away from the process which is causing deadlock then we can prevent deadlock.

This is not a good approach at all since if we take a resource away which is being used by the process then all the work which it has done till now can become inconsistent.

Consider a printer is being used by any process. If we take the printer away from that process and assign it to some other process then all the data which has been printed can become inconsistent and ineffective and also the fact that the process can't start printing again from where it has left which causes performance inefficiency.

4. Circular Wait

To violate circular wait, we can assign a priority number to each of the resource. A process can't request for a lesser priority resource. This ensures that not a single process can request a resource which is being utilized by some other process and no cycle will be formed.

Condition	Approach	Is Practically Possible?
Mutual Exclusion	Spooling	
Hold and Wait	Request for all the resources initially	
No Preemption	Snatch all the resources	
Circular Wait	Assign priority to each resources and order resources numerically	

Among all the methods, violating Circular wait is the only approach that can be implemented practically.

Deadlock avoidance

In deadlock avoidance, the request for any resource will be granted if the resulting state of the system doesn't cause deadlock in the system. The state of the system will continuously be checked for safe and unsafe states.

In order to avoid deadlocks, the process must tell OS, the maximum number of resources a process can request to complete its execution.

The simplest and most useful approach states that the process should declare the maximum number of resources of each type it may ever need. The Deadlock avoidance algorithm examines the resource allocations so that there can never be a circular wait condition.

Safe and Unsafe States

The resource allocation state of a system can be defined by the instances of available and allocated resources, and the maximum instance of the resources demanded by the processes.

A state of a system recorded at some random time is shown below.

Resources Assigned

Process	Type 1	Type 2	Type 3	Type 4
A	3	0	2	2
B	0	0	1	1
C	1	1	1	0
D	2	1	4	0

Resources still needed

Process	Type 1	Type 2	Type 3	Type 4
A	1	1	0	0
B	0	1	1	2
C	1	2	1	0
D	2	1	1	2

E = (7 6 8 4)
P = (6 2 8 3)
A = (1 4 0 1)

Above tables and vector E, P and A describes the resource allocation state of a system. There are 4 processes and 4 types of the resources in a system. Table 1 shows the instances of each resource assigned to each process.

Table 2 shows the instances of the resources, each process still needs. Vector E is the representation of total instances of each resource in the system.

Vector P represents the instances of resources that have been assigned to processes. Vector A represents the number of resources that are not in use.

A state of the system is called safe if the system can allocate all the resources requested by all the processes without entering into deadlock.

If the system cannot fulfill the request of all processes then the state of the system is called unsafe.

The key of Deadlock avoidance approach is when the request is made for resources then the request must only be approved in the case if the resulting state is also a safe state.

Bankers algorithm:

Banker's Algorithm is a resource allocation and deadlock avoidance algorithm. This algorithm test for safety simulating the allocation for predetermined maximum possible amounts of all resources, then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue.

In simple terms, it checks if allocation of any resource will lead to deadlock or not, OR is it safe to allocate a resource to a process and if not then resource is not allocated to that process. Determining a safe sequence(even if there is only 1) will assure that system will not go into deadlock.

Banker's algorithm is generally used to find if a safe sequence exist or not. But here we will determine the total number of safe sequences and print all safe sequences.

Following **Data structures** are used to implement the Banker's Algorithm:

- Available vector
- Max Matrix
- Allocation Matrix
- Need Matrix

Let **'n'** be the number of processes in the system and **'m'** be the number of resources types.

Available :

- It is a 1-d array of size **'m'** indicating the number of available resources of each type.
- Available[j] = k means there are **'k'** instances of resource type $\mathbf{R_j}$

Max :

- It is a 2-d array of size '**n*m'** that defines the maximum demand of each process in a system.
- Max[i, j] = k means process $\mathbf{P_i}$ may request at most **'k'** instances of resource type $\mathbf{R_j}$.

Allocation :

- It is a 2-d array of size **'n*m'** that defines the number of resources of each type currently allocated to each process.
- Allocation[i, j] = k means process $\mathbf{P_i}$ is currently allocated **'k'** instances of resource type $\mathbf{R_j}$

Need :

- It is a 2-d array of size **'n*m'** that indicates the remaining resource need of each process.
- Need [i, j] = k means process $\mathbf{P_i}$ currently need **'k'** instances of resource type $\mathbf{R_j}$

for its execution.

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

Example:

Input:

Total Resources	R1	R2	R3
	10	5	7

Process	Allocation			Max		
	R1	R2	R3	R1	R2	R3
P1	0	1	0	7	5	3
P2	2	0	0	3	2	2
P3	3	0	2	9	0	2
P4	2	1	1	2	2	2

```
Output: Safe sequences are:
P2--> P4--> P1--> P3
P2--> P4--> P3--> P1
P4--> P2--> P1--> P3
P4--> P2--> P3--> P1

There are total 4 safe-sequences
```

Explanation:

Total resources are R1 = 10, R2 = 5, R3 = 7 and allocated resources are R1 = (0+2+3+2 =) 7, R2 = (1+0+0+1 =) 2, R3 = (0+0+2+1 =) 3. Therefore, remaining resources are R1 = (10 – 7 =) 3, R2 = (5 – 2 =) 3, R3 = (7 – 3 =) 4.

Remaining available = Total resources – allocated resources

and

Remaining need = max – allocated

Process	Max			Allocation			Available			Needed		
	R1	R2	R3	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	7	5	3	0	1	0	3	3	4	7	4	3
P2	3	2	2	2	0	0				1	2	2
P3	9	0	2	3	0	2				6	0	0
P4	2	2	2	2	1	1				0	1	1
				7	2	3						

So, we can start from either P2 or P4. We can not satisfy remaining need from available resources of either P1 or P3 in first or second attempt step of Banker's algorithm. There are only four possible safe sequences. These are :

P2–> P4–> P1–> P3
P2–> P4–> P3–> P1
P4–> P2–> P1–> P3
P4–> P2–> P3–> P1

Banker's algorithm consists of Safety algorithm and Resource request algorithm

Safety Algorithm:

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

1) Let Work and Finish be vectors of length 'm' and 'n' respectively.
Initialize: Work = Available
Finish[i] = false; for i=1, 2, 3, 4….n

2) Find an i such that both
a) Finish*[i] = false*
b) Need$_i$ <= Work
if no such i exists goto step (4)
3) Work = Work + Allocation[i]
Finish[i] = true
goto step (2)

4) if Finish [i] = true for all i
then the system is in a safe state

Resource-Request Algorithm:

Let Request$_i$ be the request array for process P_i. Request$_i$ [j] = k means process P_i wants k instances of resource type R_j. When a request for resources is made by process P_i, the following actions are taken:

1) If Request$_i$ <= Need$_i$
Goto step (2) ; otherwise, raise an error condition, since the process has exceeded its maximum claim.
2) If Request$_i$ <= Available
Goto step (3); otherwise, P_i must wait, since the resources are not available.
3) Have the system pretend to have allocated the requested resources to process Pi by modifying the state as
follows:
Available = Available – Requesti
Allocation$_i$ = Allocation$_i$ + Request$_i$
Need$_i$ = Need$_i$– Request$_i$

Deadlock detection and recovery:

When a Deadlock Detection Algorithm determines that a deadlock has occurred in the system, the system must recover from that deadlock…..(or)

This approach let the processes fall in deadlock and then periodically check whether deadlock occur in the system or not. If it occurs then it applies some of the recovery methods to the system to get rid of deadlock.

There are two approaches of breaking a Deadlock:

OPERATING SYSTEMS -U18PCCS404

UNIT-III

1. Process Termination:

To eliminate the deadlock, we can simply kill one or more processes. For this, we use two methods:

- **(a). Abort all the Deadlocked Processes:**
  Aborting all the processes will certainly break the deadlock, but with a great expenses. The deadlocked processes may have computed for a long time and the result of those partial computations must be discarded and there is a probability to recalculate them later.

- **(b). Abort one process at a time untill deadlock is eliminated:**
  Abort one deadlocked process at a time, untill deadlock cycle is eliminated from the system. Due to this method, there may be considerable overhead, because after aborting each process, we have to run deadlock detection algorithm to check whether any processes are still deadlocked.

2. Resource Preemption:

To eliminate deadlocks using resource preemption, we preepmt some resources from processes and give those resources to other processes. This method will raise three issues –

- **(a). Selecting a victim:**
  We must determine which resources and which processes are to be preempted and also the order to minimize the cost.
- **(b). Rollback:**
  We must determine what should be done with the process from which resources are preempted. One simple idea is total rollback. That means abort the process and restart it.
- **(c). Starvation:**
  In a system, it may happen that same process is always picked as a victim. As a result, that process will never complete its designated task. This situation is called **Starvation** and must be avoided. One solution is that a process must be picked as a victim only a finite number of times.

All the best

UNIT-IV MEMORY MANAGEMENT

UNIT – IV MEMORY MANAGEMENT **9**

Memory Management: Basic concept, Logical and Physical address map, Memory allocation – Contiguous Memory allocation – Fixed and variable partition – Internal and External fragmentation and Compaction, Paging – Principle of operation – Page allocation – Hardware support for paging, Protection and Sharing, Disadvantages of Paging.

Virtual Memory: Basics of Virtual Memory – Hardware and control structures – Locality of reference, Page fault , Working Set , Dirty page/Dirty bit – Demand paging, Page Replacement algorithms: Optimal, First in First Out (FIFO), Second Chance (SC), Not recently used (NRU) and Least Recently used (LRU

MEMORY:

Computer memory can be defined as a collection of some data represented in the binary format. On the basis of various functions, memory can be classified into various categories.

A computer device that is capable to store any information or data temporally or permanently, is called storage device.

MEMORY MANAGEMENT:

- Memory management is the functionality of an operating system which handles or manages primary memory and moves processes back and forth between main memory and disk during execution.
- Memory management keeps track of each and every memory location, regardless of either it is allocated to some process or it is free.
- It checks how much memory is to be allocated to processes.
- It decides which process will get memory at what time.
- It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.

Memory Management Unit

The purpose of Memory Management Unit (MMU) is to convert the logical address into the physical address. The logical address is the address generated by the CPU for every page while the physical address is the actual address of the frame where each page will be stored.

When a page is to be accessed by the CPU by using the logical address, the operating system needs to obtain the physical address to access that page physically.

The logical address has two parts.

1. Page Number
2. Offset

Memory management unit of OS needs to convert the page number to the frame number.

Example:

Considering the above image, let's say that the CPU demands 10th word of 4th page of process P3. Since the page number 4 of process P1 gets stored at frame number 9 therefore the 10th word of 9th frame will be returned as the physical address.

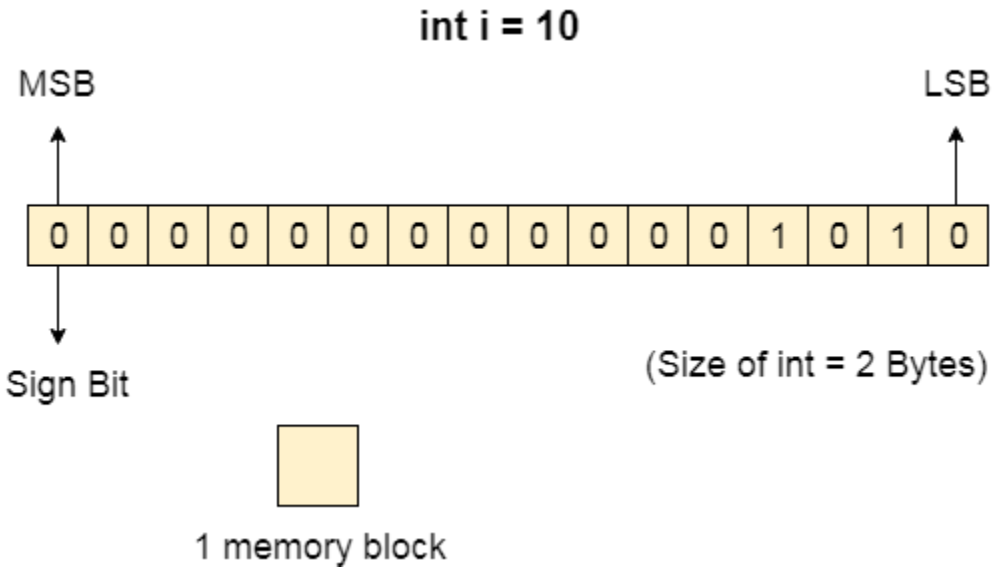
How Data is being stored in a computer system?

In order to understand memory management, we have to make everything clear about how data is being stored in a computer system.

Machine understands only binary language that is 0 or 1. Computer converts every data into binary language first and then stores it into the memory.

That means if we have a program line written as **int a = 10** then the computer converts it into the binary language and then store it into the memory blocks.

The representation of **inti = 10** is shown below.



The binary representation of 10 is 1010. Here, we are considering 32 bit system therefore, the size of int is 2 bytes i.e. 16 bit. 1 memory block stores 1 bit. If we are using signed integer then the most significant bit in the memory array is always a signed bit.

Signed bit value 0 represents positive integer while 1 represents negative integer. Here, the range of values that can be stored using the memory array is -32768 to +32767.

well, we can enlarge this range by using unsigned int. in that case, the bit which is now storing the sign will also store the bit value and therefore the range will be 0 to 65,535.

Physical and Logical Address Space

Physical Address Space

Physical address space in a system can be defined as the size of the main memory. It is really important to compare the process size with the physical address space. The process size must be less than the physical address space.

Physical Address Space = Size of the Main Memory

If, physical address space = 64 KB = 2 ^ 6 KB = 2 ^ 6 X 2 ^ 10 Bytes = 2 ^ 16 bytes

Let us consider,
word size = 8 Bytes = 2 ^ 3 Bytes

Hence,
Physical address space (in words) = (2 ^ 16) / (2 ^ 3) = 2 ^ 13 Words

Therefore,
Physical Address = 13 bits

In General,
If, Physical Address Space = N Words

then, Physical Address = $\log_2 N$

Logical Address Space

Logical address space can be defined as the size of the process. The size of the process should be less enough so that it can reside in the main memory.

Let's say,

Logical Address Space = 128 MB = (2 ^ 7 X 2 ^ 20) Bytes = 2 ^ 27 Bytes
Word size = 4 Bytes = 2 ^ 2 Bytes

Logical Address Space (in words) = (2 ^ 27) / (2 ^ 2) = 2 ^ 25 Words
Logical Address = 25 Bits

In general,
If, logical address space = L words
Then, Logical Address = Log_2L bits

What is a Word?

The Word is the smallest unit of the memory. It is the collection of bytes. Every operating system defines different word sizes after analyzing the n-bit address that is inputted to the decoder and the 2 ^ n memory locations that are produced from the decoder.

MEMORY ALLOCATION

Contiguous memory allocation.

The earliest and one of the simplest technique which can be used to load more than one processes into the main memory is Fixed partitioning or Contiguous memory allocation.

In this technique, the main memory is divided into partitions of equal or different sizes. The operating system always resides in the first partition while the other partitions can be used to store user processes. The memory is assigned to the processes in contiguous way.

IN contiguous memory allocation,

1. The partitions cannot overlap.
2. A process must be contiguously present in a partition for the execution.

There are various cons of using this technique.

1. Internal Fragmentation

If the size of the process is lesser then the total size of the partition then some size of the partition get wasted and remain unused. This is wastage of the memory and called internal fragmentation.

As shown in the image below, the 4 MB partition is used to load only 3 MB process and the remaining 1 MB got wasted.

2. External Fragmentation :The total unused space of various partitions cannot be used to load the processes even though there is space available but not in the contiguous form.

As shown in the image below, the remaining 1 MB space of each partition cannot be used as a unit to store a 4 MB process. Despite of the fact that the sufficient space is available to load the process, process will not be loaded.

3. Limitation on the size of the process

If the process size is larger than the size of maximum sized partition then that process cannot be loaded into the memory. Therefore, a limitation can be imposed on the process size that is it cannot be larger than the size of the largest partition.

4. Degree of multiprogramming is less

By Degree of multi programming, we simply mean the maximum number of processes that can be loaded into the memory at the same time. In fixed partitioning, the degree of multiprogramming is fixed and very less due to the fact that the size of the partition cannot be varied according to the size of processes.

BRIEF EXPLANATION ABOUT FRAGMENTATION

Fragmentation:

Fragmentation is a phenomenon in which storage space is used inefficiently, reducing capacity or performance and often both.

The exact consequences of fragmentation depend on the specific system of storage allocation in use and the particular form of fragmentation.

In many cases, fragmentation leads to storage space being "wasted", and in that case the term also refers to the wasted space itself. For other systems (e.g. the FAT file system) the space used to store given data (e.g. files) is the same regardless of the degree of fragmentation (from none to extreme).

Whenever a process is loaded or removed from the physical memory block, it creates a small hole in memory space which is called fragment. Due to fragmentation, the system fails in allocating the contiguous memory space to a process even though it have the requested amount of memory but, in a non-contiguous manner.

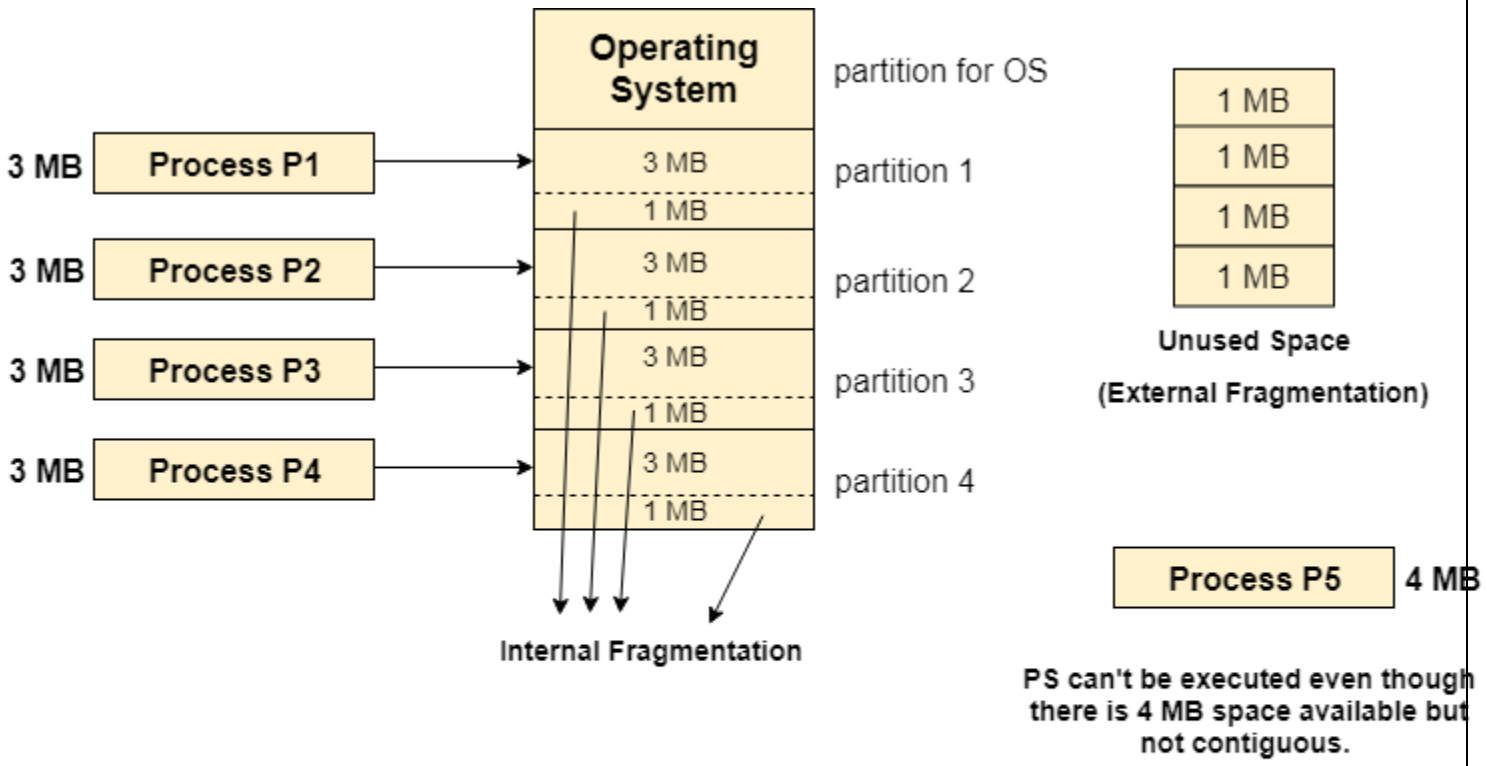
The fragmentation is further classified into two categories Internal and External Fragmentation.

Internal fragmentation: Internal fragmentation is the wasted space within each allocated block because of rounding up from the actual requested allocation to the allocation granularity.

External fragmentation: External fragmentation is the various free spaced holes that are generated in either your memory or disk space.

Both the internal and external classification affects data accessing speed of the system. They have a basic difference between them i.e. Internal fragmentation occurs when fixed sized memory blocks are allocated to the process without concerning about the size of the process, and External fragmentation occurs when the processes are allocated memory dynamically.

Basis for comparision	**Internal Fragmentation**	**External Fragmentation**
Basic	It occurs when fixed sized memory blocks are allocated to the processes.	It occurs when variable size memory space are allocated to the processes dynamically.
Occurence	When the memory assigned to the process is slightly larger than the memory requested by the process this creates free space in the allocated block causing internal fragmentation.	When the process is removed from the memory, it creates the free space in the memory causing external fragmentation.
Solution	The memory must be partitioned into variable sized blocks and assign the best fit block to the process.	Compaction, paging and segmentation.



Fixed Partitioning

(Contiguous memory allocation)

Fixed and variable partition

Dynamic Partitioning OR Fixed and variable partition

Dynamic partitioning tries to overcome the problems caused by fixed partitioning. In this technique, the partition size is not declared initially. It is declared at the time of process loading.

The first partition is reserved for the operating system. The remaining space is divided into parts. The size of each partition will be equal to the size of the process. The partition size varies according to the need of the process so that the internal fragmentation can be avoided.

Advantages of Dynamic Partitioning over fixed partitioning

1. No Internal Fragmentation:

Given the fact that the partitions in dynamic partitioning are created according to the need of the process, It is clear that there will not be any internal fragmentation because there will not be any unused remaining space in the partition.

2. No Limitation on the size of the process

In Fixed partitioning, the process with the size greater than the size of the largest partition could not be executed due to the lack of sufficient contiguous memory. Here, In Dynamic partitioning, the process size can't be restricted since the partition size is decided according to the process size.

3. Degree of multiprogramming is dynamic

Due to the absence of internal fragmentation, there will not be any unused space in the partition hence more processes can be loaded in the memory at the same time.

Disadvantages of dynamic partitioning

External Fragmentation

Absence of internal fragmentation doesn't mean that there will not be external fragmentation.

Let's consider three processes P1 (1 MB) and P2 (3 MB) and P3 (1 MB) are being loaded in the respective partitions of the main memory.

After some time P1 and P3 got completed and their assigned space is freed. Now there are two unused partitions (1 MB and 1 MB) available in the main memory but they cannot be used to load a 2 MB process in the memory since they are not contiguously located.

The rule says that the process must be contiguously present in the main memory to get executed. We need to change this rule to avoid external fragmentation

Complex Memory Allocation:

In Fixed partitioning, the list of partitions is made once and will never change but in dynamic partitioning, the allocation and deallocation is very complex since the partition size will be varied every time when it is assigned to a new process. OS has to keep track of all the partitions.

Due to the fact that the allocation and deallocation are done very frequently in dynamic memory allocation and the partition size will be changed at each time, it is going to be very difficult for OS to manage everything.

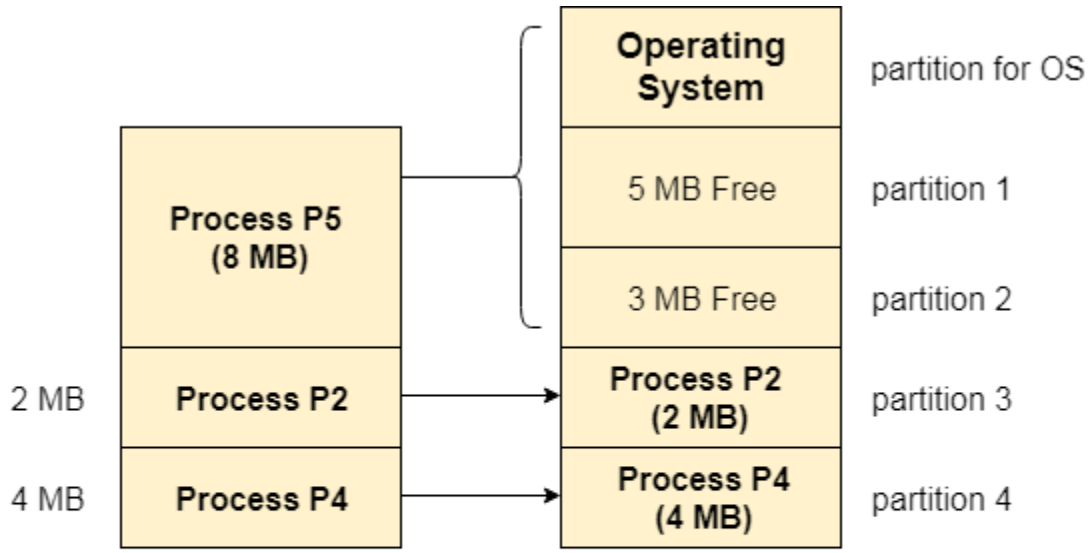
Compaction

We got to know that the dynamic partitioning suffers from external fragmentation. However, this can cause some serious problems.

To avoid compaction, we need to change the rule which says that the process can't be stored in the different places in the memory.

We can also use compaction to minimize the probability of external fragmentation. In compaction, all the free partitions are made contiguous and all the loaded partitions are brought together.

By applying this technique, we can store the bigger processes in the memory. The free partitions are merged which can now be allocated according to the needs of new processes. This technique is also called defragmentation.



Compaction

As shown in the image above, the process P5, which could not be loaded into the memory due to the lack of contiguous space, can be loaded now in the memory since the free partitions are made contiguous.

Problem with Compaction

The efficiency of the system is decreased in the case of compaction due to the fact that all the free spaces will be transferred from several places to a single place.

Huge amount of time is invested for this procedure and the CPU will remain idle for all this time. Despite of the fact that the compaction avoids external fragmentation, it makes system inefficient.

Let us consider that OS needs 6 NS to copy 1 byte from one place to another.

1 B transfer needs 6 NS
256 MB transfer needs 256 X 2^20 X 6 X 10 ^ -9 secs

hence, it is proved to some extent that the larger size memory transfer needs some huge amount of time that is in seconds.

Paging :

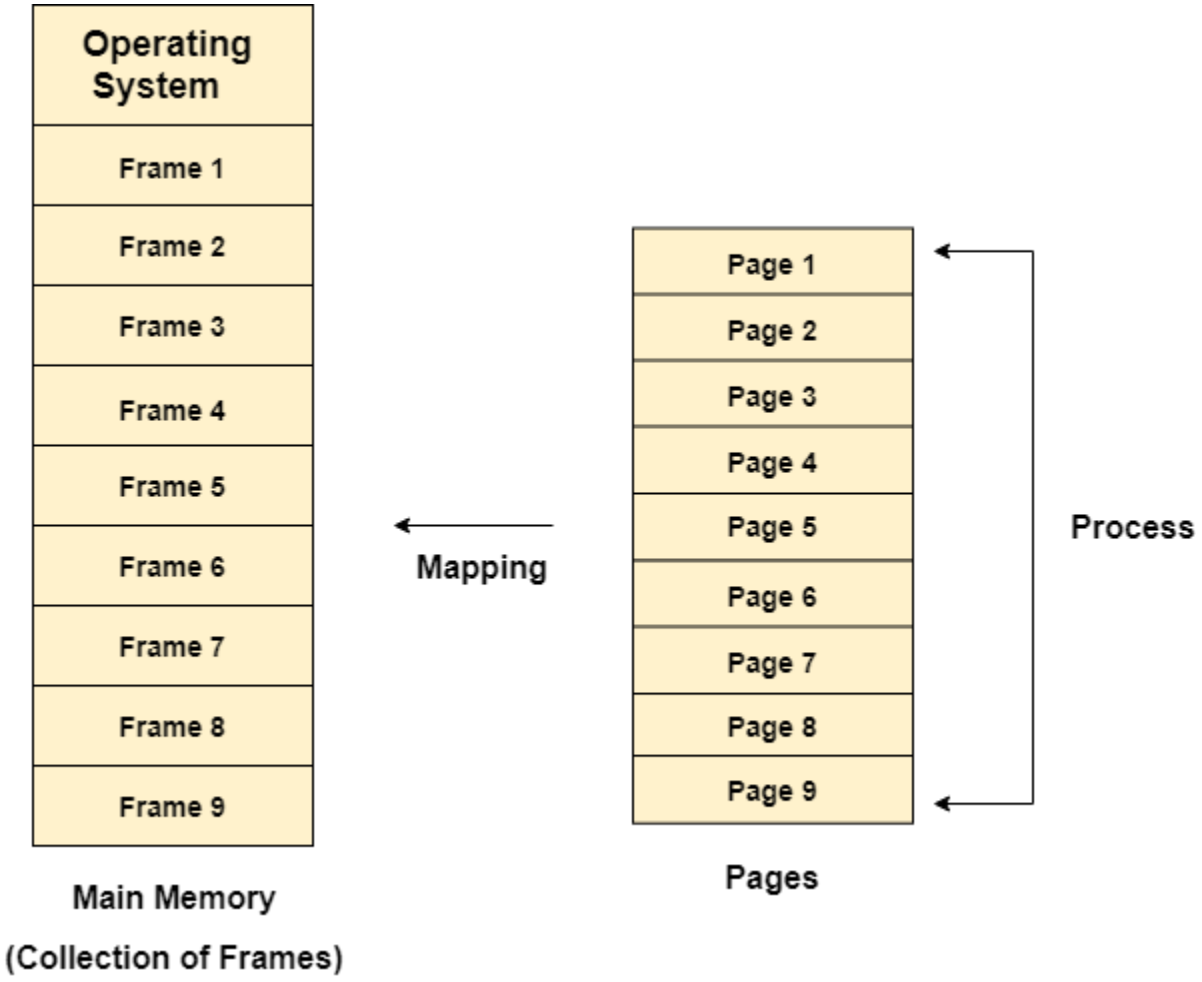
In Operating Systems, Paging is a storage mechanism used to retrieve processes from the secondary storage into the main memory in the form of pages.

The main idea behind the paging is to divide each process in the form of pages. The main memory will also be divided in the form of frames.

One page of the process is to be stored in one of the frames of the memory. The pages can be stored at the different locations of the memory but the priority is always to find the contiguous frames or holes.

Pages of the process are brought into the main memory only when they are required otherwise they reside in the secondary storage.

Different operating system defines different frame sizes. The sizes of each frame must be equal. Considering the fact that the pages are mapped to the frames in Paging, page size needs to be as same as frame size.



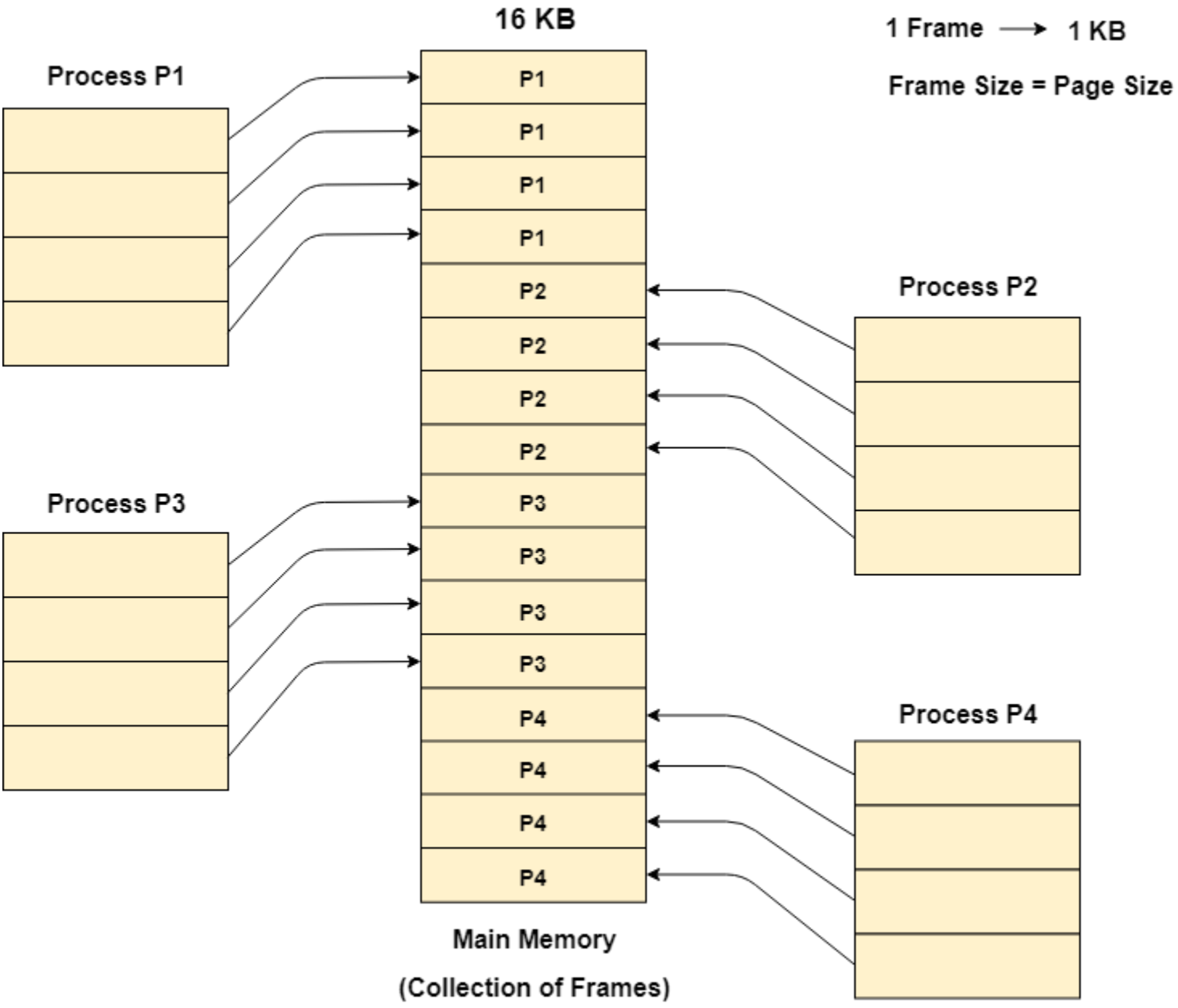
Example:

Let us consider the main memory size 16 Kb and Frame size is 1 KB therefore the main memory will be divided into the collection of 16 frames of 1 KB each.

There are 4 processes in the system that is P1, P2, P3 and P4 of 4 KB each. Each process is divided into pages of 1 KB each so that one page can be stored in one frame.

Initially, all the frames are empty therefore pages of the processes will get stored in the contiguous way.

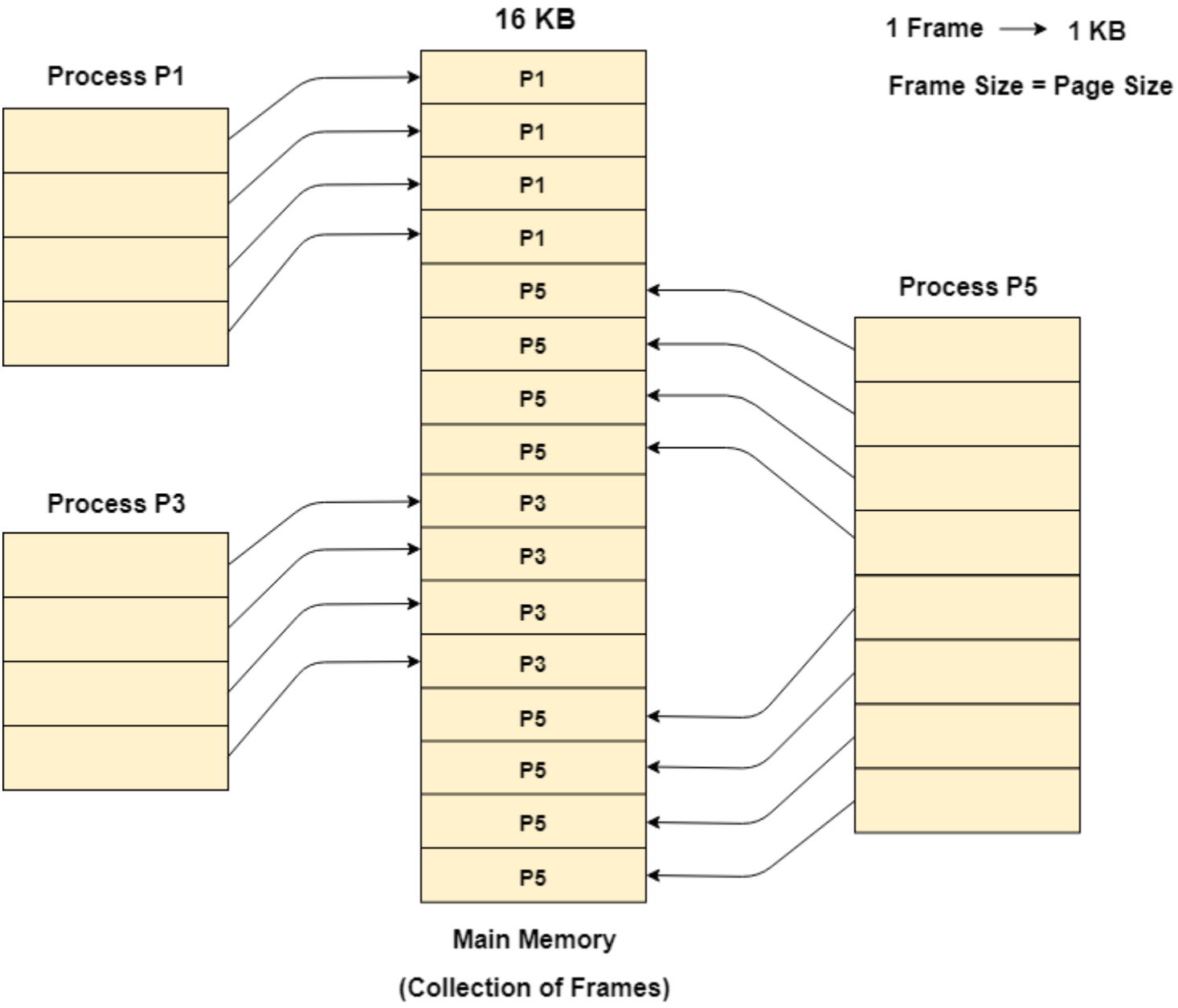
Frames, pages and the mapping between the two is shown in the image below.



Paging

Let us consider that, P2 and P4 are moved to waiting state after some time. Now, 8 frames become empty and therefore other pages can be loaded in that empty place. The process P5 of size 8 KB (8 pages) is waiting inside the ready queue.

Given the fact that, we have 8 non contiguous frames available in the memory and paging provides the flexibility of storing the process at the different places. Therefore, we can load the pages of process P5 in the place of P2 and P4.


16 KB
1 Frame → 1 KB
Frame Size = Page Size
Process P1
Process P5
Process P3
P1
P1
P1
P1
P5
P5
P5
P5
P3
P3
P3
P3
P5
P5
P5
P5
Main Memory
(Collection of Frames)

Paging

What is Paging Protection?

The paging process should be protected by using the concept of insertion of an additional bit called Valid/Invalid bit. Paging Memory protection in paging is achieved by associating protection bits with each page. These bits are associated with each page table entry and specify protection on the corresponding page.

Advantages of Paging

Here, are advantages of using Paging method:

- Easy to use memory management algorithm
- No need for external Fragmentation
- Swapping is easy between equal-sized pages and page frames.

Disadvantages of Paging

Here, are drawback/ cons of Paging:

- May cause Internal fragmentation
- Complex memory management algorithm
- Page tables consume additonal memory.
- Multi-level paging may lead to memory reference overhead.

Virtual Memory

Virtual Memory is a storage scheme that provides user an illusion of having a very big main memory. This is done by treating a part of secondary memory as the main memory.

In this scheme, User can load the bigger size processes than the available main memory by having the illusion that the memory is available to load the process.

Instead of loading one big process in the main memory, the Operating System loads the different parts of more than one process in the main memory.

By doing this, the degree of multiprogramming will be increased and therefore, the CPU utilization will also be increased.

How Virtual Memory Works?

In modern word, virtual memory has become quite common these days. In this scheme, whenever some pages needs to be loaded in the main memory for the execution and the memory is not available for those many pages, then in that case, instead of stopping the pages from entering in the main memory, the OS search for the RAM area that are least used in the recent times or that are not referenced and copy that into the secondary memory to make the space for the new pages in the main memory.

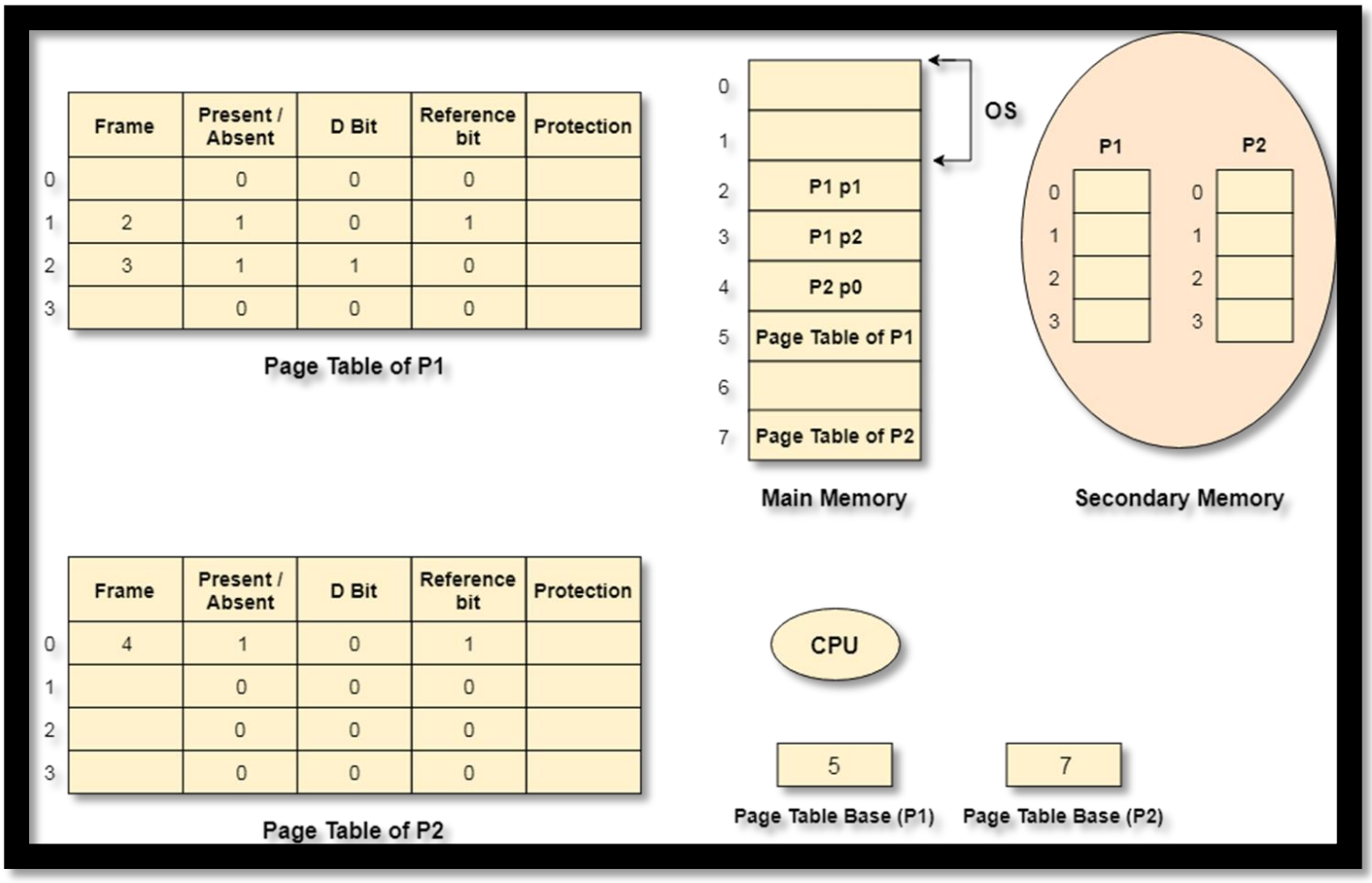
Since all this procedure happens automatically, therefore it makes the computer feel like it is having the unlimited

Virtual memory management system:

Let us assume 2 processes, P1 and P2, contains 4 pages each. Each page size is 1 KB. The main memory contains 8 frame of 1 KB each. The OS resides in the first two partitions. In the third partition, 1st page of P1 is stored and the other frames are also shown as filled with the different pages of processes in the main memory.

The page tables of both the pages are 1 KB size each and therefore they can be fit in one frame each. The page tables of both the processes contain various information that is also shown in the image.

The CPU contains a register which contains the base address of page table that is 5 in the case of P1 and 7 in the case of P2. This page table base address will be added to the page number of the Logical address when it comes to accessing the actual corresponding entry.



Advantages of Virtual Memory:

1. The degree of Multiprogramming will be increased.
2. User can run large application with less real RAM.
3. There is no need to buy more memory RAMs.

Disadvantages of Virtual Memory:

1. The system becomes slower since swapping takes time.
2. It takes more time in switching between applications.
3. The user will have the lesser hard disk space for its use.

Hardware and Control Structures

• Memory references are dynamically translated into physical addresses at run time – A process may be swapped in and out of main memory such that it occupies different regions

• A process may be broken up into pieces that do not need to be located contiguously in main memory

• All pieces of a process do not need to be loaded in main memory during execution 3 Execution of a Program

Execution of a program

• Operating system brings into main memory a few pieces of the program

• Resident set - portion of process that is in main memory

• An interrupt is generated when an address is needed that is not in main memory

• Operating system places the process in a blocking state 4 Execution of a Program

Execution of a Program

-Piece of process that contains the logical address is brought into main memory

– Operating system issues a disk I/O Read request

– Another process is dispatched to run while the disk I/O takes place

– An interrupt is issued when disk I/O complete which causes the operating system to place the affected process in the Ready state.

LOCALITY OF REFERENCE:

Locality of reference, also known as the **principle of locality**, is the tendency of a processor to access the same set of memory locations repetitively over a short period of time.

There are two basic types of reference locality –

Temporal and spatial locality.

1.**Temporal locality** refers to the reuse of specific data, and/or resources, within a relatively small time duration.

2.Spatial locality (also termed *data locality*) refers to the use of data elements within relatively close storage locations. Sequential locality, a special case of spatial locality, occurs when data elements are arranged and accessed linearly, such as, traversing the elements in a one-dimensional array.

Locality is a type of predictable behavior that occurs in computer systems. Systems that exhibit strong *locality of reference* are great candidates for performance optimization

through the use of techniques such as the caching, prefetching for memory and advanced branch predictors at the pipelining stage of a processor core.

PAGE FAULT:

A page fault occurs when a program attempts to access a block of memory that is not stored in the physical memory, or RAM. The fault notifies the operating system that it must locate the data in virtual memory, then transfer it from the storage device, such as an HDD or SSD, to the system RAM.

What is a Page Fault?

If the referred page is not present in the main memory then there will be a miss and the concept is called Page miss or page fault.

The CPU has to access the missed page from the secondary memory. If the number of page fault is very high then the effective access time of the system will become very high.

WORKING SET

- **Working set** is a concept in computer science which defines the amount of memory that a process requires in a given time interval.

DEFINITION:

Peter Denning (1968) defines "the working set of information of a process at time to be the collection of information referenced by the process during the process time interval

Typically the units of information in question are considered to be memory pages. This is suggested to be an approximation of the set of pages that the process will access in the future (say during the next time units), and more specifically is suggested to be an indication of what pages ought to be kept in main memory to allow most progress to be made in the execution of that process.

DIRTY PAGE /DIRTY BIT:

When a bit is modified by the CPU and not written back to the storage, it is called as a dirty bit. This bit is present in the memory cac`he or the virtual storage space.

What is dirty bit?

A dirty bit is a flag that indicates whether an attribute needs to be updated. Such situations usually occur when a bit in a memory cache or virtual memory page that has been changed by a processor but has not been updated in the storage.

Demand Paging

According to the concept of Virtual Memory, in order to execute some process, only a part of the process needs to be present in the main memory which means that only a few pages will only be present in the main memory at any time.

However, deciding, which pages need to be kept in the main memory and which need to be kept in the secondary memory, is going to be difficult because we cannot say in advance that a process will require a particular page at particular time.

Therefore, to overcome this problem, there is a concept called Demand Paging is introduced. It suggests keeping all pages of the frames in the secondary memory until they are required. In other words, it says that do not load any page in the main memory until it is required.

Whenever any page is referred for the first time in the main memory, then that page will be found in the secondary memory.

After that, it may or may not be present in the main memory depending upon the page replacement algorithm which will be covered later in this tutorial.

Advantages:

Following are the advantages of Demand Paging –

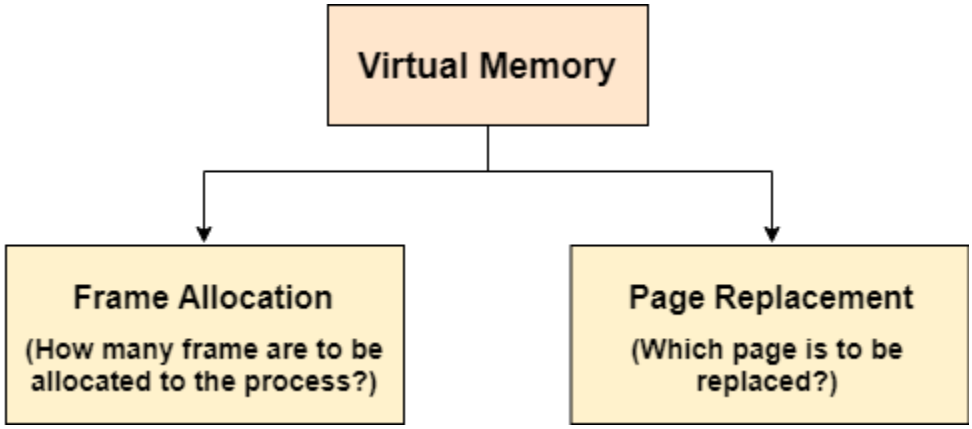
- Large virtual memory.
- More efficient use of memory.
- There is no limit on degree of multiprogramming.

Disadvantages:

- Number of tables and the amount of processor overhead for handling page interrupts are greater than in the case of the simple paged management techniques.

Page Replacement Algorithms

The page replacement algorithm decides which memory page is to be replaced. The process of replacement is sometimes called swap out or write to disk. Page replacement is done when the requested page is not found in the main memory (page fault).



There are two main aspects of virtual memory, Frame allocation and Page Replacement. It is very important to have the optimal frame allocation and page replacement algorithm. Frame allocation is all about how many frames are to be allocated to the process while the page replacement is all about determining the page number which needs to be replaced in order to make space for the requested page.

What If the algorithm is not optimal?

1. If the number of frames which are allocated to a process is not sufficient or accurate then there can be a problem of thrashing. Due to the lack of frames, most of the pages will be residing in the main memory and therefore more page faults will occur.

However, if OS allocates more frames to the process then there can be internal fragmentation.

2. If the page replacement algorithm is not optimal then there will also be the problem of thrashing. If the number of pages that are replaced by the requested pages will be referred in the near future then there will be more number of swap-in and swap-out and therefore the OS has to perform more replacements then usual which causes performance deficiency.

Therefore, the task of an optimal page replacement algorithm is to choose the page which can limit the thrashing.

Types of Page Replacement Algorithms

There are various page replacement algorithms. Each algorithm has a different method by which the pages can be replaced.

1. **Optimal Page Replacement algorithm** → this algorithms replaces the page which will not be referred for so long in future. Although it can not be practically implementable but

it can be used as a benchmark. Other algorithms are compared to this in terms of optimality.

2. **Least recent used (LRU) page replacement algorithm** → this algorithm replaces the page which has not been referred for a long time. This algorithm is just opposite to the optimal page replacement algorithm. In this, we look at the past instead of staring at future.
3. **FIFO** → in this algorithm, a queue is maintained. The page which is assigned the frame first will be replaced first. In other words, the page which resides at the rare end of the queue will be replaced on the every page fault.

IN OUR SYLLABUS WE ARE HAVING (BIHER R-2018)

1.OPTIMAL PAGE REPLACEMENT ALGORITHM

2.FIRST IN FIRST OUT(FIFO)

3.REPLACEMENT SECOND CHANCE(SC)

4.NOT RECENTLY USED(NRU)

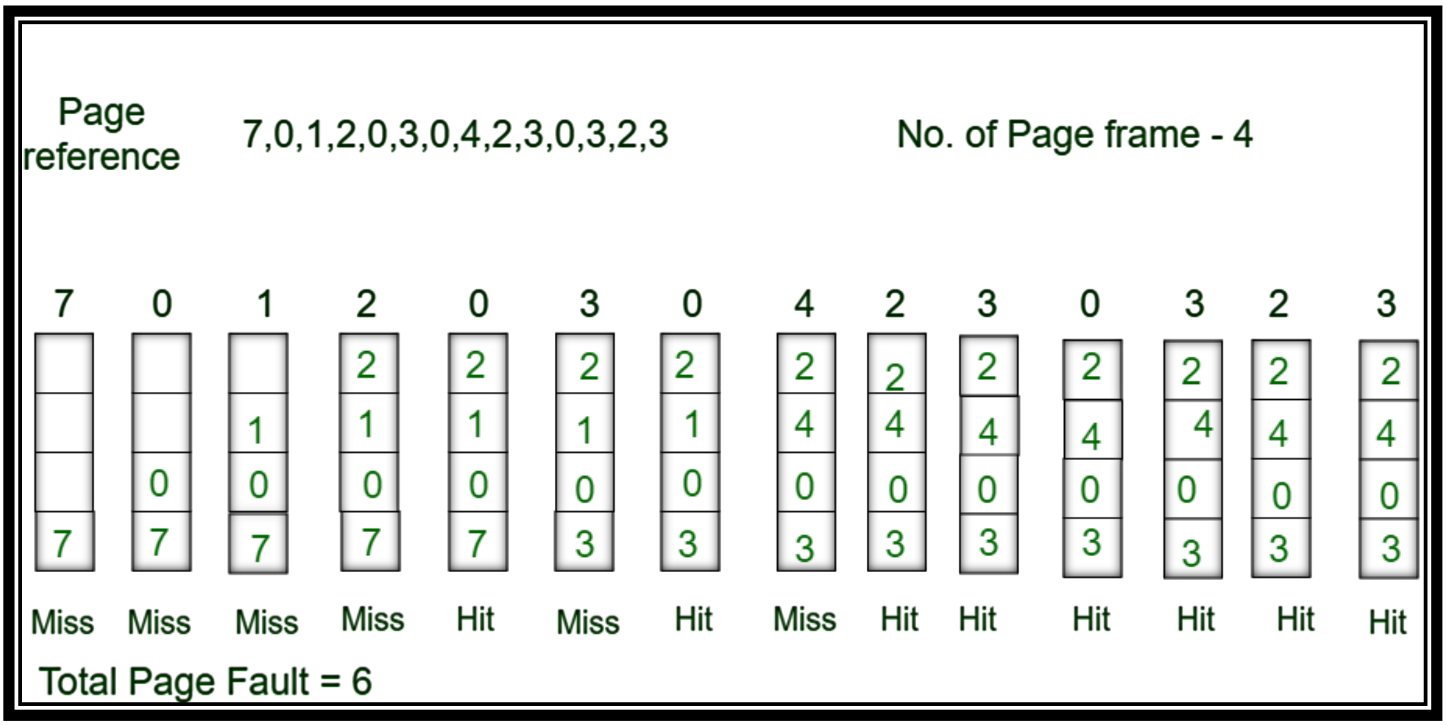
5.LEAST RECENTLY USED(LRU)

Page Replacement Algorithms :

1.OPTIMAL PAGE REPLACEMENT

In this algorithm, pages are replaced which would not be used for the longest duration of time in the future.

Example: Consider the page references 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, with 4 page frame. Find number of page fault.

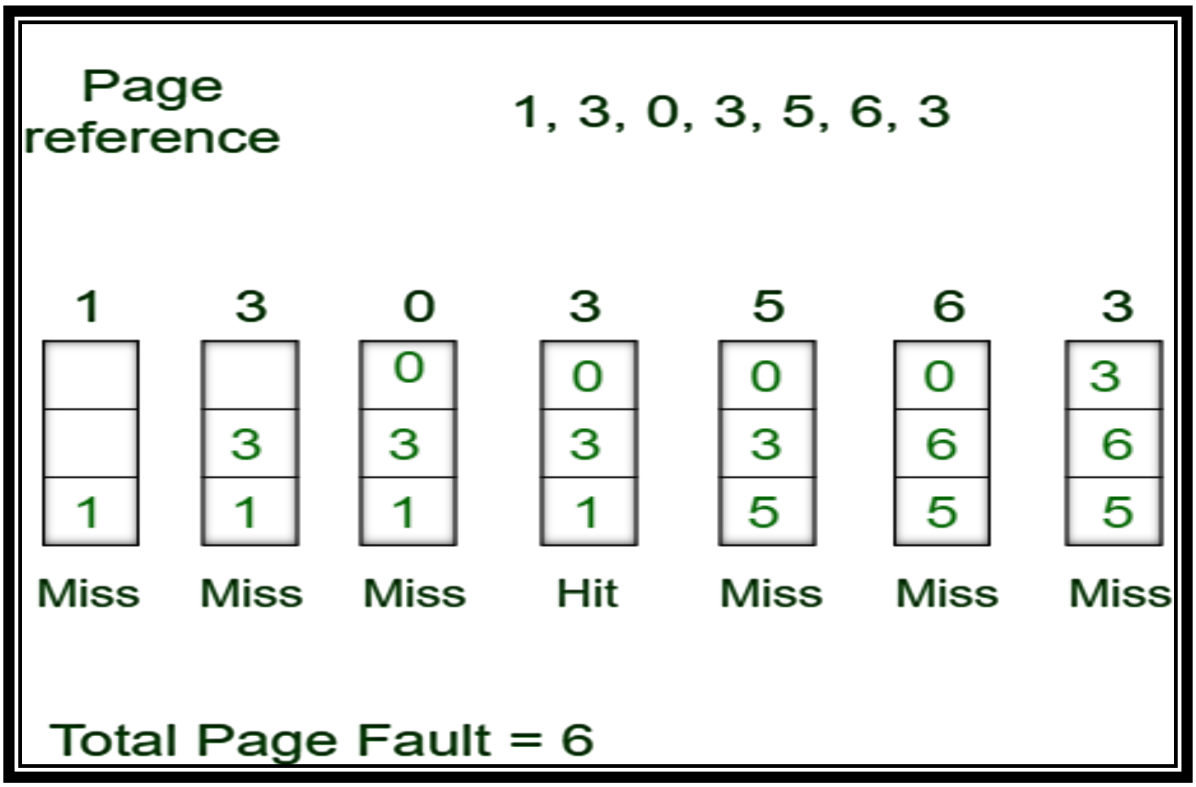


Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> **4 Page faults**
0 is already there so —> **0 Page fault.**
when 3 came it will take the place of 7 because it is not used for the longest duration of time in the future.—>**1 Page fault.**
0 is already there so —> **0 Page fault.**.
4 will takes place of 1 —> **1 Page Fault.**

Now for the further page reference string —> **0 Page fault** because they are already available in the memory.
Optimal page replacement is perfect, but not possible in practice as the operating system cannot know future requests. The use of Optimal Page replacement is to set up a benchmark so that other replacement algorithms can be analyzed against it.

2.First In First Out (FIFO) –
This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

Example: Consider page reference string 1, 3, 0, 3, 5, 6 with 3 page frames.Find number of page faults.



Initially all slots are empty, so when 1, 3, 0 came they are allocated to the empty slots —> **3 Page Faults.**
when 3 comes, it is already in memory so —> **0 Page Faults.**
Then 5 comes, it is not available in memory so it replaces the oldest page slot i.e 1. —>**1 Page Fault.**
6 comes, it is also not available in memory so it replaces the oldest page slot i.e 3 —>**1 Page Fault.**
Finally when 3 come it is not avilable so it replaces 0 **1 page fault**

3.REPLACEMENT SECOND CHANCE(SC):

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.

SOLUTON:

- 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.

4.NOT RECENTLY USED (NRU):

The not recently used (NRU) page replacement algorithm is an algorithm that favours keeping pages in memory that have been recently used. This algorithm works on the following principle: when a page is referenced, a referenced bit is set for that page, marking it as referenced. Similarly, when a page is modified (written to), a modified bit is set. The setting of the bits is usually done by the hardware, although it is possible to do so on the software level as well.

At a certain fixed time interval, a timer interrupt triggers and clears the referenced bit of all the pages, so only pages referenced within the current timer interval are marked with a referenced bit. When a page needs to be replaced, the operating system divides the pages into four classes:

3. referenced, modified
2. referenced, not modified
1. not referenced, modified
0. not referenced, not modified

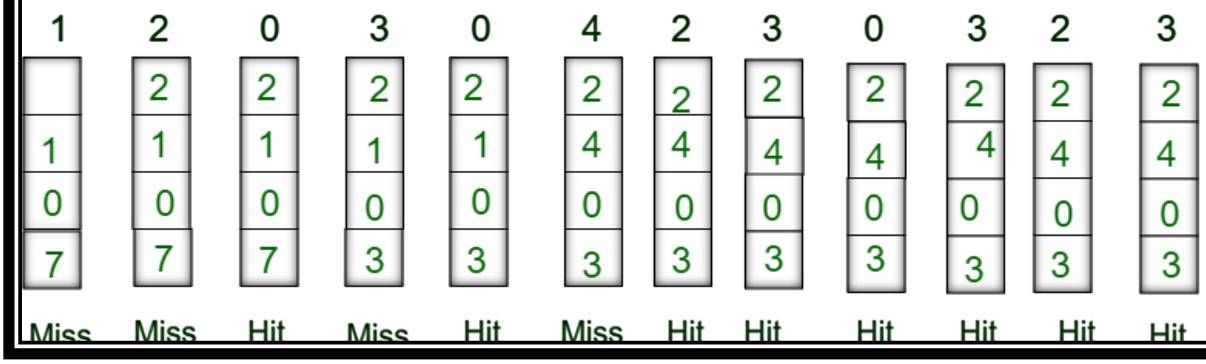
Although it does not seem possible for a page to be modified yet not referenced, this happens when a class 3 page has its referenced bit cleared by the timer interrupt. The NRU algorithm picks a random page from the lowest category for removal. So out of the above four page categories, the NRU algorithm will replace a not-referenced, not-modified page if such a page exists. Note that this algorithm implies that a modified but not-referenced (within the last timer interval) page is less important than a not-modified page that is intensely referenced.

5.LEAST RECENTLY USED (LRU):

In this algorithm page will be replaced which is least recently used.
Example: Consider the page reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2 with 4 page frames.Find number of page faults.

Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> **4 Page faults**
0 is already their so —> **0 Page fault.**
when 3 came it will take the place of 7 because it is least recently used —>**1 Page fault**
0 is already in memory so —> **0 Page fault**.
4 will takes place of 1 —> **1 Page Fault**
Now for the further page reference string —> **0 Page fault** because they are already available in the memory.

7,0,1,2,0,3,0,4,2,3,0,3,2,3 No. of Page frame - 4



----------ALL THE BEST WISHES----------