

Operating System By Javatpoint

#1.Operating System Tutorial



Operating System Tutorial provides the basic and advanced concepts of operating system . Our Operating system tutorial is designed for beginners, professionals and GATE aspirants. We have designed this tutorial after the completion of a deep research about every concept.

The content is described in detailed manner and has the ability to answer most of your queries. The tutorial also contains the numerical examples based on previous year GATE questions which will help you to address the problems in a practical manner.

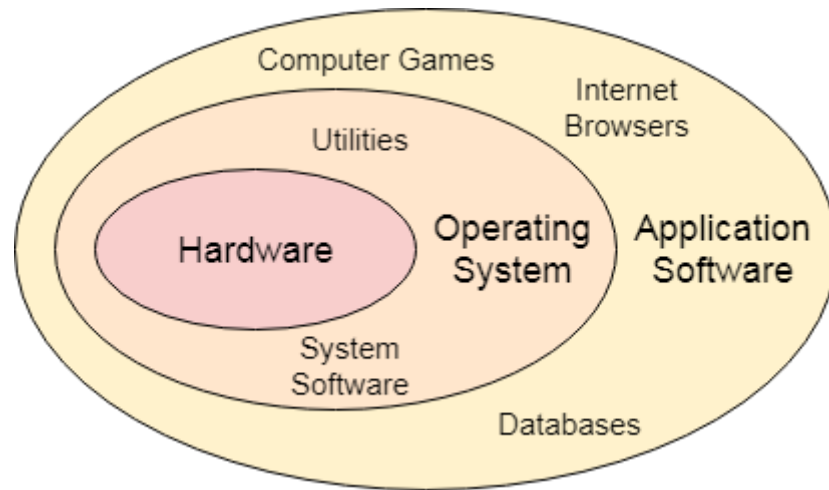
Operating System can be defined as an interface between user and the hardware. It provides an environment to the user so that, the user can perform its task in convenient and efficient way.

The Operating System Tutorial is divided into various parts based on its functions such as Process Management, Process Synchronization, Deadlocks and File Management.

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.



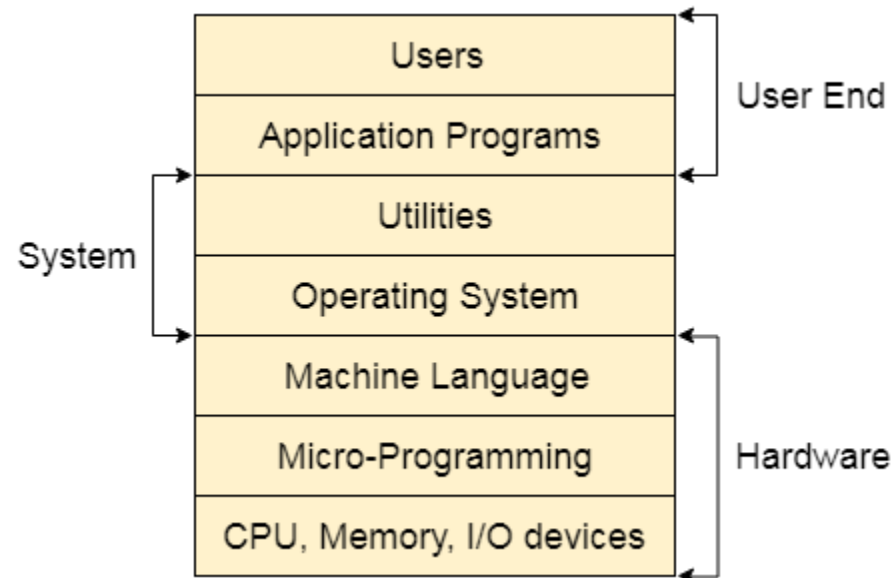
An **Operating System** can be defined as an **interface between user and hardware**. It is responsible for the execution of all the processes, Resource Allocation, [CPU](#) management, File Management and many other tasks.

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

Types of OS

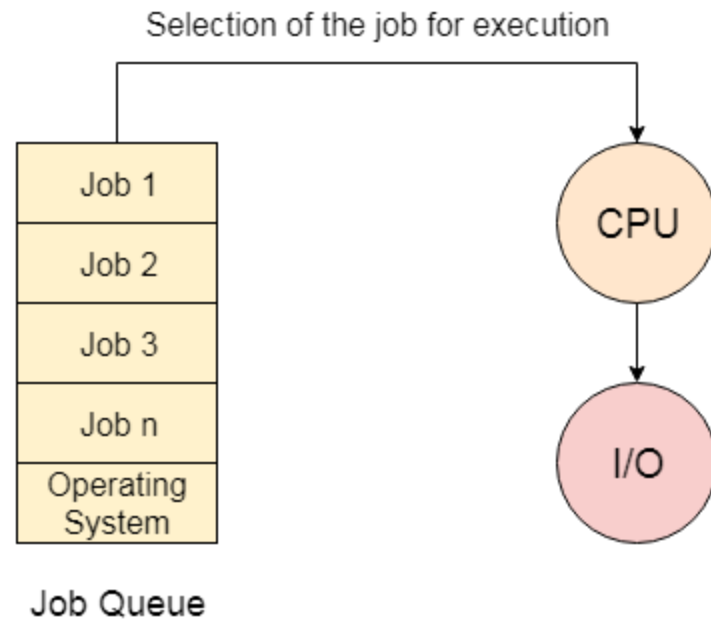
There are many types of operating system exists in the current scenario:

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.

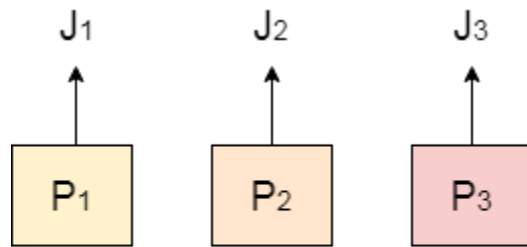
Multiprogramming Operating System

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

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.

#2.Process Management

Process Management Introduction

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.

5. Providing mechanisms for process communication.

Attributes of a process

The Attributes of the process are used by the Operating System to create the process control block (PCB) for each of them. This is also called context of the process. Attributes which are stored in the PCB are described below.

1. Process ID

When a process is created, a unique id is assigned to the process which is used for unique identification of the process in the system.

2. Program counter

A program counter stores the address of the last instruction of the process on which the process was suspended. The CPU uses this address when the execution of this process is resumed.

3. Process State

The Process, from its creation to the completion, goes through various states which are new, ready, running and waiting. We will discuss about them later in detail.

4. Priority

Every process has its own priority. The process with the highest priority among the processes gets the CPU first. This is also stored on the process control block.

5. General Purpose Registers

Every process has its own set of registers which are used to hold the data which is generated during the execution of the process.

6. List of open files

During the Execution, Every process uses some files which need to be present in the main memory. OS also maintains a list of open files in the PCB.

7. List of open devices

OS also maintain the list of all open devices which are used during the execution of the process.

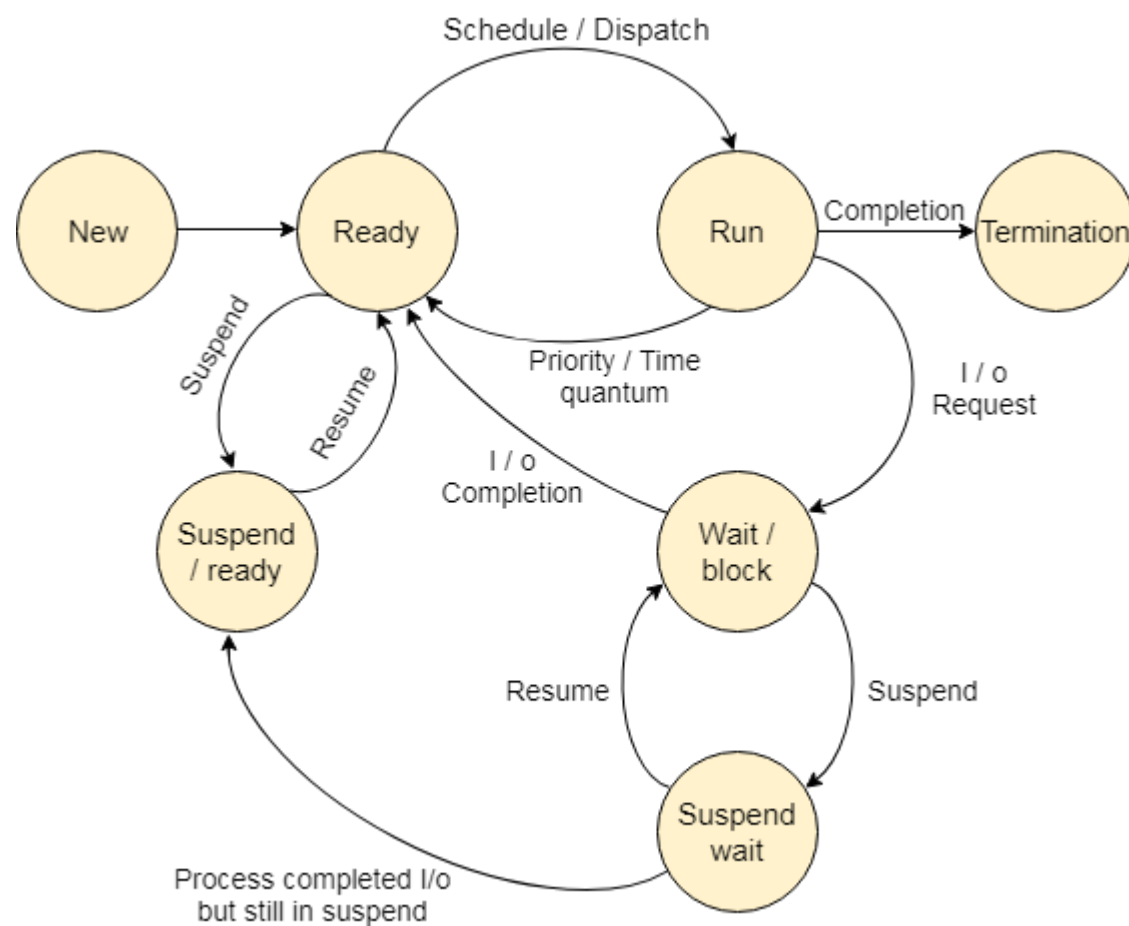
Process ID
Program Counter
Process State
Priority
General Purpose Registers
List of Open Files
List of Open Devices

Process Attributes

[Next →](#) [← Prev](#)

Process States

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.

Operations on the Process

1. Creation

Once the process is created, it will be ready and come into the ready queue (main memory) and will be ready for the execution.

2. Scheduling

Out of the many processes present in the ready queue, the Operating system chooses one process and start executing it. Selecting the process which is to be executed next, is known as scheduling.

3. Execution

Once the process is scheduled for the execution, the processor starts executing it. Process may come to the blocked or wait state during the execution then in that case the processor starts executing the other processes.

4. Deletion/killing

Once the purpose of the process gets over then the OS will kill the process. The Context of the process (PCB) will be deleted and the process gets terminated by the Operating system.

Process Schedulers

Operating system uses various schedulers for the process scheduling described below.

1. Long term scheduler

Long term scheduler is also known as job scheduler. It chooses the processes from the pool (secondary memory) and keeps them in the ready queue maintained in the primary memory.

Long Term scheduler mainly controls the degree of Multiprogramming. The purpose of long term scheduler is to choose a perfect mix of IO bound and CPU bound processes among the jobs present in the pool.

If the job scheduler chooses more IO bound processes then all of the jobs may reside in the blocked state all the time and the CPU will remain idle most of the time. This will reduce the degree of Multiprogramming. Therefore, the Job of long term scheduler is very critical and may affect the system for a very long time.

2. Short term scheduler

Short term scheduler is also known as CPU scheduler. It selects one of the Jobs from the ready queue and dispatch to the CPU for the execution.

A scheduling algorithm is used to select which job is going to be dispatched for the execution. The Job of the short term scheduler can be very critical in the sense that if it selects job whose CPU burst time is very high then all the jobs after that, will have to wait in the ready queue for a very long time.

This problem is called starvation which may arise if the short term scheduler makes some mistakes while selecting the job.

3. Medium term scheduler

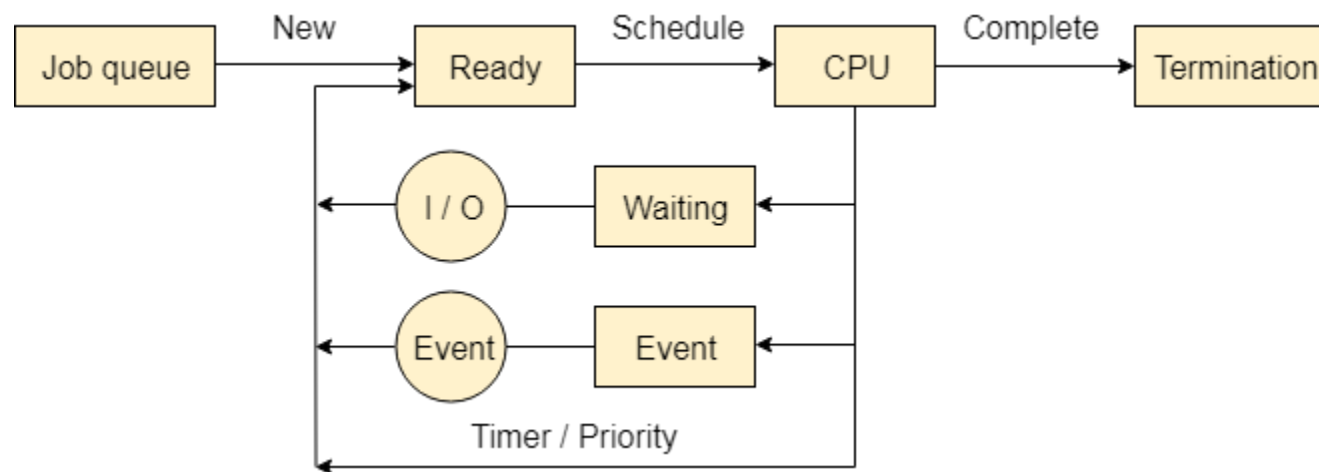
Medium term scheduler takes care of the swapped out processes. If the running state processes needs some IO time for the completion then there is a need to change its state from running to waiting.

Medium term scheduler is used for this purpose. It removes the process from the running state to make room for the other processes. Such processes are the swapped out processes and this procedure is called swapping. The medium term scheduler is responsible for suspending and resuming the processes.

It reduces the degree of multiprogramming. The swapping is necessary to have a perfect mix of processes in the ready queue.

Process Queues

The Operating system manages various types of queues for each of the process states. The PCB related to the process is also stored in the queue of the same state. If the Process is moved from one state to another state then its PCB is also unlinked from the corresponding queue and added to the other state queue in which the transition is made.



There are the following queues maintained by the Operating system.

1. Job Queue

In starting, all the processes get stored in the job queue. It is maintained in the secondary memory. The long term scheduler (Job scheduler) picks some of the jobs and put them in the primary memory.

2. Ready Queue

Ready queue is maintained in primary memory. The short term scheduler picks the job from the ready queue and dispatch to the CPU for the execution.

3. Waiting Queue

When the process needs some IO operation in order to complete its execution, OS changes the state of the process from running to waiting. The context (PCB) associated with the process gets stored on the waiting queue which will be used by the Processor when the process finishes the IO.

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.

[Next →](#) [← Prev](#)

CPU Scheduling

In the uniprogramming systems like MS DOS, when a process waits for any I/O operation to be done, the CPU remains idle. This is an overhead since it wastes the time and causes the problem of starvation. However, In Multiprogramming systems, the CPU doesn't remain idle during the waiting time of the Process and it starts executing other processes. Operating System has to define which process the CPU will be given.

In Multiprogramming systems, the Operating system schedules the processes on the CPU to have the maximum utilization of it and this procedure is called **CPU scheduling**. The Operating System uses various scheduling algorithm to schedule the processes.

This is a task of the short term scheduler to schedule the CPU for the number of processes present in the Job Pool. Whenever the running process requests some IO operation then the short term scheduler saves the current context of the process (also called PCB) and changes its state from running to waiting. During the time, process is in waiting state; the Short term scheduler picks another process from the ready queue and assigns the CPU to this process. This procedure is called **context switching**.

What is saved in the Process Control Block?

The Operating system maintains a process control block during the lifetime of the process. The Process control block is deleted when the process is terminated or killed. There is the following information which is saved in the process control block and is changing with the state of the process.

Process ID
Process State
Pointer
Priority
Program Counter
CPU Registers
I/O Information
Accounting Information
etc.

Why do we need Scheduling?

In Multiprogramming, if the long term scheduler picks more I/O bound processes then most of the time, the CPU remains idle. The task of Operating system is to optimize the utilization of resources.

If most of the running processes change their state from running to waiting then there may always be a possibility of deadlock in the system. Hence to reduce this overhead, the OS needs to schedule the jobs to get the optimal utilization of CPU and to avoid the possibility to deadlock.

Scheduling Algorithms

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

The Purpose of a Scheduling algorithm

1. Maximum CPU utilization
2. Fair allocation of CPU
3. Maximum throughput
4. Minimum turnaround time
5. Minimum waiting time
6. Minimum response time

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 get 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

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.

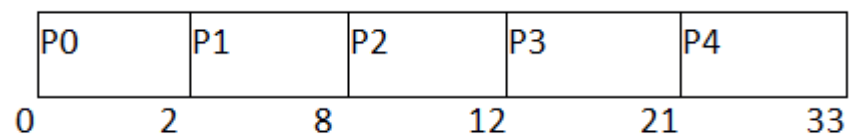
The Turnaround time and the waiting time are calculated by using the following formula.

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	Waiting Time
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= $31/5$



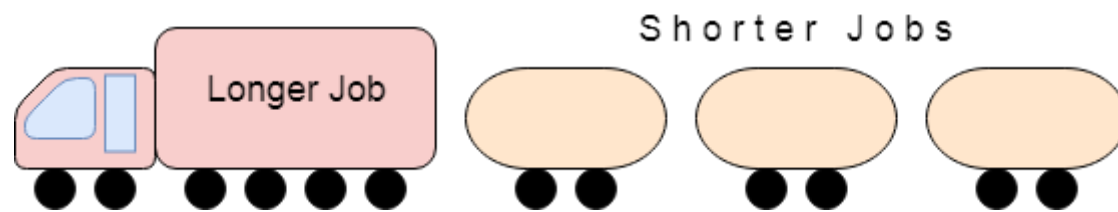
(Gantt chart)

Convoy Effect in FCFS

FCFS may suffer from the **convoy effect** if the burst time of the first job is the highest among all. As in the real life, if a convoy is passing through the road then the other persons may get blocked until it passes completely. This can be simulated in the Operating System also.

If the CPU gets the processes of the higher burst time at the front end of the ready queue then the processes of lower burst time may get blocked which means they may never get the CPU if the job in the execution has a very high burst time. This is called **convoy effect** or **starvation**.

The Convoy Effect, Visualized Starvation



Example

In the Example, We have 3 processes named as **P1, P2 and P3**. The Burt Time of process P1 is highest.

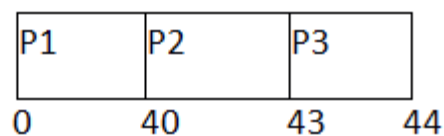
The Turnaround time and the waiting time in the following table, are calculated by the formula,

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

In the First scenario, The Process P1 arrives at the first in the queue although; the burst time of the process is the highest among all. Since, the Scheduling algorithm, we are following is FCFS hence the CPU will execute the Process P1 first.

In this schedule, the average waiting time of the system will be very high. That is because of the convoy effect. The other processes P2, P3 have to wait for their turn for 40 units of time although their burst time is very low. This schedule suffers from starvation.

Process ID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time
1	0	40	40	40	0
2	1	3	43	42	39
3	1	1	44	43	42

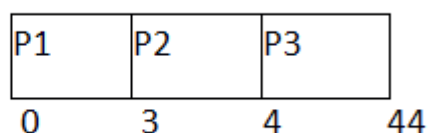


Avg waiting Time = $81/3$

In the Second scenario, If Process P1 would have arrived at the last of the queue and the other processes P2 and P3 at earlier then the problem of starvation would not be there.

Following example shows the deviation in the waiting times of both the scenarios. Although the length of the schedule is same that is 44 units but the waiting time will be lesser in this schedule.

Process ID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time
1	1	40	44	43	3
2	0	3	3	3	0
3	0	1	4	4	3



Avg Waiting Time=6/3

FCFS with Overhead

In the above Examples, we are assuming that all the processes are the CPU bound processes only. We were also neglecting the context switching time.

However if the time taken by the scheduler in context switching is considered then the average waiting time of the system will be increased which also affects the efficiency of the system.

Context Switching is always an overhead. The Following Example describes how the efficiency will be affected if the context switching time is considered in the system.

Example

In the following Example, we are considering five processes P1, P2, P3, P4, P5 and P6. Their arrival time and Burst time are given below.

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

If the context switching time of the system is 1 unit then the Gantt chart of the system will be prepared as follows.

Given $\delta=1$ unit;

δ	P1	δ	P2	δ	P3	δ	P4	Δ	P5	δ	P6	
0	1	4	5	7	8	9	10	14	15	20	21	23

The system will take extra 1 unit of time (overhead) after the execution of every process to schedule the next process.

1. **Inefficiency** = $(6/23) \times 100 \%$
- 2.
3. **Efficiency** = $(1 - 6/23) \times 100 \%$

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 Time
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).

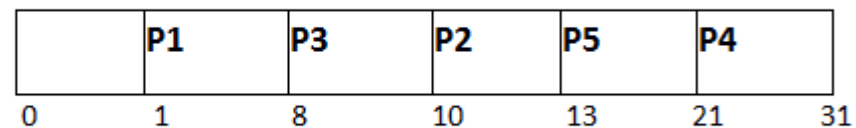
According to the algorithm, the OS schedules the process which is having the lowest burst time among the available processes in the ready queue.

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.



$$\text{Avg Waiting Time} = 27/5$$

Prediction of CPU Burst Time for a process in SJF

The SJF algorithm is one of the best scheduling algorithms since it provides the maximum throughput and minimal waiting time but the problem with the algorithm is, the CPU burst time can't be known in advance.

We can approximate the CPU burst time for a process. There are various techniques which can be used to assume the CPU Burst time for a process. Our Assumption needs to be accurate in order to utilize the algorithm optimally.

There are the following techniques used for the assumption of CPU burst time for a process.

1. Static Techniques

Process Size

We can predict the Burst Time of the process from its size. If we have two processes **T_OLD** and **T_New** and the actual burst time of the old process is known as **20 secs** and the size of the process is **20 KB**. We know that the size of **P_NEW** is **21 KB**. Then the probability of **P_New** having the similar burst time as **20 secs** is maximum.

1. If, $P_{OLD} \rightarrow 20 \text{ KB}$
2. $P_{New} \rightarrow 21 \text{ KB}$
3. $BT(P_{OLD}) \rightarrow 20 \text{ Secs}$
4. Then,
5. $BT(P_{New}) \rightarrow 20 \text{ secs}$

Hence, in this technique, we actually predict the burst time of a new process according to the burst time of an old process of similar size as of new process.

Process Type

We can also predict the burst time of the process according to its type. A Process can be of various types defined as follows.

- **OS Process**

A Process can be an Operating system process like schedulers, compilers, program managers and many more system processes. Their burst time is generally lower for example, 3 to 5 units of time.

- **User Process**

The Processes initiated by the users are called user processes. There can be three types of processes as follows.

- **Interactive Process**

The Interactive processes are the one which interact with the user time to time or Execution of which totally depends upon the User inputs for example various games are such processes. Their burst time needs to be lower since they don't need CPU for a large amount of time, they mainly depend upon the user's interactivity with the process hence they are mainly IO bound processes.

- **Foreground process**

Foreground processes are the processes which are used by the user to perform their needs such as MS office, Editors, utility software etc. These types of processes have a bit higher burst time since they are a perfect mix of CPU and IO bound processes.

- **Background process**

Background processes supports the execution of other processes. They work in hidden mode. For example, key logger is the process which records the keys pressed by the user and activities of the user on the system. They are mainly CPU bound processes and needs CPU for a higher amount of time.

2. Dynamic Techniques

Simple Averaging

In simple averaging, there are given list of n processes $P(1), \dots, P(n)$. Let $T(i)$ denotes the burst time of the process $P(i)$. Let $\tau(n)$ denotes the predicted burst time of Pth process. Then according to the simple averaging, the predicted burst time of process n+1 will be calculated as,

$$1. \tau(n+1) = (1/n) \sum T(i)$$

Where, $0 \leq i \leq n$ and $\sum T(i)$ is the summation of actual burst time of all the processes available till now.

Exponential Averaging or Aging

Let, T_n be the actual burst time of nth process. $\tau(n)$ be the predicted burst time for nth process then the CPU burst time for the next process (n+1) will be calculated as,

$$1. \tau(n+1) = \alpha \cdot T_n + (1-\alpha) \cdot \tau(n)$$

Where, α is the smoothing. Its value lies between 0 and 1.

Shortest Remaining Time First (SRTF) Scheduling Algorithm

This Algorithm is the **preemptive version** of **SJF scheduling**. In SRTF, the execution of the process can be stopped after certain amount of time. At the arrival of every process, the short term scheduler schedules the process with the least remaining burst time among the list of available processes and the running process.

Once all the processes are available in the **ready queue**, No preemption will be done and the algorithm will work as **SJF scheduling**. The context of the process is saved in the **Process Control Block** when the process is removed from the execution and the next process is scheduled. This PCB is accessed on the **next execution** of this process.

Example

In this Example, there are five jobs P1, P2, P3, P4, P5 and P6. Their arrival time and burst time are given below in the table.

Process ID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time	Response Time
1	0	8	20	20	12	0
2	1	4	10	9	5	1
3	2	2	4	2	0	2
4	3	1	5	2	1	4
5	4	3	13	9	6	10
6	5	2	7	2	0	5

P1	P2	P3	P3	P4	P6	P2	P5	P1	
0	1	2	3	4	5	7	10	13	20

$$\text{Avg Waiting Time} = 24/6$$

The Gantt chart is prepared according to the arrival and burst time given in the table.

1. Since, at time 0, the only available process is P1 with CPU burst time 8. This is the only available process in the list therefore it is scheduled.

- The next process arrives at time unit 1. Since the algorithm we are using is SRTF which is a preemptive one, the current execution is stopped and the scheduler checks for the process with the least burst time.
Till now, there are two processes available in the ready queue. The OS has executed P1 for one unit of time till now; the remaining burst time of P1 is 7 units. The burst time of Process P2 is 4 units. Hence Process P2 is scheduled on the CPU according to the algorithm.
- The next process P3 arrives at time unit 2. At this time, the execution of process P3 is stopped and the process with the least remaining burst time is searched. Since the process P3 has 2 unit of burst time hence it will be given priority over others.
- The Next Process P4 arrives at time unit 3. At this arrival, the scheduler will stop the execution of P4 and check which process is having least burst time among the available processes (P1, P2, P3 and P4). P1 and P2 are having the remaining burst time 7 units and 3 units respectively.

P3 and P4 are having the remaining burst time 1 unit each. Since, both are equal hence the scheduling will be done according to their arrival time. P3 arrives earlier than P4 and therefore it will be scheduled again.
- The Next Process P5 arrives at time unit 4. Till this time, the Process P3 has completed its execution and it is no more in the list. The scheduler will compare the remaining burst time of all the available processes. Since the burst time of process P4 is 1 which is least among all hence this will be scheduled.
- The Next Process P6 arrives at time unit 5, till this time, the Process P4 has completed its execution. We have 4 available processes till now, that are P1 (7), P2 (3), P5 (3) and P6 (2). The Burst time of P6 is the least among all hence P6 is scheduled. Since, now, all the processes are available hence the algorithm will now work same as SJF. P6 will be executed till its completion and then the process with the least remaining time will be scheduled.

Once all the processes arrive, No preemption is done and the algorithm will work as SJF.

SRTF GATE 2011 Example

If we talk about scheduling algorithm from the GATE point of view, they generally ask simple numerical questions about finding the average waiting time and Turnaround Time. Let's discuss the question asked in GATE 2011 on SRTF.

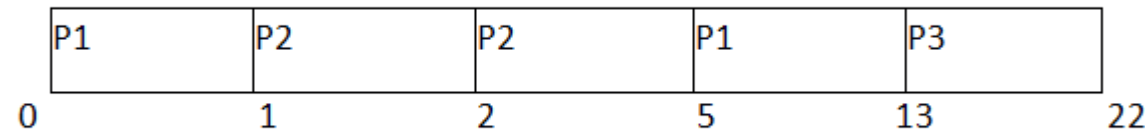
Q. Given the arrival time and burst time of 3 jobs in the table below. Calculate the Average waiting time of the system.

Process ID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time
1	0	9	13	13	4

2	1	4	5	4	0
3	2	9	22	20	11

There are three jobs P1, P2 and P3. P1 arrives at time unit 0; it will be scheduled first for the time until the next process arrives. P2 arrives at 1 unit of time. Its burst time is 4 units which is least among the jobs in the queue. Hence it will be scheduled next.

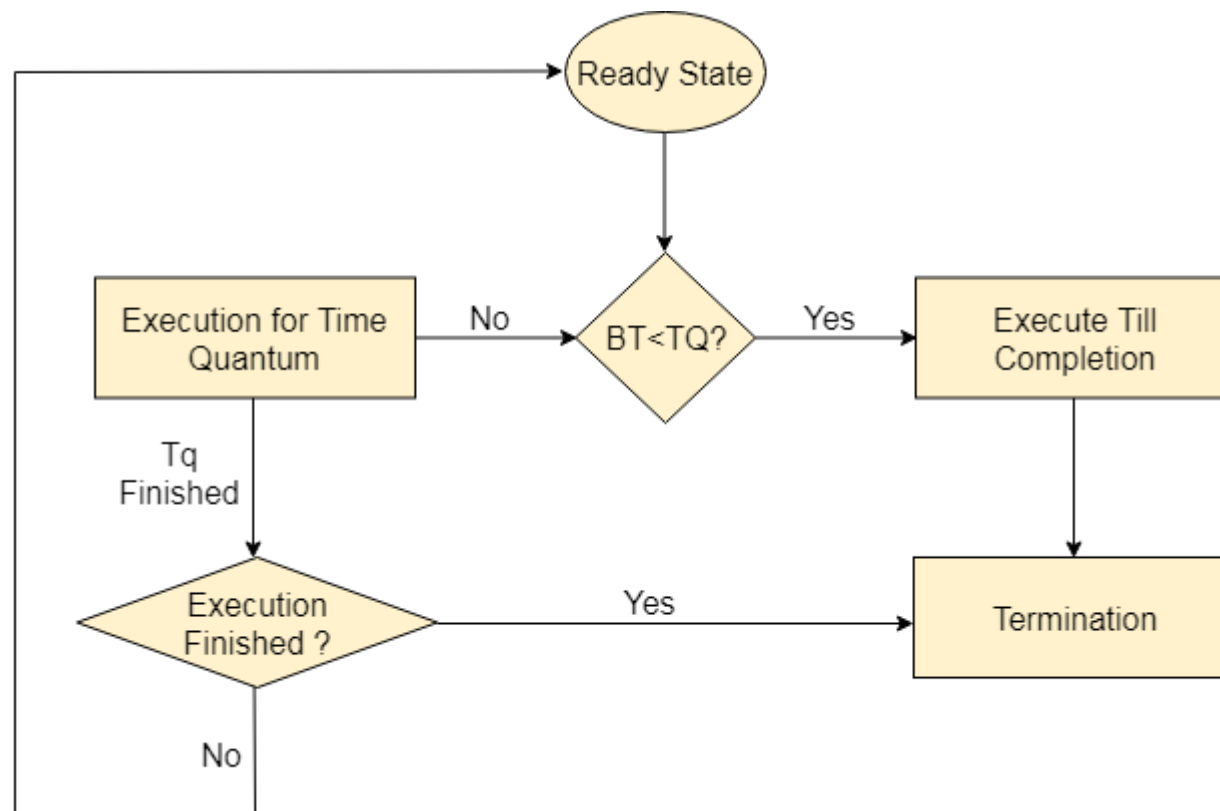
At time 2, P3 will arrive with burst time 9. Since remaining burst time of P2 is 3 units which are least among the available jobs. Hence the processor will continue its execution till its completion. Because all the jobs have been arrived so no preemption will be done now and all the jobs will be executed till the completion according to SJF.



$$\text{Avg Waiting Time} = (4+0+11)/3 = 5 \text{ units}$$

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 fair 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.

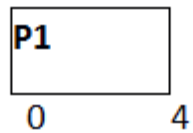
Ready Queue:

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.



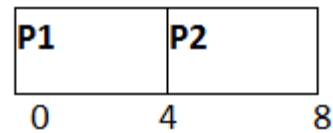
Ready Queue

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.

P1	P2	P3
0	4	8
		11

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.

P1	P2	P3	P4
0	4	8	11
			12

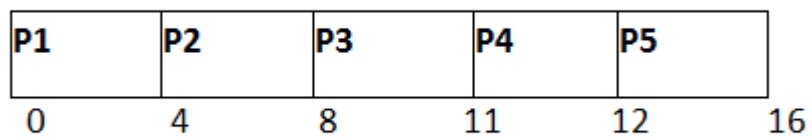
Ready Queue

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.

P1	P2	P3	P4	P5	P1	
0	4	8	11	12	16	17

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.

P1	P2	P3	P4	P5	P1	P6	
0	4	8	11	12	16	17	21

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.

P1	P2	P3	P4	P5	P1	P6	P2	
0	4	8	11	12	16	17	21	23

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.

P1	P2	P3	P4	P5	P1	P6	P2	P5	
0	4	8	11	12	16	17	21	23	24

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

Process 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

[Next](#) →← [Prev](#)

Highest Response Ratio Next (HRRN) Scheduling

Highest Response Ratio Next (HRRN) 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.

1. Response **Ratio** = $(W+S)/S$

Where,

1. $W \rightarrow$ Waiting Time
2. $S \rightarrow$ 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,

1. $HRNN \propto W$
2. $HRNN \propto (1/S)$

Hence,

1. This algorithm not only favors shorter job but it also concern the waiting time of the longer jobs.
2. Its mode is non preemptive hence context switching is minimal in this algorithm.

[Next](#) \rightarrow [Prev](#)

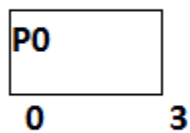
HRNN Example

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

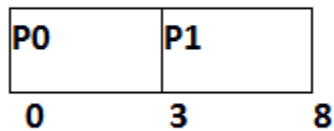
Process ID	Arrival Time	Burst Time
0	0	3
1	2	5
2	4	4
3	6	1

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 scheduled immediately.



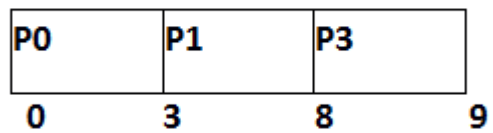
P0 is executed for 3 units, meanwhile, only one process P1 arrives at time 3. This will get scheduled immediately since the OS doesn't have a choice.



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

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

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

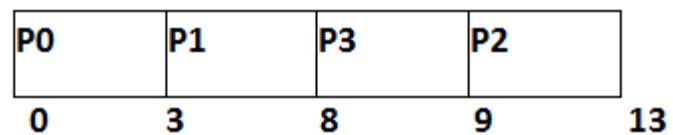


P3 is scheduled for 1 unit. The next available processes are P2 and P4. Let's calculate their Response ratio.

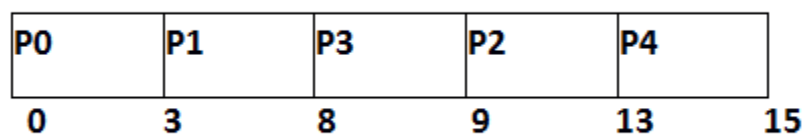
1. $RR(P2) = (5+4)/4 = 2.25$

2. $RR(P4) = (1+2)/2 = 1.5$

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 Time
0	0	3	3	3	0

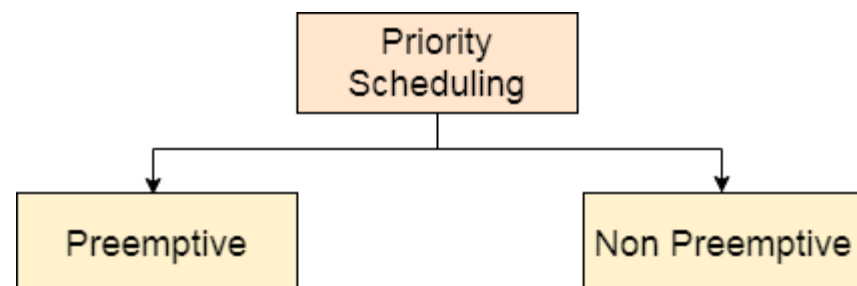
1	2	5	8	6	1
2	4	4	13	9	5
3	6	1	9	3	2
4	8	2	15	7	5

Average Waiting Time = 13/5

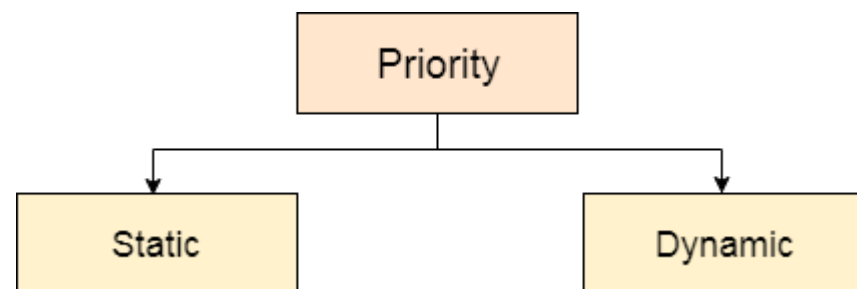
[Next →](#) [← Prev](#)

Priority Scheduling

In Priority scheduling, there is a priority number assigned to each process. In some systems, the lower the number, the higher the priority. While, in the others, the higher the number, the higher will be the priority. The Process with the higher priority among the available processes is given the CPU. There are two types of priority scheduling algorithm exists. One is Preemptive priority scheduling while the other is Non Preemptive Priority scheduling.



The priority number assigned to each of the process may or may not vary. If the priority number doesn't change itself throughout the process, it is called static priority, while if it keeps changing itself at the regular intervals, it is called dynamic priority.



Non Preemptive Priority Scheduling

In the Non Preemptive Priority scheduling, The Processes are scheduled according to the priority number assigned to them. Once the process gets scheduled, it will run till the completion. Generally, the lower the priority number, the higher is the priority of the process. The people might get confused with the priority numbers, hence in the GATE, there clearly mention which one is the highest priority and which one is the lowest one.

Example

In the Example, there are 7 processes P1, P2, P3, P4, P5, P6 and P7. Their priorities, Arrival Time and burst time are given in the table.

Process ID	Priority	Arrival Time	Burst Time
1	2	0	3
2	6	2	5
3	3	1	4
4	5	4	2
5	7	6	9
6	4	5	4
7	10	7	10

We can prepare the Gantt chart according to the Non Preemptive priority scheduling.

The Process P1 arrives at time 0 with the burst time of 3 units and the priority number 2. Since No other process has arrived till now hence the OS will schedule it immediately.

Meanwhile the execution of P1, two more Processes P2 and P3 are arrived. Since the priority of P3 is 3 hence the CPU will execute P3 over P2.

Meanwhile the execution of P3, All the processes get available in the ready queue. The Process with the lowest priority number will be given the priority. Since P6 has priority number assigned as 4 hence it will be executed just after P3.

After P6, P4 has the least priority number among the available processes; it will get executed for the whole burst time.

Since all the jobs are available in the ready queue hence All the Jobs will get executed according to their priorities. If two jobs have similar priority number assigned to them, the one with the least arrival time will be executed.

P1	P3	P6	P4	P2	P5	P7	
0	3	7	11	13	18	27	37

From the GANTT Chart prepared, we can determine the completion time of every process. The turnaround time, waiting time and response time will be determined.

1. Turn Around **Time** = **Completion** Time - Arrival Time

2. Waiting **Time** = **Turn** Around Time - Burst Time

Process Id	Priority	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time	Response Time
1	2	0	3	3	3	0	0
2	6	2	5	18	16	11	13
3	3	1	4	7	6	2	3
4	5	4	2	13	9	7	11
5	7	6	9	27	21	12	18
6	4	5	4	11	6	2	7

7	10	7	10	37	30	18	27
---	----	---	----	----	----	----	----

Avg Waiting Time = $(0+11+2+7+12+2+18)/7 = 52/7$ units

[Next →](#) [← Prev](#)

Preemptive Priority Scheduling

In Preemptive Priority Scheduling, at the time of arrival of a process in the ready queue, its Priority is compared with the priority of the other processes present in the ready queue as well as with the one which is being executed by the CPU at that point of time. The One with the highest priority among all the available processes will be given the CPU next.

The difference between preemptive priority scheduling and non preemptive priority scheduling is that, in the preemptive priority scheduling, the job which is being executed can be stopped at the arrival of a higher priority job.

Once all the jobs get available in the ready queue, the algorithm will behave as non-preemptive priority scheduling, which means the job scheduled will run till the completion and no preemption will be done.

Example

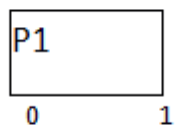
There are 7 processes P1, P2, P3, P4, P5, P6 and P7 given. Their respective priorities, Arrival Times and Burst times are given in the table below.

Process Id	Priority	Arrival Time	Burst Time
1	2(L)	0	1
2	6	1	7
3	3	2	3

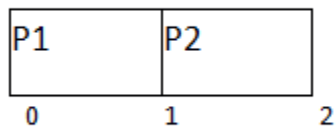
4	5	3	6
5	4	4	5
6	10(H)	5	15
7	9	15	8

GANTT chart Preparation

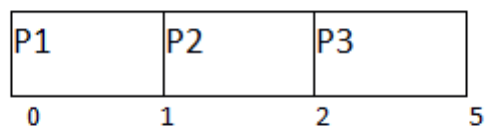
At time 0, P1 arrives with the burst time of 1 units and priority 2. Since no other process is available hence this will be scheduled till next job arrives or its completion (whichever is lesser).



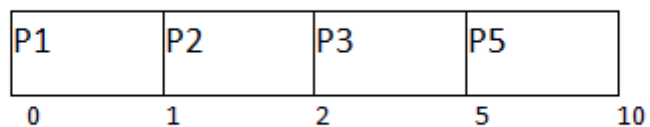
At time 1, P2 arrives. P1 has completed its execution and no other process is available at this time hence the Operating system has to schedule it regardless of the priority assigned to it.



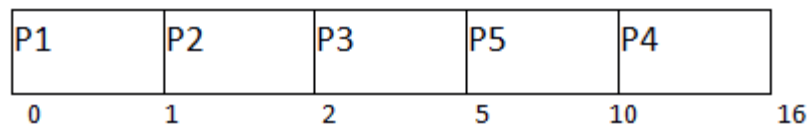
The Next process P3 arrives at time unit 2, the priority of P3 is higher to P2. Hence the execution of P2 will be stopped and P3 will be scheduled on the CPU.



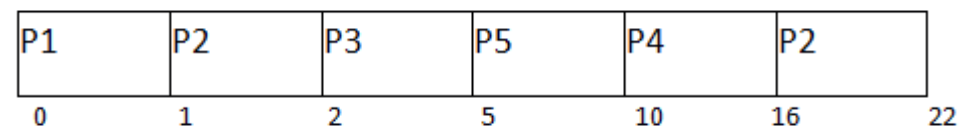
During the execution of P3, three more processes P4, P5 and P6 becomes available. Since, all these three have the priority lower to the process in execution so PS can't preempt the process. P3 will complete its execution and then P5 will be scheduled with the priority highest among the available processes.



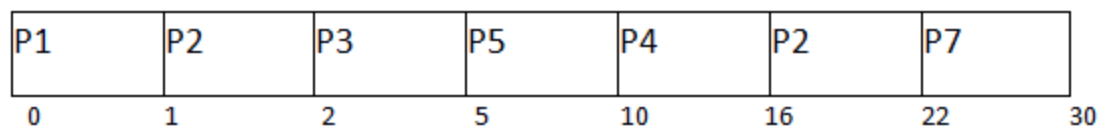
Meanwhile the execution of P5, all the processes got available in the ready queue. At this point, the algorithm will start behaving as Non Preemptive Priority Scheduling. Hence now, once all the processes get available in the ready queue, the OS just took the process with the highest priority and execute that process till completion. In this case, P4 will be scheduled and will be executed till the completion.



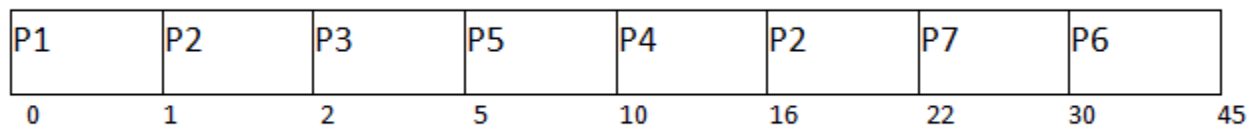
Since P4 is completed, the other process with the highest priority available in the ready queue is P2. Hence P2 will be scheduled next.



P2 is given the CPU till the completion. Since its remaining burst time is 6 units hence P7 will be scheduled after this.



The only remaining process is P6 with the least priority, the Operating System has no choice unless of executing it. This will be executed at the last.



The Completion Time of each process is determined with the help of GANTT chart. The turnaround time and the waiting time can be calculated by the following formula.

1. Turnaround **Time** = **Completion** Time - Arrival Time

2. Waiting **Time** = **Turn** Around Time - Burst Time

Process Id	Priority	Arrival Time	Burst Time	Completion Time	Turn around Time	Waiting Time
1	2	0	1	1	1	0
2	6	1	7	22	21	14
3	3	2	3	5	3	0
4	5	3	6	16	13	7
5	4	4	5	10	6	1
6	10	5	15	45	40	25
7	9	6	8	30	24	16

$$\text{Avg Waiting Time} = (0+14+0+7+1+25+16)/7 = 63/7 = 9 \text{ units}$$

SRTF with Processes contains CPU and IO Time

Till now, we were considering the CPU bound jobs only. However, the process might need some IO operation or some resource to complete its execution. In this Example, we are considering, the IO bound processes.

In the Example, there are four jobs with process ID P1, P2, P3 and P4 are available. Their Arrival Time, and the CPU Burst time are given in the table below.

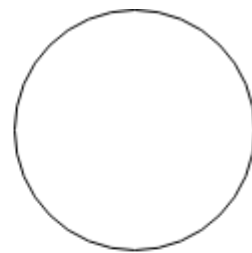
Process Id	Arrival Time	(Burst Time, IO Burst Time, Burst Time)
1	0	(3,2,2)
2	0	(1,3,1)
3	3	(3,1,2)
4	6	(5,4,5)

GANTT Chart Preparation

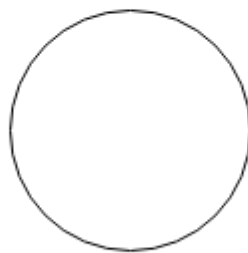
At time 0, the process P1 and P2 arrives. Since the algorithm we are using is SRTF hence, the process with the shortest burst time will be scheduled on the CPU. In this case, it is P2.



Ready State

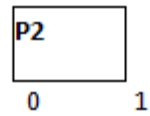


Running State

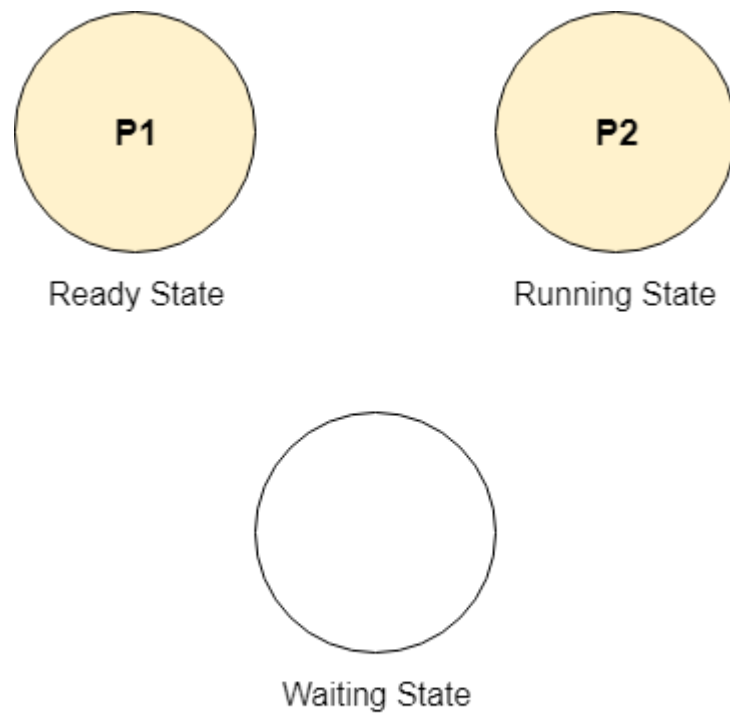


Waiting State

Before Execution



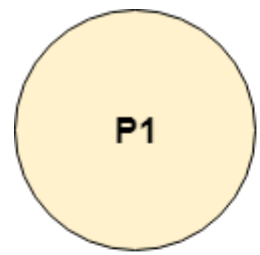
From time 0 to time 1, P2 will be in running state.



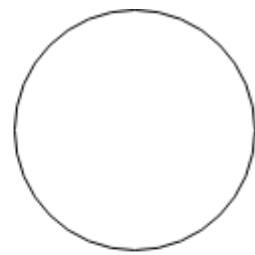
Time 0 to Time 1

P2 also needs some IO time in order to complete its execution. After 1 unit of execution, P2 will change its state from running to waiting. The processor becomes free to execute other jobs. Since No other process is available at this point of time other than P1 so P1 will get executed.

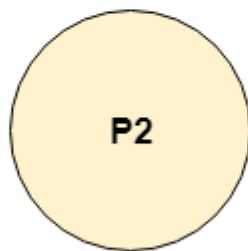
The following diagram illustrates the processes and states at Time 1. The process P2 went to waiting state and the CPU becomes idle at this time.



Ready State



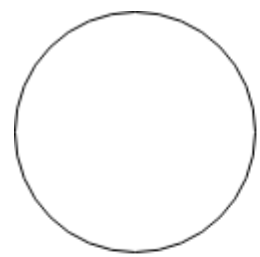
Running State



Waiting State

At Time 1

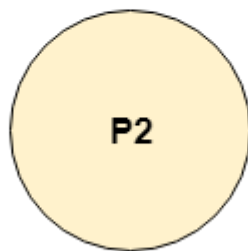
From time 1 to 3, since P2 is being in waiting state, and no other process is available in ready queue, hence the only available process P1 will be executed in this period of time.



Ready State

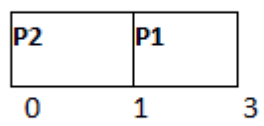


Running State



Waiting State

Time 1 to Time 3



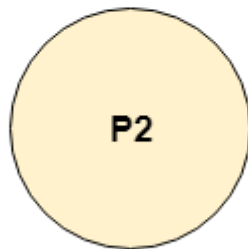
At time 3, the process P3 arrived with the total CPU burst time of 5 units. Since the remaining burst time of P1 is lesser than P3 hence CPU will continue its execution.



Ready State



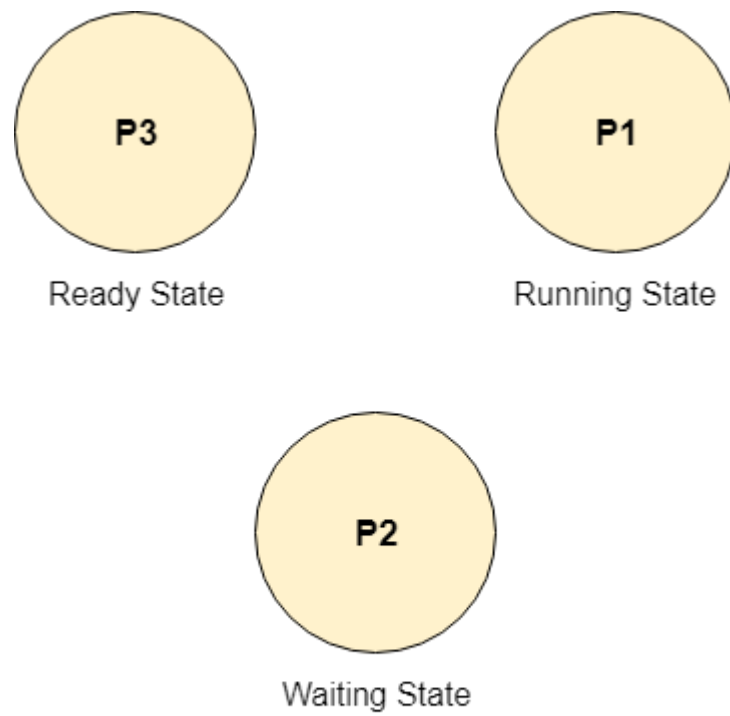
Running State



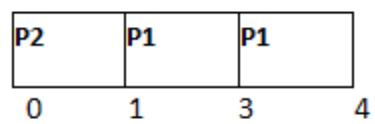
Waiting State

At Time 3

Hence, P1 will remain in the running state from time 3 to time 4.



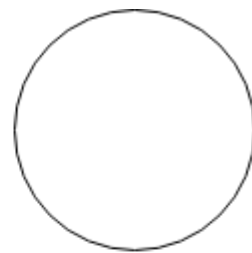
Time 3 to Time 4



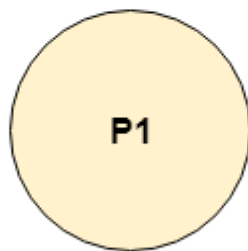
Since P1 is an IO bound process. At time unit 4, it will change its state from running to waiting. Processor becomes free for the execution of other jobs. Since P2 also becomes available at time 4 because it has completed the IO operation and now it needs another 1 unit of CPU burst time. P3 is also available and requires 5 units of total CPU burst time.



Ready State



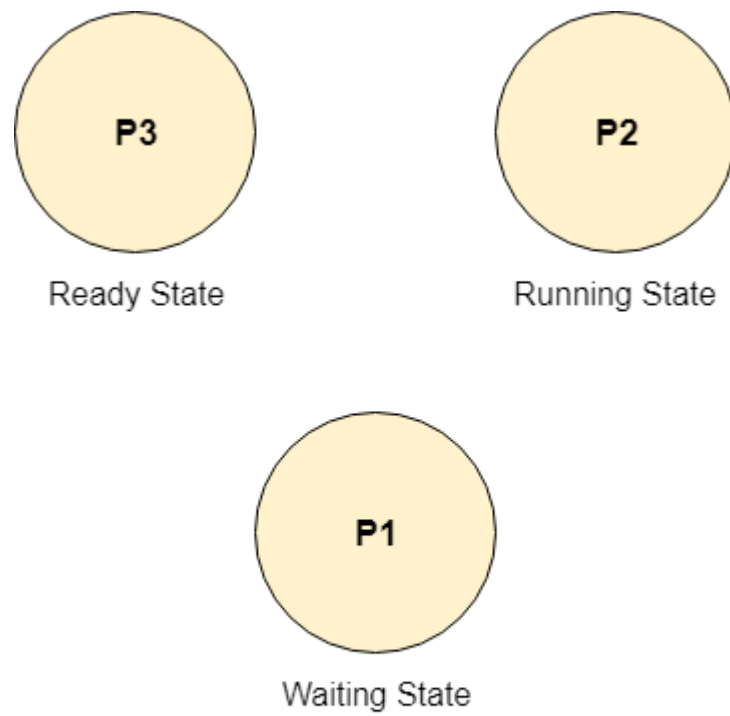
Running State



Waiting State

At Time 4

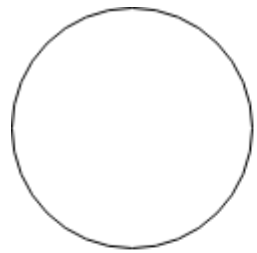
The process with the least remaining CPU burst time among the available processes will get executed. In our case, such process is P2 which requires 1 unit of burst time hence it will be given the CPU.



Time 4 to Time 5

P2	P1	P1	P2
0	1	3	4
5			

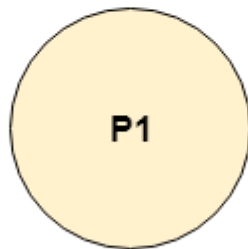
At time 5, P2 is finished. P1 is still in waiting state. At this point of time, the only available process is P3, hence it will be given the CPU.



Ready State



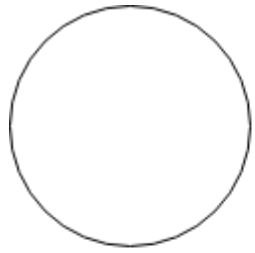
Running State



Waiting State

At Time 5

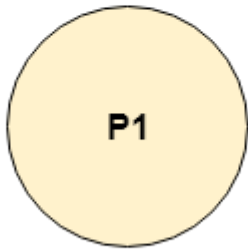
From Time 5 to time 6, P3 will be in the running state; meanwhile, P1 will still be in waiting state.



Ready State

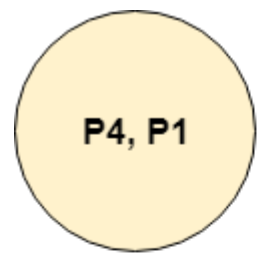


Running State



Waiting State

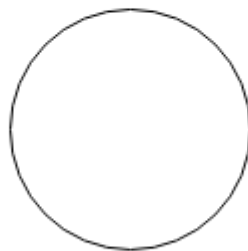
Time 5 to Time 6



Ready State

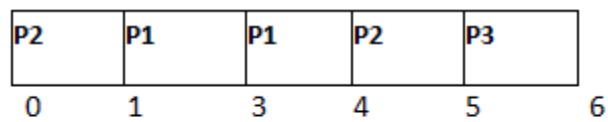


Running State



Waiting State

At Time 6

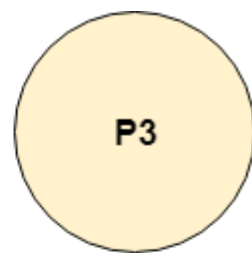


At time 6, the Process P4 arrives in the ready queue. The P1 has also done with the IO and becomes available for the execution. P3 is not yet finished and still needs another 2 unit of CPU burst time.

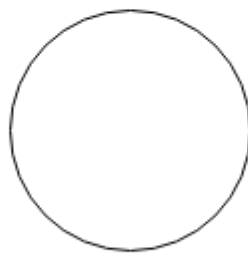
From time 6 to time 8, the reaming CPU burst time of Process P3 is least among the available processes, hence P3 will be given the CPU.



Ready State

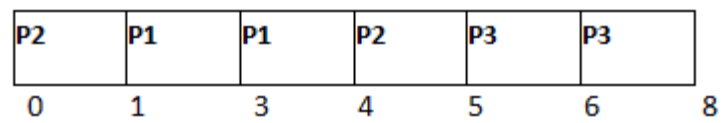


Running State



Waiting State

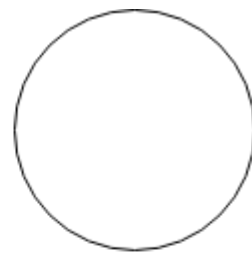
Time 6 to Time 8



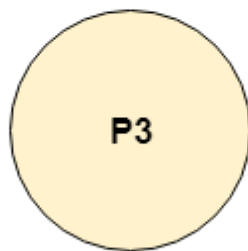
P3 needs some IO operation in order to complete its execution. At time 8, P3 will change its state from running to waiting. The CPU becomes free to execute the other processes. Process P4 and P1 are available out of which, the process with the least remaining burst time will get executed.



Ready State



Running State



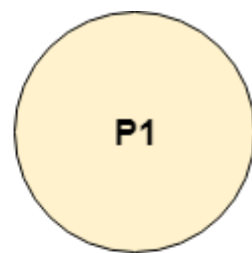
Waiting State

At Time 8

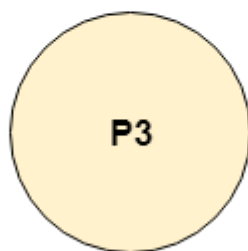
From time 8 to time 9, the process P1 will get executed.



Ready State



Running State



Waiting State

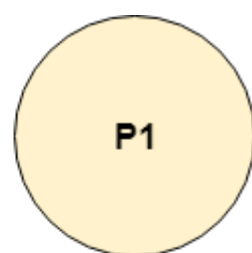
Time 8 to Time 9

P2	P1	P1	P2	P3	P3	P1	
0	1	3	4	5	6	8	9

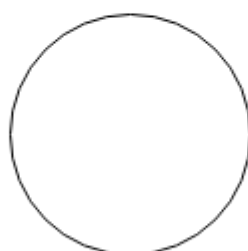
At time 9, the IO of process P3 is finished and it will now be available in the ready state along with P4 which is already waiting there for its turn. In order to complete its execution, it needs another 2 unit of burst time. P1 is in running state at this point of the time while no process is present in the waiting state.



Ready State



Running State

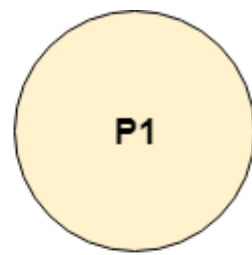


Waiting State

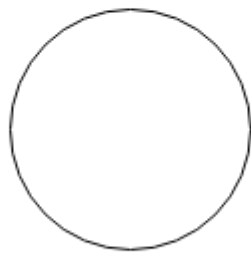
At Time 9



Ready State



Running State



Waiting State

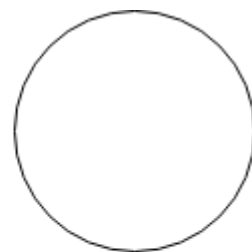
Time 9 to Time 10

from time 9 to 10 , the process P1 will get executed since its remaining CPU burst time is lesser then the processes P4 and P3 available in the ready queue.

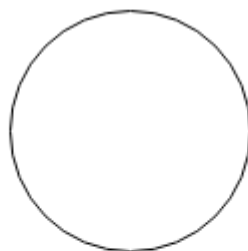
P2	P1	P1	P2	P3	P3	P1	P1	
0	1	3	4	5	6	8	9	10



Ready State



Running State



Waiting State

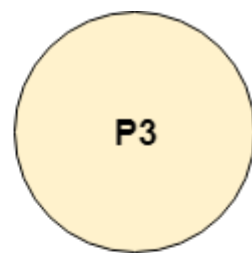
At Time 10

At time 10, execution of P1 is finished, and now the CPU becomes idle. The process with the lesser CPU burst time among the ready processes will get the CPU turn.

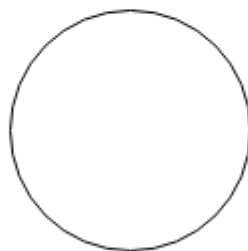
From time 10 to 12, the process P3 will get executed till its completion because of the fact that its remaining CPU burst time is the between the two available processes. It needs 2 units of more CPU burst time, since No other process will be arrived in the ready state hence No preemption will be done and it will be executed till the completion.



Ready State



Running State

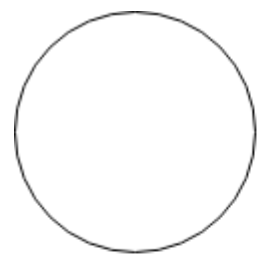


Waiting State

Time 10 to Time 12

P2	P1	P1	P2	P3	P3	P1	P1	P3	
0	1	3	4	5	6	8	9	10	12

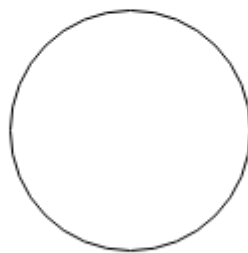
At time 12, the process P3 will get completed, since there is only one process P4 available in the ready state hence P4 will be given the CPU.



Ready State



Running State



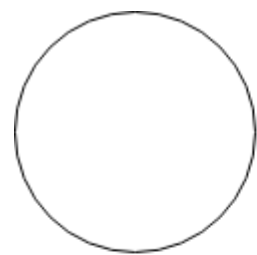
Waiting State

Time 12 to Time 17

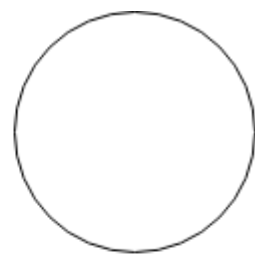
P4 needs 5 units of CPU burst time before IO, hence it will be executed till time 17 (for 5 units) and then it will change its state from running to waiting.

P2	P1	P1	P2	P3	P3	P1	P1	P3	P4	
0	1	3	4	5	6	8	9	10	12	17

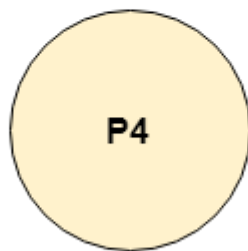
At time 17, the Process P4 changes its state from running to waiting. Since this is the only process in the system hence the CPU will remain idle until P4 becomes available again.



Ready State



Running State



Waiting State

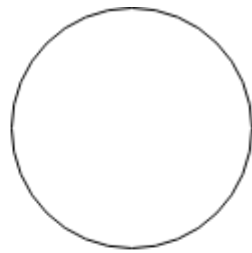
Time 17 to Time 21

P2	P1	P1	P2	P3	P3	P1	P1	P3	P4	
0	1	3	4	5	6	8	9	10	12	17 21

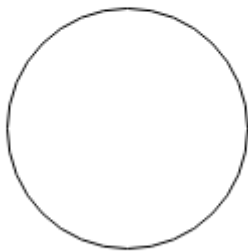
At time 21, P4 will be done with the IO operation and becomes available in the ready state.



Ready State



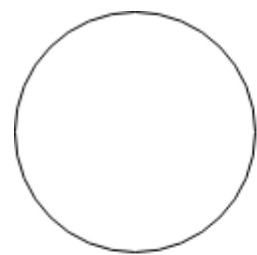
Running State



Waiting State

At Time 21

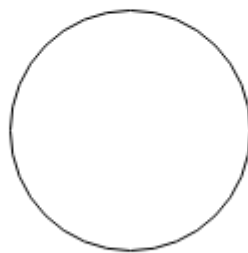
From time 21, the process P4 will get scheduled. Since No other process is in ready queue hence the processor don't have any choice. It will be executed till completion.



Ready State



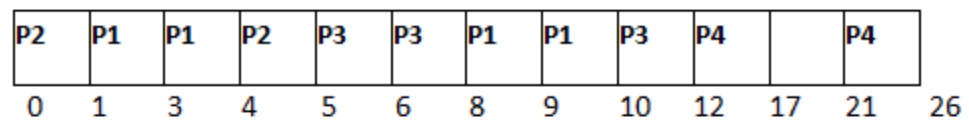
Running State



Waiting State

Time 21 to Time 26

Final Gantt chart:



Process Id	Arrival Time	Total CPU Burst Time	Completion Time	Turn Around Time	Waiting Time
------------	--------------	----------------------	-----------------	------------------	--------------

1	0	5	10	10	5
2	0	2	5	5	3
3	3	5	12	9	4
4	6	10	26	20	10

Average waiting Time = $(5+3+4+10)/4 = 22/4$ units

#3.Synchronization

Introduction

When two or more process cooperates with each other, their order of execution must be preserved otherwise there can be conflicts in their execution and inappropriate outputs can be produced.

A cooperative process is the one which can affect the execution of other process or can be affected by the execution of other process. Such processes need to be synchronized so that their order of execution can be guaranteed.

The procedure involved in preserving the appropriate order of execution of cooperative processes is known as Process Synchronization. There are various synchronization mechanisms that are used to synchronize the processes.

Race Condition

A Race Condition typically occurs when two or more threads try to read, write and possibly make the decisions based on the memory that they are accessing concurrently.

Critical Section

The regions of a program that try to access shared resources and may cause race conditions are called critical section. To avoid race condition among the processes, we need to assure that only one process at a time can execute within the critical section.

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.

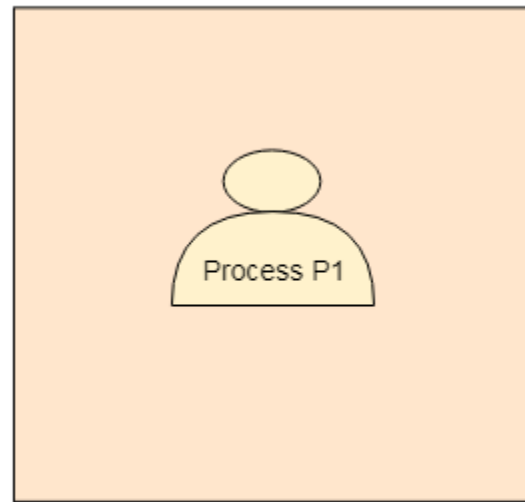
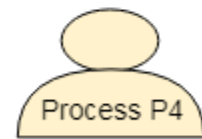
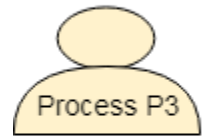
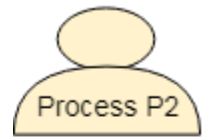
Requirements of Synchronization mechanisms

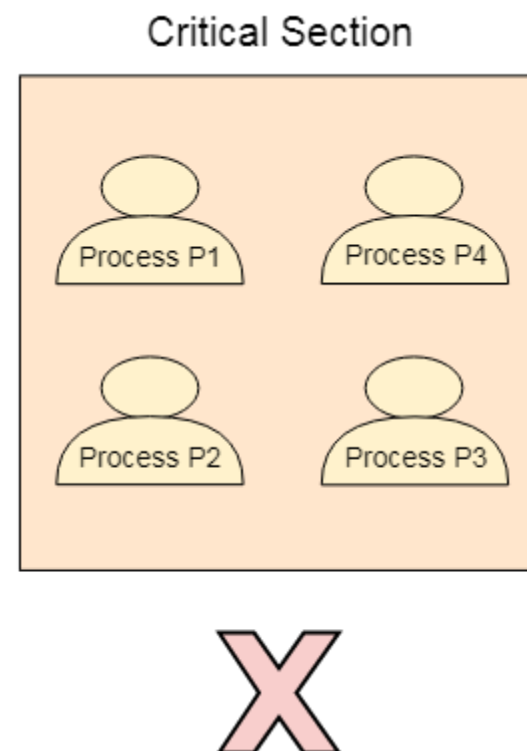
Primary

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.

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.

[Next →](#) [← Prev](#)

Lock Variable

This is the simplest synchronization mechanism. This is a Software Mechanism implemented in User mode. This is a busy waiting solution which can be used for more than two processes.

In this mechanism, a Lock variable **lock** is used. Two values of lock can be possible, either 0 or 1. Lock value 0 means that the critical section is vacant while the lock value 1 means that it is occupied.

A process which wants to get into the critical section first checks the value of the lock variable. If it is 0 then it sets the value of lock as 1 and enters into the critical section, otherwise it waits.

The pseudo code of the mechanism looks like following.

1. Entry Section →
2. While (lock != 0);
3. Lock = 1;
4. //Critical Section
5. Exit Section →
6. Lock = 0;

If we look at the Pseudo Code, we find that there are three sections in the code. Entry Section, Critical Section and the exit section.

Initially the value of **lock variable** is **0**. The process which needs to get into the **critical section**, enters into the entry section and checks the condition provided in the while loop.

The process will wait infinitely until the value of **lock** is 1 (that is implied by while loop). Since, at the very first time critical section is vacant hence the process will enter the critical section by setting the lock variable as 1.

When the process exits from the critical section, then in the exit section, it reassigns the value of **lock** as 0.

Every Synchronization mechanism is judged on the basis of four conditions.

1. Mutual Exclusion
2. Progress
3. Bounded Waiting

4. Portability

Out of the four parameters, Mutual Exclusion and Progress must be provided by any solution. Let's analyze this mechanism on the basis of the above mentioned conditions.

Mutual Exclusion

The lock variable mechanism doesn't provide Mutual Exclusion in some of the cases. This can be better described by looking at the pseudo code by the Operating System point of view I.E. Assembly code of the program. Let's convert the Code into the assembly language.

1. *Load Lock, R0*
2. *CMP R0, #0*
3. *JNZ Step 1*
4. *Store #1, Lock*
5. *Store #0, Lock*

Let us consider that we have two processes P1 and P2. The process P1 wants to execute its critical section. P1 gets into the entry section. Since the value of lock is 0 hence P1 changes its value from 0 to 1 and enters into the critical section.

Meanwhile, P1 is preempted by the CPU and P2 gets scheduled. Now there is no other process in the critical section and the value of lock variable is 0. P2 also wants to execute its critical section. It enters into the critical section by setting the lock variable to 1.

Now, CPU changes P1's state from waiting to running. P1 is yet to finish its critical section. P1 has already checked the value of lock variable and remembers that its value was 0 when it previously checked it. Hence, it also enters into the critical section without checking the updated value of lock variable.

Now, we got two processes in the critical section. According to the condition of mutual exclusion, morethan one process in the critical section must not be present at the same time. Hence, the lock variable mechanism doesn't guarantee the mutual exclusion.

The problem with the lock variable mechanism is that, at the same time, more than one process can see the vacant tag and more than one process can enter in the critical section. Hence, the lock variable doesn't provide the mutual exclusion that's why it cannot be used in general.

Since, this method is failed at the basic step; hence, there is no need to talk about the other conditions to be fulfilled.

Test Set Lock Mechanism

Modification in the assembly code

In lock variable mechanism, Sometimes Process reads the old value of lock variable and enters the critical section. Due to this reason, more than one process might get into critical section. However, the code shown in the part one of the following section can be replaced with the code shown in the part two. This doesn't affect the algorithm but, by doing this, we can manage to provide the mutual exclusion to some extent but not completely.

In the updated version of code, the value of Lock is loaded into the local register R0 and then value of lock is set to 1.

However, in step 3, the previous value of lock (that is now stored into R0) is compared with 0. if this is 0 then the process will simply enter into the critical section otherwise will wait by executing continuously in the loop.

The benefit of setting the lock immediately to 1 by the process itself is that, now the process which enters into the critical section carries the updated value of lock variable that is 1.

In the case when it gets preempted and scheduled again then also it will not enter the critical section regardless of the current value of the lock variable as it already knows what the updated value of lock variable is.

Section 1	Section 2
<div>1. Load Lock, R0</div> <div>2. CMP R0, #0</div> <div>3. JNZ step1</div> <div>4. store #1, Lock</div>	<div>1. Load Lock, R0</div> <div>2. Store #1, Lock</div> <div>3. CMP R0, #0</div> <div>4. JNZ step 1</div>

TSL Instruction

However, the solution provided in the above segment provides mutual exclusion to some extent but it doesn't make sure that the mutual exclusion will always be there. There is a possibility of having more than one process in the critical section.

What if the process gets preempted just after executing the first instruction of the assembly code written in section 2? In that case, it will carry the old value of lock variable with it and it will enter into the critical section regardless of knowing the current value of lock variable. This may make the two processes present in the critical section at the same time.

To get rid of this problem, we have to make sure that the preemption must not take place just after loading the previous value of lock variable and before setting it to 1. The problem can be solved if we can be able to merge the first two instructions.

In order to address the problem, the operating system provides a special instruction called **Test Set Lock (TSL)** instruction which simply loads the value of lock variable into the local register R0 and sets it to 1 simultaneously

The process which executes the TSL first will enter into the critical section and no other process after that can enter until the first process comes out. No process can execute the critical section even in the case of preemption of the first process.

The assembly code of the solution will look like following.

1. *TSL Lock, R0*
2. *CMP R0, #0*
3. *JNZ step 1*

Let's examine TSL on the basis of the four conditions.

- **Mutual Exclusion**

Mutual Exclusion is guaranteed in TSL mechanism since a process can never be preempted just before setting the lock variable. Only one process can see the lock variable as 0 at a particular time and that's why, the mutual exclusion is guaranteed.

- **Progress**

According to the definition of the progress, a process which doesn't want to enter in the critical section should not stop other processes to get into it. In TSL mechanism, a process will execute the TSL instruction only when it wants to get into the critical section. The value of the lock will always be 0 if no process doesn't want to enter into the critical section hence the progress is always guaranteed in TSL.

- **Bounded Waiting**

Bounded Waiting is not guaranteed in TSL. Some process might not get a chance for so long. We cannot predict for a process that it will definitely get a chance to enter in critical section after a certain time.

- **Architectural Neutrality**

TSL doesn't provide Architectural Neutrality. It depends on the hardware platform. The TSL instruction is provided by the operating system. Some platforms might not provide that. Hence it is not Architectural natural.

Mutual Exclusion	✓
Progress	✓
Bounded Waiting	✗
Portability	✗

Priority Inversion

In TSL mechanism, there can be a problem of priority inversion. Let's say that there are two cooperative processes, P1 and P2.

The priority of P1 is 2 while that of P2 is 1. P1 arrives earlier and got scheduled by the CPU. Since it is a cooperative process and wants to execute in the critical section hence it will enter in the critical section by setting the lock variable to 1.

Now, P2 arrives in the ready queue. The priority of P2 is higher than P1 hence according to priority scheduling, P2 is scheduled and P1 got preempted. P2 is also a cooperative process and wants to execute inside the critical section.

Although, P1 got preempted but the value of lock variable will be shown as 1 since P1 is not completed and it is yet to finish its critical section.

P1 needs to finish the critical section but according to the scheduling algorithm, CPU is with P2. P2 wants to execute in the critical section, but according to the synchronization mechanism, critical section is with P1.

This is a kind of lock where each of the process neither executes nor completes. Such kind of lock is called **Spin Lock**.

This is different from deadlock since they are not in blocked state. One is in ready state and the other is in running state, but neither of the two is being executed.

Turn Variable or 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 P_i and P_j . They share a variable called turn variable. The pseudo code of the program can be given as following.

For Process P_i

1. Non - CS
2. while (turn ! = i);
3. Critical Section
4. turn = j;
5. Non - CS

For Process P_j

1. Non - CS
2. while (turn ! = j);
3. Critical Section
4. turn = i ;
5. Non - CS

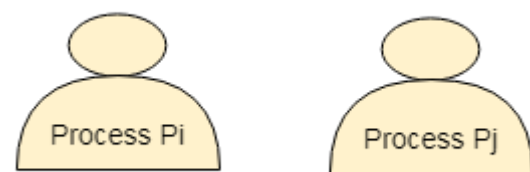
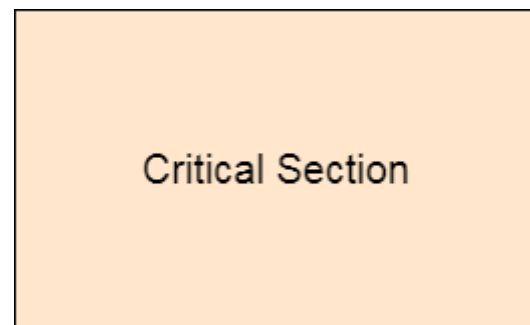
The actual problem of the lock variable approach was the fact that the process was entering in the critical section only when the lock variable is 1. More than one process could see the lock variable as 1 at the same time hence the mutual exclusion was not guaranteed there.

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.

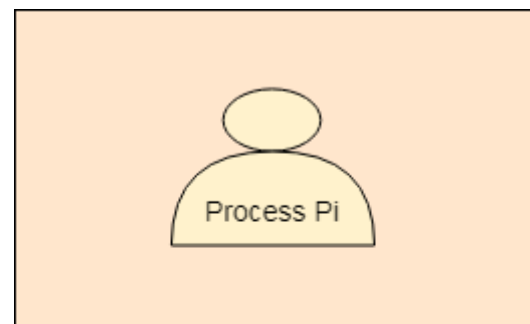
In the entry section, in general, the process P_i will not enter in the critical section until its value is j or the process P_j will not enter in the critical section until its value is i.

Initially, two processes P_i and P_j are available and want to execute into critical section.

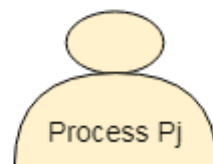


Turn = i

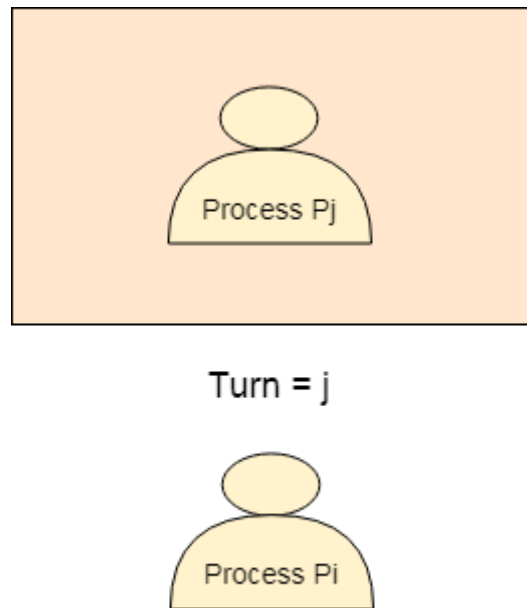
The turn variable is equal to i hence Pi will get the chance to enter into the critical section. The value of Pi remains I until Pi finishes critical section.



Turn = i



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.



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.

Mutual Exclusion	✓
Progress	✗
Bounded Waiting	✓
Portability	✓

[Next →](#) [← Prev](#)

Interested Variable Mechanism

We have to make sure that the progress must be provided by our synchronization mechanism. In the turn variable mechanism, progress was not provided due to the fact that the process which doesn't want to enter in the critical section does not consider the other interested process as well.

The other process will also have to wait regardless of the fact that there is no one inside the critical section. If the operating system can make use of an extra variable along with the turn variable then this problem can be solved and our problem can provide progress to most of the extent.

Interested variable mechanism makes use of an extra Boolean variable to make sure that the progress is provided.

For Process P_i

1. Non CS
2. $Int[i] = T$;

3. while (Int[j] == T) ;
4. Critical Section
5. Int[i] = F ;

For Process Pj

1. Non CS
2. Int [1] = T ;
3. while (Int[i] == T) ;
4. Critical Section
5. Int[j]=F ;

In this mechanism, an extra variable **interested** is used. This is a Boolean variable used to store the interest of the processes to get enter inside the critical section.

A process which wants to enter in the critical section first checks in the entry section whether the other process is interested to get inside. The process will wait for the time until the other process is interested.

In exit section, the process makes the value of its interest variable false so that the other process can get into the critical section.

The table shows the possible values of interest variable of both the processes and the process which get the chance in the scenario.

Interest [Pi]	Interest [Pj]	Process which get the chance
True	True	The process which first shows interest.
True	False	Pi
False	True	Pj

False	False	X
-------	-------	---

Let's analyze the mechanism on the basis of the requirements.

Mutual Exclusion

In interested variable mechanism, if one process is interested in getting into the CPU then the other process will wait until it becomes uninterested. Therefore, more than one process can never be present in the critical section at the same time hence the mechanism guarantees mutual exclusion.

Progress

In this mechanism, if a process is not interested in getting into the critical section then it will not stop the other process from getting into the critical section. Therefore the progress will definitely be provided by this method.

Bounded Waiting

To analyze bounded waiting, let us consider two processes P_i and P_j , are the cooperative processes wants to execute in the critical section. The instructions executed by the processes are shown below in relative manner.

Process P_i	Process P_j	Process P_i	Process P_j
1. $Int [P_i] = True$ 2. $while (Int [P_j] == True);$ 3. Critical Section	1. $Int [P_j] = True$ 2. $while (Int[P_i]==True);$	1. $Int [P_i] = False$ 2. $Int [P_i] = True$ 3. $while (Int [P_j] == True);$ //waiting for P_j	1. $While (Int [P_i] == True);$ //waiting for P_j

Initially, the interest variable of both the processes is **false**. The process P_i shows the interest to get inside the critical section.

It sets its Interest Variable to true and check whether the Pj is also interested or not. Since the other process's interest variable is false hence Pi will get enter into the critical section.

Meanwhile, the process Pi is preempted and Pj is scheduled. Pj is a cooperative process and therefore, it also wants to enter in the critical section. It shows its interest by setting the interest variable to true.

It also checks whether the other process is also interested or not. We should notice that Pi is preempted but its interested variable is true that means it needs to further execute in the critical section. Therefore Pj will not get the chance and gets stuck in the while loop.

Meanwhile, CPU changes Pi's state from blocked to running. Pi is yet to finish its critical section hence it finishes the critical section and makes an exit by setting the interest variable to False.

Now, a case can be possible when Pi again wants to enter in the critical section and set its interested variable to true and checks whether the interested variable of Pj is true. Here, Pj's interest variable is True hence Pi will get stuck in the while loop and waits for Pj become uninterested.

Since, Pj still stuck in the while loop waiting for the Pi' interested variable to become false. Therefore, both the processes are waiting for each other and none of them is getting into the critical section.

This is a condition of deadlock and bounded waiting can never be provided in the case of deadlock.

Therefore, we can say that the interested variable mechanism doesn't guarantee deadlock.

Architectural Neutrality

The mechanism is a complete software mechanism executed in the user mode therefore it guarantees portability or architectural neutrality.

Mutual Exclusion	✓
Progress	✓
Bounded Waiting	✗
Portability	✓

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

```

1. # define N 2
2. # define TRUE 1
3. # define FALSE 0
4. int interested[N] = FALSE;
5. int turn;
6. voidEntry_Section (int process)
7. {
8.     int other;
9.     other = 1-process;
10.    interested[process] = TRUE;
11.    turn = process;
12.    while (interested [other] =True && TURN=process);
13.}

```

```

14. voidExit_Section (int process)
15. {
16.     interested [process] = FALSE;
17. }

```

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

```

1. voidEntry_Section (int process)
2. {
3.     1. int other;
4.     2. other = 1-process;
5.     3. interested[process] = TRUE;
6.     4. turn = process;
7.     5. while (interested [other] =True && TURN=process);
8. }
9.
10. Critical Section
11.
12. voidExit_Section (int process)
13. {
14.     6. interested [process] = FALSE;
15. }

```

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.

1. 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.

1. 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.

1. P1 → 6
2. 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	
Progress	
Bounded Waiting	
Portability	

Synchronization Mechanism without busy waiting

All the solutions we have seen till now were intended to provide mutual exclusion with busy waiting. However, busy waiting is not the optimal allocation of resources because it keeps CPU busy all the time in checking the while loops condition continuously although the process is waiting for the critical section to become available.

All the synchronization mechanism with busy waiting are also suffering from the priority inversion problem that is there is always a possibility of spin lock whenever there is a process with the higher priority has to wait outside the critical section since the mechanism intends to execute the lower priority process in the critical section.

However these problems need a proper solution without busy waiting and priority inversion.

[Next →](#) [← Prev](#)

Sleep and Wake

(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 using a buffer. The code that simulates the sleep and wake mechanism in terms of providing the solution to producer consumer problem is shown below.

```
1. #define N 100 //maximum slots in buffer
2. #define count=0 //items in the buffer
3. void producer (void)
4. {
5.     int item;
6.     while(True)
7.     {
8.         item = produce_item(); //producer produces an item
9.         if(count == N) //if the buffer is full then the producer will sleep
10.            Sleep();
11.        insert_item (item); //the item is inserted into buffer
12.        count=count+1;
13.        if(count==1) //The producer will wake up the
14.            //consumer if there is at least 1 item in the buffer
15.            wake-up(consumer);
16.    }
17.}
18.
19.void consumer (void)
20.{
21.    int item;
```

```

22. while(True)
23. {
24.     {
25.         if(count == 0) //The consumer will sleep if the buffer is empty.
26.         sleep();
27.         item = remove_item();
28.         count = count - 1;
29.         if(count == N-1) //if there is at least one slot available in the buffer
30.         //then the consumer will wake up producer
31.         wake-up(producer);
32.         consume_item(item); //the item is read by consumer.
33.     }
34. }
35.}

```

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 store 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.

```

1. struct Semaphore
2. {
3.     int value; // processes that can enter in the critical section simultaneously.
4.     queue type L; // L contains set of processes which get blocked
5. }
6. Down (Semaphore S)
7. {
8.     SS.value = S.value - 1; //semaphore's value will get decreased when a new
9.     //process enter in the critical section
10.    if (S.value < 0)
11.    {
12.        put_process(PCB) in L; //if the value is negative then
13.        //the process will get into the blocked state.
14.        Sleep();
15.    }
16.    else
17.        return;
18.}
19.up (Semaphore s)
20.{
21.    SS.value = S.value+1; //semaphore value will get increased when

```

```

22. //it makes an exit from the critical section.
23. if(S.value<=0)
24. {
25.     select a process from L; //if the value of semaphore is positive
26.     //then wake one of the processes in the blocked queue.
27.     wake-up();
28. }
29. }
30.}

```

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.

Problem on Counting Semaphore

The questions are being asked on counting semaphore in GATE. Generally the questions are very simple that contains only subtraction and addition.

1. Wait → Decre → Down → P
2. Signal → Inc → Up → V

The following type questions can be asked in GATE.

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

1. $S = 12$ (initial)
2. 10 p (wait) :
3. $S = S - 10 = 12 - 10 = 2$
4. then 4 V :
5. $S = S + 4 = 2 + 4 = 6$

Hence, the final value of counting semaphore is 6.

Binary Semaphore or Mutex

In counting semaphore, Mutual exclusion was not provided because we have 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.

1. StructBsemaphore
2. {
3. enum Value(0,1); //value is enumerated data type which can only have two values 0 or 1.
4. Queue type L;
5. }
6. /* L contains all PCBs corresponding to process
7. Blocked while processing down operation unsuccessfully.
8. */
9. Down (Bsemaphore S)
10. {
11. if ($S.value == 1$) // if a slot is available in the
12. //critical section then let the process enter in the queue.
13. {
14. $S.value = 0$; // initialize the value to 0 so that no other process can read it as 1.

```
15. }
16. else
17. {
18.     put the process (PCB) in S.L; //if no slot is available
19.     //then let the process wait in the blocked queue.
20.     sleep();
21. }
22.}
23.Up (Bsemaphore S)
24.{
25.    if (S.L is empty) //an empty blocked processes list implies that no process
26.    //has ever tried to get enter in the critical section.
27.    {
28.        S.Value = 1;
29.    }
30.    else
31.    {
32.        Select a process from S.L;
33.        Wakeup(); // if it is not empty then wake the first process of the blocked queue.
34.    }
35.}
```

#4. Deadlocks

Introduction to Deadlock

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

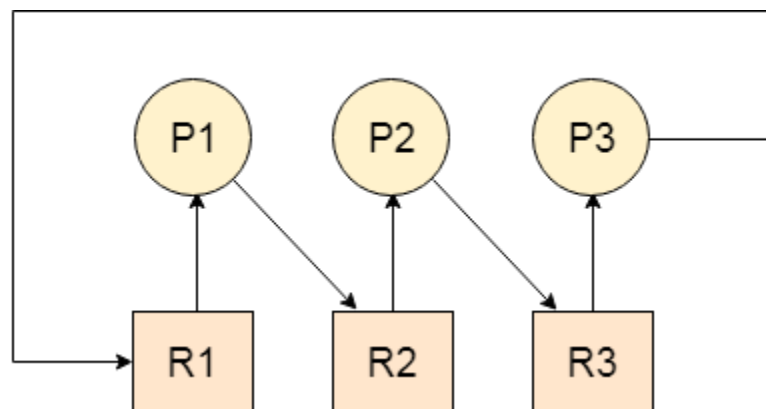
1. The process requests for some resource.
2. OS grant the resource if it is available otherwise let the process waits.
3. The process uses it and release on the completion.

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.



Difference between Starvation and Deadlock

Sr.	Deadlock	Starvation
1	Deadlock is a situation where no process got blocked and no process proceeds	Starvation is a situation where the low priority process got blocked and the high priority processes proceed.
2	Deadlock is an infinite waiting.	Starvation is a long waiting but not infinite.
3	Every Deadlock is always a starvation.	Every starvation need not be deadlock.

4	The requested resource is blocked by the other process.	The requested resource is continuously be used by the higher priority processes.
5	Deadlock happens when Mutual exclusion, hold and wait, No preemption and circular wait occurs simultaneously.	It occurs due to the uncontrolled priority and resource management.

Necessary conditions for Deadlocks

1. Mutual Exclusion

A resource can only be shared in mutually exclusive manner. It implies, if two process cannot use the same resource at the same time.

2. Hold and Wait

A process waits for some resources while holding another resource at the same time.

3. No preemption

The process which once scheduled will be executed till the completion. No other process can be scheduled by the scheduler meanwhile.

4. Circular Wait

All the processes must be waiting for the resources in a cyclic manner so that the last process is waiting for the resource which is being held by the first process.

Strategies for handling Deadlock

1. Deadlock Ignorance

Deadlock Ignorance is the most widely used approach among all the mechanism. This is being used by many operating systems mainly for end user uses. In this approach, the Operating system assumes that deadlock never occurs. It simply ignores deadlock. This approach is best suitable for a single end user system where User uses the system only for browsing and all other normal stuff.

There is always a tradeoff between Correctness and performance. The operating systems like Windows and Linux mainly focus upon performance. However, the performance of the system decreases if it uses deadlock handling mechanism all the time if deadlock happens 1 out of 100 times then it is completely unnecessary to use the deadlock handling mechanism all the time.

In these types of systems, the user has to simply restart the computer in the case of deadlock. Windows and Linux are mainly using this approach.

2. 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.

We will discuss it later in detail.

3. Deadlock avoidance

In deadlock avoidance, the operating system checks whether the system is in safe state or in unsafe state at every step which the operating system performs. The process continues until the system is in safe state. Once the system moves to unsafe state, the OS has to backtrack one step.

In simple words, The OS reviews each allocation so that the allocation doesn't cause the deadlock in the system.

We will discuss Deadlock avoidance later in detail.

4. Deadlock detection and recovery

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.

We will discuss deadlock detection and recovery later in more detail since it is a matter of discussion.

Deadlock Prevention

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.

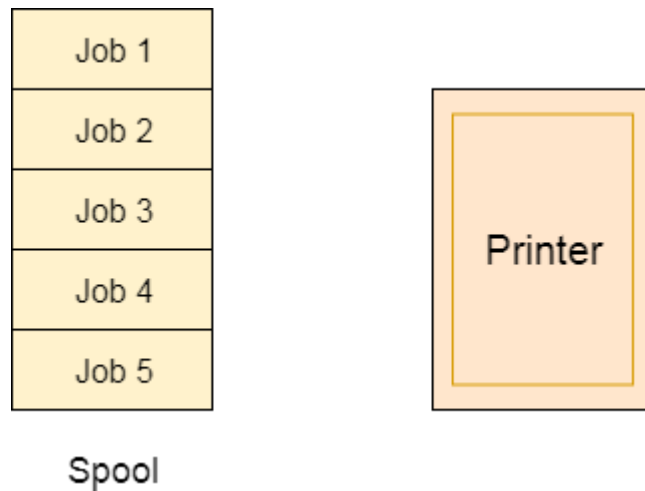
1. Mutual Exclusion

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

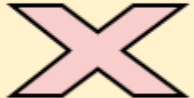



Deadlock arises due to the fact that a process can't be stopped once it starts. However, if we take the resource away from the process which is causing deadlock then we can prevent deadlock.

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

1. **E** = (7 6 8 4)
2. **P** = (6 2 8 3)
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.

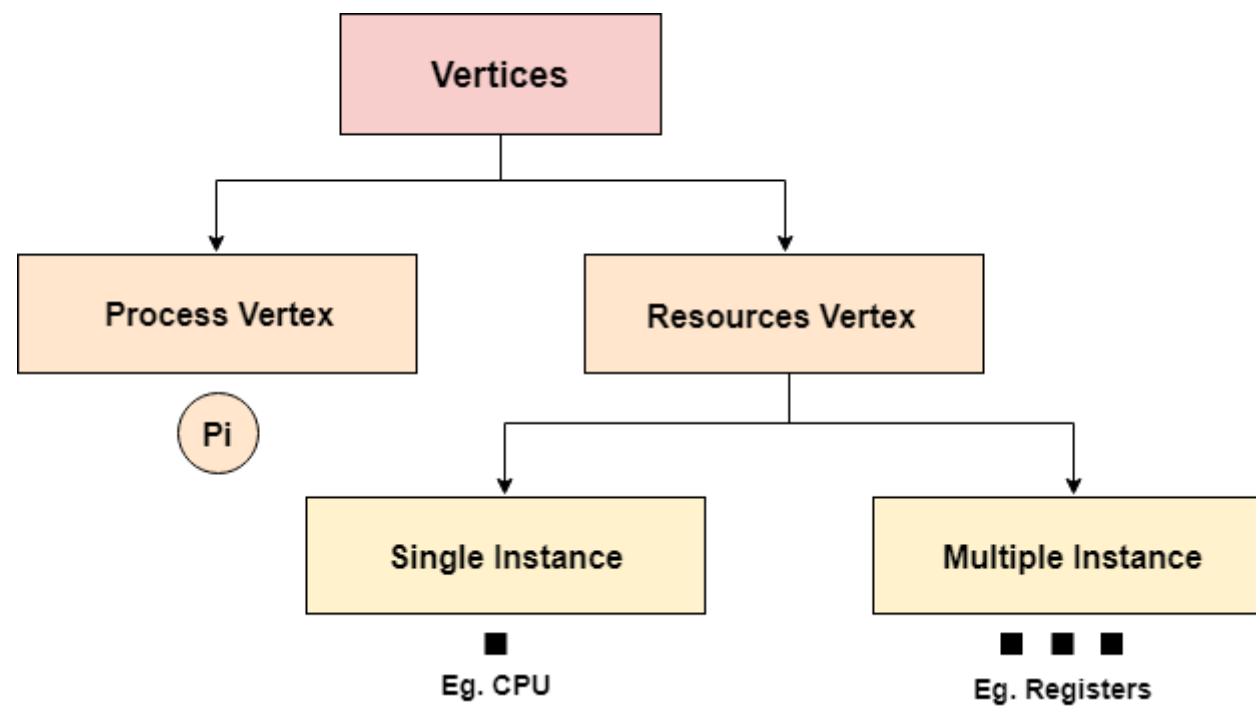
The key of Deadlock avoidance approach is when the request is made for resources then the request must only be approved in the case if the resulting state is also a safe state.

Resource Allocation Graph

The resource allocation graph is the pictorial representation of the state of a system. As its name suggests, the resource allocation graph is the complete information about all the processes which are holding some resources or waiting for some resources.

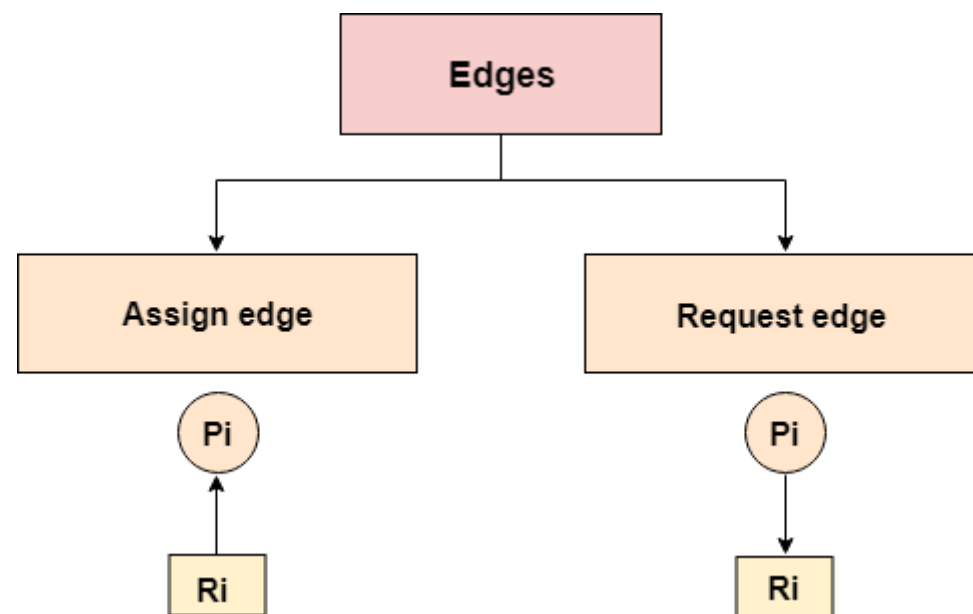
It also contains the information about all the instances of all the resources whether they are available or being used by the processes.

In Resource allocation graph, the process is represented by a Circle while the Resource is represented by a rectangle. Let's see the types of vertices and edges in detail.



Vertices are mainly of two types, Resource and process. Each of them will be represented by a different shape. Circle represents process while rectangle represents resource.

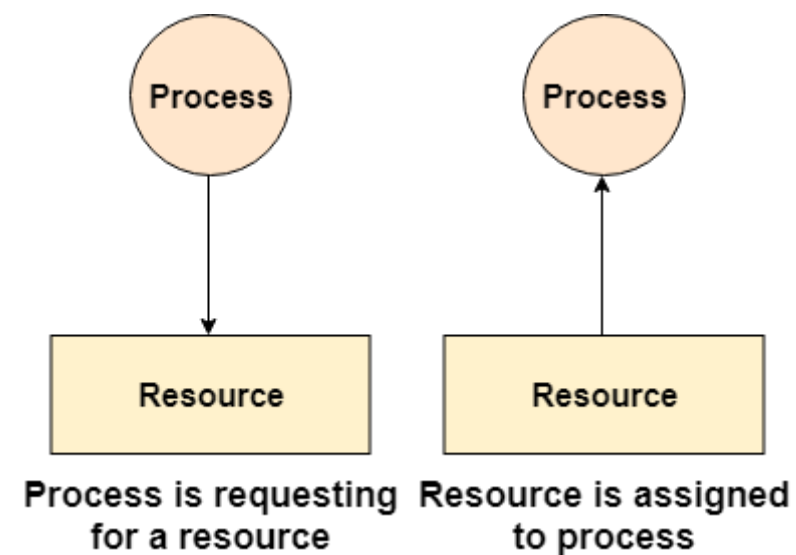
A resource can have more than one instance. Each instance will be represented by a dot inside the rectangle.



Edges in RAG are also of two types, one represents assignment and other represents the wait of a process for a resource. The above image shows each of them.

A resource is shown as assigned to a process if the tail of the arrow is attached to an instance to the resource and the head is attached to a process.

A process is shown as waiting for a resource if the tail of an arrow is attached to the process while the head is pointing towards the resource.

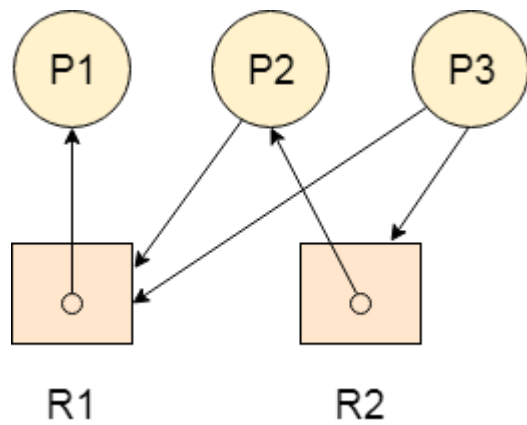


Example

Let's consider 3 processes P1, P2 and P3, and two types of resources R1 and R2. The resources are having 1 instance each.

According to the graph, R1 is being used by P1, P2 is holding R2 and waiting for R1, P3 is waiting for R1 as well as R2.

The graph is deadlock free since no cycle is being formed in the graph.

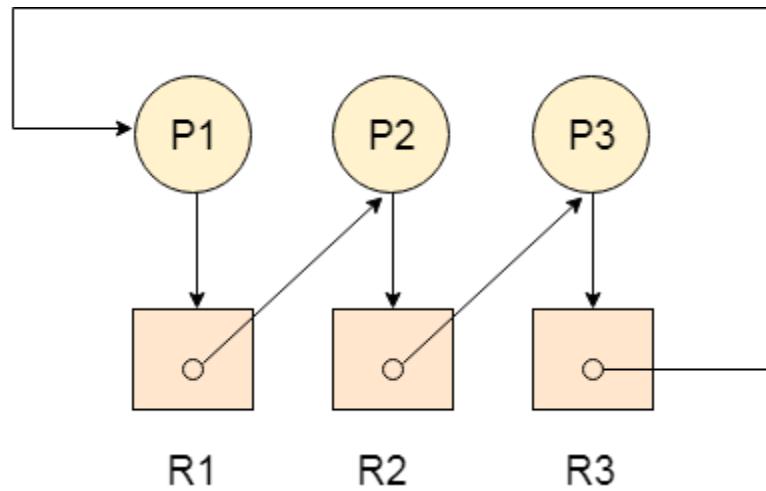


Deadlock Detection using RAG

If a cycle is being formed in a Resource allocation graph where all the resources have the single instance then the system is deadlocked.

In Case of Resource allocation graph with multi-instanced resource types, Cycle is a necessary condition of deadlock but not the sufficient condition.

The following example contains three processes P1, P2, P3 and three resources R2, R2, R3. All the resources are having single instances each.



If we analyze the graph then we can find out that there is a cycle formed in the graph since the system is satisfying all the four conditions of deadlock.

Allocation Matrix

Allocation matrix can be formed by using the Resource allocation graph of a system. In Allocation matrix, an entry will be made for each of the resource assigned. For Example, in the following matrix, an entry is being made in front of P1 and below R3 since R3 is assigned to P1.

Process	R1	R2	R3
P1	0	0	1
P2	1	0	0
P3	0	1	0

Request Matrix

In request matrix, an entry will be made for each of the resource requested. As in the following example, P1 needs R1 therefore an entry is being made in front of P1 and below R1.

Process	R1	R2	R3
P1	1	0	0
P2	0	1	0
P3	0	0	1

Avial = (0,0,0)

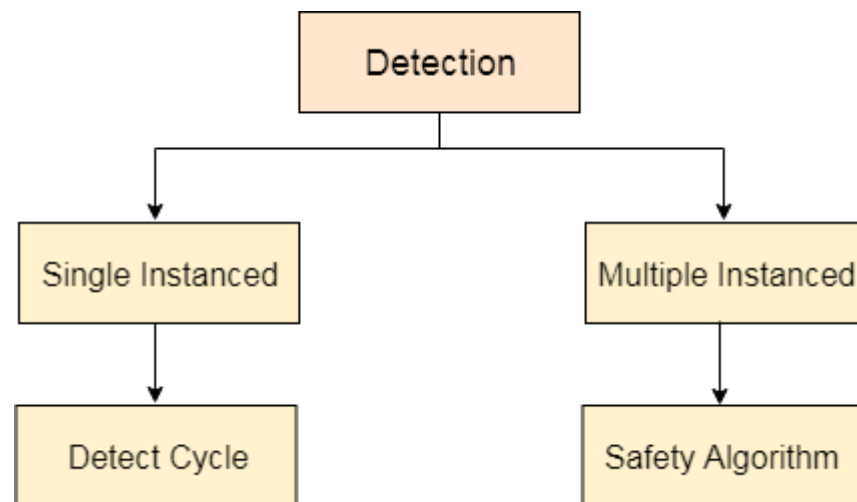
Neither we are having any resource available in the system nor a process going to release. Each of the process needs at least single resource to complete therefore they will continuously be holding each one of them.

We cannot fulfill the demand of at least one process using the available resources therefore the system is deadlocked as determined earlier when we detected a cycle in the graph.

Deadlock Detection and Recovery

In this approach, The OS doesn't apply any mechanism to avoid or prevent the deadlocks. Therefore the system considers that the deadlock will definitely occur. In order to get rid of deadlocks, The OS periodically checks the system for any deadlock. In case, it finds any of the deadlock then the OS will recover the system using some recovery techniques.

The main task of the OS is detecting the deadlocks. The OS can detect the deadlocks with the help of Resource allocation graph.



In single instanced resource types, if a cycle is being formed in the system then there will definitely be a deadlock. On the other hand, in multiple instanced resource type graph, detecting a cycle is not just enough. We have to apply the safety algorithm on the system by converting the resource allocation graph into the allocation matrix and request matrix.

In order to recover the system from deadlocks, either OS considers resources or processes.

For Resource

Preempt the resource

We can snatch one of the resources from the owner of the resource (process) and give it to the other process with the expectation that it will complete the execution and will release this resource sooner. Well, choosing a resource which will be snatched is going to be a bit difficult.

Rollback to a safe state

System passes through various states to get into the deadlock state. The operating system can rollback the system to the previous safe state. For this purpose, OS needs to implement check pointing at every state.

The moment, we get into deadlock, we will rollback all the allocations to get into the previous safe state.

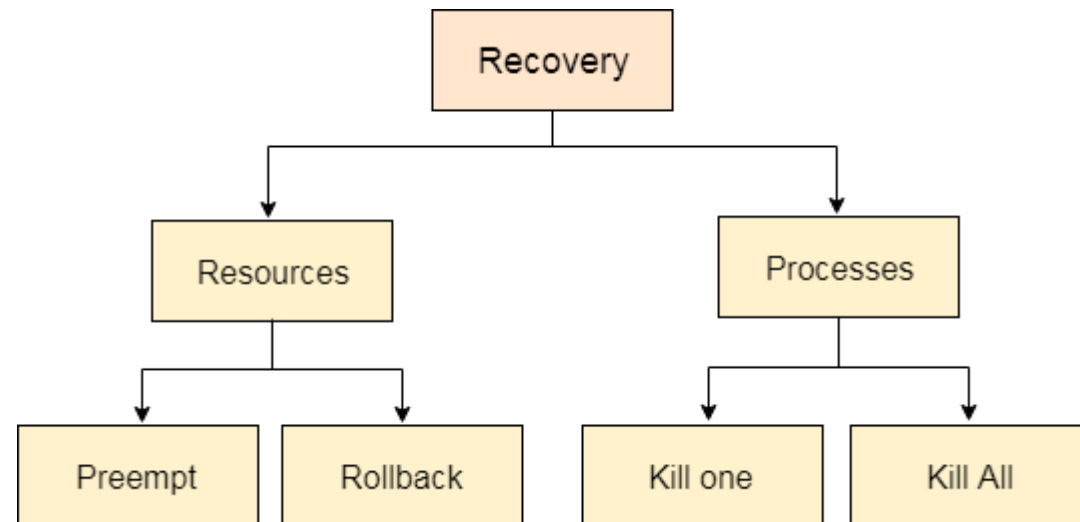
For Process

Kill a process

Killing a process can solve our problem but the bigger concern is to decide which process to kill. Generally, Operating system kills a process which has done least amount of work until now.

Kill all process

This is not a suggestible approach but can be implemented if the problem becomes very serious. Killing all process will lead to inefficiency in the system because all the processes will execute again from starting.



#5.Memory Management

What is 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. We will discuss each one of them later in detail.

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

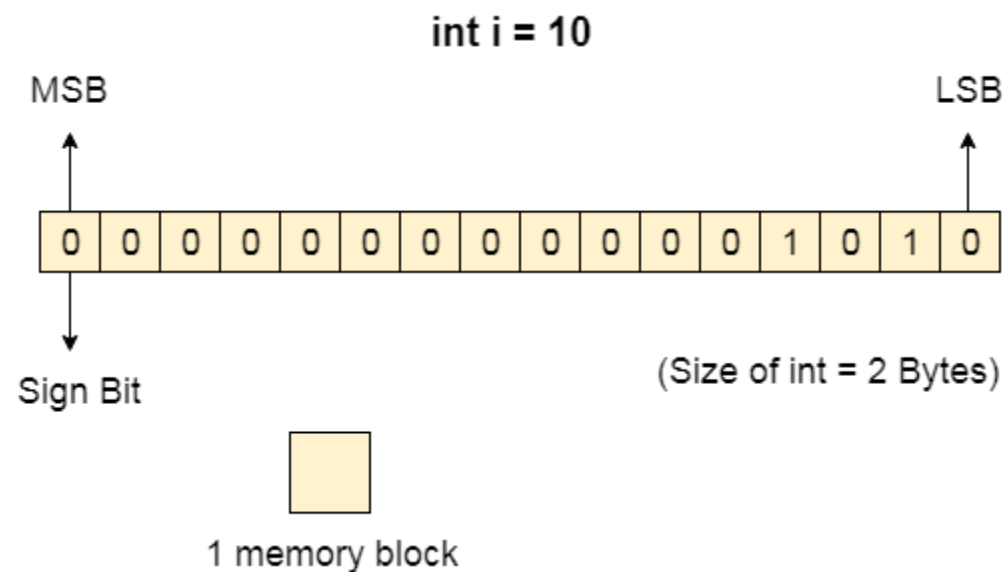
How Data is being stored in a computer system?

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.

Need for Multi programming

However, The CPU can directly access the main memory, Registers and cache of the system. The program always executes in main memory. The size of main memory affects degree of Multi programming to most of the extent. If the size of the main memory is larger than CPU can load more processes in the main memory at the same time and therefore will increase degree of Multi programming as well as CPU utilization.

1. Let's consider,
2. Process **Size** = 4 MB
3. Main memory **size** = 4 MB
4. The process can only reside in the main memory at any time.
5. If the time for which the process does IO is P,
- 6.
7. Then,
- 8.
9. CPU **utilization** = $(1-P)$
10. let's say,
11. **P** = 70%
12. CPU **utilization** = 30 %
13. Now, increase the memory size, Let's say it is 8 MB.
14. Process **Size** = 4 MB
15. Two processes can reside in the main memory at the same time.
16. Let's say the time for which, one process does its IO is P,
- 17.
18. Then
- 19.
20. CPU **utilization** = $(1-P^2)$
21. let's say **P** = 70 %
22. CPU **utilization** = $(1-0.49) = 0.51 = 51 \%$

Therefore, we can state that the CPU utilization will be increased if the memory size gets increased.

Fixed Partitioning

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 fixed partitioning,

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 than 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.

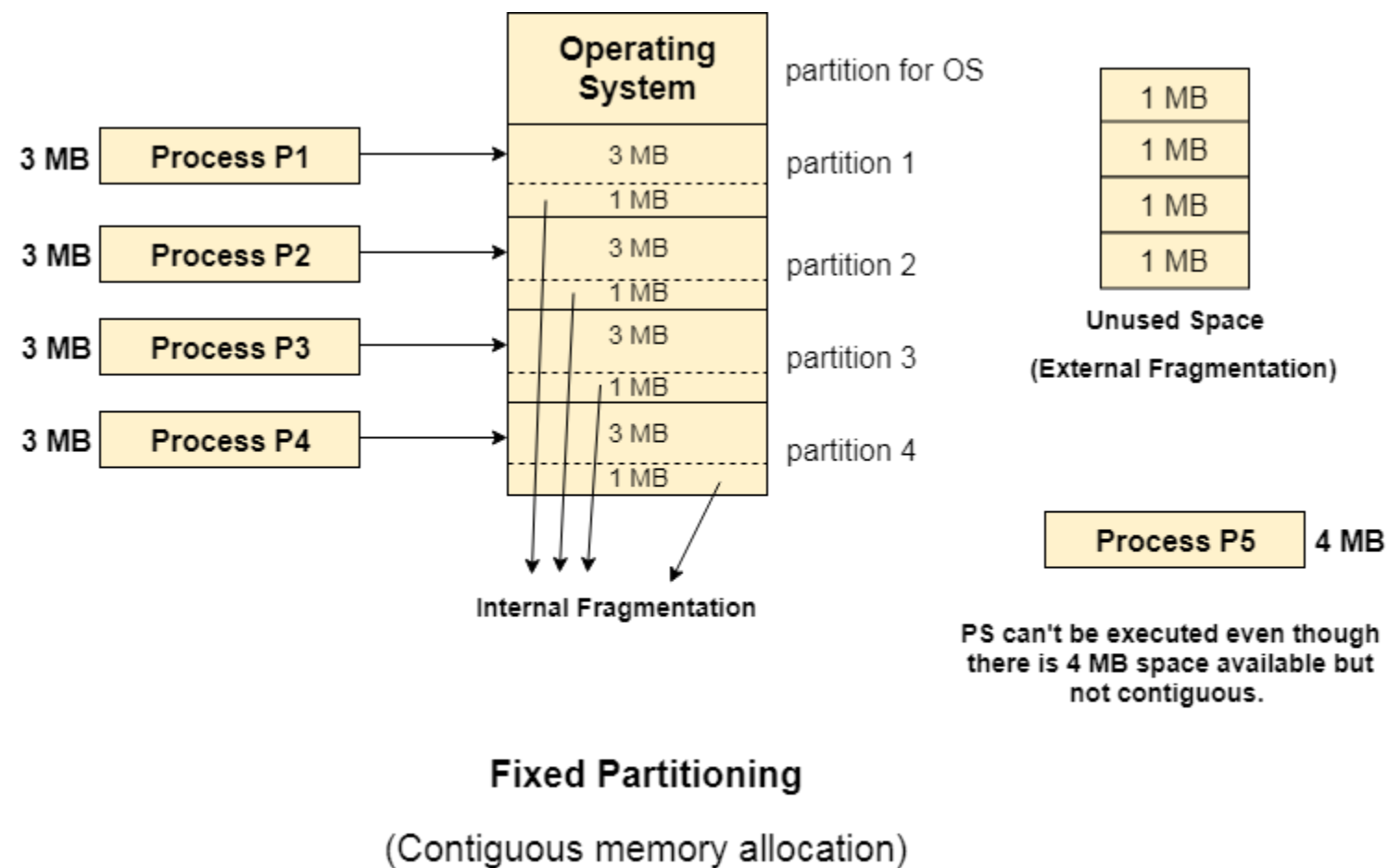
As shown in the image below, the remaining 1 MB space of each partition cannot be used as a unit to store a 4 MB process. Despite of the fact that the sufficient space is available to load the process, process will not be loaded.

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.

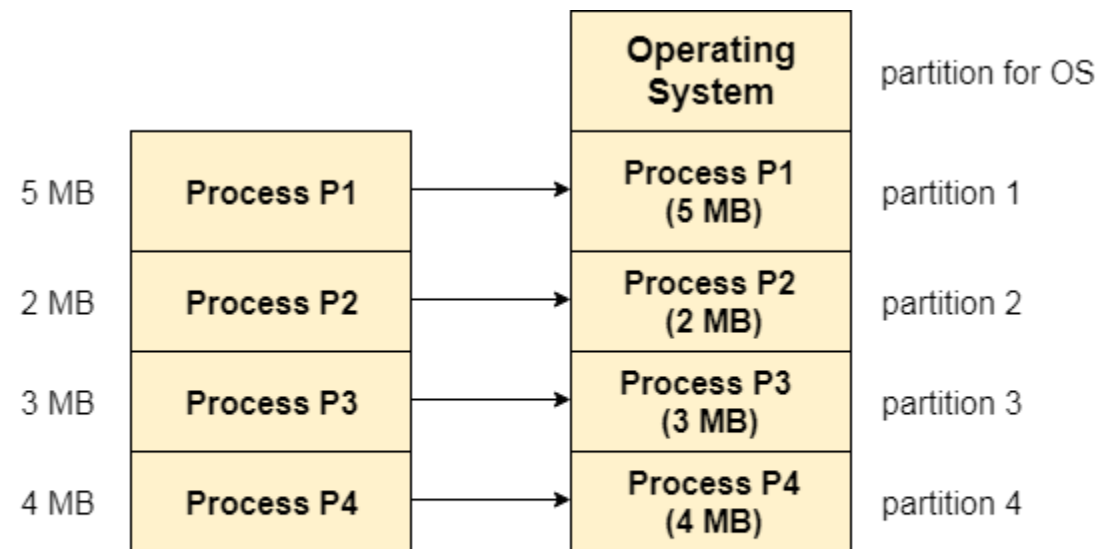


[Next →](#) [← Prev](#)

Dynamic Partitioning

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.



Dynamic Partitioning
(Process Size = Partition Size)

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

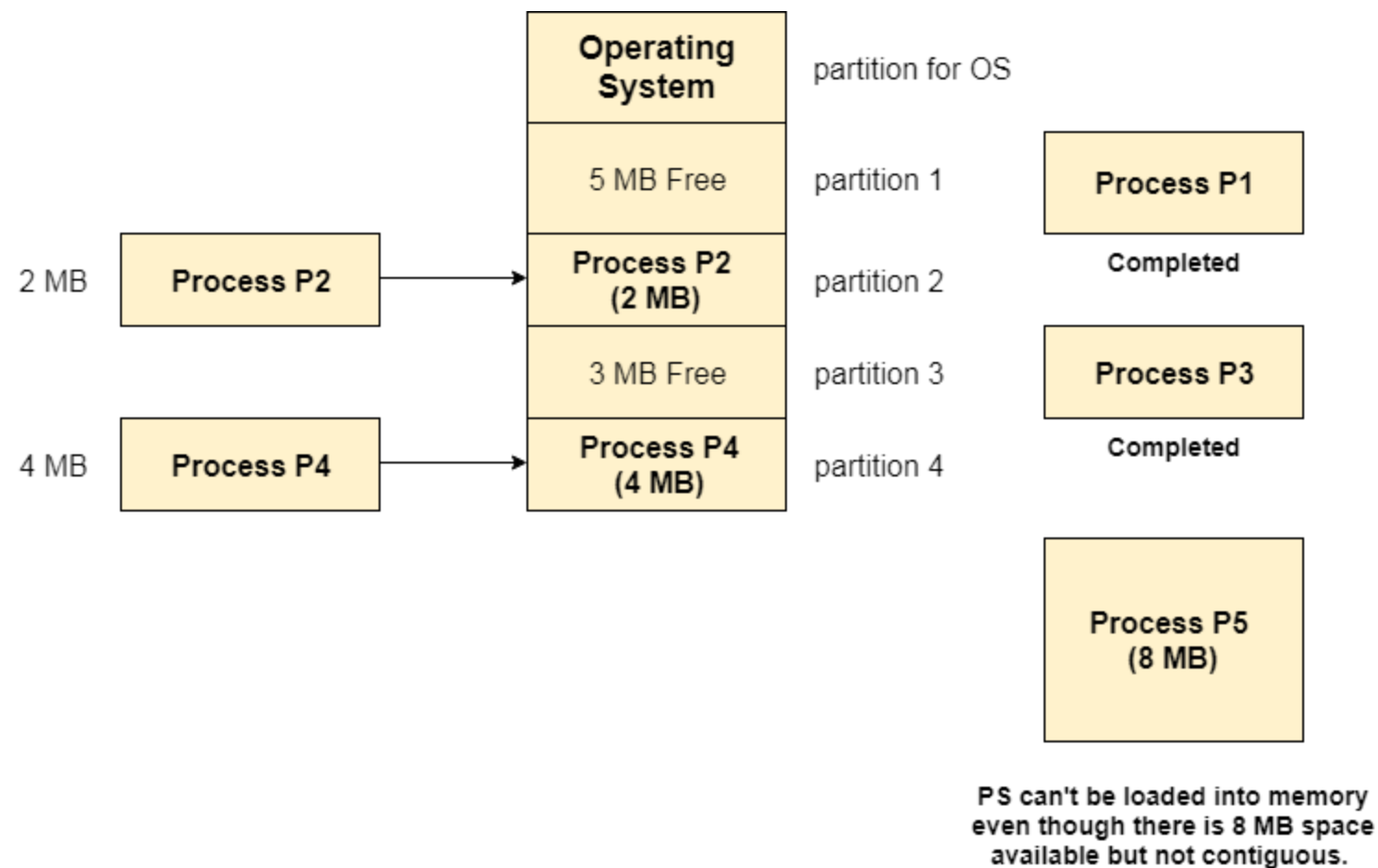
External Fragmentation

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.



External Fragmentation in Dynamic Partitioning

X

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.

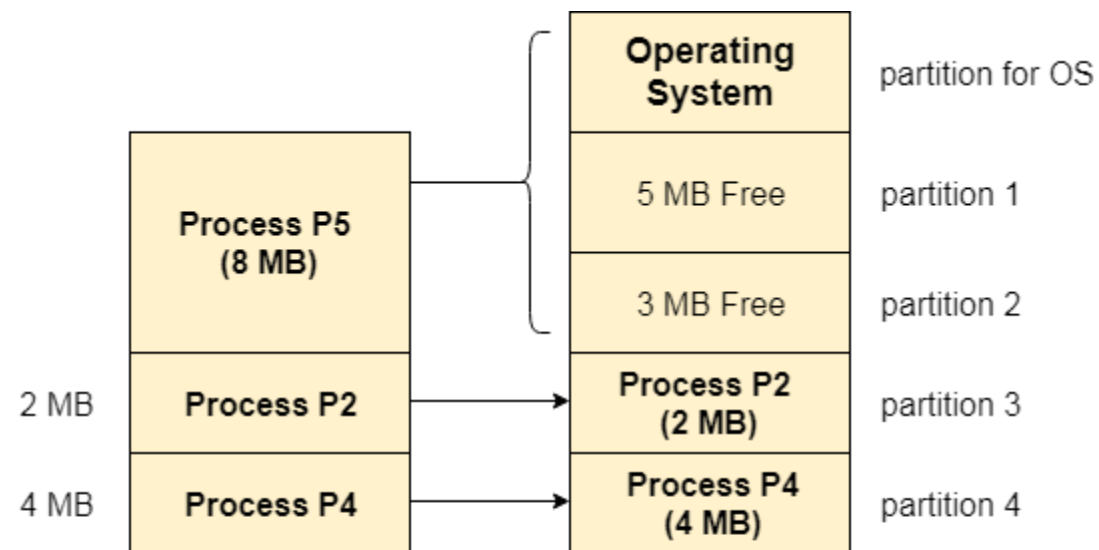
Compaction

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.



Now P5 can be loaded into memory
because the free space is now made
contiguous by compaction

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. 1 B transfer needs 6 NS
2. 256 MB transfer needs $256 \times 2^{20} \times 6 \times 10^{-9}$ secs

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

Bit Map for Dynamic Partitioning

The Main concern for dynamic partitioning is keeping track of all the free and allocated partitions. However, the Operating system uses following data structures for this task.

1. Bit Map
2. Linked List

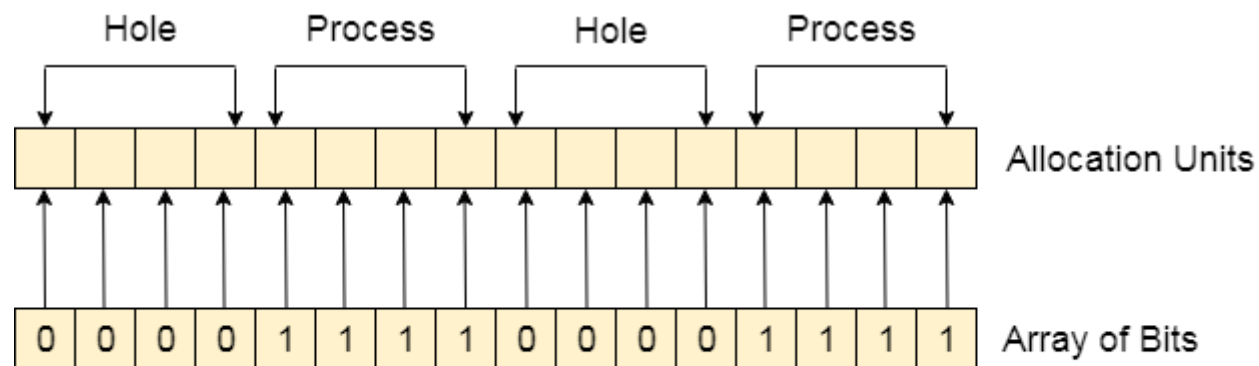
Bit Map is the least famous data structure to store the details. In this scheme, the main memory is divided into the collection of allocation units. One or more allocation units may be allocated to a process according to the need of that process. However, the size of the allocation unit is fixed that is defined by the Operating System and never changed. Although the partition size may vary but the allocation size is fixed.

The main task of the operating system is to keep track of whether the partition is free or filled. For this purpose, the operating system also manages another data structure that is called bitmap.

The process or the hole in Allocation units is represented by a flag bit of bitmap. In the image shown below, a flag bit is defined for every bit of allocation units. However, it is not the general case, it depends on the OS that, for how many bits of the allocation units, it wants to store the flag bit.

The flag bit is set to 1 if there is a contiguously present process at the adjacent bit in allocation unit otherwise it is set to 0.

A string of 0s in the bitmap shows that there is a hole in the relative Allocation unit while the string of 1s represents the process in the relative allocation unit.



1 Allocation Unit = 1/2 byte

1 Bit of Bit Map \longrightarrow 1 Bit of Allocation Unit

1/5 of the total memory is taken by Bit Map

0 \longrightarrow Hole

1 \longrightarrow Process

Bit Map for Dynamic Partitioning

Disadvantages of using Bitmap

1. The OS has to assign some memory for bitmap as well since it stores the details about allocation units. That much amount of memory cannot be used to load any process therefore that decreases the degree of multiprogramming as well as throughput.

In the above image,

The allocation unit is of 4 bits that is 0.5 bytes. Here, 1 bit of the bitmap is representing 1 bit of allocation unit.

1. Size of 1 allocation unit = 4 bits

2. Size of **bitmap** = $1/(4+1) = 1/5$ of total main memory.

Therefore, in this bitmap configuration, $1/5$ of total main memory is wasted.

2. To identify any hole in the memory, the OS need to search the string of 0s in the bitmap. This searching takes a huge amount of time which makes the system inefficient to some extent

[Next →](#) [← Prev](#)

Linked List for Dynamic Partitioning

The better and the most popular approach to keep track the free or filled partitions is using Linked List.

In this approach, the Operating system maintains a linked list where each node represents each partition. Every node has three fields.

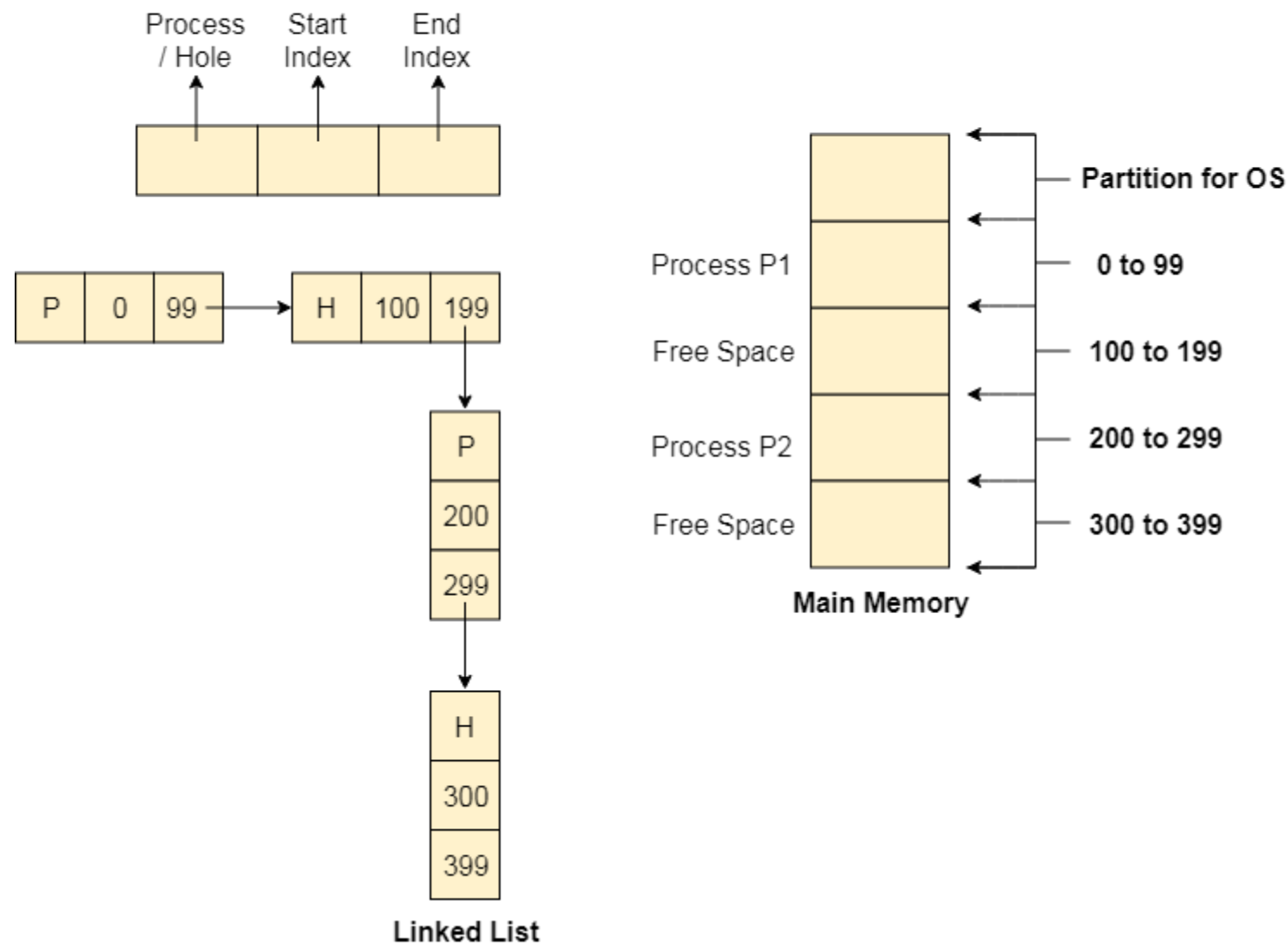
1. First field of the node stores a flag bit which shows whether the partition is a hole or some process is inside.
2. Second field stores the starting index of the partition.
3. Third field stores the end index of the partition.

If a partition is freed at some point of time then that partition will be merged with its adjacent free partition without doing any extra effort.

There are some points which need to be focused while using this approach.

1. The OS must be very clear about the location of the new node which is to be added in the linked list. However, adding the node according to the increasing order of starting index is suggestible.
2. Using a doubly linked list will make some positive effects on the performance due to the fact that a node in the doubly link list can also keep track of its previous node.

Linked List for Dynamic Partitioning



Partitioning Algorithms

There are various algorithms which are implemented by the Operating System in order to find out the holes in the linked list and allocate them to the processes.

The explanation about each of the algorithm is given below.

1. First Fit Algorithm

First Fit algorithm scans the linked list and whenever it finds the first big enough hole to store a process, it stops scanning and load the process into that hole. This procedure produces two partitions. Out of them, one partition will be a hole while the other partition will store the process.

First Fit algorithm maintains the linked list according to the increasing order of starting index. This is the simplest to implement among all the algorithms and produces bigger holes as compare to the other algorithms.

2. Next Fit Algorithm

Next Fit algorithm is similar to First Fit algorithm except the fact that, Next fit scans the linked list from the node where it previously allocated a hole.

Next fit doesn't scan the whole list, it starts scanning the list from the next node. The idea behind the next fit is the fact that the list has been scanned once therefore the probability of finding the hole is larger in the remaining part of the list.

Experiments over the algorithm have shown that the next fit is not better then the first fit. So it is not being used these days in most of the cases.

3. Best Fit Algorithm

The Best Fit algorithm tries to find out the smallest hole possible in the list that can accommodate the size requirement of the process.

Using Best Fit has some disadvantages.

1. 1. It is slower because it scans the entire list every time and tries to find out the smallest hole which can satisfy the requirement the process.
2. Due to the fact that the difference between the whole size and the process size is very small, the holes produced will be as small as it cannot be used to load any process and therefore it remains useless.
Despite of the fact that the name of the algorithm is best fit, It is not the best algorithm among all.
- 3.

4. Worst Fit Algorithm

The worst fit algorithm scans the entire list every time and tries to find out the biggest hole in the list which can fulfill the requirement of the process.

Despite of the fact that this algorithm produces the larger holes to load the other processes, this is not the better approach due to the fact that it is slower because it searches the entire list every time again and again.

5. Quick Fit Algorithm

The quick fit algorithm suggests maintaining the different lists of frequently used sizes. Although, it is not practically suggestible because the procedure takes so much time to create the different lists and then expending the holes to load a process.

The first fit algorithm is **the best algorithm** among all because

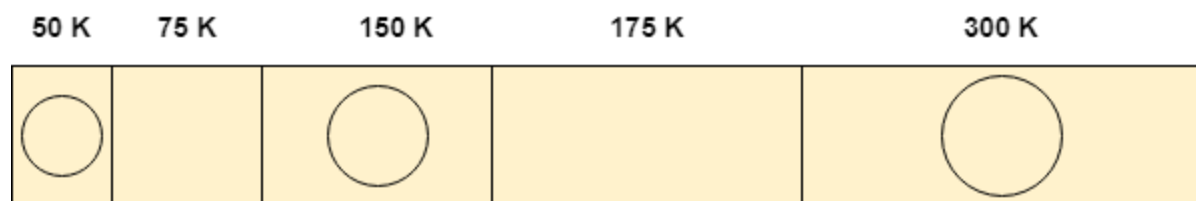
1. It takes lesser time compare to the other algorithms.
2. It produces bigger holes that can be used to load other processes later on.
3. It is easiest to implement.

GATE question on best fit and first fit

From the GATE point of view, Numerical on best fit and first fit are being asked frequently in 1 mark. Let's have a look on the one given as below.

Q. Process requests are given as;

25 K , 50 K , 100 K , 75 K



Determine the algorithm which can optimally satisfy this requirement.

1. First Fit algorithm
2. Best Fit Algorithm
3. Neither of the two
4. Both of them

In the question, there are five partitions in the memory. 3 partitions are having processes inside them and two partitions are holes.

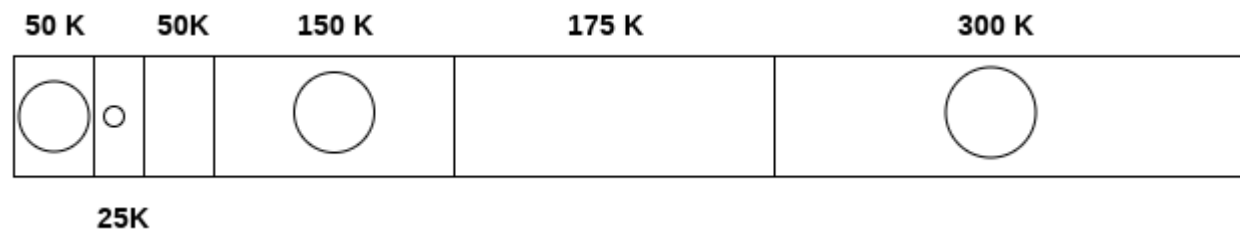
Our task is to check the algorithm which can satisfy the request optimally.

Using First Fit algorithm

Let's see, how first fit algorithm works on this problem.

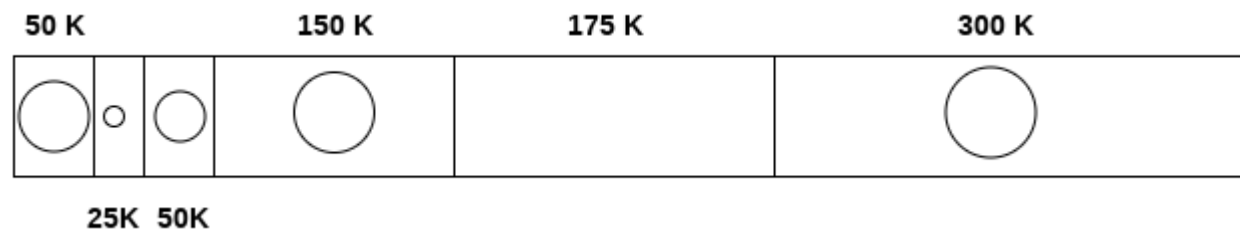
1. 25 K requirement

The algorithm scans the list until it gets first hole which should be big enough to satisfy the request of 25 K. it gets the space in the second partition which is free hence it allocates 25 K out of 75 K to the process and the remaining 50 K is produced as hole.



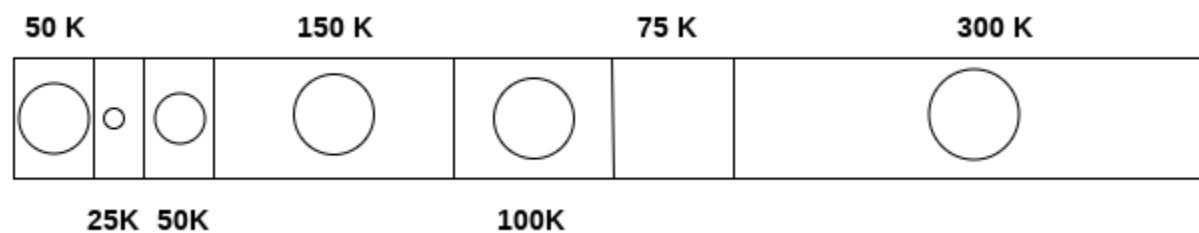
2. 50 K requirement

The 50 K requirement can be fulfilled by allocating the third partition which is 50 K in size to the process. No free space is produced as free space.



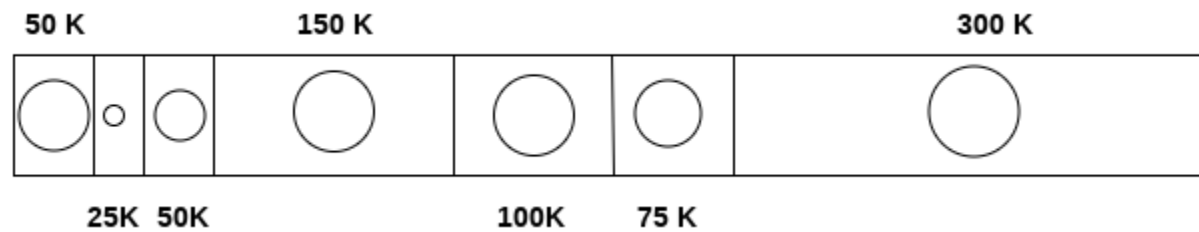
3. 100 K requirement

100 K requirement can be fulfilled by using the fifth partition of 175 K size. Out of 175 K, 100 K will be allocated and remaining 75 K will be there as a hole.



4. 75 K requirement

Since we are having a 75 K free partition hence we can allocate that much space to the process which is demanding just 75 K space.



Using first fit algorithm, we have fulfilled the entire request optimally and no useless space is remaining.

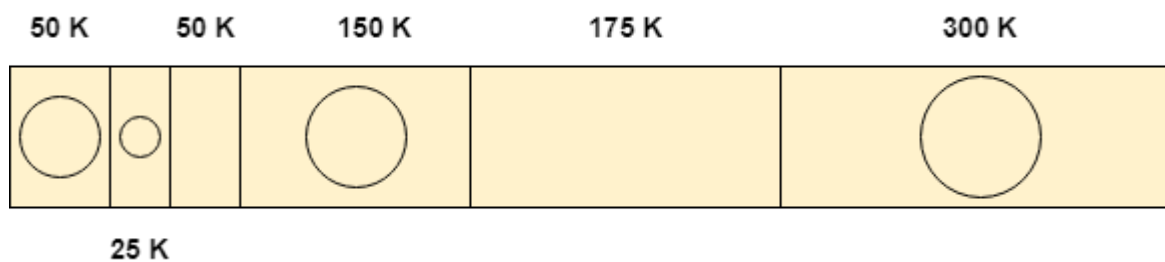
Let's see, How Best Fit algorithm performs for the problem.

Using Best Fit Algorithm

1. 25 K requirement

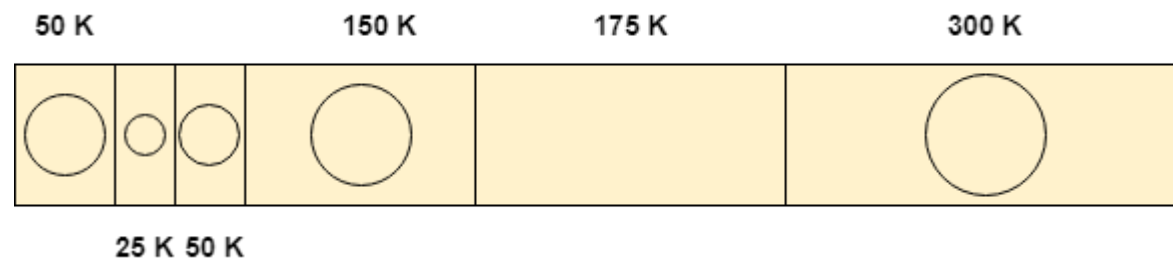
To allocate 25 K space using best fit approach, need to scan the whole list and then we find that a 75 K partition is free and the smallest among all, which can accommodate the need of the process.

Therefore 25 K out of those 75 K free partition is allocated to the process and the remaining 50 K is produced as a hole.



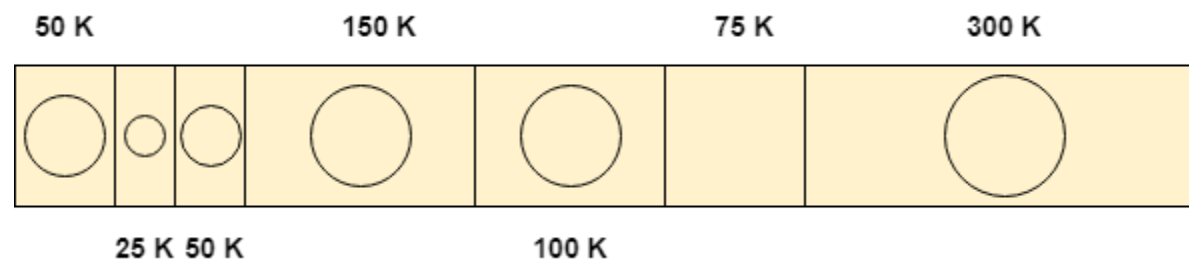
2. 50 K requirement

To satisfy this need, we will again scan the whole list and then find the 50 K space is free which the exact match of the need is. Therefore, it will be allocated for the process.



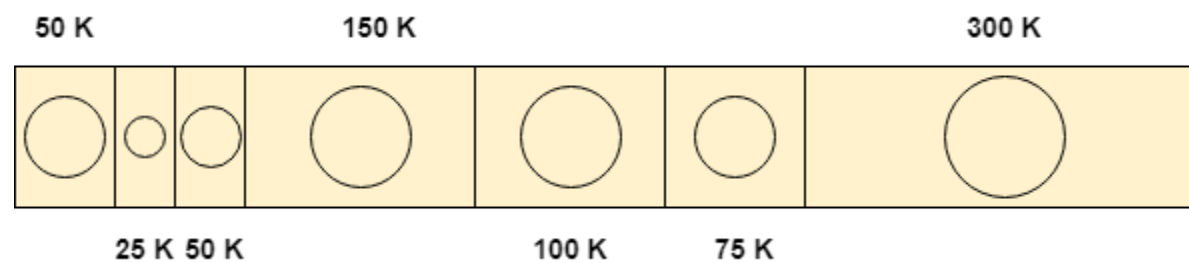
3. 100 K requirement

100 K need is close enough to the 175 K space. The algorithm scans the whole list and then allocates 100 K out of 175 K from the 5th free partition.



4. 75 K requirement

75 K requirement will get the space of 75 K from the 6th free partition but the algorithm will scan the whole list in the process of taking this decision.



By following both of the algorithms, we have noticed that both the algorithms perform similar to most of the extant in this case.

Both can satisfy the need of the processes but however, the best fit algorithm scans the list again and again which takes lot of time.

Therefore, if you ask me that which algorithm performs in more optimal way then it will be **First Fit algorithm** for sure.

Therefore, the answer in this case is A.

Need for Paging

Disadvantage of Dynamic Partitioning

The main disadvantage of Dynamic Partitioning is External fragmentation. Although, this can be removed by Compaction but as we have discussed earlier, the compaction makes the system inefficient.

We need to find out a mechanism which can load the processes in the partitions in a more optimal way. Let us discuss a dynamic and flexible mechanism called paging.

Need for Paging

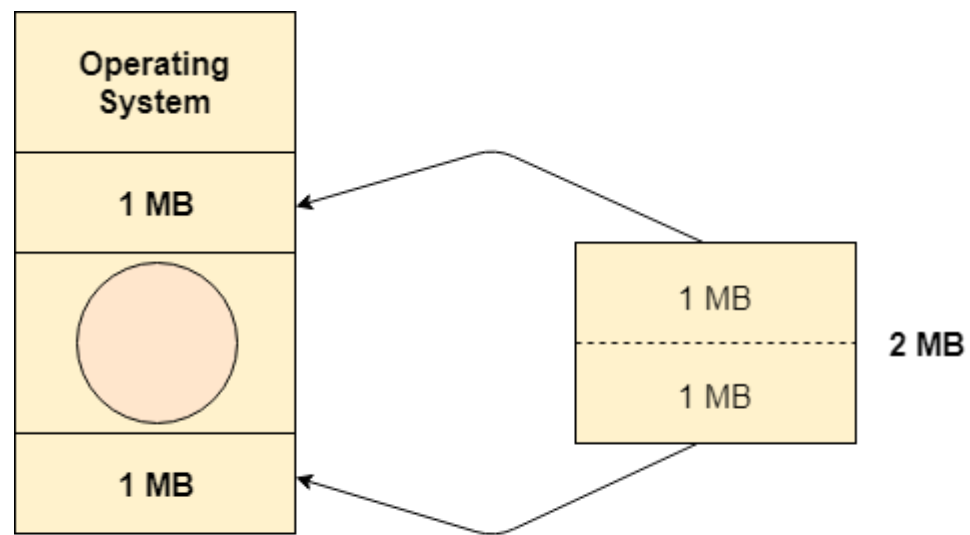
Lets consider a process P1 of size 2 MB and the main memory which is divided into three partitions. Out of the three partitions, two partitions are holes of size 1 MB each.

P1 needs 2 MB space in the main memory to be loaded. We have two holes of 1 MB each but they are not contiguous.

Although, there is 2 MB space available in the main memory in the form of those holes but that remains useless until it become contiguous. This is a serious problem to address.

We need to have some kind of mechanism which can store one process at different locations of the memory.

The Idea behind paging is to divide the process in pages so that, we can store them in the memory at different holes. We will discuss paging with the examples in the next sections.



The process needs to be divided into two parts to get stored at two different places.

[Next →](#) [← Prev](#)

Paging with Example

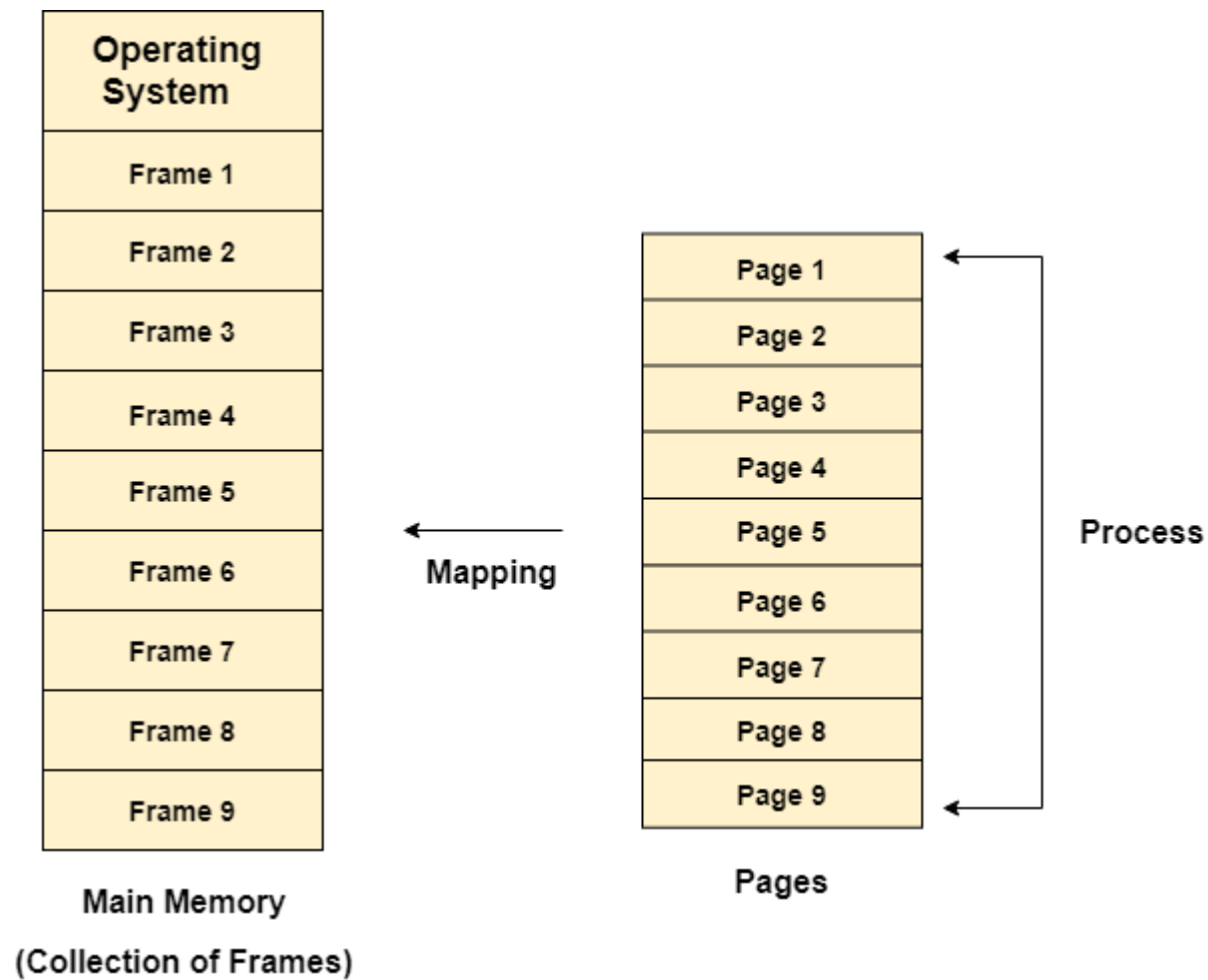
In Operating Systems, Paging is a storage mechanism used to retrieve processes from the secondary storage into the main memory in the form of pages.

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.



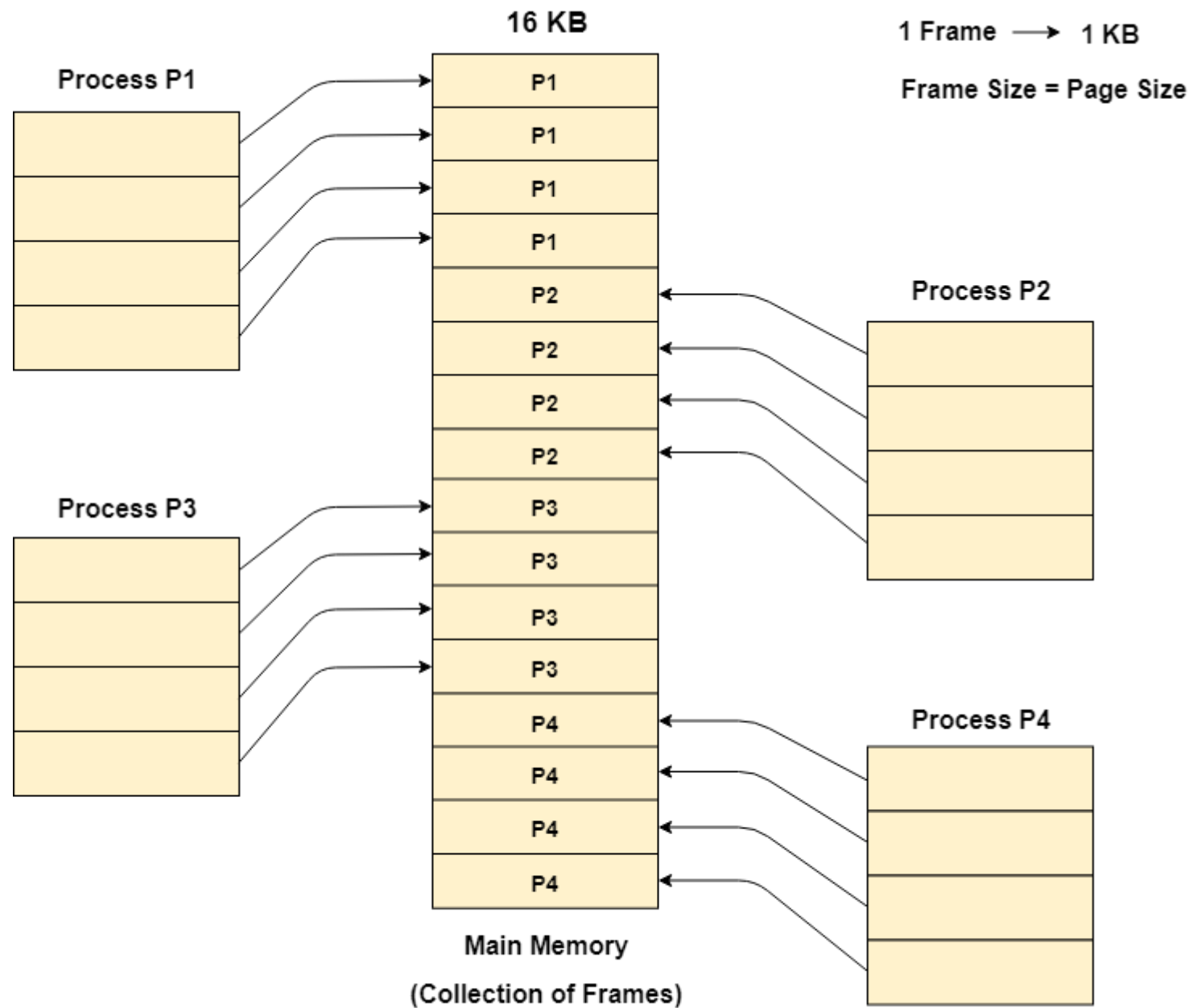
Example

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.

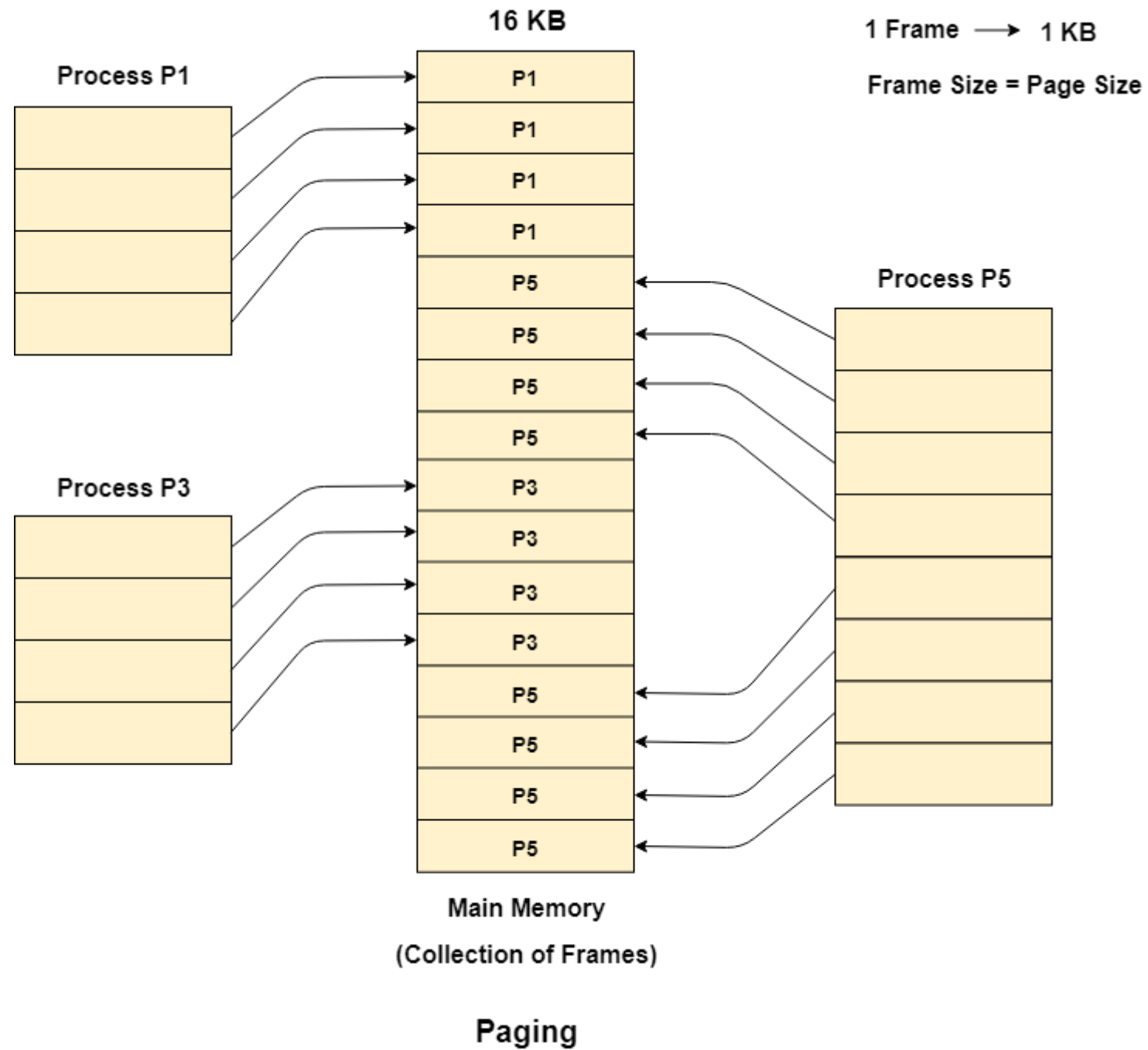
Frames, pages and the mapping between the two is shown in the image below.



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.



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.

Basics of Binary Addresses

Computer system assigns the binary addresses to the memory locations. However, The system uses amount of bits to address a memory location.

Using 1 bit, we can address two memory locations. Using 2 bits we can address 4 and using 3 bits we can address 8 memory locations.

A pattern can be identified in the mapping between the number of bits in the address and the range of the memory locations.

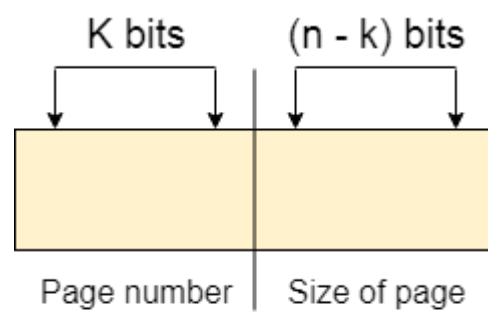
We know,

1. Using 1 Bit we can represent 2^1 i.e 2 memory locations.
2. Using 2 bits, we can represent 2^2 i.e. 4 memory locations.
3. Using 3 bits, we can represent 2^3 i.e. 8 memory locations.
4. Therefore, if we generalize this,
5. Using n bits, we can assign 2^n memory locations.
- 6.

7. n bits of address $\rightarrow 2^n$ memory locations

1 Bit	2 Bits	3 Bits
0	0 0	0 0 0
1	0 1	0 0 1
	1 0	0 1 0
	1 1	0 1 1
		1 0 0
		1 0 1
		1 1 0
		1 1 1

these n bits can be divided into two parts, that are, **K** bits and **(n-k)** bits.



$$2^n = 2^k \times 2^{n-k}$$

[Next](#) \rightarrow [Prev](#)

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 = $2^6 \times 2^{10}$ 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 = $(2^7 \times 2^{20})$ 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 = $\log_2 L$ 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.

[Next →](#) [← Prev](#)

Page Table

Page Table is a data structure used by the virtual memory system to store the mapping between logical addresses and physical addresses.

Logical addresses are generated by the CPU for the pages of the processes therefore they are generally used by the processes.

Physical addresses are the actual frame address of the memory. They are generally used by the hardware or more specifically by RAM subsystems.

The image given below considers,

Physical Address Space = M words

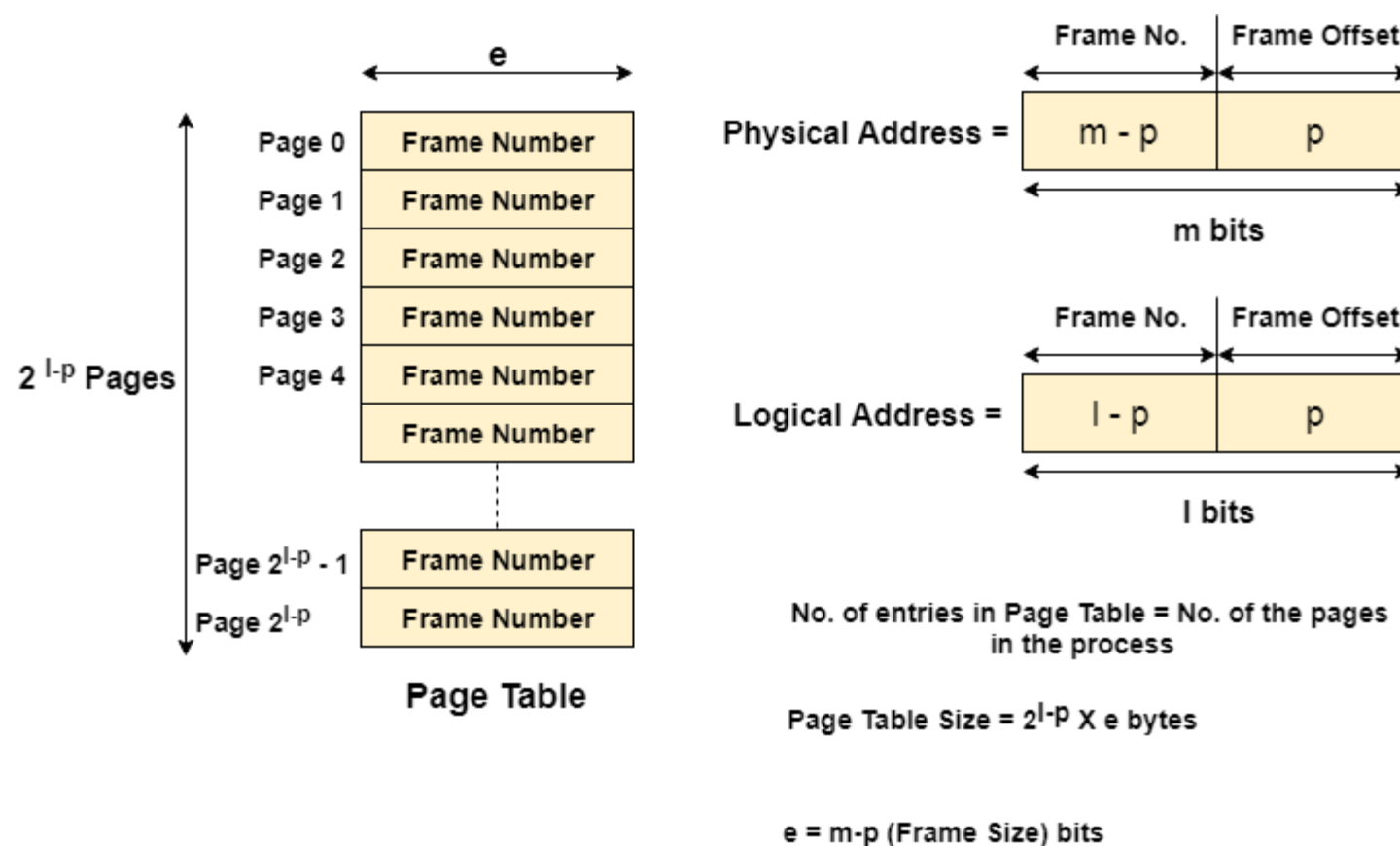
Logical Address Space = L words

Page Size = P words

Physical Address = $\log_2 M = m$ bits

Logical Address = $\log_2 L = l$ bits

page offset = $\log_2 P = p$ bits



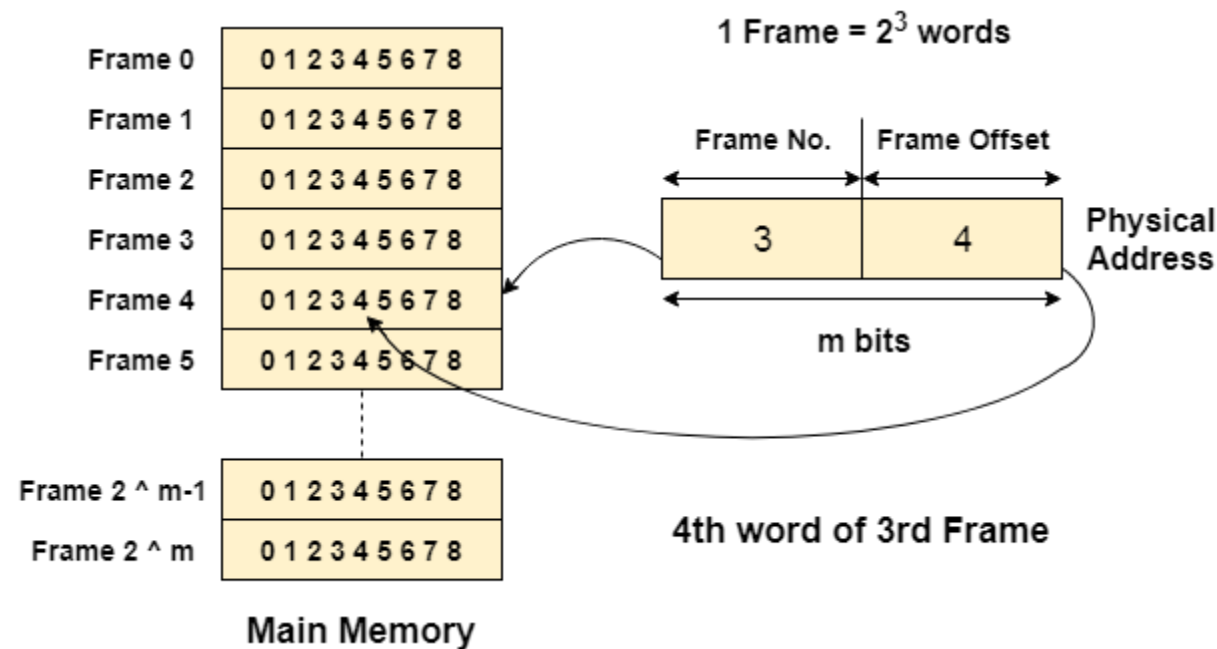
The CPU always accesses the processes through their logical addresses. However, the main memory recognizes physical address only.

In this situation, a unit named as Memory Management Unit comes into the picture. It converts the page number of the logical address to the frame number of the physical address. The offset remains same in both the addresses.

To perform this task, Memory Management unit needs a special kind of mapping which is done by page table. The page table stores all the Frame numbers corresponding to the page numbers of the page table.

In other words, the page table maps the page number to its actual location (frame number) in the memory.

In the image given below shows, how the required word of the frame is accessed with the help of offset.



Mapping from page table to main memory

In operating systems, there is always a requirement of mapping from logical address to the physical address. However, this process involves various steps which are defined as follows.

1. Generation of logical address

CPU generates logical address for each page of the process. This contains two parts: page number and offset.

2. Scaling

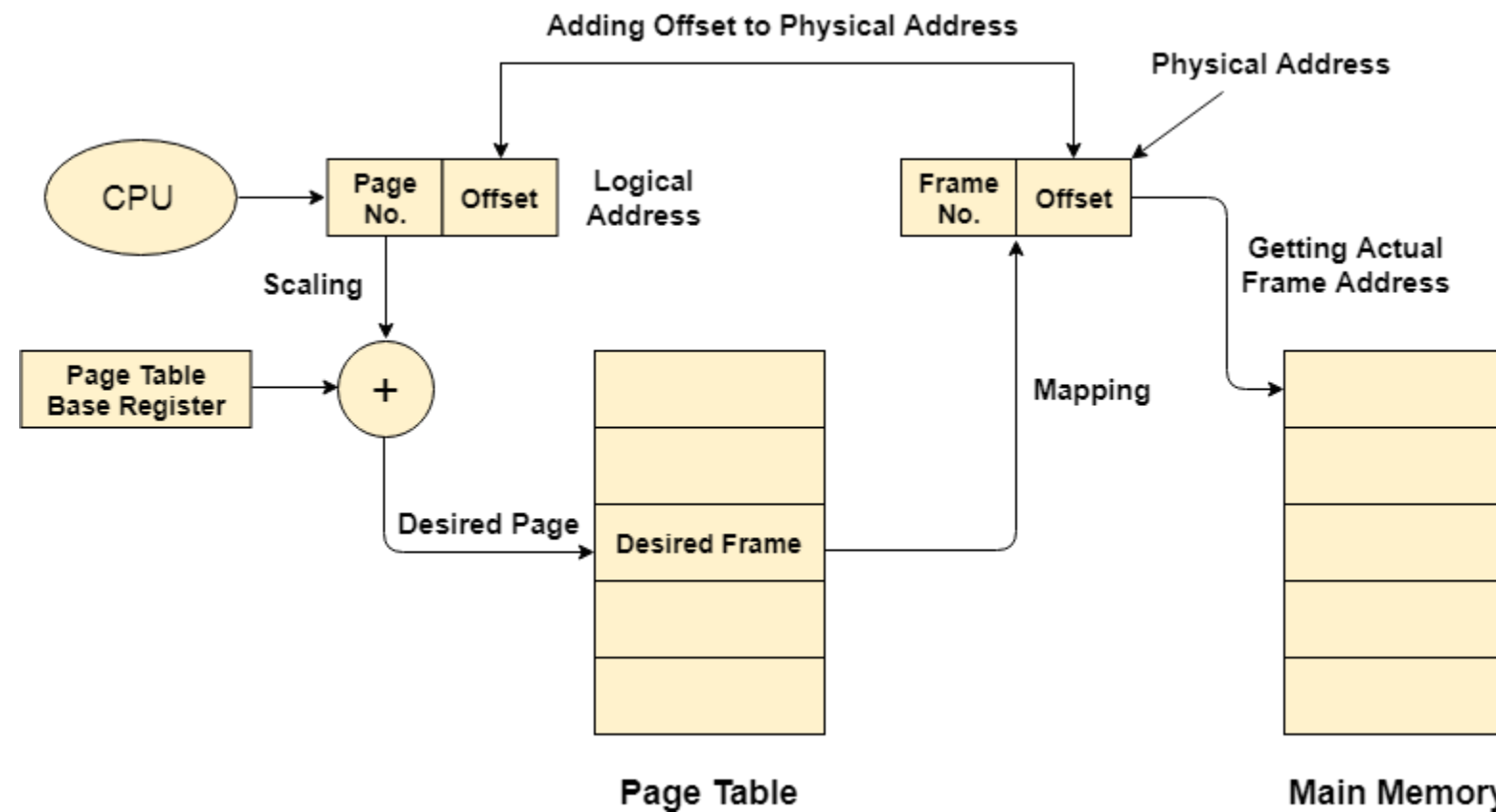
To determine the actual page number of the process, CPU stores the page table base in a special register. Each time the address is generated, the value of the page table base is added to the page number to get the actual location of the page entry in the table. This process is called scaling.

3. Generation of physical Address

The frame number of the desired page is determined by its entry in the page table. A physical address is generated which also contains two parts : frame number and offset. The Offset will be similar to the offset of the logical address therefore it will be copied from the logical address.

4. Getting Actual Frame Number

The frame number and the offset from the physical address is mapped to the main memory in order to get the actual word address.



Page Table Entry

Along with page frame number, the page table also contains some of the bits representing the extra information regarding the page.

Let's see what the each bit represents about the page.

1. Caching Disabled

Sometimes, there are differences between the information closest to the CPU and the information closest to the user. Operating system always wants CPU to access user's data as soon as possible. CPU accesses cache which can be inaccurate in some of the cases, therefore, OS can disable the cache for the required pages. This bit is set to 1 if the cache is disabled.

2. Referenced

There are various page replacement algorithms which will be covered later in this tutorial. This bit is set to 1 if the page is referred in the last clock cycle otherwise it remains 0.

3. Modified

This bit will be set if the page has been modified otherwise it remains 0.

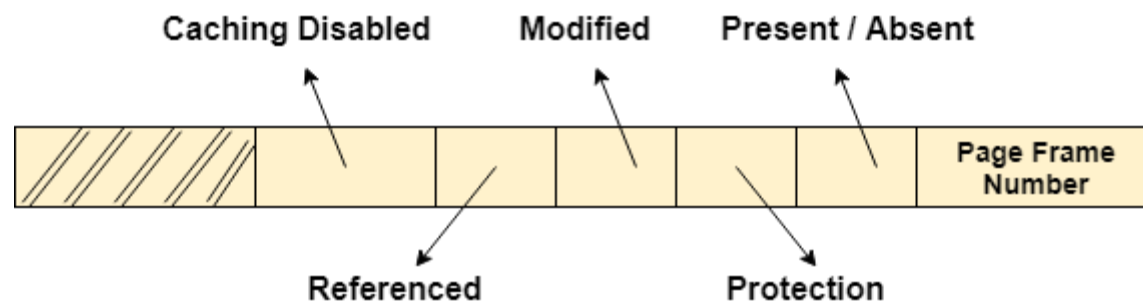
4. Protection

The protection field represents the protection level which is applied on the page. It can be read only or read & write or execute. We need to remember that it is not a bit rather it is a field which contains many bits.

5. Present/Absent

In the concept of demand paging, all the pages doesn't need to be present in the main memory. Therefore, for all the pages that are present in the main memory, this bit will be set to 1 and the bit will be 0 for all the pages which are absent.

If some page is not present in the main memory then it is called page fault.



Size of the page table

However, the part of the process which is being executed by the CPU must be present in the main memory during that time period. The page table must also be present in the main memory all the time because it has the entry for all the pages.

The size of the page table depends upon the number of entries in the table and the bytes stored in one entry.

Let's consider,

1. Logical Address = 24 bits
2. Logical Address space = 2^{24} bytes
3. Let's say, Page size = 4 KB = 2^{12} Bytes
4. Page offset = 12
5. Number of bits in a page = Logical Address - Page Offset = $24 - 12 = 12$ bits
6. Number of pages = $2^{12} = 2 \times 2 \times 10^3 = 4$ KB
7. Let's say, Page table entry = 1 Byte
8. Therefore, the size of the page table = 4 KB \times 1 Byte = 4 KB

Here we are lucky enough to get the page table size equal to the frame size. Now, the page table will be simply stored in one of the frames of the main memory. The CPU maintains a register which contains the base address of that frame, every page number from the logical address will first be added to that base address so that we can access the actual location of the word being asked.

However, in some cases, the page table size and the frame size might not be same. In those cases, the page table is considered as the collection of frames and will be stored in the different frames.

[Next →](#) [← Prev](#)

Finding Optimal Page Size

We have seen that the bigger page table size cause an extra overhead because we have to divide that table into the pages and then store that into the main memory.

Our concern must be about executing processes not on the execution of page table. Page table provides a support for the execution of the process. The larger the page Table, the higher the overhead.

We know that,

1. Page Table Size = number of page entries in page table \times size of one page entry
2. Let's consider an example,
3. Virtual Address Space = 2 GB = 2×2^{30} Bytes
4. Page Size = 2 KB = 2×2^{10} Bytes
5. Number of Pages in Page Table = $(2 \times 2^{30}) / (2 \times 2^{10}) = 1$ M pages

There will be 1 million pages which is quite big number. However, try to make page size larger, say 2 MB.

Then, Number of pages in page table = $(2 \times 2^{30}) / (2 \times 2^{20}) = 1 \text{ K pages}$.

If we compare the two scenarios, we can find out that the page table size is anti proportional to Page Size.

In Paging, there is always wastage on the last page. If the virtual address space is not a multiple of page size, then there will be some bytes remaining and we have to assign a full page to those many bytes. This is simply a overhead.

Let's consider,

1. Page Size = 2 KB
2. Virtual Address Space = 17 KB
3. Then number of pages = 17 KB / 2 KB

The number of pages will be 9 although the 9th page will only contain 1 byte and the remaining page will be wasted.

In general,

1. If page size = p bytes
2. Entry size = e bytes
3. Virtual Address Space = S bytes
4. Then, overhead $O = (S/p) \times e + (p/2)$

On an average, the wasted number of pages in a virtual space is $p/2$ (the half of total number of pages).

For, the minimal overhead,

1. $\partial O / \partial p = 0$
2. $-S/(p^2) + 1/2 = 0$
3. $p = \sqrt{2.S.e}$ bytes

Hence, if the page size $\sqrt{2.S.e}$ bytes then the overhead will be minimal.

[Next →](#) [← Prev](#)

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 RAM.

Demand Paging

Demand Paging is a popular method of virtual memory management. In demand paging, the pages of a process which are least used, get stored in the secondary memory.

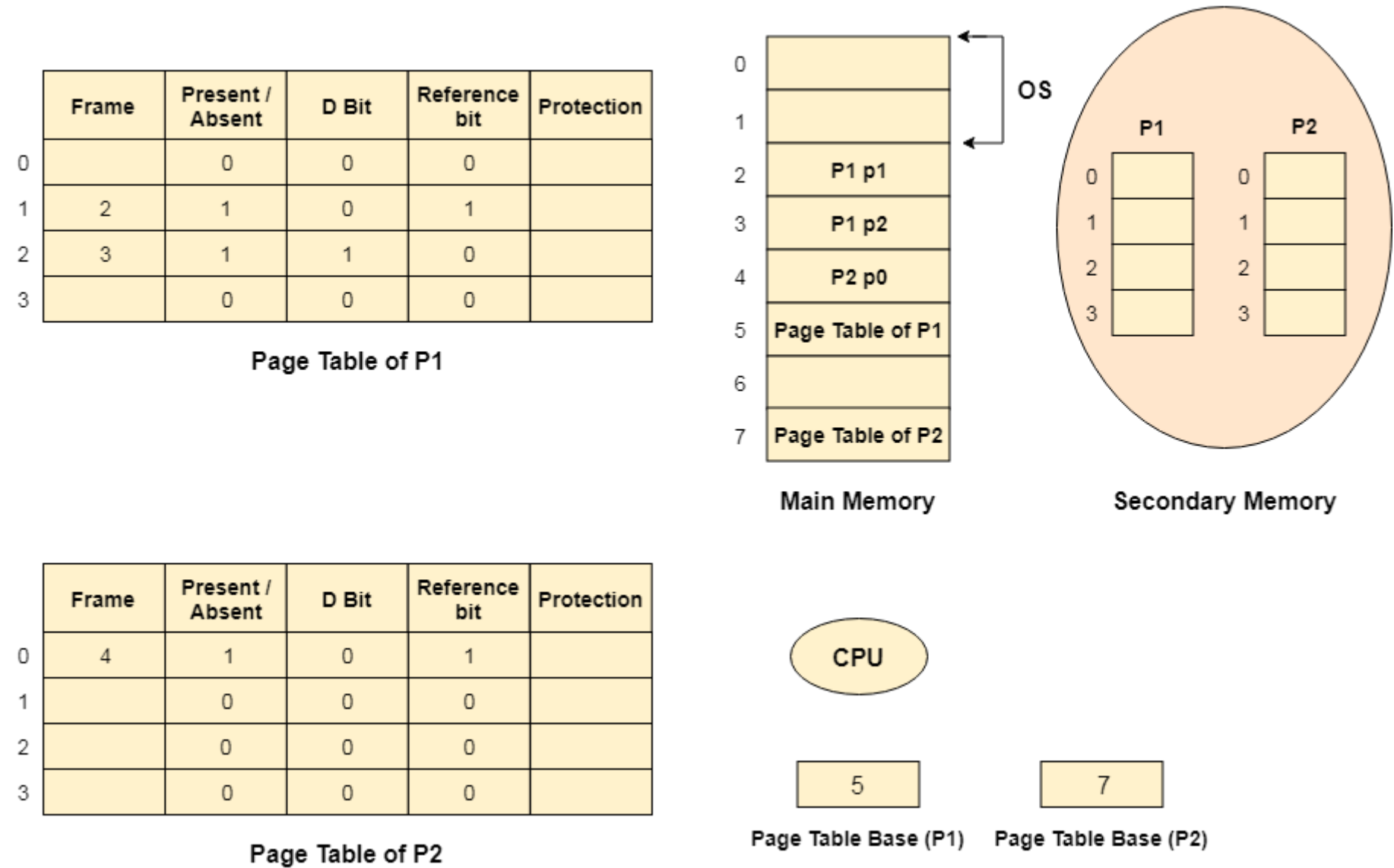
A page is copied to the main memory when its demand is made or page fault occurs. There are various page replacement algorithms which are used to determine the pages which will be replaced. We will discuss each one of them later in detail.

Snapshot of a 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.



CPU

5

7

Page Table Base (P1)

Page Table Base (P2)

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.

Translation Look aside buffer

Drawbacks of Paging

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

How to decrease the page table size

1. The page table size can be decreased by increasing the page size but it will cause internal fragmentation and there will also be page wastage.
2. Other way is to use multilevel paging but that increases the effective access time therefore this is not a practical approach.

How to decrease the effective access time

1. CPU can use a register having the page table stored inside it so that the access time to access page table can become quite less but the register are not cheaper and they are very small in compare to the page table size therefore, this is also not a practical approach.
2. To overcome these many drawbacks in paging, we have to look for a memory that is cheaper than the register and faster than the main memory so that the time taken by the CPU to access page table again and again can be reduced and it can only focus to access the actual word.

Locality of reference

In operating systems, the concept of locality of reference states that, instead of loading the entire process in the main memory, OS can load only those number of pages in the main memory that are frequently accessed by the CPU and along with that, the OS can also load only those page table entries which are corresponding to those many pages.

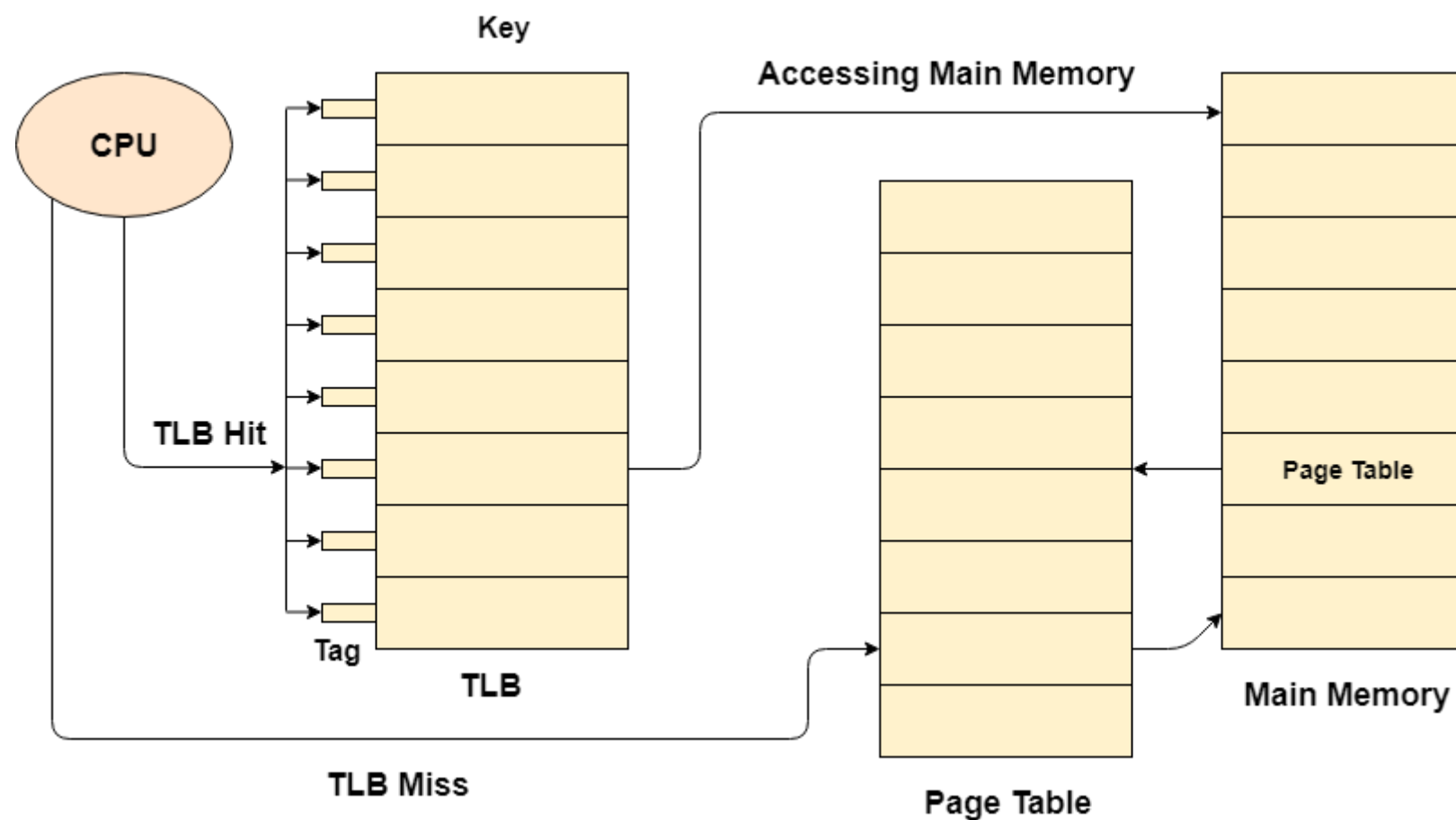
Translation look aside buffer (TLB)

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

It is a memory cache which is closer to the CPU and the time taken by CPU to access TLB is lesser then that taken to access main memory.

In other words, we can say that TLB is faster and smaller than the main memory but cheaper and bigger than the register.

TLB follows the concept of locality of reference which means that it contains only the entries of those many pages that are frequently accessed by the CPU.



In translation look aside buffers, there are tags and keys with the help of which, the mapping is done.

TLB hit is a condition where the desired entry is found in translation look aside buffer. If this happens then the CPU simply access the actual location in the main memory.

However, if the entry is not found in TLB (TLB miss) then CPU has to access page table in the main memory and then access the actual frame in the main memory.

Therefore, in the case of TLB hit, the effective access time will be lesser as compare to the case of TLB miss.

If the probability of TLB hit is P% (TLB hit rate) then the probability of TLB miss (TLB miss rate) will be (1-P) %.

Therefore, the effective access time can be defined as;

$$1. \text{ EAT} = P(t + m) + (1 - p)(t + k.m + m)$$

Where, $p \rightarrow$ TLB hit rate, $t \rightarrow$ time taken to access TLB, $m \rightarrow$ time taken to access main memory $k = 1$, if the single level paging has been implemented.

By the formula, we come to know that

1. Effective access time will be decreased if the TLB hit rate is increased.
2. Effective access time will be increased in the case of multilevel paging.

GATE Question on TLB

GATE | GATE-CS-2014-(Set-3)

Consider a paging hardware with a TLB. Assume that the entire page table and all the pages are in the physical memory. It takes 10 milliseconds to search the TLB and 80 milliseconds to access the physical memory. If the TLB hit ratio is 0.6, the effective memory access time (in milliseconds) is _____.

- A. 120
- B. 122
- C. 124
- D. 118

Given,

1. TLB hit ratio = 0.6
2. Therefore, TLB miss ratio = 0.4
3. Time taken to access TLB (t) = 10 ms
4. Time taken to access main memory (m) = 80 ms

$$\text{Effective Access Time (EAT)} = 0.6 (10 + 80) + 0.4 (10 + 80 + 80) = 90 \times 0.6 + 0.4 \times 170 = 122$$

Hence, the right answer is option B.

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.

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.

What is Thrashing?

If the number of page faults is equal to the number of referred pages or the number of page faults are so high so that the CPU remains busy in just reading the pages from the secondary memory then the effective access time will be the time taken by the CPU to read one word from the secondary memory and it will be so high. The concept is called thrashing.

If the page fault rate is PF %, the time taken in getting a page from the secondary memory and again restarting is S (service time) and the memory access time is ma then the effective access time can be given as;

$$1. \text{ EAT} = \text{PF} \times S + (1 - \text{PF}) \times (\text{ma})$$

[Next →](#) [← Prev](#)

Inverted Page Table

Inverted Page Table is the global page table which is maintained by the Operating System for all the processes. In inverted page table, the number of entries is equal to the number of frames in the main memory. It can be used to overcome the drawbacks of page table.

There is always a space reserved for the page regardless of the fact that whether it is present in the main memory or not. However, this is simply the wastage of the memory if the page is not present.

Pages	Frames
0	X
1	X
2	F1
3	F3
4	F6
5	X
6	F5

Page Table of P1

Pages	Frames
0	F2
1	F4
2	F7
3	X
4	X
5	X
6	F0

Page Table of P2

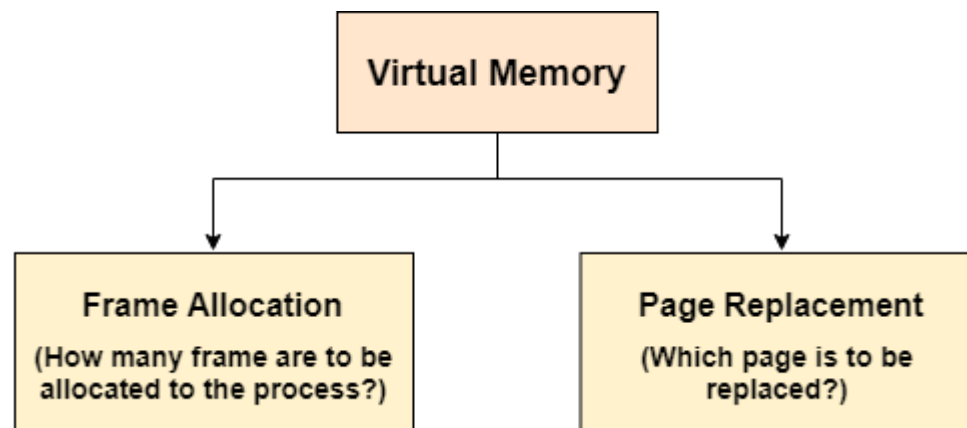
We can save this wastage by just inverting the page table. We can save the details only for the pages which are present in the main memory. Frames are the indices and the information saved inside the block will be Process ID and page number.

Pages	Frames
0	OS
1	P1 p2
2	P2 p0
3	P1 p3
4	P2 p1
5	P1 p6
6	P1 p4
7	P2 p2

Inverted Page Table

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 than 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 algorithm 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.

[Next →](#) [← Prev](#)

GATE 2015 question on LRU and FIFO

Q. Consider a main memory with five page frames and the following sequence of page references: 3, 8, 2, 3, 9, 1, 6, 3, 8, 9, 3, 6, 2, 1, 3. which one of the following is true with respect to page replacement policies First-In-First-out (FIFO) and Least Recently Used (LRU)?

- A.** Both incur the same number of page faults
- B.** FIFO incurs 2 more page faults than LRU
- C.** LRU incurs 2 more page faults than FIFO
- D.** FIFO incurs 1 more page faults than LRU

Solution:

Number of frames = 5

FIFO

According to FIFO, the page which first comes in the memory will first goes out.

Request	3	8	2	3	9	1	6	3	8	9	3	6	2	1	3
Frame 5						1	1	1	1	1	1	1	1	1	1
Frame 4					9	9	9	9	9	9	9	9	2	2	2
Frame 3			2	2	2	2	2	2	8	8	8	8	8	8	8
Frame 2		8	8	8	8	8	8	3	3	3	3	3	3	3	3
Frame 1	3	3	3	3	3	3	6	6	6	6	6	6	6	6	6
Miss/Hit	Miss	Miss	Miss	Hit	Miss	Miss	Miss	Miss	Miss	Hit	Hit	Hit	Miss	Hit	Hit

Number of Page Faults = 9

Number of hits = 6

LRU

According to LRU, the page which has not been requested for a long time will get replaced with the new one.

Request	3	8	2	3	9	1	6	3	8	9	3	6	2	1	3
Frame 5						1	1	1	1	1	1	1	2	2	2
Frame 4					9	9	9	9	9	9	9	9	9	9	9
Frame 3			2	2	2	2	2	2	8	8	8	8	8	1	1
Frame 2		8	8	8	8	8	6	6	6	6	6	6	6	6	6
Frame 1	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
Miss/Hit	Miss	Miss	Miss	Hit	Miss	Miss	Hit	Hit	Miss	Hit	Miss	Hit	Miss	Miss	Hit

Number of Page Faults = 9

Number of Hits = 6

The Number of page faults in both the cases is equal therefore the Answer is **option (A)**.

Numerical on Optimal, LRU and FIFO

Q. Consider a reference string: 4, 7, 6, 1, 7, 6, 1, 2, 7, 2. the number of frames in the memory is 3. Find out the number of page faults respective to:

1. Optimal Page Replacement Algorithm
2. FIFO Page Replacement Algorithm
3. LRU Page Replacement Algorithm

Optimal Page Replacement Algorithm

Request	4	7	6	1	7	6	1	2	7	2
Frame 3			6	6	6	6	6	2	2	2
Frame 2		7	7	7	7	7	7	7	7	7
Frame 1	4	4	4	1	1	1	1	1	1	1
Miss/Hit	Miss	Miss	Miss	Miss	Hit	Hit	Hit	Miss	Hit	Hit

Number of Page Faults in Optimal Page Replacement Algorithm = 5

LRU Page Replacement Algorithm

Request	4	7	6	1	7	6	1	2	7	2
Frame 3			6	6	6	6	6	6	7	7
Frame 2		7	7	7	7	7	7	2	2	2
Frame 1	4	4	4	1	1	1	1	1	1	1
Miss/Hit	Miss	Miss	Miss	Miss	Hit	Hit	Hit	Miss	Miss	Hit

Number of Page Faults in LRU = 6

FIFO Page Replacement Algorithm

Request	4	7	6	1	7	6	1	2	7	2
Frame 3			6	6	6	6	6	6	7	7
Frame 2		7	7	7	7	7	7	2	2	2
Frame 1	4	4	4	1	1	1	1	1	1	1
Miss/Hit	Miss	Miss	Miss	Miss	Hit	Hit	Hit	Miss	Miss	Hit

Number of Page Faults in FIFO = 6

Belady'sAnomaly

In the case of LRU and optimal page replacement algorithms, it is seen that the number of page faults will be reduced if we increase the number of frames. However, Balady found that, In FIFO page replacement algorithm, the number of page faults will get increased with the increment in number of frames.

This is the strange behavior shown by FIFO algorithm in some of the cases. This is an Anomaly called as Belady'sAnomaly.

Let's examine such example :

The reference String is given as 0 1 5 3 0 1 4 0 1 5 3 4. Let's analyze the behavior of FIFO algorithm in two cases.

Case 1: Number of frames = 3

Request	0	1	5	3	0	1	4	0	1	5	3	4
Frame 3			5	5	5	1	1	1	1	1	3	3
Frame 2		1	1	1	0	0	0	0	0	5	5	5

Frame 1	0	0	0	3	3	3	4	4	4	4	4	4
Miss/Hit	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Hit	Hit	Miss	Miss	Hit

Number of Page Faults = 9

Case 2: Number of frames = 4

Request	0	1	5	3	0	1	4	0	1	5	3	4
Frame 4				3	3	3	3	3	3	5	5	5
Frame 3			5	5	5	5	5	5	1	1	1	1
Frame 2		1	1	1	1	1	1	0	0	0	0	4
Frame 1	0	0	0	0	0	0	4	4	4	4	3	3
Miss/Hit	Miss	Miss	Miss	Miss	Hit	Hit	Miss	Miss	Miss	Miss	Miss	Miss

Number of Page Faults = 10

Therefore, in this example, the number of page faults is increasing by increasing the number of frames hence this suffers from Belady's Anomaly.

[Next →](#) [← Prev](#)

Segmentation

In Operating Systems, Segmentation is a memory management technique in which, the memory is divided into the variable size parts. Each part is known as segment which can be allocated to a process.

The details about each segment are stored in a table called as segment table. Segment table is stored in one (or many) of the segments.

Segment table contains mainly two information about segment:

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

Why Segmentation is required?

Till now, we were using Paging as our main memory management technique. Paging is more close to Operating system rather than the User. It divides all the process into the form of pages regardless of the fact that a process can have some relative parts of functions which needs to be loaded in the same page.

Operating system doesn't care about the User's view of the process. It may divide the same function into different pages and those pages may or may not be loaded at the same time into the memory. It decreases the efficiency of the system.

It is better to have segmentation which divides the process into the segments. Each segment contain same type of functions such as main function can be included in one segment and the library functions can be included in the other segment,

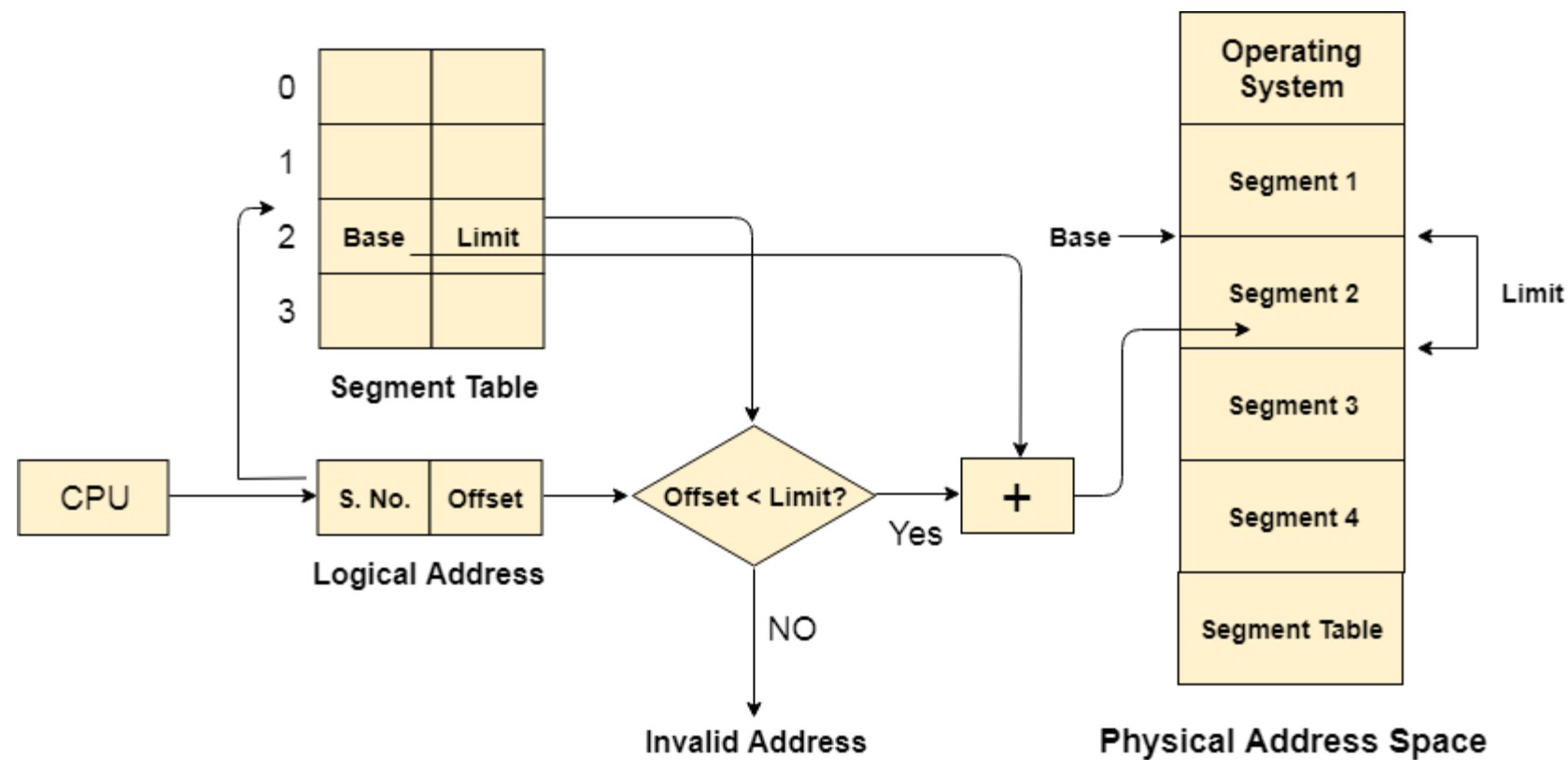
Translation of Logical address into physical address by segment table

CPU generates a logical address which contains two parts:

1. Segment Number
2. Offset

The Segment number is mapped to the segment table. The limit of the respective segment is compared with the offset. If the offset is less than the limit then the address is valid otherwise it throws an error as the address is invalid.

In the case of valid address, the base address of the segment is added to the offset to get the physical address of actual word in the main memory.



Advantages of Segmentation

1. No internal fragmentation
2. Average Segment Size is larger than the actual page size.
3. Less overhead
4. It is easier to relocate segments than entire address space.
5. The segment table is of lesser size as compare to the page table in paging.

Disadvantages

1. It can have external fragmentation.

2. it is difficult to allocate contiguous memory to variable sized partition.
3. Costly memory management algorithms.

Paging VS Segmentation

Sr No.	Paging	Segmentation
1	Non-Contiguous memory allocation	Non-contiguous memory allocation
2	Paging divides program into fixed size pages.	Segmentation divides program into variable size segments.
3	OS is responsible	Compiler is responsible.
4	Paging is faster than segmentation	Segmentation is slower than paging
5	Paging is closer to Operating System	Segmentation is closer to User
6	It suffers from internal fragmentation	It suffers from external fragmentation
7	There is no external fragmentation	There is no external fragmentation
8	Logical address is divided into page number and page offset	Logical address is divided into segment number and segment offset
9	Page table is used to maintain the page information.	Segment Table maintains the segment information

10	Page table entry has the frame number and some flag bits to represent details about pages.	Segment table entry has the base address of the segment and some protection bits for the segments.
----	--	--

Segmented Paging

Pure segmentation is not very popular and not being used in many of the operating systems. However, Segmentation can be combined with Paging to get the best features out of both the techniques.

In Segmented Paging, the main memory is divided into variable size segments which are further divided into fixed size pages.

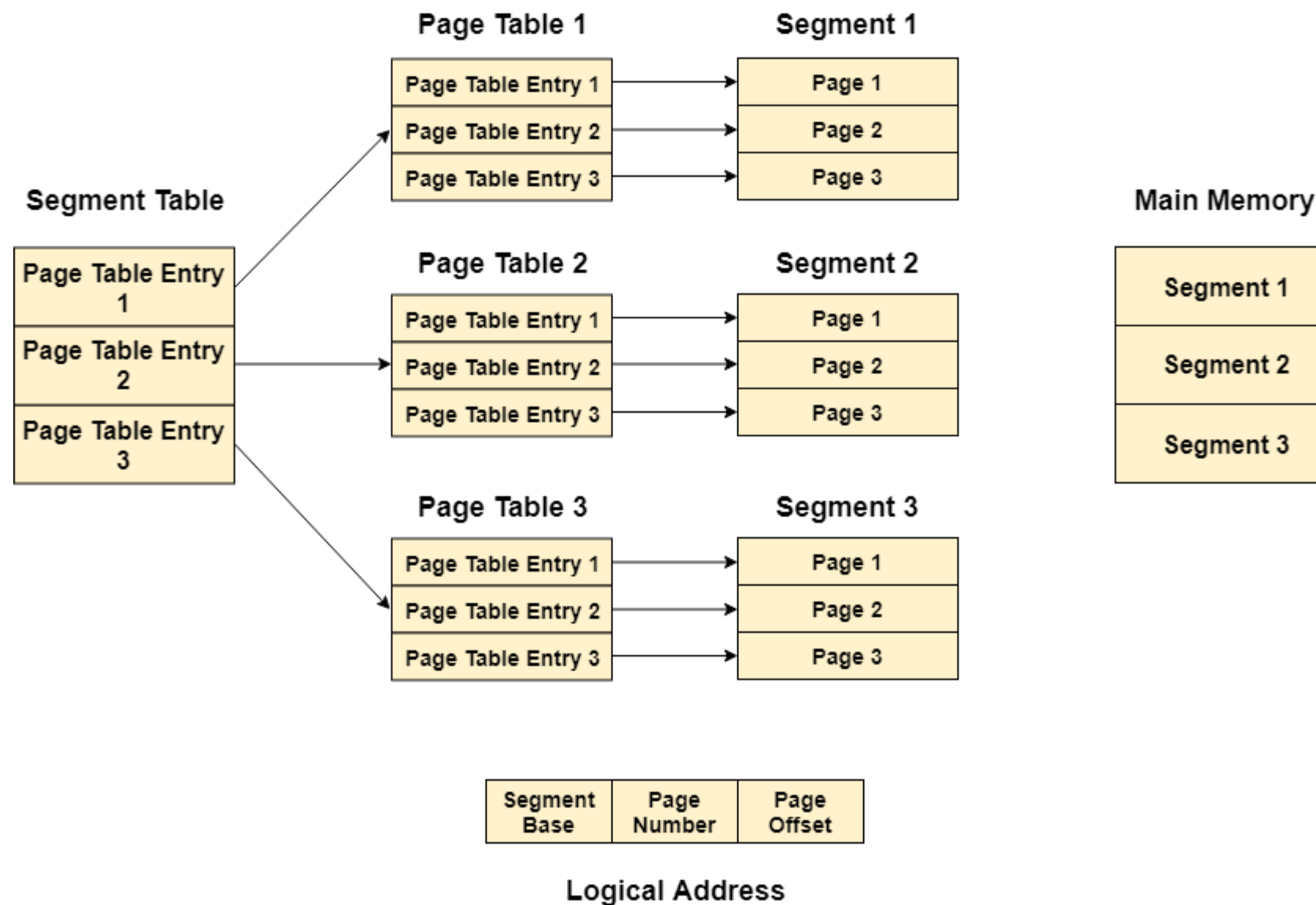
1. Pages are smaller than segments.
2. Each Segment has a page table which means every program has multiple page tables.
3. The logical address is represented as Segment Number (base address), Page number and page offset.

Segment Number → It points to the appropriate Segment Number.

Page Number → It Points to the exact page within the segment

Page Offset → Used as an offset within the page frame

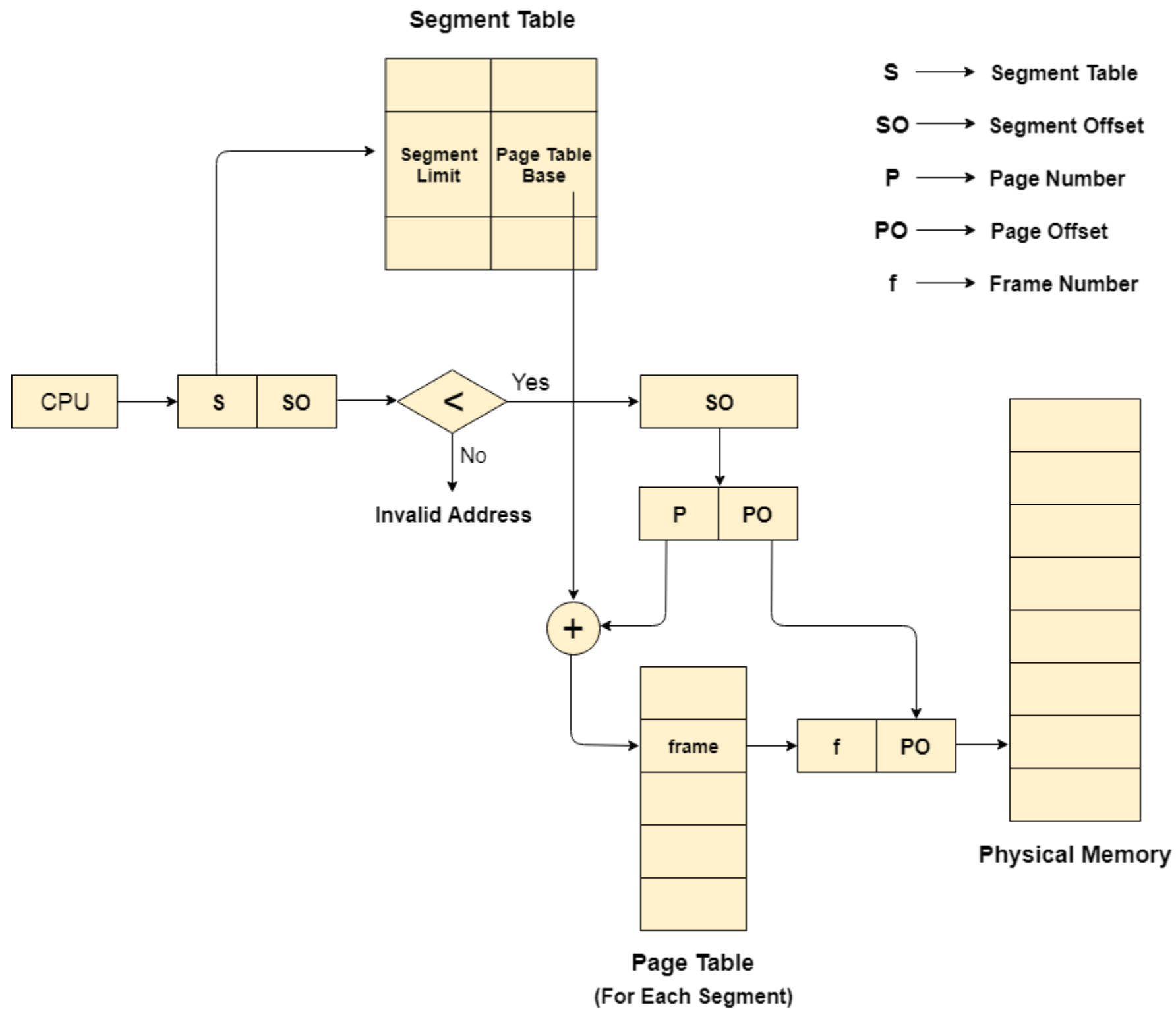
Each Page table contains the various information about every page of the segment. The Segment Table contains the information about every segment. Each segment table entry points to a page table entry and every page table entry is mapped to one of the page within a segment.



Translation of logical address to physical address

The CPU generates a logical address which is divided into two parts: Segment Number and Segment Offset. The Segment Offset must be less than the segment limit. Offset is further divided into Page number and Page Offset. To map the exact page number in the page table, the page number is added into the page table base.

The actual frame number with the page offset is mapped to the main memory to get the desired word in the page of the certain segment of the process.



Advantages of Segmented Paging

1. It reduces memory usage.
2. Page table size is limited by the segment size.
3. Segment table has only one entry corresponding to one actual segment.
4. External Fragmentation is not there.
5. It simplifies memory allocation.

Disadvantages of Segmented Paging

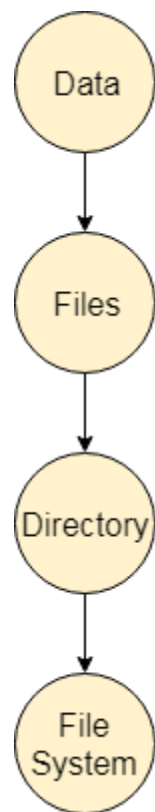
1. Internal Fragmentation will be there.
2. The complexity level will be much higher as compare to paging.
3. Page Tables need to be contiguously stored in the memory.

#6.File Management

What is a File ?

A file can be defined as a data structure which stores the sequence of records. Files are stored in a file system, which may exist on a disk or in the main memory. Files can be simple (plain text) or complex (specially-formatted).

The collection of files is known as Directory. The collection of directories at the different levels, is known as File System.



Attributes of the File

1.Name

Every file carries a name by which the file is recognized in the file system. One directory cannot have two files with the same name.

2.Identifier

Along with the name, Each File has its own extension which identifies the type of the file. For example, a text file has the extension **.txt**, A video file can have the extension **.mp4**.

3.Type

In a File System, the Files are classified in different types such as video files, audio files, text files, executable files, etc.

4.Location

In the File System, there are several locations on which, the files can be stored. Each file carries its location as its attribute.

5.Size

The Size of the File is one of its most important attribute. By size of the file, we mean the number of bytes acquired by the file in the memory.

6.Protection

The Admin of the computer may want the different protections for the different files. Therefore each file carries its own set of permissions to the different group of Users.

7.Time and Date

Every file carries a time stamp which contains the time and date on which the file is last modified.

[Next →](#) [← Prev](#)

Operations on the File

There are various operations which can be implemented on a file. We will see all of them in detail.

1.Create

Creation of the file is the most important operation on the file. Different types of files are created by different methods for example text editors are used to create a text file, word processors are used to create a word file and Image editors are used to create the image files.

2.Write

Writing the file is different from creating the file. The OS maintains a write pointer for every file which points to the position in the file from which, the data needs to be written.

3.Read

Every file is opened in three different modes : Read, Write and append. A Read pointer is maintained by the OS, pointing to the position up to which, the data has been read.

4.Re-position

Re-positioning is simply moving the file pointers forward or backward depending upon the user's requirement. It is also called as seeking.

5.Delete

Deleting the file will not only delete all the data stored inside the file, It also deletes all the attributes of the file. The space which is allocated to the file will now become available and can be allocated to the other files.

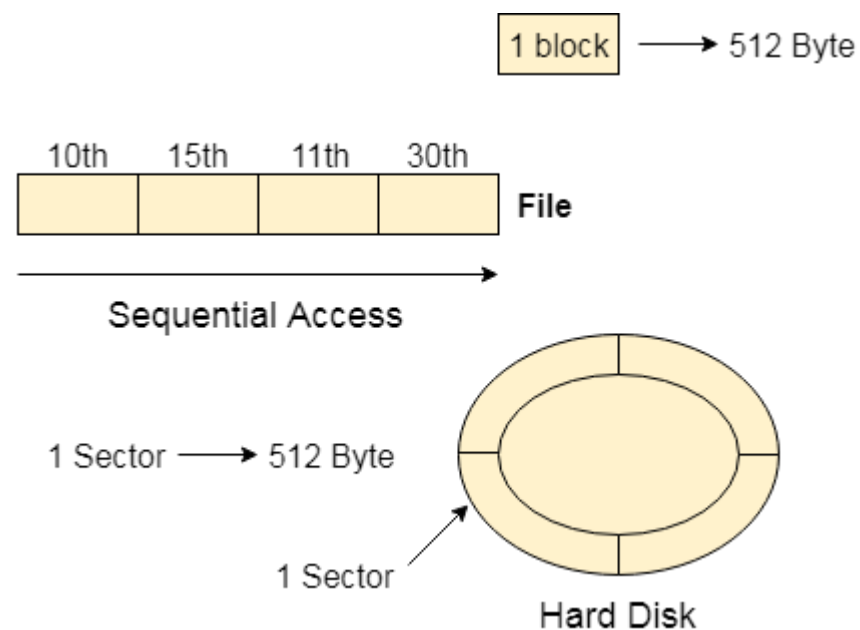
6.Truncate

Truncating is simply deleting the file except deleting attributes. The file is not completely deleted although the information stored inside the file get replaced.

File Access Methods

Let's look at various ways to access files stored in secondary memory.

Sequential Access



Most of the operating systems access the file sequentially. In other words, we can say that most of the files need to be accessed sequentially by the operating system.

In sequential access, the OS read the file word by word. A pointer is maintained which initially points to the base address of the file. If the user wants to read first word of the file then the pointer provides that word to the user and increases its value by 1 word. This process continues till the end of the file.

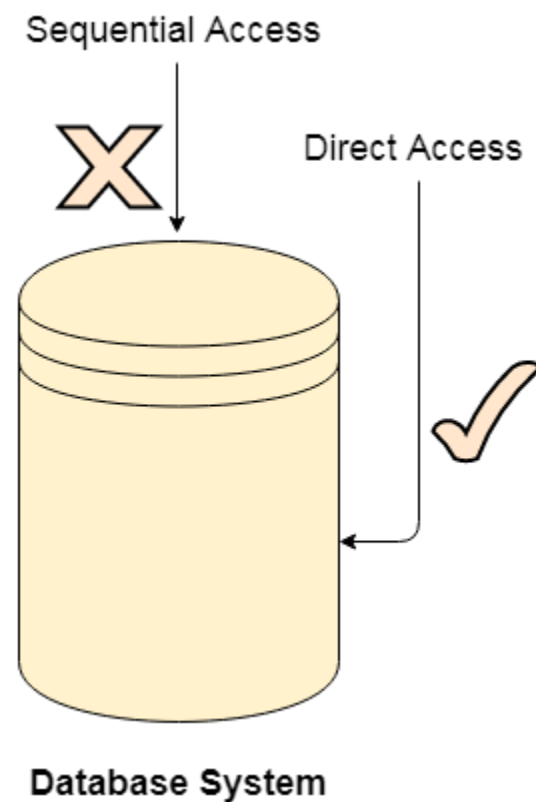
Modern word systems do provide the concept of direct access and indexed access but the most used method is sequential access due to the fact that most of the files such as text files, audio files, video files, etc need to be sequentially accessed.

Direct Access

The Direct Access is mostly required in the case of database systems. In most of the cases, we need filtered information from the database. The sequential access can be very slow and inefficient in such cases.

Suppose every block of the storage stores 4 records and we know that the record we needed is stored in 10th block. In that case, the sequential access will not be implemented because it will traverse all the blocks in order to access the needed record.

Direct access will give the required result despite of the fact that the operating system has to perform some complex tasks such as determining the desired block number. However, that is generally implemented in database applications.



Indexed Access

If a file can be sorted on any of the filed then an index can be assigned to a group of certain records. However, A particular record can be accessed by its index. The index is nothing but the address of a record in the file.

In index accessing, searching in a large database became very quick and easy but we need to have some extra space in the memory to store the index value.

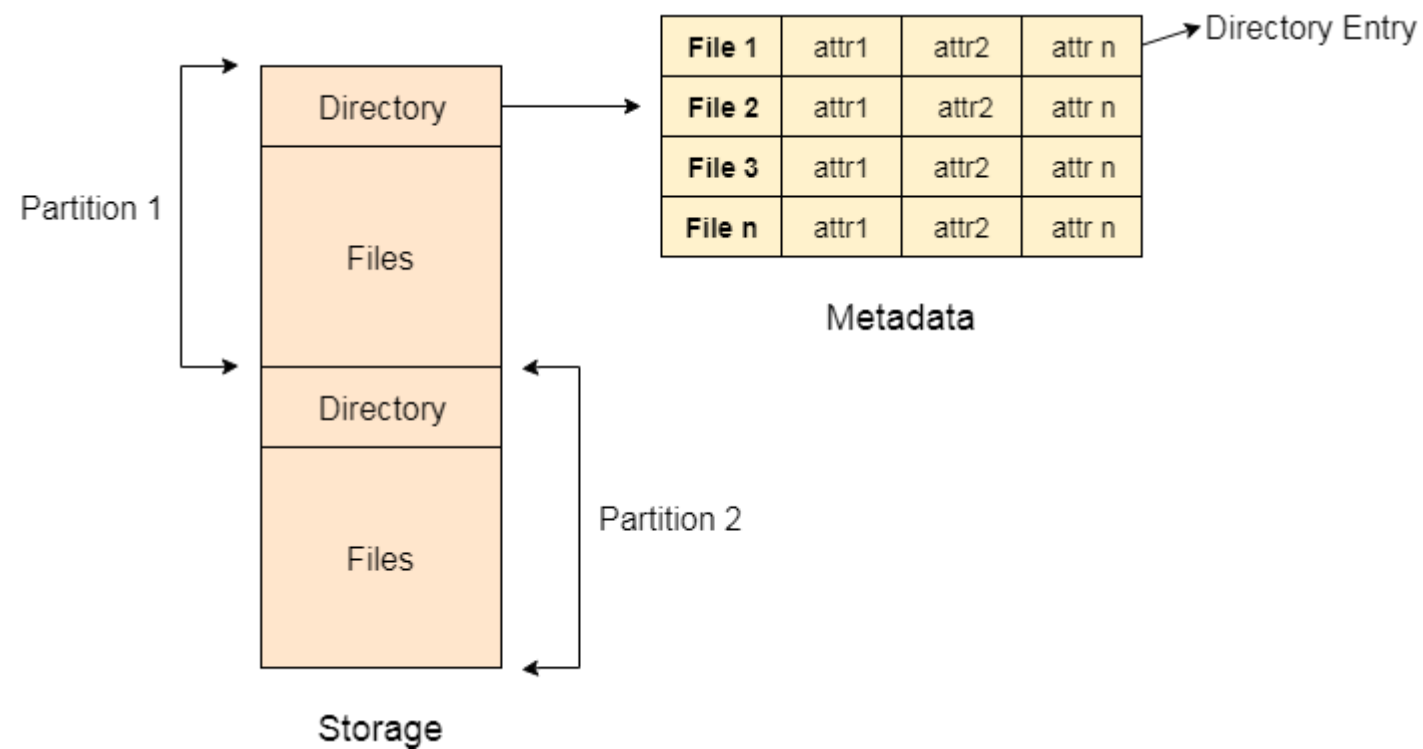
Directory Structure

What is a directory?

Directory can be defined as the listing of the related files on the disk. The directory may store some or the entire file attributes.

To get the benefit of different file systems on the different operating systems, A hard disk can be divided into the number of partitions of different sizes. The partitions are also called volumes or mini disks.

Each partition must have at least one directory in which, all the files of the partition can be listed. A directory entry is maintained for each file in the directory which stores all the information related to that file.



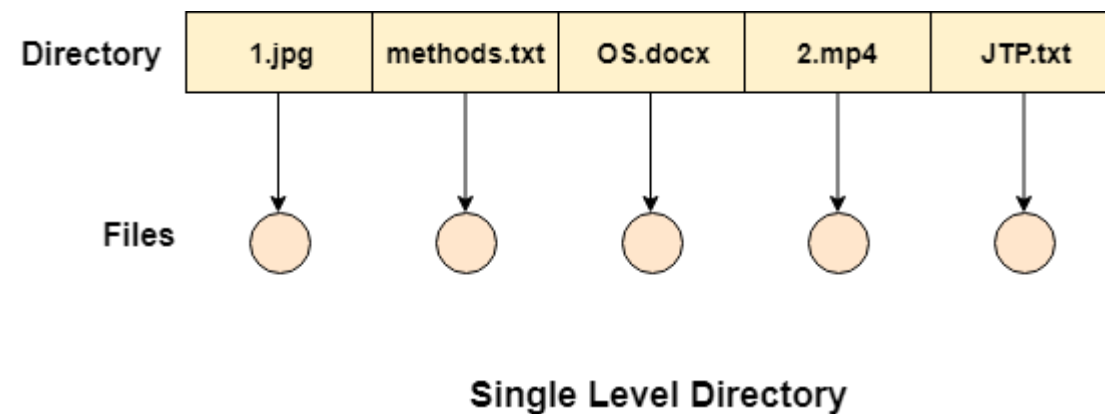
A directory can be viewed as a file which contains the Meta data of the bunch of files.

Every Directory supports a number of common operations on the file:

1. File Creation
2. Search for the file
3. File deletion
4. Renaming the file
5. Traversing Files
6. Listing of files

Single Level Directory

The simplest method is to have one big list of all the files on the disk. The entire system will contain only one directory which is supposed to mention all the files present in the file system. The directory contains one entry per each file present on the file system.



This type of directories can be used for a simple system.

Advantages

1. Implementation is very simple.
2. If the sizes of the files are very small then the searching becomes faster.
3. File creation, searching, deletion is very simple since we have only one directory.

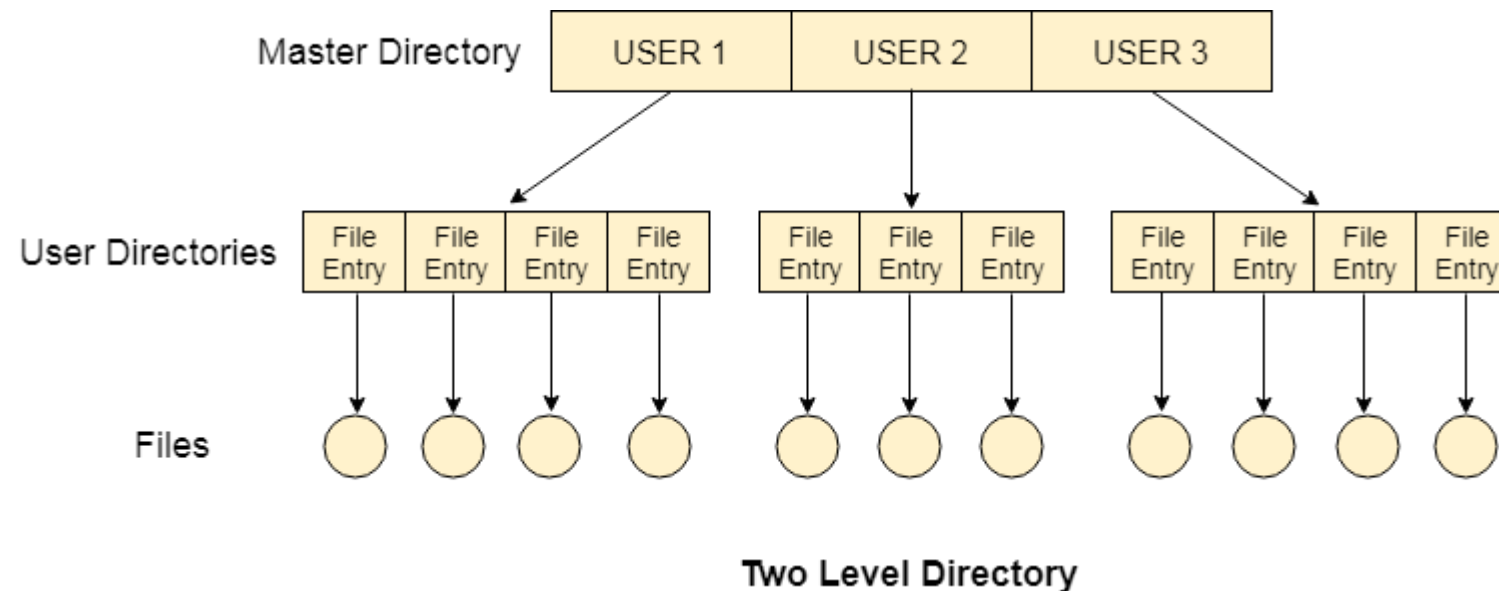
Disadvantages

1. We cannot have two files with the same name.
2. The directory may be very big therefore searching for a file may take so much time.
3. Protection cannot be implemented for multiple users.
4. There are no ways to group same kind of files.
5. Choosing the unique name for every file is a bit complex and limits the number of files in the system because most of the Operating System limits the number of characters used to construct the file name.

[Next →](#) [← Prev](#)

Two Level Directory

In two level directory systems, we can create a separate directory for each user. There is one master directory which contains separate directories dedicated to each user. For each user, there is a different directory present at the second level, containing group of user's file. The system doesn't let a user to enter in the other user's directory without permission.



Characteristics of two level directory system

1. Each files has a path name as ***/User-name/directory-name/***
2. Different users can have the same file name.
3. Searching becomes more efficient as only one user's list needs to be traversed.
4. The same kind of files cannot be grouped into a single directory for a particular user.

Every Operating System maintains a variable as **PWD** which contains the present directory name (present user name) so that the searching can be done appropriately.

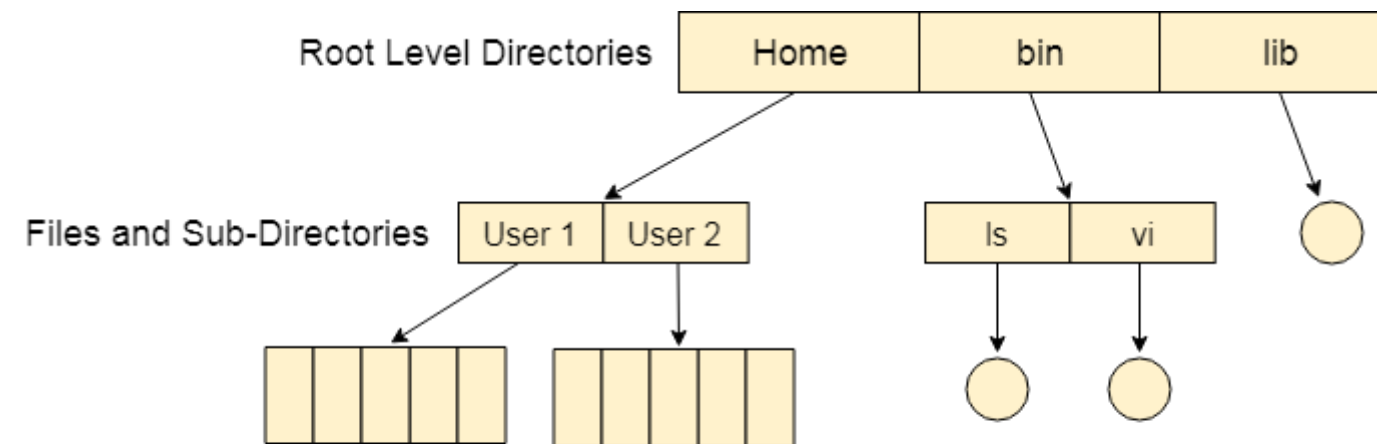
Tree Structured Directory

In Tree structured directory system, any directory entry can either be a file or sub directory. Tree structured directory system overcomes the drawbacks of two level directory system. The similar kind of files can now be grouped in one directory.

Each user has its own directory and it cannot enter in the other user's directory. However, the user has the permission to read the root's data but he cannot write or modify this. Only administrator of the system has the complete access of root directory.

Searching is more efficient in this directory structure. The concept of current working directory is used. A file can be accessed by two types of path, either relative or absolute.

Absolute path is the path of the file with respect to the root directory of the system while relative path is the path with respect to the current working directory of the system. In tree structured directory systems, the user is given the privilege to create the files as well as directories.



The Structured Directory System

Permissions on the file and directory

A tree structured directory system may consist of various levels therefore there is a set of permissions assigned to each file and directory.

The permissions are **R W X** which are regarding reading, writing and the execution of the files or directory. The permissions are assigned to three types of users: owner, group and others.

There is a identification bit which differentiate between directory and file. For a directory, it is **d** and for a file, it is dot **(.)**

The following snapshot shows the permissions assigned to a file in a Linux based system. Initial bit **d** represents that it is a directory.

```
javatpoint@localhost:~/javatpoint
File Edit View Search Terminal Help
[javatpoint@localhost javatpoint]$ ls -l
total 0
drwxr-xr-x. 3 root root 27 Jan 11 14:37 CentOS
[javatpoint@localhost javatpoint]$
```

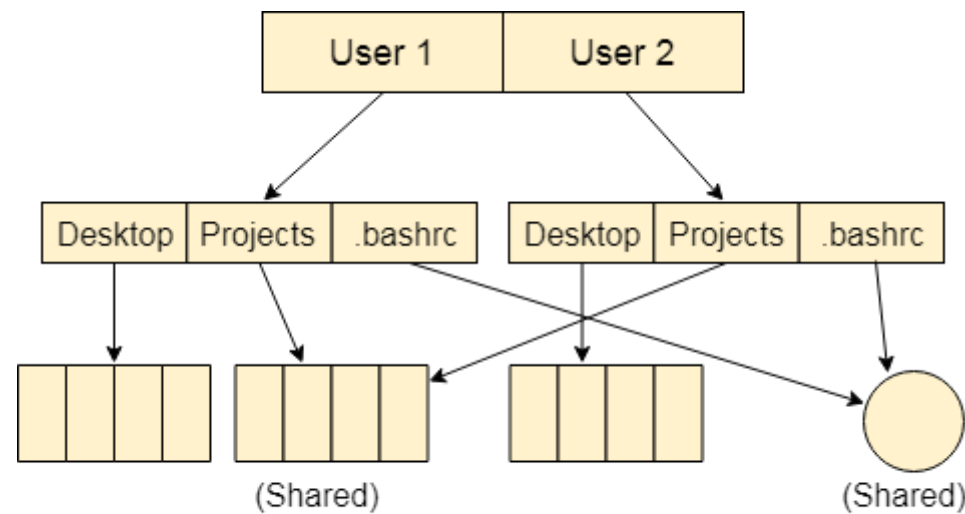
Acyclic-Graph Structured Directories

The tree structured directory system doesn't allow the same file to exist in multiple directories therefore sharing is major concern in tree structured directory system. We can provide sharing by making the directory an acyclic graph. In this system, two or more directory entry can point to the same file or sub directory. That file or sub directory is shared between the two directory entries.

These kinds of directory graphs can be made using links or aliases. We can have multiple paths for a same file. Links can either be symbolic (logical) or hard link (physical).

If a file gets deleted in acyclic graph structured directory system, then

1. In the case of soft link, the file just gets deleted and we are left with a dangling pointer.
2. In the case of hard link, the actual file will be deleted only if all the references to it gets deleted.



Acyclic-Graph Structured Directory System

File Systems

File system is the part of the operating system which is responsible for file management. It provides a mechanism to store the data and access to the file contents including data and programs. Some Operating systems treats everything as a file for example Ubuntu.

The File system takes care of the following issues

- **File Structure**

We have seen various data structures in which the file can be stored. The task of the file system is to maintain an optimal file structure.

- **Recovering Free space**

Whenever a file gets deleted from the hard disk, there is a free space created in the disk. There can be many such spaces which need to be recovered in order to reallocate them to other files.

- **disk space assignment to the files**

The major concern about the file is deciding where to store the files on the hard disk. There are various disks scheduling algorithm which will be covered later in this tutorial.

- **tracking data location**

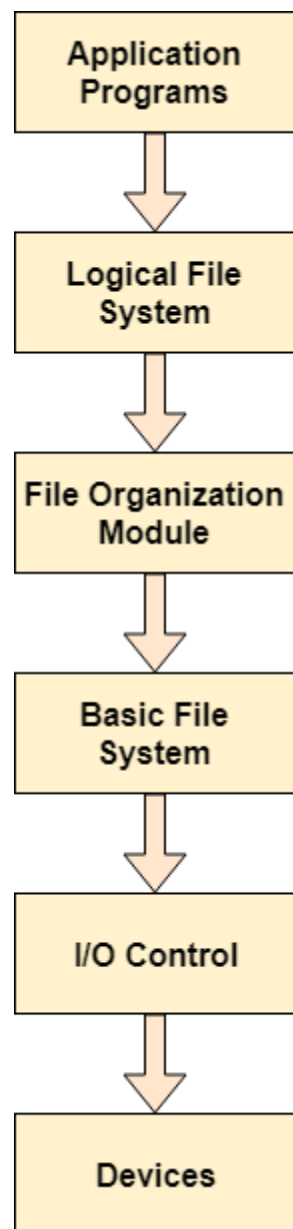
A File may or may not be stored within only one block. It can be stored in the non contiguous blocks on the disk. We need to keep track of all the blocks on which the part of the files reside.

File System Structure

File System provide efficient access to the disk by allowing data to be stored, located and retrieved in a convenient way. A file System must be able to store the file, locate the file and retrieve the file.

Most of the Operating Systems use layering approach for every task including file systems. Every layer of the file system is responsible for some activities.

The image shown below, elaborates how the file system is divided in different layers, and also the functionality of each layer.



- When an application program asks for a file, the first request is directed to the logical file system. The logical file system contains the Meta data of the file and directory structure. If the application program doesn't have the required permissions of the file then this layer will throw an error. Logical file systems also verify the path to the file.

- Generally, files are divided into various logical blocks. Files are to be stored in the hard disk and to be retrieved from the hard disk. Hard disk is divided into various tracks and sectors. Therefore, in order to store and retrieve the files, the logical blocks need to be mapped to physical blocks. This mapping is done by File organization module. It is also responsible for free space management.
- Once File organization module decided which physical block the application program needs, it passes this information to basic file system. The basic file system is responsible for issuing the commands to I/O control in order to fetch those blocks.
- I/O controls contain the codes by using which it can access hard disk. These codes are known as device drivers. I/O controls are also responsible for handling interrupts.

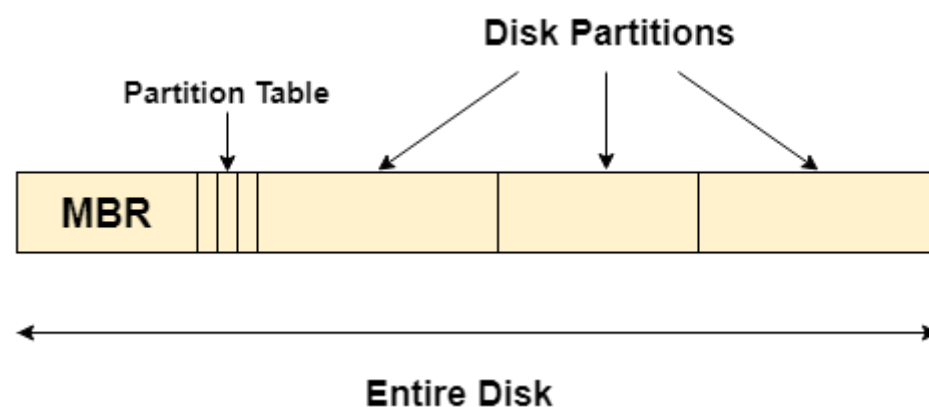
[Next →](#) [← Prev](#)

Master Boot Record (MBR)

Master boot record is the information present in the first sector of any hard disk. It contains the information regarding how and where the Operating system is located in the hard disk so that it can be booted in the RAM.

MBR is sometimes called master partition table because it includes a partition table which locates every partition in the hard disk.

Master boot record (MBR) also includes a program which reads the boot sector record of the partition that contains operating system.



What happens when you turn on your computer?

Due to the fact that the main memory is volatile, when we turn on our computer, CPU

cannot access the main memory directly. However, there is a special program called as BIOS stored in ROM is accessed for the first time by the CPU.

BIOS contains the code, by executing which, the CPU access the very first partition of hard disk that is MBR. It contains a partition table for all the partitions of the hard disk.

Since, MBR contains the information about where the operating system is being stored and it also contains a program which can read the boot sector record of the partition, hence the CPU fetches all this information and load the operating system into the main memory.

[Next →](#) [← Prev](#)

On Disk Data Structures

There are various on disk data structures that are used to implement a file system. This structure may vary depending upon the operating system.

1. **Boot Control Block**

Boot Control Block contains all the information which is needed to boot an operating system from that volume. It is called boot block in UNIX file system. In NTFS, it is called the partition boot sector.

2. **Volume Control Block**

Volume control block all the information regarding that volume such as number of blocks, size of each block, partition table, pointers to free blocks and free FCB blocks. In UNIX file system, it is known as super block. In NTFS, this information is stored inside master file table.

3. **Directory Structure (per file system)**

A directory structure (per file system) contains file names and pointers to corresponding FCBs. In UNIX, it includes inode numbers associated to file names.

4. **File Control Block**

File Control block contains all the details about the file such as ownership details, permission details, file size, etc. In UFS, this detail is stored in inode. In NTFS, this information is stored inside master file table as a relational database structure. A typical file control block is shown in the image below.

File Permissions
File Dates (Create, Access, Write)
File Owner, Group, ACL
File Size
File Data Blocks

File Control Block

[Next →](#) [← Prev](#)

In Memory Data Structure

Till now, we have discussed the data structures that are required to be present on the hard disk in order to implement file systems. Here, we will discuss the data structures required to be present in memory in order to implement the file system.

The in-memory data structures are used for file system management as well as performance improvement via caching. This information is loaded on the mount time and discarded on ejection.

1. In-memory Mount Table

In-memory mount table contains the list of all the devices which are being mounted to the system. Whenever the connection is maintained to a device, its entry will be done in the mount table.

2. In-memory Directory structure cache

This is the list of directory which is recently accessed by the CPU. The directories present in the list can also be accessed in the near future so it will be better to store them temporally in cache.

3. **System-wide open file table**

This is the list of all the open files in the system at a particular time. Whenever the user open any file for reading or writing, the entry will be made in this open file table.

4. **Per process Open file table**

It is the list of open files subjected to every process. Since there is already a list which is there for every open file in the system therefore It only contains Pointers to the appropriate entry in the system wide table.

[Next →](#) [← Prev](#)

Directory Implementation

There is the number of algorithms by using which, the directories can be implemented. However, the selection of an appropriate directory implementation algorithm may significantly affect the performance of the system.

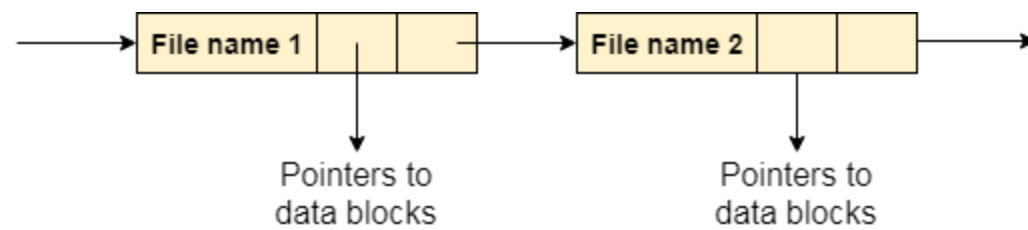
The directory implementation algorithms are classified according to the data structure they are using. There are mainly two algorithms which are used in these days.

1. **Linear List**

In this algorithm, all the files in a directory are maintained as singly lined list. Each file contains the pointers to the data blocks which are assigned to it and the next file in the directory.

Characteristics

1. When a new file is created, then the entire list is checked whether the new file name is matching to a existing file name or not. In case, it doesn't exist, the file can be created at the beginning or at the end. Therefore, searching for a unique name is a big concern because traversing the whole list takes time.
2. The list needs to be traversed in case of every operation (creation, deletion, updating, etc) on the files therefore the systems become inefficient.



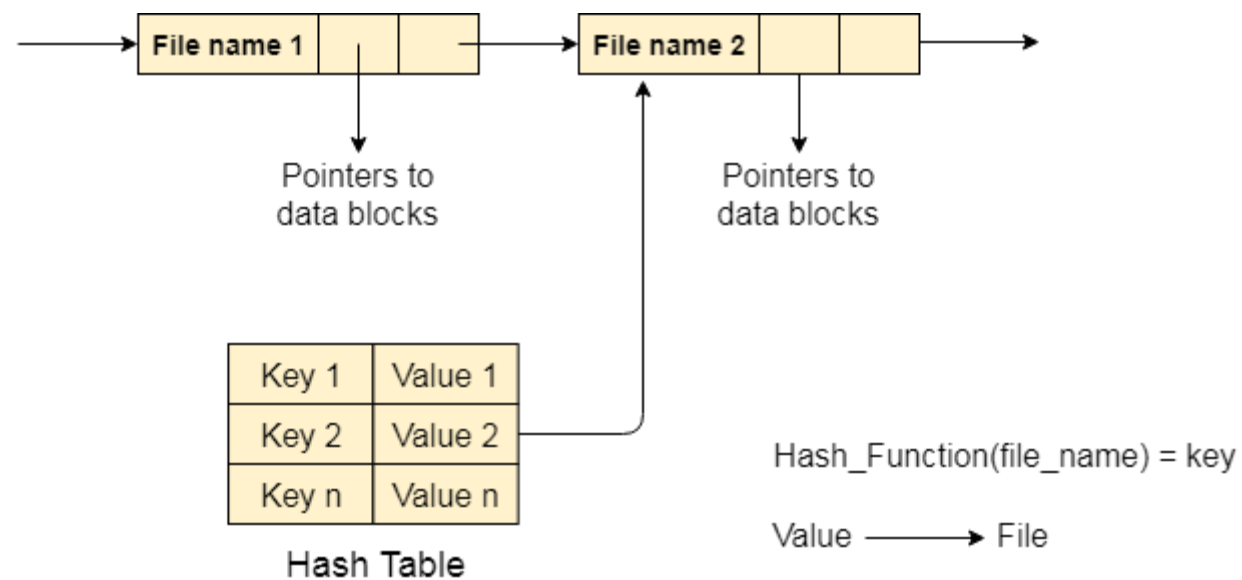
Linear List

2. Hash Table

To overcome the drawbacks of singly linked list implementation of directories, there is an alternative approach that is hash table. This approach suggests to use hash table along with the linked lists.

A key-value pair for each file in the directory gets generated and stored in the hash table. The key can be determined by applying the hash function on the file name while the key points to the corresponding file stored in the directory.

Now, searching becomes efficient due to the fact that now, entire list will not be searched on every operating. Only hash table entries are checked using the key and if an entry found then the corresponding file will be fetched using the value.



Allocation Methods

There are various methods which can be used to allocate disk space to the files. Selection of an appropriate allocation method will significantly affect the performance and efficiency of the system. Allocation method provides a way in which the disk will be utilized and the files will be accessed.

There are following methods which can be used for allocation.

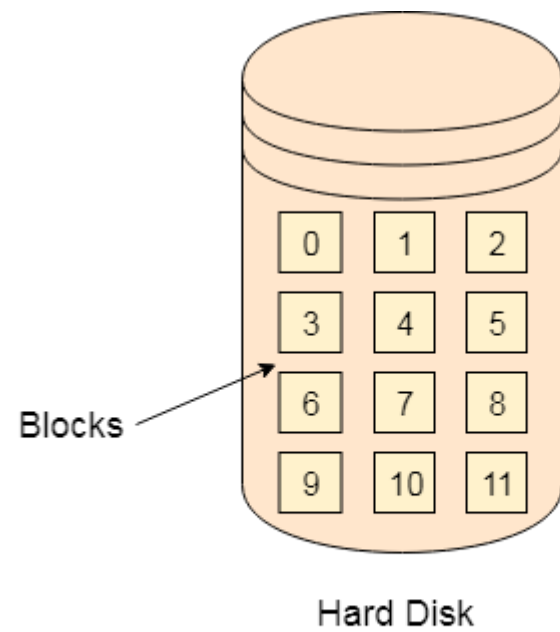
1. Contiguous Allocation.
2. Extents
3. Linked Allocation
4. Clustering
5. FAT
6. Indexed Allocation
7. Linked Indexed Allocation
8. Multilevel Indexed Allocation
9. Inode

We will discuss three of the most used methods in detail.

Contiguous Allocation

If the blocks are allocated to the file in such a way that all the logical blocks of the file get the contiguous physical block in the hard disk then such allocation scheme is known as contiguous allocation.

In the image shown below, there are three files in the directory. The starting block and the length of each file are mentioned in the table. We can check in the table that the contiguous blocks are assigned to each file as per its need.



File Name	Start	Length	Allocated Blocks
abc.text	0	3	0,1,2
video.mp4	4	2	4,5
jtp.docx	9	3	9,10,11

Directory

Contiguous Allocation

Advantages

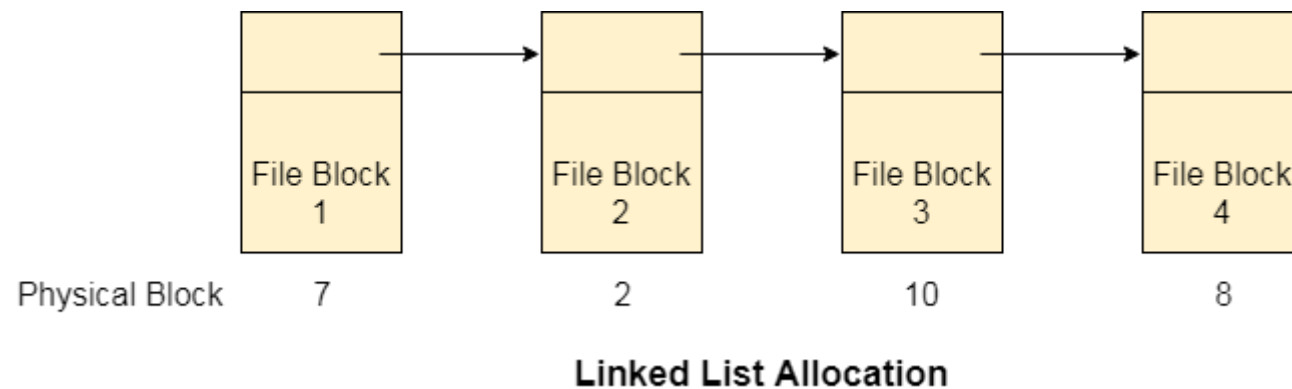
1. It is simple to implement.
2. We will get Excellent read performance.
3. Supports Random Access into files.

Disadvantages

1. The disk will become fragmented.
2. It may be difficult to have a file grow.

Linked List Allocation

Linked List allocation solves all problems of contiguous allocation. In linked list allocation, each file is considered as the linked list of disk blocks. However, the disks blocks allocated to a particular file need not to be contiguous on the disk. Each disk block allocated to a file contains a pointer which points to the next disk block allocated to the same file.



Advantages

1. There is no external fragmentation with linked allocation.
2. Any free block can be utilized in order to satisfy the file block requests.
3. File can continue to grow as long as the free blocks are available.
4. Directory entry will only contain the starting block address.

Disadvantages

1. Random Access is not provided.
2. Pointers require some space in the disk blocks.
3. Any of the pointers in the linked list must not be broken otherwise the file will get corrupted.
4. Need to traverse each block.

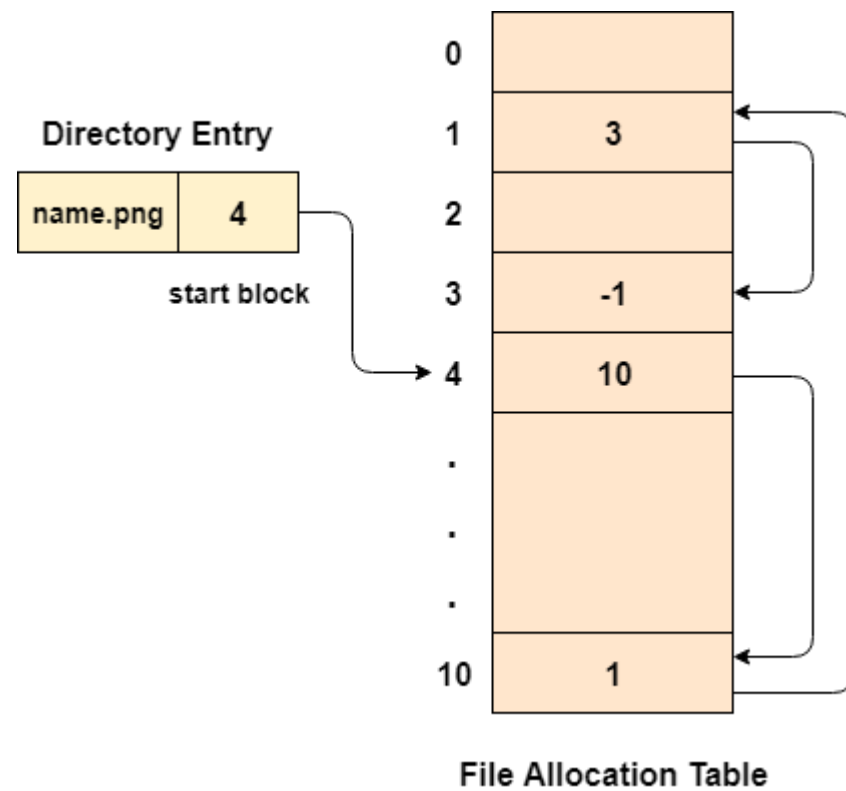
File Allocation Table

The main disadvantage of linked list allocation is that the Random access to a particular block is not provided. In order to access a block, we need to access all its previous blocks.

File Allocation Table overcomes this drawback of linked list allocation. In this scheme, a file allocation table is maintained, which gathers all the disk block links. The table has one entry for each disk block and is indexed by block number.

File allocation table needs to be cached in order to reduce the number of head seeks. Now the head doesn't need to traverse all the disk blocks in order to access one successive block.

It simply accesses the file allocation table, read the desired block entry from there and access that block. This is the way by which the random access is accomplished by using FAT. It is used by MS-DOS and pre-NT Windows versions.



Advantages

1. Uses the whole disk block for data.
2. A bad disk block doesn't cause all successive blocks lost.
3. Random access is provided although its not too fast.
4. Only FAT needs to be traversed in each file operation.

Disadvantages

1. Each Disk block needs a FAT entry.
2. FAT size may be very big depending upon the number of FAT entries.
3. Number of FAT entries can be reduced by increasing the block size but it will also increase Internal Fragmentation.

Indexed Allocation

Limitation of FAT

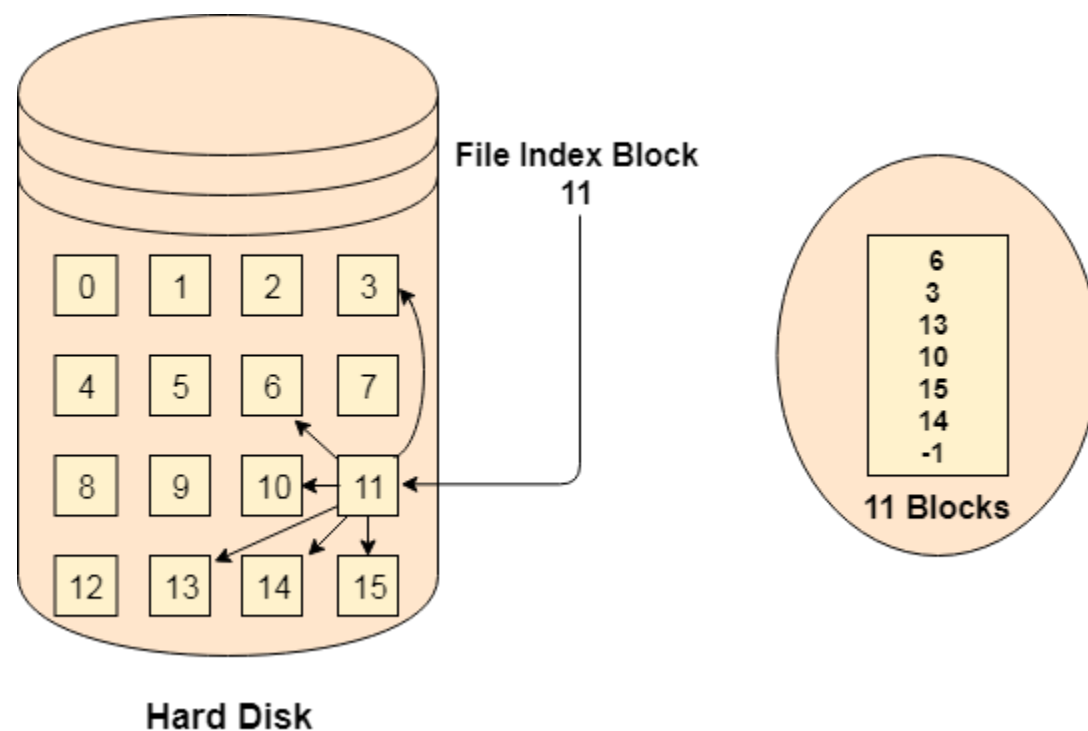
Limitation in the existing technology causes the evolution of a new technology. Till now, we have seen various allocation methods; each of them was carrying several advantages and disadvantages.

File allocation table tries to solve as many problems as possible but leads to a drawback. The more the number of blocks, the more will be the size of FAT.

Therefore, we need to allocate more space to a file allocation table. Since, file allocation table needs to be cached therefore it is impossible to have as many space in cache. Here we need a new technology which can solve such problems.

Indexed Allocation Scheme

Instead of maintaining a file allocation table of all the disk pointers, Indexed allocation scheme stores all the disk pointers in one of the blocks called as indexed block. Indexed block doesn't hold the file data, but it holds the pointers to all the disk blocks allocated to that particular file. Directory entry will only contain the index block address.



Advantages

1. Supports direct access
2. A bad data block causes the lost of only that block.

Disadvantages

1. A bad index block could cause the lost of entire file.
2. Size of a file depends upon the number of pointers, a index block can hold.
3. Having an index block for a small file is totally wastage.
4. More pointer overhead

[Next →](#) [← Prev](#)

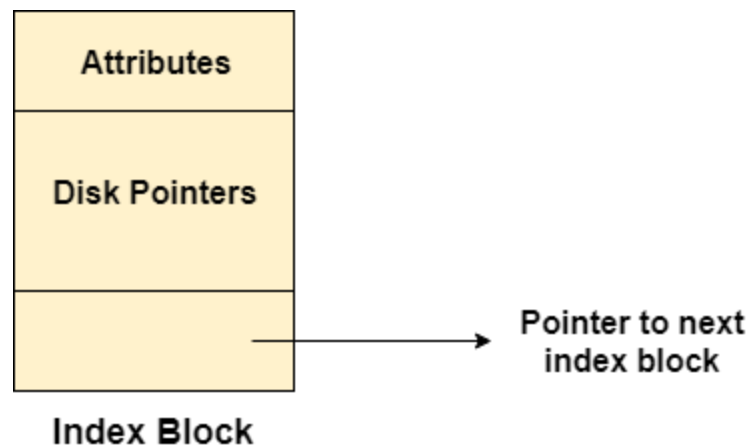
Linked Index Allocation

Single level linked Index Allocation

In index allocation, the file size depends on the size of a disk block. To allow large files, we have to link several index blocks together. In linked index allocation,

- Small header giving the name of the file
- Set of the first 100 block addresses
- Pointer to another index block

For the larger files, the last entry of the index block is a pointer which points to another index block. This is also called as linked schema.



Advantage: It removes file size limitations

Disadvantage: Random Access becomes a bit harder

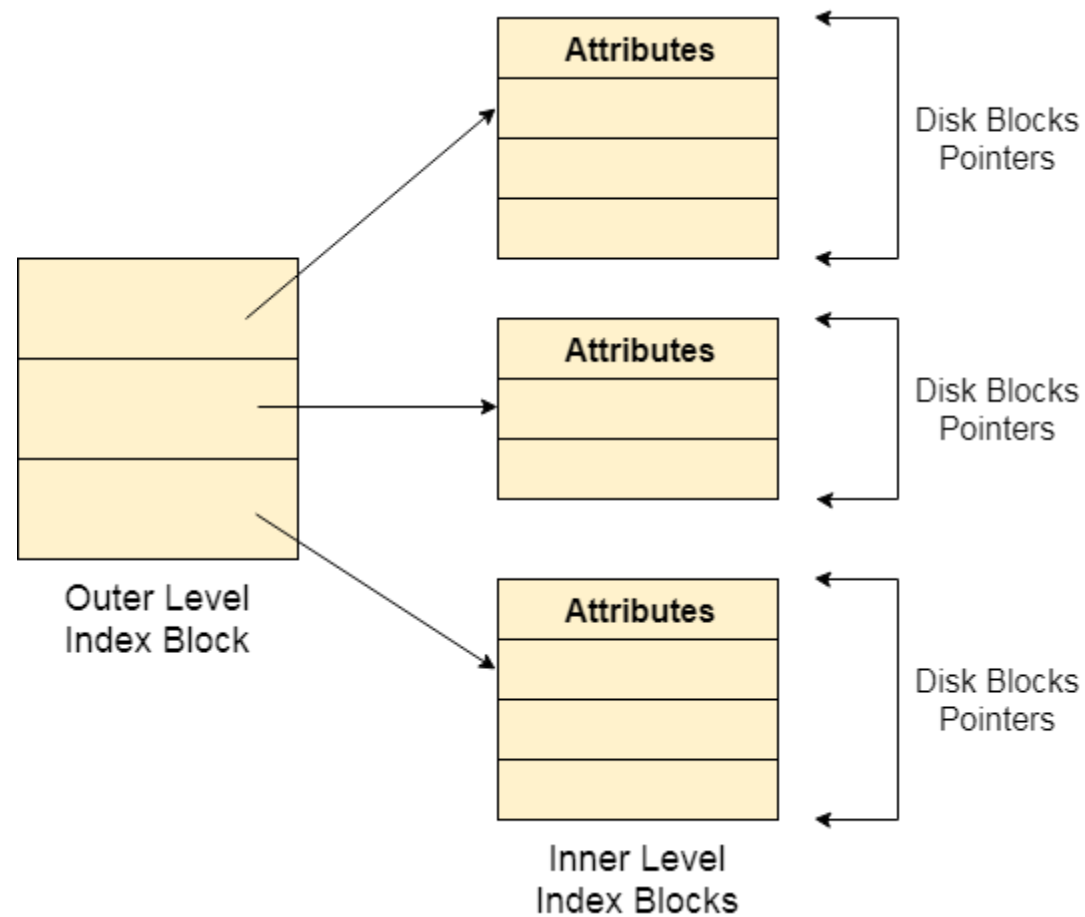
Multilevel Index Allocation

In Multilevel index allocation, we have various levels of indices. There are outer level index blocks which contain the pointers to the inner level index blocks and the inner level index blocks contain the pointers to the file data.

- The outer level index is used to find the inner level index.
- The inner level index is used to find the desired data block.

Advantage: Random Access becomes better and efficient.

Disadvantage: Access time for a file will be higher.



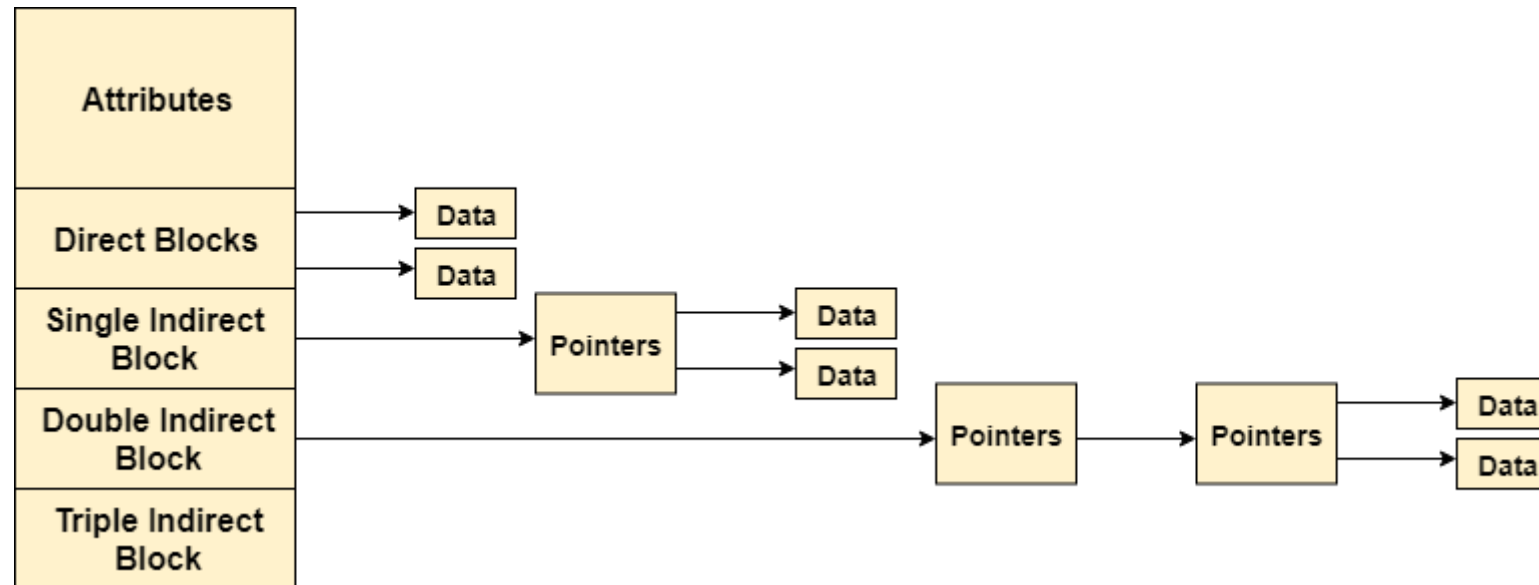
Inode

In UNIX based operating systems, each file is indexed by an Inode. Inode are the special disk block which is created with the creation of the file system. The number of files or directories in a file system depends on the number of Inodes in the file system.

An Inode includes the following information

1. Attributes (permissions, time stamp, ownership details, etc) of the file
2. A number of direct blocks which contains the pointers to first 12 blocks of the file.
3. A single indirect pointer which points to an index block. If the file cannot be indexed entirely by the direct blocks then the single indirect pointer is used.
4. A double indirect pointer which points to a disk block that is a collection of the pointers to the disk blocks which are index blocks. Double index pointer is used if the file is too big to be indexed entirely by the direct blocks as well as the single indirect pointer.

5. A triple index pointer that points to a disk block that is a collection of pointers. Each of the pointers is separately pointing to a disk block which also contains a collection of pointers which are separately pointing to an index block that contains the pointers to the file blocks.



[Next →](#) [← Prev](#)

Free Space Management

A file system is responsible to allocate the free blocks to the file therefore it has to keep track of all the free blocks present in the disk. There are mainly two approaches by using which, the free blocks in the disk are managed.

1. Bit Vector

In this approach, the free space list is implemented as a bit map vector. It contains the number of bits where each bit represents each block.

If the block is empty then the bit is 1 otherwise it is 0. Initially all the blocks are empty therefore each bit in the bit map vector contains 1.

As the space allocation proceeds, the file system starts allocating blocks to the files and setting the respective bit to 0.

2. Linked List

It is another approach for free space management. This approach suggests linking together all the free blocks and keeping a pointer in the cache which points to the first free block.

Therefore, all the free blocks on the disks will be linked together with a pointer. Whenever a block gets allocated, its previous free block will be linked to its next free block.

Disk Scheduling

As we know, a process needs two type of time, CPU time and IO time. For I/O, it requests the Operating system to access the disk.

However, the operating system must be fare enough to satisfy each request and at the same time, operating system must maintain the efficiency and speed of process execution.

The technique that operating system uses to determine the request which is to be satisfied next is called disk scheduling.

Let's discuss some important terms related to disk scheduling.

Seek Time

Seek time is the time taken in locating the disk arm to a specified track where the read/write request will be satisfied.

Rotational Latency

It is the time taken by the desired sector to rotate itself to the position from where it can access the R/W heads.

Transfer Time

It is the time taken to transfer the data.

Disk Access Time

Disk access time is given as,

Disk Access Time = Rotational Latency + Seek Time + Transfer Time

Disk Response Time

It is the average of time spent by each request waiting for the IO operation.

Purpose of Disk Scheduling

The main purpose of disk scheduling algorithm is to select a disk request from the queue of IO requests and decide the schedule when this request will be processed.

Goal of Disk Scheduling Algorithm

- Fairness
- High throughput
- Minimal traveling head time

Disk Scheduling Algorithms

The list of various disks scheduling algorithm is given below. Each algorithm is carrying some advantages and disadvantages. The limitation of each algorithm leads to the evolution of a new algorithm.

- FCFS scheduling algorithm
- SSTF (shortest seek time first) algorithm
- SCAN scheduling
- C-SCAN scheduling
- LOOK Scheduling
- C-LOOK scheduling

FCFS Scheduling Algorithm

It is the simplest Disk Scheduling algorithm. It services the IO requests in the order in which they arrive. There is no starvation in this algorithm, every request is serviced.

Disadvantages

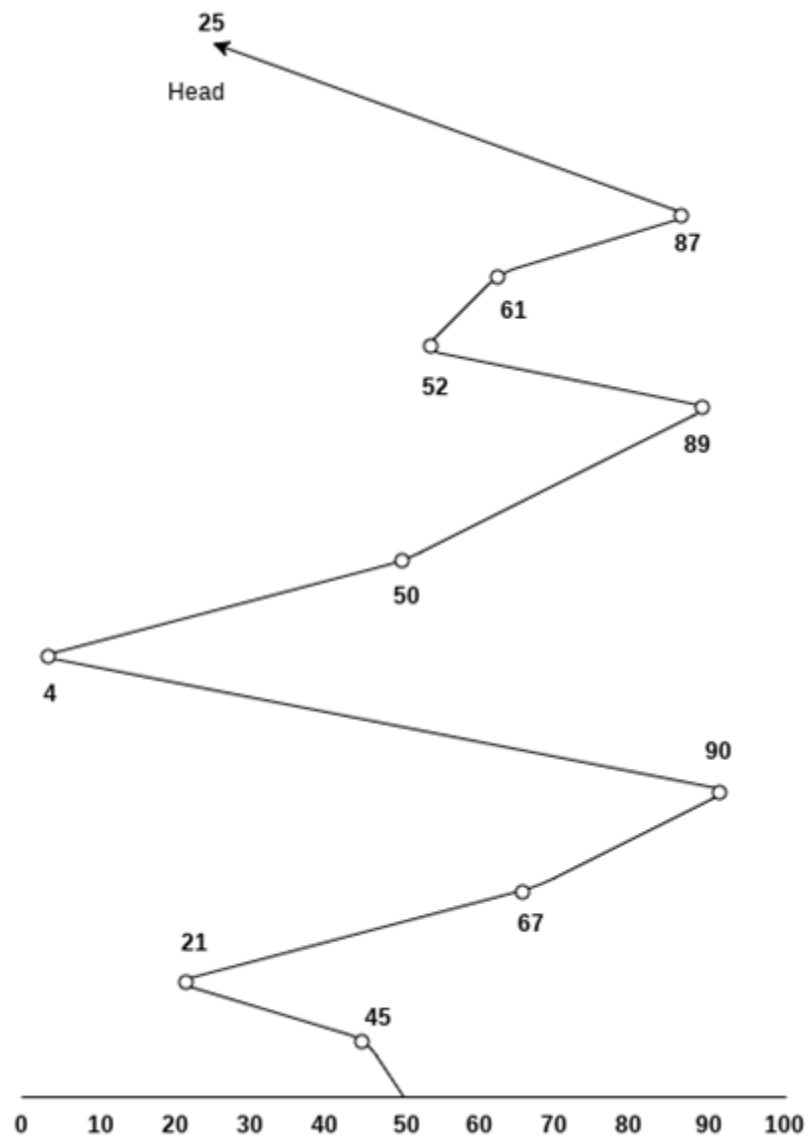
- The scheme does not optimize the seek time.
- The request may come from different processes therefore there is the possibility of inappropriate movement of the head.

Example

Consider the following disk request sequence for a disk with 100 tracks 45, 21, 67, 90, 4, 50, 89, 52, 61, 87, 25

Head pointer starting at 50 and moving in left direction. Find the number of head movements in cylinders using FCFS scheduling.

Solution



Number of cylinders moved by the head

$$= (50-45)+(45-21)+(67-21)+(90-67)+(90-4)+(50-4)+(89-50)+(61-52)+(87-61)+(87-25)$$

$$= 5 + 24 + 46 + 23 + 86 + 46 + 49 + 9 + 26 + 62$$

$$= 376$$

SSTF Scheduling Algorithm

Shortest seek time first (SSTF) algorithm selects the disk I/O request which requires the least disk arm movement from its current position regardless of the direction. It reduces the total seek time as compared to FCFS.

It allows the head to move to the closest track in the service queue.

Disadvantages

- It may cause starvation for some requests.
- Switching direction on the frequent basis slows the working of algorithm.
- It is not the most optimal algorithm.

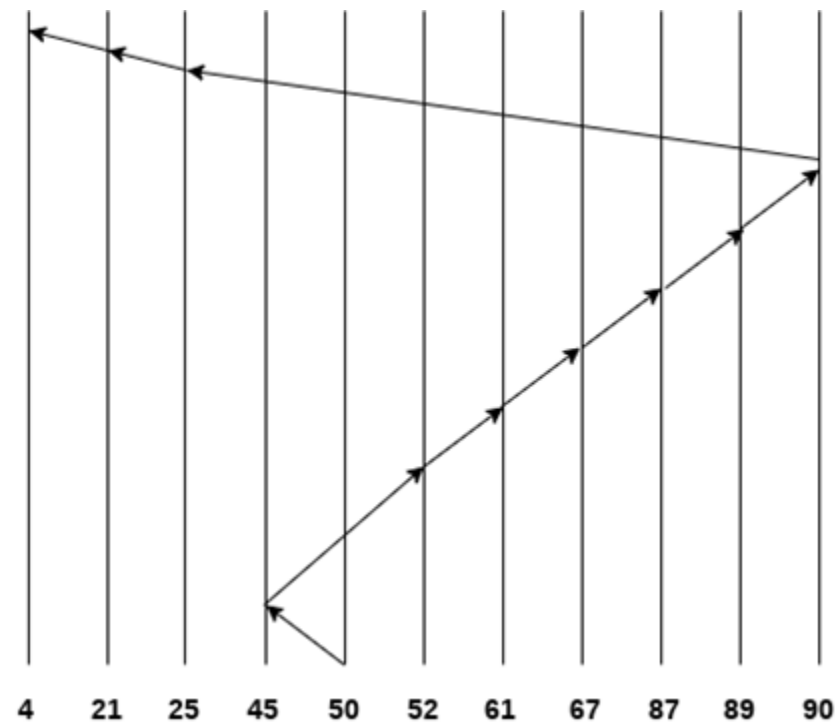
Example

Consider the following disk request sequence for a disk with 100 tracks

45, 21, 67, 90, 4, 89, 52, 61, 87, 25

Head pointer starting at 50. Find the number of head movements in cylinders using SSTF scheduling.

Solution:



Number of cylinders = $5 + 7 + 9 + 6 + 20 + 2 + 1 + 65 + 4 + 17 = 136$

[Next →](#) [← Prev](#)

SCAN and C-SCAN algorithm

Scan Algorithm

It is also called as Elevator Algorithm. In this algorithm, the disk arm moves into a particular direction till the end, satisfying all the requests coming in its path, and then it turns back and moves in the reverse direction satisfying requests coming in its path.

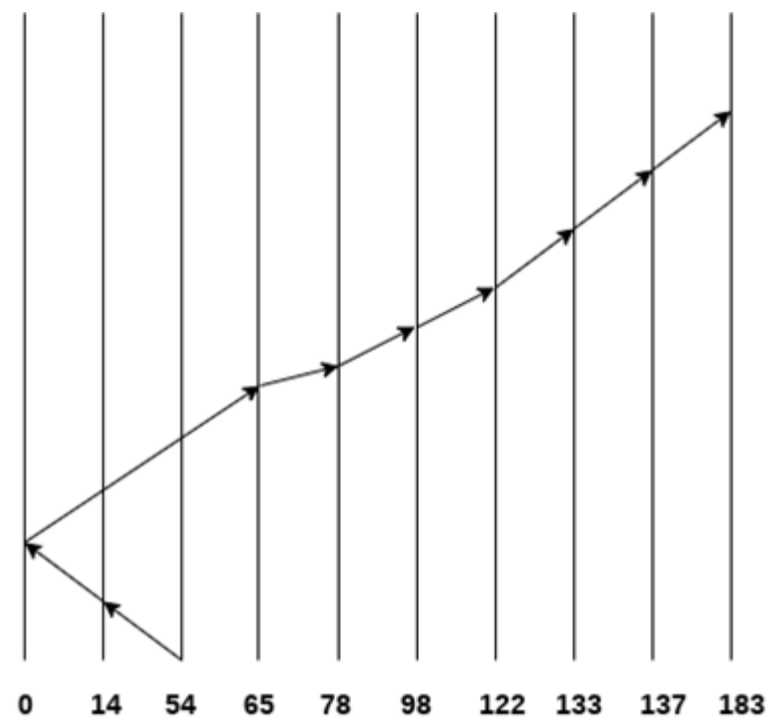
It works in the way an elevator works, elevator moves in a direction completely till the last floor of that direction and then turns back.

Example

Consider the following disk request sequence for a disk with 100 tracks

98, 137, 122, 183, 14, 133, 65, 78

Head pointer starting at 54 and moving in left direction. Find the number of head movements in cylinders using SCAN scheduling.



Number of Cylinders = 40 + 14 + 65 + 13 + 20 + 24 + 11 + 4 + 46 = 237

C-SCAN algorithm

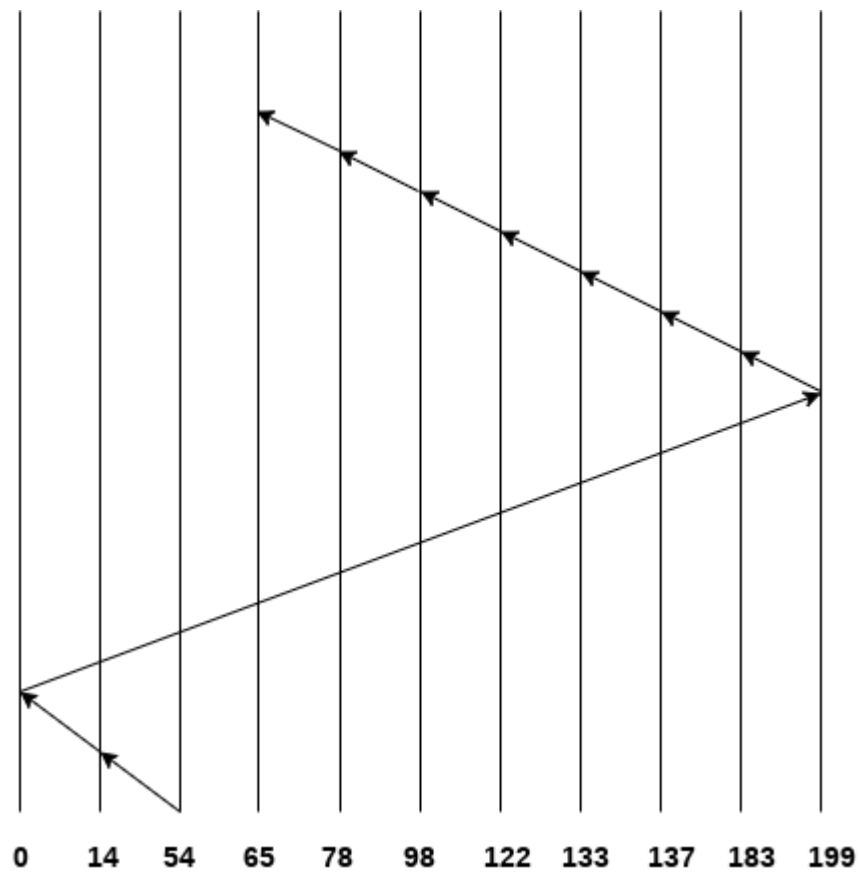
In C-SCAN algorithm, the arm of the disk moves in a particular direction servicing requests until it reaches the last cylinder, then it jumps to the last cylinder of the opposite direction without servicing any request then it turns back and start moving in that direction servicing the remaining requests.

Example

Consider the following disk request sequence for a disk with 100 tracks

98, 137, 122, 183, 14, 133, 65, 78

Head pointer starting at 54 and moving in left direction. Find the number of head movements in cylinders using C-SCAN scheduling.



No. of cylinders crossed = $40 + 14 + 199 + 16 + 46 + 4 + 11 + 24 + 20 + 13 = 387$

[Next →](#) [← Prev](#)

Look Scheduling

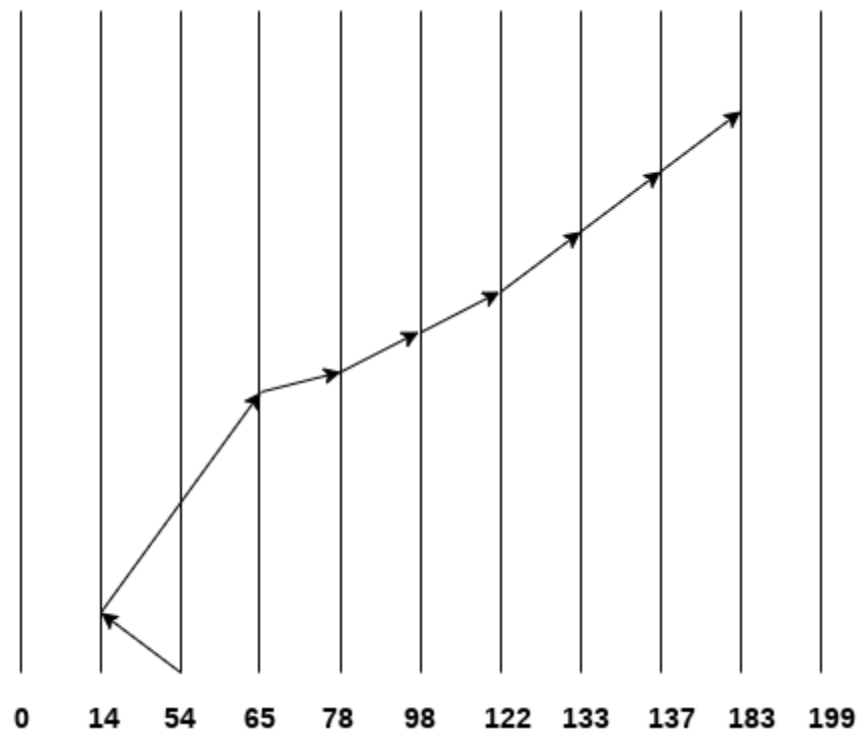
It is like SCAN scheduling Algorithm to some extent except the difference that, in this scheduling algorithm, the arm of the disk stops moving inwards (or outwards) when no more request in that direction exists. This algorithm tries to overcome the overhead of SCAN algorithm which forces disk arm to move in one direction till the end regardless of knowing if any request exists in the direction or not.

Example

Consider the following disk request sequence for a disk with 100 tracks

98, 137, 122, 183, 14, 133, 65, 78

Head pointer starting at 54 and moving in left direction. Find the number of head movements in cylinders using LOOK scheduling.



Number of cylinders crossed = $40 + 51 + 13 + 20 + 24 + 11 + 4 + 46 = 209$

C Look Scheduling

C Look Algorithm is similar to C-SCAN algorithm to some extent. In this algorithm, the arm of the disk moves outwards servicing requests until it reaches the highest request cylinder, then it jumps to the lowest request cylinder without servicing any request then it again start moving outwards servicing the remaining requests.

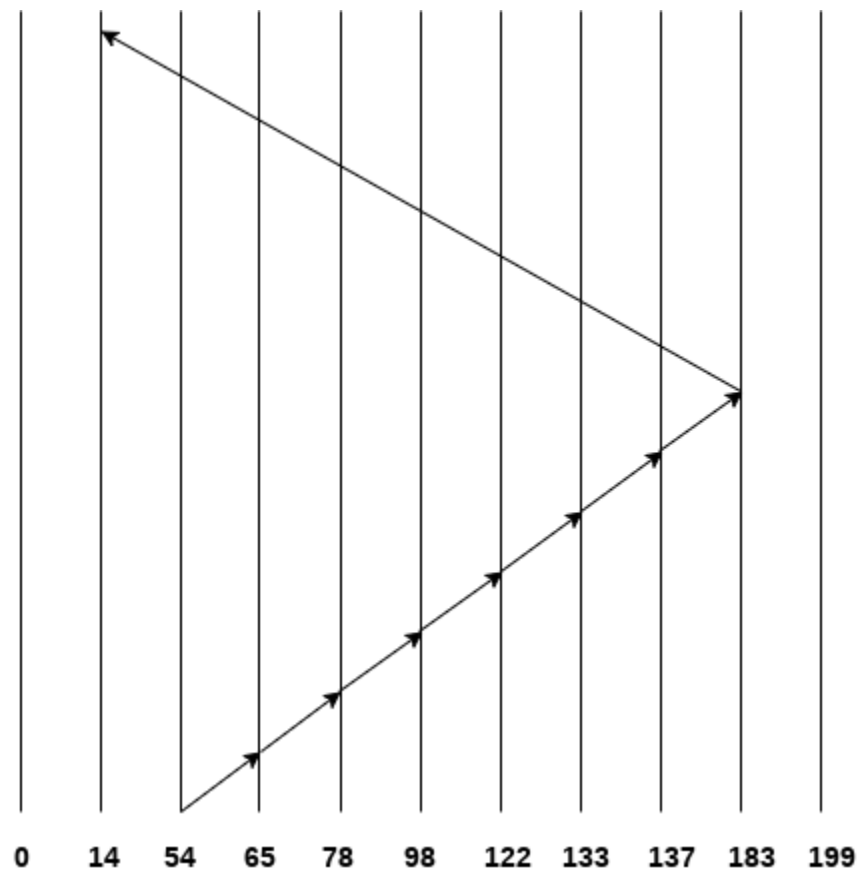
It is different from C SCAN algorithm in the sense that, C SCAN force the disk arm to move till the last cylinder regardless of knowing whether any request is to be serviced on that cylinder or not.

Example

Consider the following disk request sequence for a disk with 100 tracks

98, 137, 122, 183, 14, 133, 65, 78

Head pointer starting at 54 and moving in left direction. Find the number of head movements in cylinders using C LOOK scheduling.



Number of cylinders crossed = $11 + 13 + 20 + 24 + 11 + 4 + 46 + 169 = 298$

Numerical on SSTF and SCAN

Question:

Suppose the following disk request sequence (track numbers) for a disk with 100 tracks is given: 45, 20, 90, 10, 50, 60, 80 and 70. Assume that the initial position of the R/W head is on track 50. The additional distance that will be traversed by the R/W head when the Shortest Seek Time First (SSTF) algorithm is used compared to the SCAN (Elevator) algorithm (assuming that SCAN algorithm moves towards 100 when it starts execution) is _____ tracks

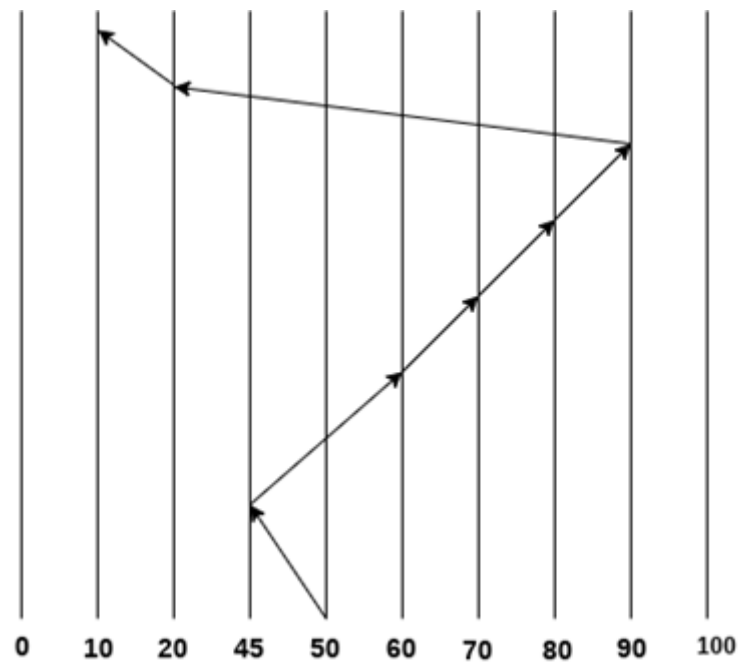
- (A) 5
- (B) 9
- (C) 10
- (D) 11

Using SSTF Algorithm

Number of track are 100.

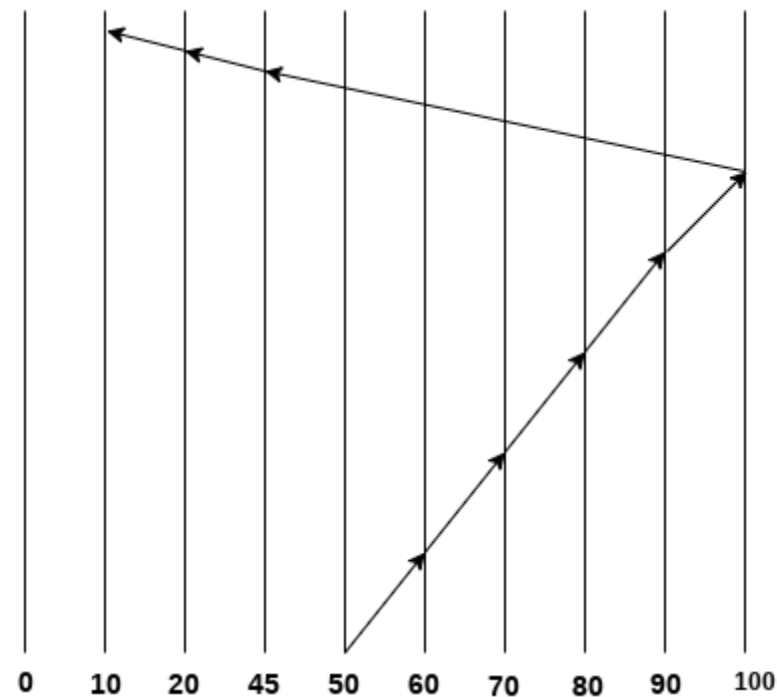
Initial Position of R/W head is 50.

The requests are: 45, 20, 90, 10, 50, 60, 80 and 70



Number of crossed cylinders = $5 + 15 + 10 + 10 + 10 + 70 + 10 = 130$

Using SCAN Algorithm



Number of cylinders crosses = $0 + 10 + 10 + 10 + 10 + 10 + 55 + 20 + 10 = 135$

Therefore the answer is (A). The SCAN algorithm travels for 5 additional tracks.

[Next →](#) [← Prev](#)

Numerical on Disk Scheduling Algorithms

Q. Consider a disk with 200 tracks and the queue has random requests from different processes in the order:

55, 58, 39, 18, 90, 160, 150, 38, 184

Initially arm is at 100. Find the Average Seek length using FIFO, SSTF, SCAN and C-SCAN algorithm.

Solution :

Using FCFS		Using SSTF		Using SCAN		Using C - SCAN	
Next request to be serviced	No. of cylinders travelled	Next request to be serviced	No. of cylinders travelled	Next request to be serviced	No. of cylinders travelled	Next request to be serviced	No. of cylinders travelled
55	45	90	10	150	50	150	50
58	3	58	32	160	10	160	10
39	19	55	3	184	24	184	24
18	21	39	16	90	94	18	166
90	72	38	1	58	32	38	20
160	70	18	20	55	3	39	1
150	10	150	132	39	16	55	16
38	112	160	10	38	1	58	3
184	146	184	24	18	20	90	32
Average Seek Length	$498 / 9 = 55.3$	Average Seek Length	27.5	Average Seek Length	27.8	Average Seek Length	35.6

#7.Misc

Functions of Operation System

An operating system is a program that acts as a user-computer GUI (Graphical user interface). It controls the execution of all types of applications.

The operating system performs the following functions in a device.

1. Instruction
2. Input/output Management
3. Memory Management
4. File Management
5. Processor Management
6. Job Priority

7. Special Control Program
8. Scheduling of resources and jobs
9. Security
10. Monitoring activities
11. Job accounting

Instruction: The [operating system](#) establishes a mutual understanding between the various instructions given by the user.

Input/output Management: What output will come from the input given by the user, the operating system runs this program. This management involves coordinating various input and output devices. It assigns the functions of those devices where one or more applications are executed.

Memory Management: The operating system handles the responsibility of storing any data, system programs, and user programs in memory. This function of the operating system is called memory management.

File Management: The operating system is helpful in making changes in the stored files and in replacing them. It also plays an important role in transferring various files to a device.

Processor Management: The processor is the execution of a program that accomplishes the specified work in that program. It can be defined as an execution unit where a program runs.

Job Priority: The work of job priority is creation and promotion. It determines what action should be done first in a computer system.

Special Control Program: The operating systems make automatic changes to the task through specific control programs. These programs are called Special Control Program.

Scheduling of resources and jobs: The operating system prepares the list of tasks to be performed for the device of the computer system. The [operating system](#) decides which device to use for which task. This action becomes complicated when multiple tasks are to be performed simultaneously in a computer system. The scheduling programs of the operating system determine the order in which tasks are completed. It performs these tasks based on the priority of performing the tasks given by the user. It makes the tasks available based on the priority of the device.

Security: Computer security is a very important aspect of any operating system. The reliability of an operating system is determined by how much better security it provides us. Modern operating systems use a firewall for security. A firewall is a security system that monitors every activity happening in the computer and blocks that activity in case of any threat.

Monitoring activities: The operating system takes care of the activities of the computer system during various processes. This aborts the program if there are errors. The operating system sends instant messages to the user for any unexpected error in the input/output device. It also provides security to the system when the operating system is used in systems operated by multiple users. So that illegal users cannot get data from the system.

Job accounting: It keeps track of time & resources used by various jobs and users.

Mobile Operating System

A mobile operating system is an operating system that helps to run other application software on mobile devices. It is the same kind of software as the famous computer operating systems like Linux and Windows, but now they are light and simple to some extent.

The [operating systems](#) found on smartphones include Symbian OS, iPhone OS, RIM's BlackBerry, [Windows](#) Mobile, Palm WebOS, Android, and Maemo. Android, WebOS, and Maemo are all derived from [Linux](#). The iPhone OS originated from BSD and NeXTSTEP, which are related to Unix.

It combines the beauty of computer and hand use devices. It typically contains a cellular built-in modem and SIM tray for telephony and internet connections. If you buy a mobile, the manufacturer company chooses the OS for that specific device.

Popular platforms of the Mobile OS

- 1. Android OS:** The [Android operating system](#) is the most popular [operating system](#) today. It is a mobile OS based on the **Linux Kernel** and **open-source software**. The android operating system was developed by **Google**. The first Android device was launched in **2008**.
- 2. Bada (Samsung Electronics):** Bada is a Samsung mobile operating system that was launched in 2010. The Samsung wave was the first mobile to use the bada operating system. The bada operating system offers many mobile features, such as 3-D graphics, application installation, and multipoint-touch.
- 3. BlackBerry OS:** The BlackBerry [operating system](#) is a mobile operating system developed by **Research In Motion** (RIM). This operating system was designed specifically for BlackBerry handheld devices. This operating system is beneficial for the corporate users because it provides synchronization with Microsoft Exchange, Novell GroupWise email, Lotus Domino, and other business software when used with the BlackBerry Enterprise Server.
- 4. iPhone OS / iOS:** The iOS was developed by the Apple inc for the use on its device. The iOS operating system is the most popular operating system today. It is a very secure operating system. The iOS operating system is not available for any other mobiles.
- 5. Symbian OS:** Symbian operating system is a mobile operating system that provides a high-level of integration with communication. The Symbian operating system is based on the java language. It combines middleware of wireless communications and personal information management (PIM) functionality. The Symbian operating system was developed by **Symbian Ltd** in **1998** for the use of mobile phones. **Nokia** was the first company to release Symbian OS on its mobile phone at that time.
- 6. Windows Mobile OS:** The window mobile OS is a mobile operating system that was developed by **Microsoft**. It was designed for the pocket PCs and smart mobiles.
- 7. Harmony OS:** The harmony operating system is the latest mobile operating system that was developed by Huawei for the use of its devices. It is designed primarily for IoT devices.
- 8. Palm OS:** The palm operating system is a mobile operating system that was developed by **Palm Ltd** for use on personal digital assistants (PADs). It was introduced in **1996**. Palm OS is also known as the **Garnet OS**.
- 9. WebOS (Palm/HP):** The WebOS is a mobile operating system that was developed by **Palm**. It based on the **Linux Kernel**. The HP uses this operating system in its mobile and touchpads.

Swapping in Operating System

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

The purpose of the swapping in [operating system](#) is to access the data present in the hard disk and bring it to [RAM](#) so that the application programs can use it. The thing to remember is that swapping is used only when data is not present in [RAM](#).

Although the process of swapping affects the performance of the system, it helps to run larger and more than one process. This is the reason why swapping is also referred to as memory compaction.

The concept of swapping has divided into two more concepts: Swap-in and Swap-out.

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

Example: Suppose the user process's size is 2048KB and is a standard hard disk where swapping has a data transfer rate of 1Mbps. Now we will calculate how long it will take to transfer from main memory to secondary memory.

1. User process size is 2048Kb
2. Data transfer rate is 1Mbps = 1024 kbps
3. Time = process size / transfer rate
4. = 2048 / 1024
5. = 2 seconds
6. = 2000 milliseconds
7. Now taking swap-in and swap-out time, the process will take 4000 milliseconds.

Advantages of Swapping

1. It helps the CPU to manage multiple processes within a single main memory.
2. It helps to create and use virtual memory.

3. Swapping allows the CPU to perform multiple tasks simultaneously. Therefore, processes do not have to wait very long before they are executed.
4. It improves the main memory utilization.

Disadvantages of Swapping

1. If the computer system loses power, the user may lose all information related to the program in case of substantial swapping activity.
2. If the swapping algorithm is not good, the composite method can increase the number of Page Fault and decrease the overall processing performance.

Note:

- In a single tasking operating system, only one process occupies the user program area of memory and stays in memory until the process is complete.
- In a multitasking operating system, a situation arises when all the active processes cannot coordinate in the main memory, then a process is swap out from the main memory so that other processes can enter it.

Threads in Operating System

There is a way of thread execution inside the process of any operating system. Apart from this, there can be more than one thread inside a process. Thread is often referred to as a lightweight process.

The process can be split down into so many threads. For example, in a [browser](#), many tabs can be viewed as threads. MS Word uses many threads - formatting text from one thread, processing input from another thread, etc.

Types of Threads

In the [operating system](#), there are two types of threads.

1. Kernel level thread.
2. User-level thread.

User-level thread

The [operating system](#) does not recognize the user-level thread. User threads can be easily implemented and it is implemented by the user. If a user performs a user-level thread blocking operation, the whole process is blocked. The kernel level thread does not know nothing about the user level thread. The kernel-level thread manages user-level threads as if they are single-threaded processes?examples: [Java](#) thread, POSIX threads, etc.

Advantages of User-level threads

1. The user threads can be easily implemented than the kernel thread.
2. User-level threads can be applied to such types of operating systems that do not support threads at the kernel-level.
3. It is faster and efficient.
4. Context switch time is shorter than the kernel-level threads.
5. It does not require modifications of the operating system.
6. User-level threads representation is very simple. The register, PC, stack, and mini thread control blocks are stored in the address space of the user-level process.
7. It is simple to create, switch, and synchronize threads without the intervention of the process.

Disadvantages of User-level threads

1. User-level threads lack coordination between the thread and the kernel.
2. If a thread causes a page fault, the entire process is blocked.

Kernel level thread

The kernel thread recognizes the operating system. There are a thread control block and process control block in the system for each thread and process in the kernel-level thread. The kernel-level thread is implemented by the operating system. The kernel knows about all the threads and manages them. The kernel-level thread offers a system call to create and manage the threads from user-space. The implementation of kernel threads is difficult than the user thread. Context switch time is longer in the kernel thread. If a kernel thread performs a blocking operation, the Banky thread execution can continue. Example: Window Solaris.

Advantages of Kernel-level threads

1. The kernel-level thread is fully aware of all threads.
2. The scheduler may decide to spend more CPU time in the process of threads being large numerical.
3. The kernel-level thread is good for those applications that block the frequency.

Disadvantages of Kernel-level threads

1. The kernel thread manages and schedules all threads.
2. The implementation of kernel threads is difficult than the user thread.
3. The kernel-level thread is slower than user-level threads.

Components of Threads

Any thread has the following components.

1. Program counter
2. Register set
3. Stack space

Benefits of Threads

- **Enhanced throughput of the system:** When the process is split into many threads, and each thread is treated as a job, the number of jobs done in the unit time increases. That is why the throughput of the system also increases.
- **Effective Utilization of Multiprocessor system:** When you have more than one thread in one process, you can schedule more than one thread in more than one processor.
- **Faster context switch:** The context switching period between threads is less than the process context switching. The process context switch means more overhead for the CPU.
- **Responsiveness:** When the process is split into several threads, and when a thread completes its execution, that process can be responded to as soon as possible.
- **Communication:** Multiple-thread communication is simple because the threads share the same address space, while in process, we adopt just a few exclusive communication strategies for communication between two processes.
- **Resource sharing:** Resources can be shared between all threads within a process, such as code, data, and files. Note: The stack and register cannot be shared between threads. There are a stack and register for each thread.

[Next →](#) [← Prev](#)

Fedora Operating System

Fedora operating system is an open-source operating system that is based on the Linux OS kernel architecture. A group of developers was developed the Fedora operating system under the Fedora Project. It is sponsored by Red Hat. It is designed as a secure operating system for the general-purpose. Fedora [operating system](#) offers a suite of virus protection, system tools, office productivity services, media playback, and other desktop application.

According to the Fedora Project, it is always free to use, modify, and distribute. Fedora OS is integrated with applications and packaged software. This operating system enhances the abilities of the software. It offers the same consistency, procedures, and functionality as a traditional [OS](#). Fedora operating system is the second most commonly used distribution of [Linux](#) after Ubuntu.

There are over 100 distributions based on the Fedora operating system, including the XO operating system of Red Hat Enterprise Linux.

Features of Fedora Operating System

List of the Fedora OS features:

- Fedora OS offers many architectures.
- Fedora OS is a very reliable and stable operating system.
- It provides unique security features.
- Fedora OS provides a very powerful firewall.
- Fedora OS is very easy to use.
- It supports a large community.
- Fedora OS is actively developed.
- Fedora OS is an open-source OS.
- The interface of Fedora OS is very attractive.
- This operating system offers live mode tools.
- This operating system enhances internet speed.

Fedora OS comes with many pre-installed applications and tools, such as [Internet Browser](#), PDF and Word files Viewer, Pre-installed Games, Libre Office Suite, Programming language Support, etc.

Fedora is a very stable, secure, and light-weight operating system. It supports different types of architectures, such as IBM Z, AMD x86-x64, Intel i686, IBM Power64le, ARM-hfp, MIPS-64el, ARM AArch64, IBM Power64, etc. Usually, it also works on the latest Linux kernel.

Fedora Server

Fedora Server is a very flexible and powerful OS. It keeps all your infrastructure and services under your control. Fedora operating system offers the latest data center technologies.

Advantages of Fedora Operating System

1. Fedora OS is a very reliable and stable operating system.
2. It enhances the security in this operating system.
3. It offers many graphical tools.
4. This operating system updates automatically.
5. This OS supports many file formats.
6. It also offers many education software.
7. It supports a large community.
8. It provides unique security features.

Disadvantages of Fedora Operating System

1. It requires a long time to set up.
2. It requires additional software tools for the server.
3. It does not provide any standard model for multi-file objects.
4. Fedora has its own server, so we can't work on another server in real-time.

Uses of Operating System

The operating system is used everywhere today, such as banks, schools, hospitals, companies, mobiles, etc. No device can operate without an operating system because it controls all the user's commands.

- **LINUX/UNIX** operating system is used in the bank because it is a very secure operating system.
- **Symbian OS, Windows Mobile, iOS, and Android OS** are used in mobile phone operating systems as these operating systems are a lightweight operating system.

Features of Operating System

The [operating system](#) has many notable features that are developing day by day. The growth of the operating system is commendable as it was developed in 1950 to handle storage tape. It acts as an interface. The features of [operating system](#) are given below.

- Error detection and handling
- Handling I/O operations
- Virtual Memory Multitasking
- Program Execution
- Allows disk access and file systems
- Memory management
- Protected and supervisor mode
- Security
- Resource allocation
- Easy to run
- Information and Resource Protection
- Manipulation of the file system

Characteristics of Operating System

Memory Management: The operating system manages memory. It has complete knowledge of primary memory; which part of the memory is used by which program. Whenever a program requests, it allocates memory.

Processor Management: It allocates the program to the processor (CPU) and also deallocates it when a program runs out of the [CPU](#) needs.

Device Management: The operating system keeps the information about all devices. It is also called the I/O controller, and the operating system also decides which devices are used to which program, when, and for how long.

Security: It prevents unauthorized access to any program. It uses passwords and other technologies.

Reliability: It is very reliable because no any virus and harmful code can be detected in it.

File Management: It allocates and deallocates resources and decides which program to allocate resources.

Easy to use: It can be easily used as it also has a GUI interface.

Needs of Operating System

The following points indicate the need for the operating system:

- More than one program runs at a time in a computer, and all of them require your computer's CPU and memory. The operating system manages resources for all those programs. That is why the operating system is required.
- Multitasking is a very critical feature of the OS. With its help, we can run many programs simultaneously.
- The operating system provides a platform to run any application program in the computer. Due to which we can do our work with the help of that application.
- It helps the user in file management. Through this, the user can save the data according to his needs.
- You use your mouse to open the application and click on the menu. All this is possible due to the modern operating system. This operating system allows you to do this with the help of GUI (Graphical user interface).
- The operating system creates a communication link between the user and the computer, allowing the user to run any application program and obtain the required output properly.
- It is almost impossible for a user to use a computer system without an operating system. Many processes run simultaneously when a program is executed, which is not easy for a person to manage.

[Next →](#) [← Prev](#)

Producer-Consumer problem

The Producer-Consumer problem is a classical multi-process synchronization problem, that is we are trying to achieve synchronization between more than one process.

There is one Producer in the producer-consumer problem, Producer is producing some items, whereas there is one Consumer that is consuming the items produced by the Producer. The same memory buffer is shared by both producers and consumers which is of fixed-size.

The task of the Producer is to produce the item, put it into the memory buffer, and again start producing items. Whereas the task of the Consumer is to consume the item from the memory buffer.

Let's understand what is the problem?

Below are a few points that considered as the problems occur in Producer-Consumer:

- The producer should produce data only when the buffer is not full. In case it is found that the buffer is full, the producer is not allowed to store any data into the memory buffer.
- Data can only be consumed by the consumer if and only if the memory buffer is not empty. In case it is found that the buffer is empty, the consumer is not allowed to use any data from the memory buffer.
- Accessing memory buffer should not be allowed to producer and consumer at the same time.

Let's see the code for the above problem:

Producer Code

```
int count = 0;
void producer( void )
{
    int itemP;
    while(1)
    {
        Produce_item(item P)
        while( count == n);
        buffer[ in ] = item P;
        in = (in + 1)mod n
        count = count + 1;
    }
}
```

Load Rp, m[count]
Increment Rp
Store m[count], Rp

Producer Code

```
int count;
void consumer(void)
{
    int itemC;
    while( 1 )
    {
        while( count == 0 );
        itemC = buffer[ out ];
        out = ( out + 1 ) mod n;
        count = count -1;
    }
}
```

Load Rc, m[count]
Decrement Rc
Store m[count], Rc

Let's understand above Producer and Consumer code:

Before Starting an explanation of code, first, understand the few terms used in the above code:

1. **"in"** used in a producer code represent the next **empty buffer**
2. **"out"** used in consumer code represent first **filled buffer**
3. count keeps the count number of elements in the buffer
4. count is further divided into 3 lines code represented in the block in both the producer and consumer code.

If we talk about Producer code first:

--Rp is a register which keeps the value of m[count]

--Rp is incremented (As element has been added to buffer)

--an Incremented value of Rp is stored back to m[count]

Similarly, if we talk about Consumer code next:

--Rc is a register which keeps the value of m[count]

--Rc is decremented (As element has been removed out of buffer)

--the decremented value of Rc is stored back to m[count].

BUFFER

(pointer to the next empty space) → in = 5
(pointer to the first filled space) → out = 0
(number of elements in Buffer) → count = 5
(Size of Buffer) → n = 8

0	A
1	B
2	C
3	D
4	E
5	
6	
7	

As we can see from Fig: Buffer has total 8 spaces out of which the first 5 are filled, in = 5(pointing next empty position) and out = 0(pointing first filled position).

Let's start with **the producer** who wanted to produce an element " F ", according to code it will enter into the producer() function, while(1) will always be true, itemP = F will be tried to insert into the buffer, before that while(count == n); will evaluate to be False.

Note: Semicolon after while loop will not let the code to go ahead if it turns out to be True(i.e. infinite loop/ Buffer is full)

Buffer[in] = itemP → Buffer[5] = F. (F is inserted now)

in = (in + 1) mod n → (5 + 1) mod 8 → 6, therefore in = 6; (next empty buffer)

After insertion of F, Buffer looks like this

Where **out = 0**, but **in = 6**

0	A
1	B
2	C
3	D
4	E
5	F
6	
7	

Since $\text{count} = \text{count} + 1$; is divided into three parts:

Load $R_p, m[\text{count}] \rightarrow$ will copy count value which is 5 to register R_p .

Increment $R_p \rightarrow$ will increment R_p to 6.

Suppose just after Increment and before the execution of third line (store $m[\text{count}], R_p$) **Context Switch** occurs and code jumps to **consumer code**. . .

Consumer Code:

Now starting **consumer** who wanted to consume the first element " A ", according to code it will enter into the consumer() function, while(1) will always be true, while(count == 0); will evaluate to be False(since the count is still 5, which is not equal to 0.

Note: Semicolon after while loop will not let the code to go ahead if it turns out to be True(i.e. infinite loop/ no element in buffer)

$\text{itemC} = \text{Buffer}[\text{out}] \rightarrow \text{itemC} = \text{A}$ (since out is 0)

$\text{out} = (\text{out} + 1) \bmod n \rightarrow (0 + 1) \bmod 8 \rightarrow 1$, therefore out = 1(first filled position)

A is removed now

After removal of A, Buffer look like this

Where **out = 1**, and **in = 6**

0	
1	B
2	C
3	D
4	E
5	F
6	
7	

Since $\text{count} = \text{count} - 1$; is divided into three parts:

Load Rc, m[count] → will copy count value which is 5 to register Rp.

Decrement Rc → will decrement Rc to 4.

store m[count], Rc → count = 4.

Now the current value of count is 4

Suppose after this **Context Switch** occurs back to the leftover part of producer code. . .

Since context switch at producer code was occurred after Increment and before the execution of the third line (store m[count], Rp)

So we resume from here since Rp holds 6 as incremented value

Hence store m[count], Rp → count = 6

Now the current value of count is 6, which is wrong as Buffer has only 5 elements, this condition is known as Race Condition and Problem is Producer-Consumer Problem.

The solution of Producer-Consumer Problem using Semaphore

The above problems of Producer and Consumer which occurred due to context switch and producing inconsistent result can be solved with the help of semaphores.

To solve the problem occurred above of race condition, we are going to use **Binary Semaphore and Counting Semaphore**

Binary Semaphore: In Binary Semaphore, only two processes can compete to enter into its **CRITICAL SECTION** at any point in time, apart from this the condition of mutual exclusion is also preserved.

Counting Semaphore: In counting semaphore, more than two processes can compete to enter into its **CRITICAL SECTION** at any point of time apart from this the condition of mutual exclusion is also preserved.

Semaphore: A semaphore is an integer variable in S, that apart from initialization is accessed by only two standard atomic operations - wait and signal, whose definitions are as follows:

```
1. 1. wait( S )
2. {
3.  while( S <= 0 ) ;
4.  S--;
5. }
```

```
1. 2. signal( S )
2. {
3.  S++;
4. }
```

From the above definitions of wait, it is clear that if the value of $S \leq 0$ then it will enter into an infinite loop (because of the semicolon; after while loop). Whereas the job of the signal is to increment the value of S.

Let's see the code as a solution of producer and consumer problem using semaphore (Both Binary and Counting Semaphore):

Producer Code- solution

```
1. void producer( void )
```

```

2. {
3.  wait ( empty );
4.  wait(S);
5.  Produce_item(item P)
6.  buffer[ in ] = item P;
7.  in = (in + 1)mod n
8.  signal(S);
9.  signal(full);
10.
11.}

```

Consumer Code- solution

```

1. void consumer(void)
2. {
3.  wait ( empty );
4.  wait(S);
5.  itemC = buffer[ out ];
6.  out = ( out + 1 ) mod n;
7.  signal(S);
8.  signal(empty);
9. }

```

Let's understand the above Solution of Producer and Consumer code:

Before Starting an explanation of code, first, understand the few terms used in the above code:

1. "in" used in a producer code represent the next **empty buffer**
2. "out" used in consumer code represent first **filled buffer**
3. "empty" is counting semaphore which keeps a score of no. of empty buffer
4. "full" is counting semaphore which scores of no. of full buffer
5. "S" is a binary semaphore **BUFFER**

If we see the current situation of Buffer



S = 1(init. Value of Binary semaphore)

in = 5(next empty buffer)

out = 0(first filled buffer)

0	A
1	B
2	C
3	D
4	E
5	
6	
7	

As we can see from Fig: Buffer has total 8 spaces out of which the first 5 are filled, in = 5(pointing next empty position) and out = 0(pointing first filled position).

Semaphores used in Producer Code:

6. wait(empty) will decrease the value of the counting semaphore variable **empty** by 1, that is when the producer produces some element then the value of the space gets automatically decreased by one in the buffer. In case the buffer is full, that is the value of the counting semaphore variable "empty" is 0, then wait(empty); will trap the process (as per definition of wait) and does not allow to go further.

7. wait(S) decreases the binary semaphore variable S to 0 so that no other process which is willing to enter into its critical section is allowed.

8. signal(s) increases the binary semaphore variable S to 1 so that other processes who are willing to enter into its critical section can now be allowed.

9. signal(full) increases the counting semaphore variable full by 1, as on adding the item into the buffer, one space is occupied in the buffer and the variable full must be updated.

Semaphores used in Producer Code:

10.0wait(full) will decrease the value of the counting semaphore variable full by 1, that is when the consumer consumes some element then the value of the full space gets automatically decreased by one in the buffer. In case the buffer is empty, that is the value of the counting semaphore variable full is 0, then wait(full); will trap the process(as per definition of wait) and does not allow to go further.

11. wait(S) decreases the binary semaphore variable S to 0 so that no other process which is willing to enter into its critical section is allowed.

12. signal(S) increases the binary semaphore variable S to 1 so that other processes who are willing to enter into its critical section can now be allowed.

13. signal(empty) increases the counting semaphore variable empty by 1, as on removing an item from the buffer, one space is vacant in the buffer and the variable empty must be updated accordingly.

Producer Code:

Let's start with producer() who wanted to produce an element " F ", according to code it will enter into the producer() function.

wait(empty); will decrease the value of empty by one, i.e. empty = 2

Suppose just after this context switch occurs and jumps to consumer code.

Consumer Code:

Now starting consumer who wanted to consume first element " A ", according to code it will enter into consumer() function,

wait(full); will decrease the value of full by one, i.e. full = 4

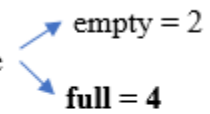
wait (S); will decrease the value of S to 0

itemC = Buffer[out]; → itemC = A (since out is 0)

A is removed now

$\text{out} = (\text{out} + 1) \bmod n \rightarrow (0 + 1) \bmod 8 \rightarrow 1$, therefore $\text{out} = 1$ (first filled position)

current situation of Buffer

Counting semaphore 

0	
1	B
2	C
3	D
4	E
5	
6	
7	

$S = 0$ (Value of Binary semaphore)

$\text{in} = 5$ (next empty buffer)

$\text{out} = 1$ (first filled buffer)

Suppose just after this context, switch occurs back to producer code

Since the next instruction of `producer()` is `wait(S);`, this will trap the producer process, as the current value of S is 0, and `wait(0);` is an infinite loop: as per the definition of wait, hence producer cannot move further.

Therefore, we move back to the consumer process next instruction.

`signal(S);` will now increment the value of S to 1.

`signal(empty);` will increment `empty` by 1, i.e. $\text{empty} = 3$

current situation of Buffer

Counting semaphore \nearrow empty = 3
 \searrow full = 4

$S = 1$ (Value of Binary semaphore)
in = 5 (next empty buffer)
out = 1 (first filled buffer)

0	
1	B
2	C
3	D
4	E
5	
6	
7	

Now moving back to producer() code;

Since the next instruction of producer() is wait(S); will successfully execute, as S is now 1 and it will decrease the value of S by 1, i.e. $S = 0$

Buffer[in] = itemP; \rightarrow Buffer[5] = F. (F is inserted now)

in = (in + 1) mod n \rightarrow (5 + 1) mod 8 \rightarrow 6, therefore in = 6; (next empty buffer)

signal(S); will increment S by 1,

signal(full); will increment full by 1, i.e. full = 5

current situation of Buffer

Counting semaphore \nearrow empty = 3
 \searrow full = 5

$S = 1$ (Value of Binary semaphore)
in = 6 (next empty buffer)
out = 1 (first filled buffer)

0	
1	B
2	C
3	D
4	E
5	F
6	
7	

Now add current value of full and empty, i.e. full + empty = 5 + 3 = 8 (which is absolutely fine) No inconsistent result is generated even after so many context switches. But in the previous condition of producer and consumer without semaphore, we see the inconsistent result in case of context switches.

This is the solution to the Producer consumer problem.

THE DINING PHILOSOPHERS PROBLEM

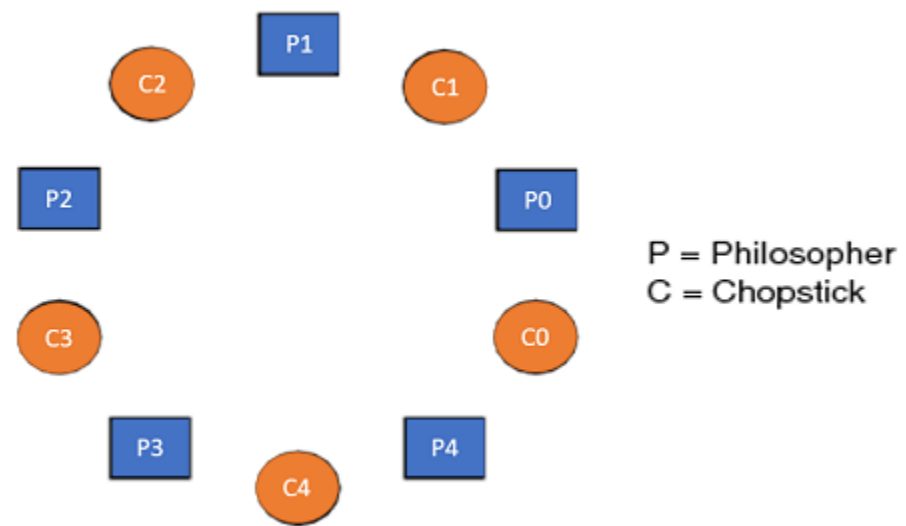
The dining philosopher's problem is the classical problem of synchronization which says that Five philosophers are sitting around a circular table and their job is to think and eat alternatively. A bowl of noodles is placed at the center of the table along with five chopsticks for each of the philosophers. To eat a philosopher needs both their right and a left chopstick. A philosopher can only eat if both immediate left and right chopsticks of the philosopher is available. In case if both immediate left and right chopsticks of the philosopher are not available then the philosopher puts down their (either left or right) chopstick and starts thinking again.

The dining philosopher demonstrates a large class of concurrency control problems hence it's a classic synchronization problem.



Five Philosophers sitting around the table

Dining Philosophers Problem- Let's understand the Dining Philosophers Problem with the below code, we have used fig 1 as a reference to make you understand the problem exactly. The five Philosophers are represented as P0, P1, P2, P3, and P4 and five chopsticks by C0, C1, C2, C3, and C4.



```
1. Void Philosopher
2. {
3.   while(1)
4.   {
5.     take_chopstick[i];
6.     take_chopstick[ (i+1) % 5] ;
7.     . .
8.     . EATING THE NOODLE
9.     .
10.    put_chopstick[i] );
11.    put_chopstick[ (i+1) % 5] ;
12.    .
13.    . THINKING
14.  }
15.}
```

Let's discuss the above code:

Suppose Philosopher P0 wants to eat, it will enter in Philosopher() function, and execute **take_chopstick[i];** by doing this it holds **C0 chopstick** after that it execute **take_chopstick[(i+1) % 5];** by doing this it holds **C1 chopstick**(since $i = 0$, therefore $(0 + 1) \% 5 = 1$)

Similarly suppose now Philosopher P1 wants to eat, it will enter in Philosopher() function, and execute **take_chopstick[i];** by doing this it holds **C1 chopstick** after that it execute **take_chopstick[(i+1) % 5];** by doing this it holds **C2 chopstick**(since $i = 1$, therefore $(1 + 1) \% 5 = 2$)

But Practically Chopstick C1 is not available as it has already been taken by philosopher P0, hence the above code generates problems and produces race condition.

The solution of the Dining Philosophers Problem

We use a semaphore to represent a chopstick and this truly acts as a solution of the Dining Philosophers Problem. Wait and Signal operations will be used for the solution of the Dining Philosophers Problem, for picking a chopstick wait operation can be executed while for releasing a chopstick signal semaphore can be executed.

Semaphore: A semaphore is an integer variable in S, that apart from initialization is accessed by only two standard atomic operations - wait and signal, whose definitions are as follows:

```
1. 1. wait( S )
2. {
3.  while( S <= 0 ) ;
4.  S--;
5. }
6.
7. 2. signal( S )
8. {
9.  S++;
10.}
```

From the above definitions of wait, it is clear that if the value of $S \leq 0$ then it will enter into an infinite loop(because of the semicolon; after while loop). Whereas the job of the signal is to increment the value of S.

The structure of the chopstick is an array of a semaphore which is represented as shown below -

1. semaphore C[5];

Initially, each element of the semaphore C0, C1, C2, C3, and C4 are initialized to 1 as the chopsticks are on the table and not picked up by any of the philosophers.

Let's modify the above code of the Dining Philosopher Problem by using semaphore operations wait and signal, the desired code looks like

1. **void** Philosopher
2. {
3. **while**(1)
4. {
5. Wait(take_chopstickC[i]);
6. Wait(take_chopstickC[(i+1) % 5]) ;
7. ..
8. . EATING THE NOODLE
9. .
10. Signal(put_chopstickC[i]);
11. Signal(put_chopstickC[(i+1) % 5]) ;
12. .
13. . THINKING
14. }
15. }

In the above code, first wait operation is performed on take_chopstickC[i] and take_chopstickC [(i+1) % 5]. This shows philosopher i have picked up the chopsticks from its left and right. The eating function is performed after that.

On completion of eating by philosopher i the, signal operation is performed on take_chopstickC[i] and take_chopstickC [(i+1) % 5]. This shows that the philosopher i have eaten and put down both the left and right chopsticks. Finally, the philosopher starts thinking again.

Let's understand how the above code is giving a solution to the dining philosopher problem?

Let value of $i = 0$ (initial value), Suppose Philosopher P0 wants to eat, it will enter in Philosopher() function, and execute **Wait(take_chopstickC[i]);** by doing this it holds **C0 chopstick** and reduces semaphore C0 to 0, after that it execute **Wait(take_chopstickC[(i+1) % 5]);** by doing this it holds **C1 chopstick** (since $i = 0$, therefore $(0 + 1) \% 5 = 1$) and reduces semaphore C1 to 0

Similarly, suppose now Philosopher P1 wants to eat, it will enter in Philosopher() function, and execute **Wait(take_chopstickC[i]);** by doing this it will try to hold **C1 chopstick** but will not be able to do that, since the value of semaphore C1 has already been set to 0 by philosopher P0, therefore it will enter into an infinite loop because of which philosopher P1 will not be able to pick chopstick C1 whereas if Philosopher P2 wants to eat, it will enter in Philosopher() function, and execute **Wait(take_chopstickC[i]);** by doing this it holds **C2 chopstick** and reduces semaphore C2 to 0, after that, it executes **Wait(take_chopstickC[(i+1) % 5]);** by doing this it holds **C3 chopstick** (since $i = 2$, therefore $(2 + 1) \% 5 = 3$) and reduces semaphore C3 to 0.

Hence the above code is providing a solution to the dining philosopher problem, A philosopher can only eat if both immediate left and right chopsticks of the philosopher are available else philosopher needs to wait. Also at one go two independent philosophers can eat simultaneously (i.e., philosopher **P0 and P2, P1 and P3 & P2 and P4** can eat simultaneously as all are the independent processes and they are following the above constraint of dining philosopher problem)

The drawback of the above solution of the dining philosopher problem

From the above solution of the dining philosopher problem, we have proved that no two neighboring philosophers can eat at the same point in time. The drawback of the above solution is that this solution can lead to a deadlock condition. This situation happens if all the philosophers pick their left chopstick at the same time, which leads to the condition of deadlock and none of the philosophers can eat.

To avoid deadlock, some of the solutions are as follows -

- Maximum number of philosophers on the table should not be more than four, in this case, chopstick C4 will be available for philosopher P3, so P3 will start eating and after the finish of his eating procedure, he will put down his both the chopstick C3 and C4, i.e. semaphore C3 and C4 will now be incremented to 1. Now philosopher P2 which was holding chopstick C2 will also have chopstick C3 available, hence similarly, he will put down his chopstick after eating and enable other philosophers to eat.
- A philosopher at an even position should pick the right chopstick and then the left chopstick while a philosopher at an odd position should pick the left chopstick and then the right chopstick.
- Only in case if both the chopsticks (left and right) are available at the same time, only then a philosopher should be allowed to pick their chopsticks

- All the four starting philosophers (P0, P1, P2, and P3) should pick the left chopstick and then the right chopstick, whereas the last philosopher P4 should pick the right chopstick and then the left chopstick. This will force P4 to hold his right chopstick first since the right chopstick of P4 is C0, which is already held by philosopher P0 and its value is set to 0, i.e C0 is already 0, because of which P4 will get trapped into an infinite loop and chopstick C4 remains vacant. Hence philosopher P3 has both left C3 and right C4 chopstick available, therefore it will start eating and will put down its both chopsticks once finishes and let others eat which removes the problem of deadlock.

The design of the problem was to illustrate the challenges of avoiding deadlock, a deadlock state of a system is a state in which no progress of system is possible. Consider a proposal where each philosopher is instructed to behave as follows:

- The philosopher is instructed to think till the left fork is available, when it is available, hold it.
- The philosopher is instructed to think till the right fork is available, when it is available, hold it.
- The philosopher is instructed to eat when both forks are available.
- then, put the right fork down first
- then, put the left fork down next
- repeat from the beginning.

[Next →](#) [← Prev](#)

READERS WRITERS PROBLEM

The readers-writers problem is a classical problem of process synchronization, it relates to a data set such as a file that is shared between more than one process at a time. Among these various processes, some are Readers - which can only read the data set; they do not perform any updates, some are Writers - can both read and write in the data sets.

The readers-writers problem is used for managing synchronization among various reader and writer process so that there are no problems with the data sets, i.e. no inconsistency is generated.

Let's understand with an example - If two or more than two readers want to access the file at the same point in time there will be no problem. However, in other situations like when two writers or one reader and one writer wants to access the file at the same point of time, there may occur some problems, hence the task is to design the code in such a manner that if one reader is reading then no writer is allowed to update at the same point of time, similarly, if one writer is writing no

reader is allowed to read the file at that point of time and if one writer is updating a file other writers should not be allowed to update the file at the same point of time. However, multiple readers can access the object at the same time.

Let us understand the possibility of reading and writing with the table given below:

TABLE 1

Case	Process 1	Process 2	Allowed / Not Allowed
Case 1	Writing	Writing	Not Allowed
Case 2	Reading	Writing	Not Allowed
Case 3	Writing	Reading	Not Allowed
Case 4	Reading	Reading	Allowed

The solution of readers and writers can be implemented using binary semaphores.

We use two binary semaphores "write" and "mutex", where binary semaphore can be defined as:

Semaphore: A semaphore is an integer variable in S, that apart from initialization is accessed by only two standard atomic operations - wait and signal, whose definitions are as follows:

1. wait(S)
2. {
3. **while**(S <= 0) ;
4. S--;
5. }
- 6.

```
7. 2. signal( S )
8. {
9. S++;
10.}
```

From the above definitions of wait, it is clear that if the value of $S \leq 0$ then it will enter into an infinite loop (because of the semicolon; after while loop). Whereas the job of the signal is to increment the value of S.

The below code will provide the solution of the reader-writer problem, reader and writer process codes are given as follows -

Code for Reader Process

The code of the reader process is given below -

```
1. static int readcount = 0;
2. wait (mutex);
3. readcount ++; // on each entry of reader increment readcount
4. if (readcount == 1)
5. {
6.   wait (write);
7. }
8. signal(mutex);
9.
10.--READ THE FILE?
11.
12.wait(mutex);
13.readcount --; // on every exit of reader decrement readcount
14.if (readcount == 0)
15.{
16.  signal (write);
17.}
18.signal(mutex);
```

In the above code of reader, **mutex** and **write** are **semaphores** that have an initial value of 1, whereas **the readcount** variable has an initial value as 0. Both **mutex** and **write** are common in reader and writer process code, semaphore **mutex** ensures mutual exclusion and semaphore **write** handles the writing mechanism.

The **readcount** variable denotes the number of readers accessing the file concurrently. The moment variable **readcount** becomes 1, **wait** operation is used to write semaphore which decreases the value by one. This means that a writer is not allowed how to access the file anymore. On completion of the read operation, **readcount** is decremented by one. When **readcount** becomes 0, the signal operation which is used to **write** permits a writer to access the file.

Code for Writer Process

The code that defines the writer process is given below:

1. wait(write);
2. WRITE INTO THE **FILE**
3. signal(wrt);

If a writer wishes to access the file, **wait** operation is performed on **write** semaphore, which decrements **write** to 0 and no other writer can access the file. On completion of the writing job by the writer who was accessing the file, the signal operation is performed on **write**.

Let's see the proof of each case mentioned in Table 1

CASE 1: WRITING - WRITING → NOT ALLOWED. That is when two or more than two processes are willing to write, then it is not allowed. Let us see that our code is working accordingly or not?

1. Explanation :
2. The initial value of semaphore write = 1
3. Suppose two processes P0 and P1 wants to write, let P0 enter first the writer code, The moment P0 enters
4. Wait(write); will decrease semaphore write by one, now write = 0
5. And **continue** WRITE INTO THE **FILE**
6. Now suppose P1 wants to write at the same time (will it be allowed?) let's see.
7. P1 does Wait(write), since the write value is already 0, therefore from the definition of wait, it will go into an infinite loop (i.e. Trap), hence P1 can never write anything, till P0 is writing.

8. Now suppose P0 has finished the task, it will
9. `signal(write);` will increase semaphore write by 1, now `write = 1`
10. **if** now P1 wants to write it since semaphore `write > 0`
11. This proves that, **if** one process is writing, no other process is allowed to write.

CASE 2: READING - WRITING → NOT ALLOWED. That is when one or more than one process is reading the file, then writing by another process is not allowed. Let us see that our code is working accordingly or not?

1. Explanation:
2. Initial value of semaphore `mutex = 1` and variable `readcount = 0`
3. Suppose two processes P0 and P1 are in a system, P0 wants to read **while** P1 wants to write, P0 enter first into the reader code, the moment P0 enters
4. `Wait(mutex);` will decrease semaphore `mutex` by 1, now `mutex = 0`
5. Increment `readcount` by 1, now `readcount = 1`, next
6. **if** (`readcount == 1`)// evaluates to TRUE
7. {
8. `wait (write);` // decrement write by 1, i.e. `write = 0`(which
9. clearly proves that **if** one or more than one
10. reader is reading then no writer will be
11. allowed.
12. }
- 13.
14. `signal(mutex);` // will increase semaphore `mutex` by 1, now `mutex = 1` i.e. other readers are allowed to enter.
- 15.
16. And reader continues to --READ THE **FILE**?
- 17.
18. Suppose now any writer wants to enter into its code then:
- 19.
20. As the first reader has executed `wait (write);` because of which `write` value is 0, therefore `wait(writer);` of the writer, code will go into an infinite loop and no writer will be allowed.
21. This proves that, **if** one process is reading, no other process is allowed to write.
22. Now suppose P0 wants to stop the reading and wanted to exit then

23. Following sequence of instructions will take place:

24. `wait(mutex); // decrease mutex by 1, i.e. mutex = 0`

25. `readcount --; // readcount = 0, i.e. no one is currently reading`

26. `if (readcount == 0) // evaluates TRUE`

27. {

28. `signal(write); // increase write by one, i.e. write = 1`

29. }

30. `signal(mutex); // increase mutex by one, i.e. mutex = 1`

31.

32. Now `if` again any writer wants to write, it can `do` it now, since `write > 0`

CASE 3: WRITING -- READING → NOT ALLOWED. That is when if one process is writing into the file, then reading by another process is not allowed. Let us see that our code is working accordingly or not?

1. Explanation:
2. The initial value of semaphore `write` = 1
3. Suppose two processes P0 and P1 are in a system, P0 wants to write `while` P1 wants to read, P0 enter first into the writer code, The moment P0 enters
4. `Wait(write);` will decrease semaphore `write` by 1, now `write` = 0
5. And `continue` WRITE INTO THE **FILE**
6. Now suppose P1 wants to read the same time (will it be allowed?) let's see.
7. P1 enters reader's code
8. Initial value of semaphore `mutex` = 1 and variable `readcount` = 0
9. `Wait(mutex);` will decrease semaphore `mutex` by 1, now `mutex` = 0
10. Increment `readcount` by 1, now `readcount` = 1, next
11. `if (readcount == 1) // evaluates to TRUE`
12. {
13. `wait(write); // since value of write is already 0, hence it`
14. `will enter into an infinite loop and will not be`
15. `allowed to proceed further (which clearly`
16. `proves that if one writer is writing then no`
17. `reader will be allowed.`

- 18.}
- 19.
- 20.The moment writer stops writing and willing to exit then
- 21.This proves that, **if** one process is writing, no other process is allowed to read.
- 22.
- 23.The moment writer stops writing and willing to exit then it will execute:
- 24.signal(write); will increase semaphore write by 1, now write = 1
- 25.**if** now P1 wants to read it can since semaphore write > 0

CASE 4: READING - READING → ALLOWED. That is when one process is reading the file, and other process or processes is willing to read, then they all are allowed i.e. reading - reading is not mutually exclusive. Let us see that our code is working accordingly or not?

1. Explanation :
2. Initial value of semaphore mutex = 1 and variable readcount = 0
3. Suppose three processes P0, P1 and P2 are in a system, all the three processes P0, P1, and P2 want to read, let P0 enter first into the reader code, the moment P0 enters
4. Wait(mutex); will decrease semaphore mutex by 1, now mutex = 0
5. Increment readcount by 1, now readcount = 1, next
6. **if** (readcount == 1)// evaluates to TRUE
7. {
8. wait (write); // decrement write by 1, i.e. write = 0(which
9. clearly proves that **if** one or more than one
10. reader is reading then no writer will be
11. allowed.
- 12.}
- 13.
- 14.signal(mutex); // will increase semaphore mutex by 1, now mutex = 1 i.e. other readers are allowed to enter.
- 15.
- 16.And P0 continues to --READ THE **FILE**?
- 17.
- 18.→Now P1 wants to enter the reader code

19. current value of semaphore mutex = 1 and variable readcount = 1
20. let P1 enter into the reader code, the moment P1 enters
21. Wait(mutex); will decrease semaphore mutex by 1, now mutex = 0
22. Increment readcount by 1, now readcount = 2, next
23. **if** (readcount == 1) // eval. to False, it will not enter if block
24.
25. signal(mutex); // will increase semaphore mutex by 1, now mutex = 1 i.e. other readers are allowed to enter.
26.
27. Now P0 and P1 continues to --READ THE **FILE**?
28.
29. → Now P2 wants to enter the reader code
30. current value of semaphore mutex = 1 and variable readcount = 2
31. let P2 enter into the reader code, The moment P2 enters
32. Wait(mutex); will decrease semaphore mutex by 1, now mutex = 0
33. Increment readcount by 1, now readcount = 3, next
34. **if** (readcount == 1) // eval. to False, it will not enter if block
35.
36. signal(mutex); // will increase semaphore mutex by 1, now mutex = 1 i.e. other readers are allowed to enter.
37.
38. Now P0, P1, and P2 continues to --READ THE **FILE**?
39.
40.
41. Suppose now any writer wants to enter into its code then:
42.
43. As the first reader P0 has executed wait (write); because of which write value is 0, therefore wait(writer); of the writer, code will go into an infinite loop and no writer will be allowed.
44.
45. Now suppose P0 wants to come out of system(stop reading) then
46. wait(mutex); // will decrease semaphore mutex by 1, now mutex = 0
47. readcount --; // on every exit of reader decrement readcount by


```
48.         one i.e. readcount = 2
49.
50. if (readcount == 0) // eval. to FALSE it will not enter if block
51.
52. signal(mutex); // will increase semaphore mutex by 1, now mutex = 1 i.e. other readers are allowed to exit
53.
54.
55. → Now suppose P1 wants to come out of system (stop reading) then
56. wait(mutex); // will decrease semaphore mutex by 1, now mutex = 0
57. readcount --; // on every exit of reader decrement readcount by
58.         one i.e. readcount = 1
59.
60. if (readcount == 0) // eval. to FALSE it will not enter if block
61.
62. signal(mutex); // will increase semaphore mutex by 1, now mutex = 1 i.e. other readers are allowed to exit
63.
64. → Now suppose P2 (last process) wants to come out of system (stop reading) then
65. wait(mutex); // will decrease semaphore mutex by 1, now mutex = 0
66. readcount --; // on every exit of reader decrement readcount by
67.         one i.e. readcount = 0
68.
69. if (readcount == 0) // eval. to TRUE it will enter into if block
70. {
71.     signal (write); // will increment semaphore write by one, i.e.
72.         now write = 1, since P2 was the last process
73.         which was reading, since now it is going out,
74.         so by making write = 1 it is allowing the writer
75.         to write now.
76. }
77.
```

78. signal(mutex); // will increase semaphore mutex by 1, now mutex = 1

79.

80.

81. The above explanation proves that **if** one or more than one processes are willing to read simultaneously then they are

Banker's Algorithm in Operating System

It is a banker algorithm used to **avoid deadlock** and **allocate resources** safely to each process in the computer system. The '**S-State**' examines all possible tests or activities before deciding whether the allocation should be allowed to each process. It also helps the operating system to successfully share the resources between all the processes. The banker's algorithm is named because it checks whether a person should be sanctioned a loan amount or not to help the bank system safely simulate allocation resources. In this section, we will learn the **Banker's Algorithm** in detail. Also, we will solve problems based on the **Banker's Algorithm**. To understand the Banker's Algorithm first we will see a real word example of it.

Suppose the number of account holders in a particular bank is 'n', and the total money in a bank is 'T'. If an account holder applies for a loan; first, the bank subtracts the loan amount from full cash and then estimates the cash difference is greater than T to approve the loan amount. These steps are taken because if another person applies for a loan or withdraws some amount from the bank, it helps the bank manage and operate all things without any restriction in the functionality of the banking system.

Similarly, it works in an **operating system**. When a new process is created in a computer system, the process must provide all types of information to the **operating system** like upcoming processes, requests for their resources, counting them, and delays. Based on these criteria, the operating system decides which process sequence should be executed or waited so that no deadlock occurs in a system. Therefore, it is also known as **deadlock avoidance algorithm** or **deadlock detection** in the operating system.

Advantages

Following are the essential characteristics of the Banker's algorithm:

1. It contains various resources that meet the requirements of each process.
2. Each process should provide information to the operating system for upcoming resource requests, the number of resources, and how long the resources will be held.
3. It helps the operating system manage and control process requests for each type of resource in the computer system.
4. The algorithm has a Max resource attribute that represents indicates each process can hold the maximum number of resources in a system.

Disadvantages

1. It requires a fixed number of processes, and no additional processes can be started in the system while executing the process.
2. The algorithm does no longer allows the processes to exchange its maximum needs while processing its tasks.

3. Each process has to know and state their maximum resource requirement in advance for the system.
4. The number of resource requests can be granted in a finite time, but the time limit for allocating the resources is one year.

When working with a banker's algorithm, it requests to know about three things:

1. How much each process can request for each resource in the system. It is denoted by the [**MAX**] request.
2. How much each process is currently holding each resource in a system. It is denoted by the [**ALLOCATED**] resource.
3. It represents the number of each resource currently available in the system. It is denoted by the [**AVAILABLE**] resource.

Following are the important data structures terms applied in the banker's algorithm as follows:

Suppose n is the number of processes, and m is the number of each type of resource used in a computer system.

1. **Available:** It is an array of length ' m ' that defines each type of resource available in the system. When $Available[j] = K$, means that ' K ' instances of Resources type $R[j]$ are available in the system.
2. **Max:** It is a $[n \times m]$ matrix that indicates each process $P[i]$ can store the maximum number of resources $R[j]$ (each type) in a system.
3. **Allocation:** It is a matrix of $m \times n$ orders that indicates the type of resources currently allocated to each process in the system. When $Allocation[i, j] = K$, it means that process $P[i]$ is currently allocated K instances of Resources type $R[j]$ in the system.
4. **Need:** It is an $M \times N$ matrix sequence representing the number of remaining resources for each process. When the $Need[i][j] = k$, then process $P[i]$ may require K more instances of resources type R_j to complete the assigned work.
 $Need[i][j] = Max[i][j] - Allocation[i][j]$.
5. **Finish:** It is the vector of the order m . It includes a Boolean value (true/false) indicating whether the process has been allocated to the requested resources, and all resources have been released after finishing its task.

The Banker's Algorithm is the combination of the safety algorithm and the resource request algorithm to control the processes and avoid deadlock in a system:

Safety Algorithm

It is a safety algorithm used to check whether or not a system is in a safe state or follows the safe sequence in a banker's algorithm:

1. There are two vectors **Work** and **Finish** of length m and n in a safety algorithm.

Initialize: $Work = Available$

$Finish[i] = false$; for $i = 0, 1, 2, 3, 4 \dots n - 1$.

2. Check the availability status for each type of resources [i], such as:

Need[i] <= Work

Finish[i] == false

If the i does not exist, go to step 4.

3. Work = Work + Allocation(i) // to get new resource allocation

Finish[i] = true

Go to step 2 to check the status of resource availability for the next process.

4. If Finish[i] == true; it means that the system is safe for all processes.

Resource Request Algorithm

A resource request algorithm checks how a system will behave when a process makes each type of resource request in a system as a request matrix.

Let create a resource request array R[i] for each process P[i]. If the Resource Request_i [j] equal to 'K', which means the process P[i] requires 'k' instances of Resources type R[j] in the system.

1. When the number of **requested resources** of each type is less than the **Need** resources, go to step 2 and if the condition fails, which means that the process P[i] exceeds its maximum claim for the resource. As the expression suggests:

If Request(i) <= Need

Go to step 2;

2. And when the number of requested resources of each type is less than the available resource for each process, go to step (3). As the expression suggests:

If Request(i) <= Available

Else Process P[i] must wait for the resource since it is not available for use.

3. When the requested resource is allocated to the process by changing state:

Available = Available - Request

Allocation(i) = Allocation(i) + Request (i)

Need_i = Need_i - Request_i

When the resource allocation state is safe, its resources are allocated to the process P(i). And if the new state is unsafe, the Process P (i) has to wait for each type of Request R(i) and restore the old resource-allocation state.

Example: Consider a system that contains five processes P1, P2, P3, P4, P5 and the three resource types A, B and C. Following are the resources types: A has 10, B has 5 and the resource type C has 7 instances.

Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P1	0	1	0	7	5	3	3	3	2
P2	2	0	0	3	2	2			
P3	3	0	2	9	0	2			
P4	2	1	1	2	2	2			
P5	0	0	2	4	3	3			

Answer the following questions using the banker's algorithm:

1. What is the reference of the need matrix?
2. Determine if the system is safe or not.
3. What will happen if the resource request (1, 0, 0) for process P1 can the system accept this request immediately?

Ans. 2: Context of the need matrix is as follows:

Need [i] = Max [i] - Allocation [i]

Need for P1: (7, 5, 3) - (0, 1, 0) = 7, 4, 3

Need for P2: (3, 2, 2) - (2, 0, 0) = 1, 2, 2

Need for P3: (9, 0, 2) - (3, 0, 2) = 6, 0, 0

Need for P4: (2, 2, 2) - (2, 1, 1) = 0, 1, 1

Need for P5: (4, 3, 3) - (0, 0, 2) = 4, 3, 1

Process	Need		
	A	B	C

P1	7	4	3
P2	1	2	2
P3	6	0	0
P4	0	1	1
P5	4	3	1

Hence, we created the context of need matrix.

Ans. 2: Apply the Banker's Algorithm:

Available Resources of A, B and C are 3, 3, and 2.

Now we check if each type of resource request is available for each process.

Step 1: For Process P1:

Need \leq Available

7, 4, 3 \leq 3, 3, 2 condition is **false**.

So, we examine another process, P2.

Step 2: For Process P2:

Need \leq Available

1, 2, 2 \leq 3, 3, 2 condition **true**

New available = available + Allocation

(3, 3, 2) + (2, 0, 0) \Rightarrow 5, 3, 2

Similarly, we examine another process P3.

Step 3: For Process P3:

P3 Need \leq Available

6, 0, 0 \leq 5, 3, 2 condition is **false**.

Similarly, we examine another process, P4.

Step 4: For Process P4:

P4 Need \leq Available

0, 1, 1 \leq 5, 3, 2 condition is **true**

New Available resource = Available + Allocation

5, 3, 2 + 2, 1, 1 \Rightarrow 7, 4, 3

Similarly, we examine another process P5.

Step 5: For Process P5:

P5 Need \leq Available

4, 3, 1 \leq 7, 4, 3 condition is **true**

New available resource = Available + Allocation

7, 4, 3 + 0, 0, 2 \Rightarrow 7, 4, 5

Now, we again examine each type of resource request for processes P1 and P3.

Step 6: For Process P1:

P1 Need \leq Available

7, 4, 3 \leq 7, 4, 5 condition is **true**

New Available Resource = Available + Allocation

7, 4, 5 + 0, 1, 0 \Rightarrow 7, 5, 5

So, we examine another process P2.

Step 7: For Process P3:

P3 Need \leq Available

6, 0, 0 \leq 7, 5, 5 condition is true

New Available Resource = Available + Allocation

7, 5, 5 + 3, 0, 2 \Rightarrow 10, 5, 7

Hence, we execute the banker's algorithm to find the safe state and the safe sequence like P2, P4, P5, P1 and P3.

Ans. 3: For granting the Request (1, 0, 2), first we have to check that **Request \leq Available**, that is (1, 0, 2) \leq (3, 3, 2), since the condition is true. So the process P1 gets the request immediately.

What is the context switching in the operating system?

The Context switching is a technique or method used by the operating system to switch a process from one state to another to execute its function using CPUs in the system. When switching perform in the system, it stores the old running process's status in the form of registers and assigns the [CPU](#) to a new process to execute its tasks. While a new process is running in the system, the previous process must wait in a ready queue. The execution of the old process starts at that point where another process stopped it. It defines the characteristics of a multitasking operating system in which multiple processes shared the same [CPU](#) to perform multiple tasks without the need for additional processors in the system.

The need for Context switching

A context switching helps to share a single CPU across all processes to complete its execution and store the system's tasks status. When the process reloads in the system, the execution of the process starts at the same point where there is conflicting.

Following are the reasons that describe the need for context switching in the Operating system.

1. The switching of one process to another process is not directly in the system. A context switching helps the operating system that switches between the multiple processes to use the CPU's resource to accomplish its tasks and store its context. We can resume the service of the process at the same point later. If we do not store the currently running process's data or context, the stored data may be lost while switching between processes.
2. If a high priority process falls into the ready queue, the currently running process will be shut down or stopped by a high priority process to complete its tasks in the system.
3. If any running process requires I/O resources in the system, the current process will be switched by another process to use the CPUs. And when the I/O requirement is met, the old process goes into a ready state to wait for its execution in the CPU. Context switching stores the state of the process to resume its tasks in an operating system. Otherwise, the process needs to restart its execution from the initials level.

4. If any interrupts occur while running a process in the operating system, the process status is saved as registers using context switching. After resolving the interrupts, the process switches from a wait state to a ready state to resume its execution at the same point later, where the operating system interrupted occurs.
5. A context switching allows a single CPU to handle multiple process requests simultaneously without the need for any additional processors.

Example of Context Switching

Suppose that multiple processes are stored in a Process Control Block (PCB). One process is running state to execute its task with the use of CPUs. As the process is running, another process arrives in the ready queue, which has a high priority of completing its task using CPU. Here we used context switching that switches the current process with the new process requiring the CPU to finish its tasks. While switching the process, a context switch saves the status of the old process in registers. When the process reloads into the CPU, it starts the execution of the process when the new process stops the old process. If we do not save the state of the process, we have to start its execution at the initial level. In this way, context switching helps the operating system to switch between the processes, store or reload the process when it requires executing its tasks.

Context switching triggers

Following are the three types of context switching triggers as follows.

1. Interrupts
2. Multitasking
3. Kernel/User switch

Interrupts: A CPU requests for the data to read from a disk, and if there are any interrupts, the context switching automatic switches a part of the hardware that requires less time to handle the interrupts.

Multitasking: A context switching is the characteristic of multitasking that allows the process to be switched from the CPU so that another process can be run. When switching the process, the old state is saved to resume the process's execution at the same point in the system.

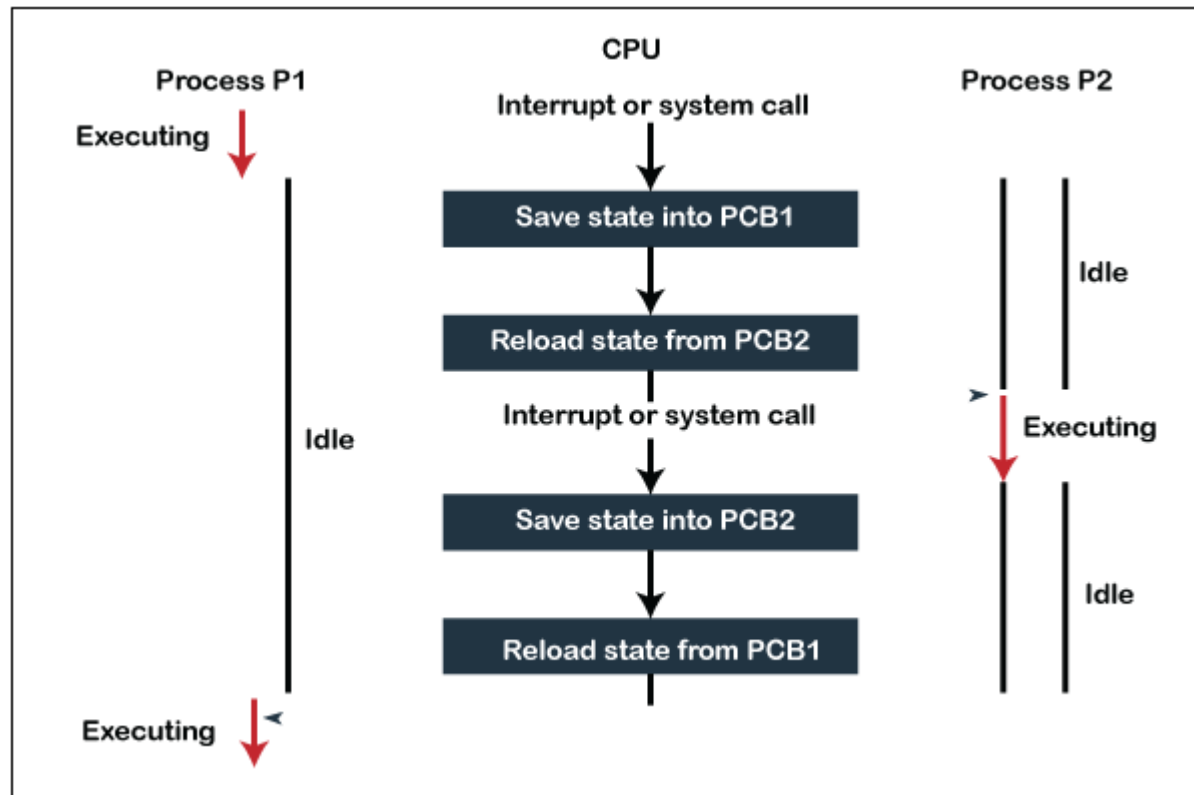
Kernel/User Switch: It is used in the operating systems when switching between the user mode, and the kernel/user mode is performed.

What is the PCB?

A PCB (Process Control Block) is a data structure used in the operating system to store all data related information to the process. For example, when a process is created in the operating system, updated information of the process, switching information of the process, terminated process in the PCB.

Steps for Context Switching

There are several steps involved in context switching of the processes. The following diagram represents the context switching of two processes, P1 to P2, when an interrupt, I/O needs, or priority-based process occurs in the ready queue of PCB.



As we can see in the diagram, initially, the P1 process is running on the CPU to execute its task, and at the same time, another process, P2, is in the ready state. If an error or interruption has occurred or the process requires input/output, the P1 process switches its state from running to the waiting state. Before changing the state of the process P1, context switching saves the context of the process P1 in the form of registers and the program counter to the **PCB1**. After that, it loads the state of the P2 process from the ready state of the **PCB2** to the running state.

The following steps are taken when switching Process P1 to Process 2:

1. First, the context switching needs to save the state of process P1 in the form of the program counter and the registers to the PCB (Program Counter Block), which is in the running state.
2. Now update PCB1 to process P1 and moves the process to the appropriate queue, such as the ready queue, I/O queue and waiting queue.
3. After that, another process gets into the running state, or we can select a new process from the ready state, which is to be executed, or the process has a high priority to execute its task.

4. Now, we have to update the PCB (Process Control Block) for the selected process P2. It includes switching the process state from ready to running state or from another state like blocked, exit, or suspend.
5. If the CPU already executes process P2, we need to get the status of process P2 to resume its execution at the same time point where the system interrupt occurs.

Similarly, process P2 is switched off from the CPU so that the process P1 can resume execution. P1 process is reloaded from PCB1 to the running state to resume its task at the same point. Otherwise, the information is lost, and when the process is executed again, it starts execution at the initial level.

[Next →](#) [← Prev](#)

Internal vs. External Fragmentation

What is Fragmentation?

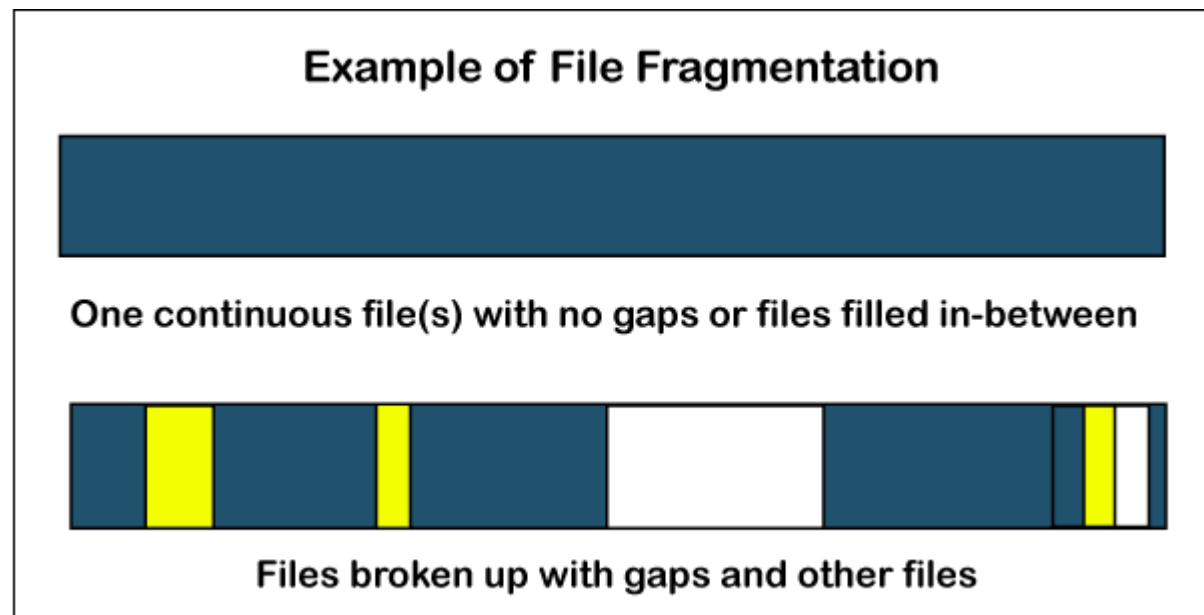
"Fragmentation is a process of data storage in which memory space is used inadequately, decreasing ability or efficiency and sometimes both." The precise implications of fragmentation depend on the specific storage space allocation scheme in operation and the particular fragmentation type. In certain instances, fragmentation contributes to "unused" storage capacity, and the concept also applies to the unusable space generated in that situation. The memory used to preserve the data set (- for example file format) is similar for other systems (- for example, the FAT file system), regardless of the amount of fragmentation (from null to the extreme).

There are three distinct fragmentation kinds: internal fragmentation, external fragmentation, and data fragmentation that can exist beside or a combination. In preference for enhancements, inefficiency, or usability, fragmentation is often acknowledged. For other tools, such as processors, similar things happen.

The Fundamental concept of Fragmentation

When a computer program demands fragments of storage from the [operating system \(OS\)](#), the elements are assigned in chunks. When a chunk of the software program is completed, it can be released back to the system, making it ready to be transferred to the next or the similar program again afterward. Software differs in the size and duration of time a chunk is kept by it. A computer program can demand and release several blocks of storage throughout its lifetime.

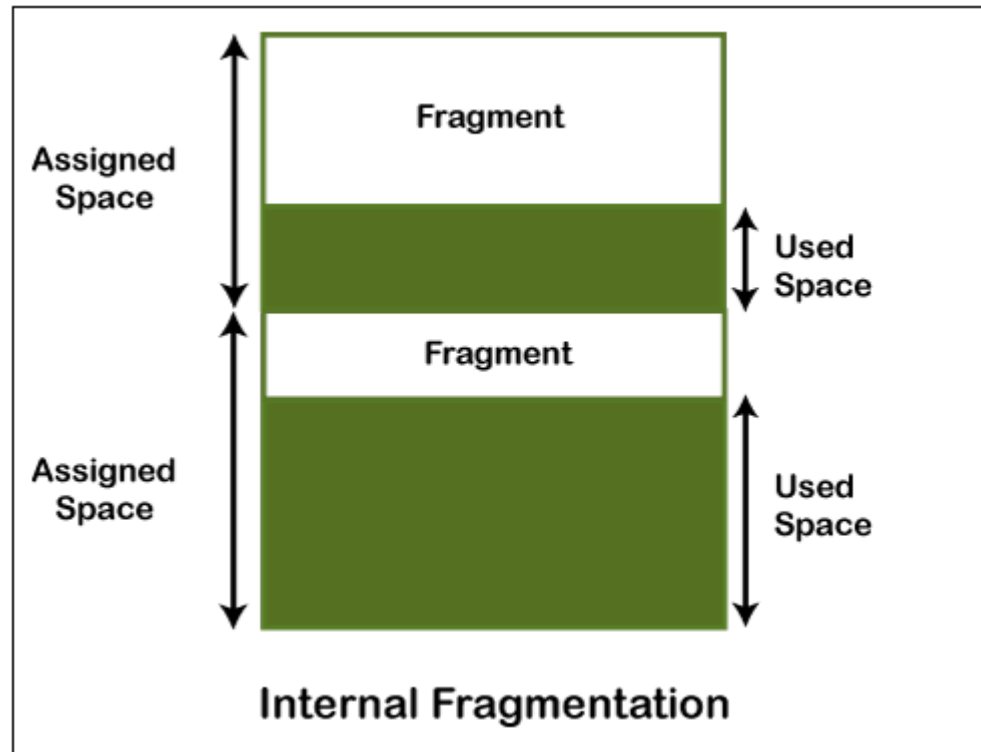
The unused memory sections are large and continuous when a system is initiated. The large continuous sectors become fragmented into a smaller part of the regions through time and utilization. Ultimately, accessing large contiguous blocks of storage can become difficult for the system.



Internal Fragmentation

Most memory space is often reserved than is required to adhere to the restrictions regulating storage space. For instance, memory can only be supplied in blocks (multiple of 4) to systems, and as an outcome, if a program demands maybe 29 bytes, it will get a coalition of 32 bytes. The surplus storage goes to waste when this occurs. The useless space is found inside an assigned area in this case. This structure, called fixed segments, struggles from excessive memory-any process consumes an enormous chunk, no matter how insignificant. Internal fragmentation is what this garbage is termed. Unlike many other forms of fragmentation, it is impossible to restore inner fragmentation, typically, the only way to eliminate it is with a new design.

For instance, in dynamic storage allocation, storage reservoirs reduce internal fragmentation significantly by extending the space overhead over a more significant number of elements.



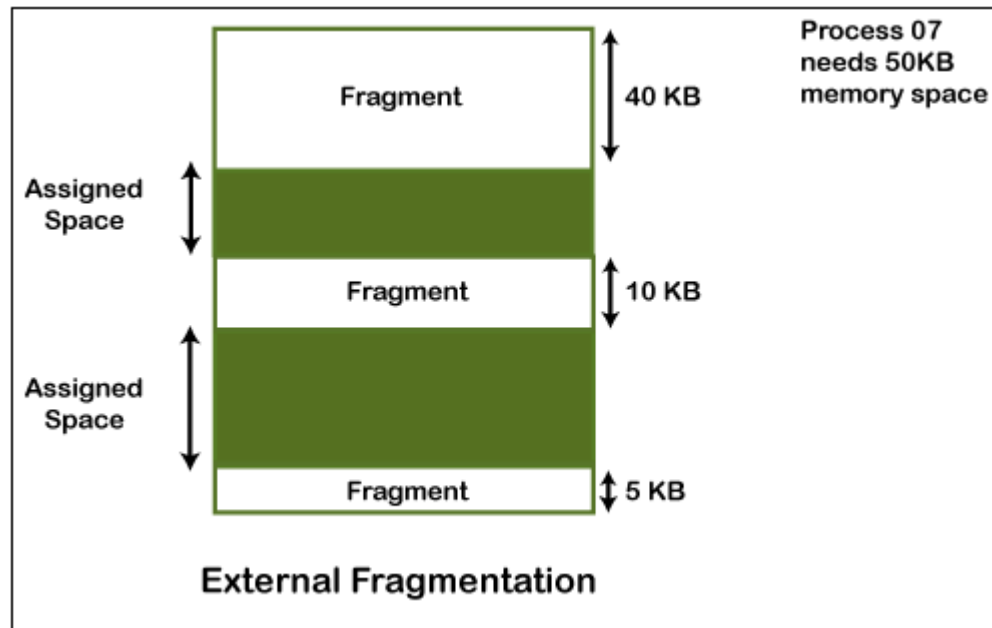
The figure mentioned above demonstrates internal fragmentation because internal fragmentation is considered the distinction between the assigned storage space and the needed space or memory.

External Fragmentation

When used storage is differentiated into smaller lots and is punctuated by assigned memory space, external fragmentation occurs. It is a weak point of many storage allocation methodologies when they cannot effectively schedule memory used by systems. The consequence is that, while unused storage is available, it is essentially inaccessible since it is separately split into fragments that are too limited to meet the software's requirements. The word "external" derives from the fact that the inaccessible space is stored outside the assigned regions.

Consider, for instance, a scenario in which a system assigns three consecutive memory blocks and then relieves the middle block. The memory allocator can use this unused allocation of the storage for future assignments. Fortunately, if the storage to be reserved is more generous in size than this available region, it will not use this component.

In data files, external fragmentation often exists when several files of various sizes are formed, resized, and discarded. If a broken document into several small chunks is removed, the impact is much worse since this retains equally small free space sections.



You can see in the figure mentioned above that there is sufficient memory space (55 KB) to execute a process-07 (50 KB mandated), but the storage (fragment) is not adjacent. Here, to use the empty room to run a procedure, you can use compression, paging, or segmentation strategies.

Internal fragmentation vs. External fragmentation



Here, the differences between Internal and External Fragmentation are discussed below in the tabular format.

Sr. No.	Internal Fragmentation	External Fragmentation
1.	Frames square measure designated for processing in internal fragmentation of fixed-sized storage.	Variable-sized memory frames square measure designated to the process during external fragmentation.
2.	When the system or procedure is greater than the storage, internal fragmentation occurs.	Whenever the system or procedure is withdrawn, external fragmentation occurs.
3.	The internal fragmentation approach is the frame with the perfect match.	Compression, paging, and differentiation are alternatives to external fragmentation.

4.	Internal fragmentation happens whenever the storage is split into fragments of a fixed length.	External fragmentation happens whenever the storage is split into segments of variable size depending on the process length.
5.	The distinction between the assigned memory and the storage or memory needed is considered as internal fragmentation.	The empty spaces created among non-contiguous pieces of storage are too tiny for a new system to operate, considered as external fragmentation.