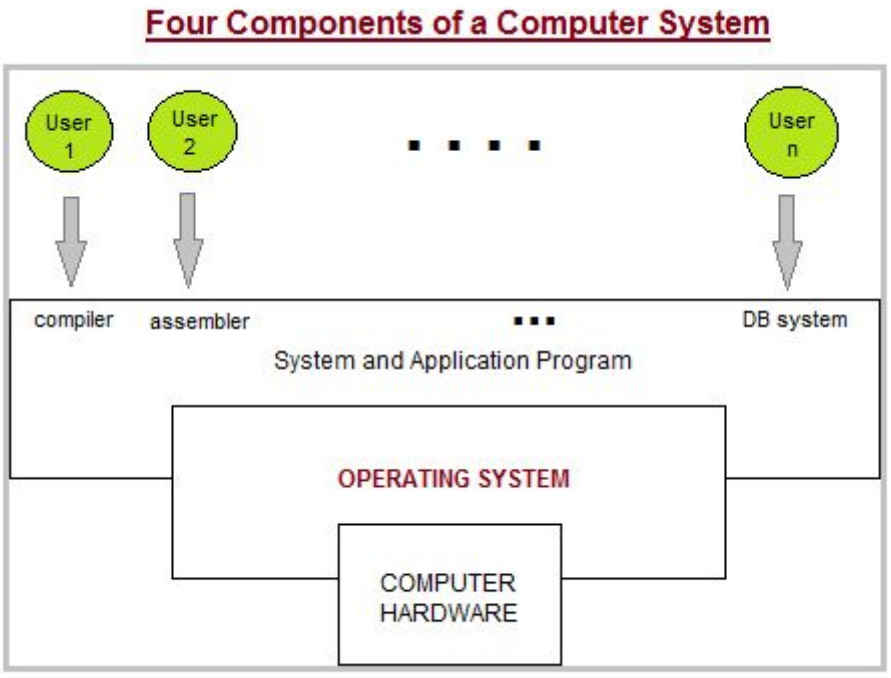
OPERATING SYSTEM NOTES

A computer system has many resources (hardware and software), which may be required to complete a task. The commonly required resources are input/output devices, memory, file storage space, CPU etc. The operating system acts as a manager of the above resources and allocates them to specific programs and users as necessary for their task. Therefore operating system is the resource manager i.e. it can manage the resource of a computer system internally. The resources are processor, memory, files, and I/O devices.



Functions of Operating System

1. It boots the computer
2. It performs basic computer tasks e.g. managing the various peripheral devices e.g. mouse, keyboard
3. It provides a user interface, e.g. command line, graphical user interface (GUI)
4. It handles system resources such as computer's memory and sharing of the central processing unit(CPU) time by various applications or peripheral devices.

5. It provides file management which refers to the way that the operating system manipulates, stores, retrieves and saves data.
6. Error Handling is done by the operating system. It takes preventive measures whenever required to avoid errors

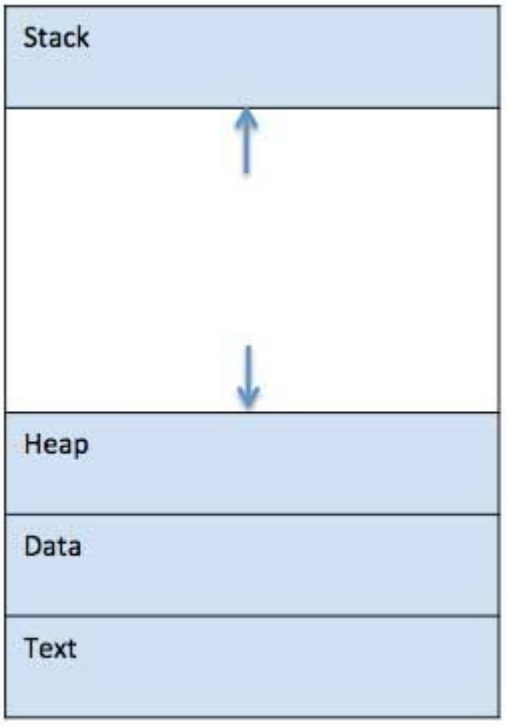
You can read about different types of operating system in detail . What important is understanding what operating system actually is.

OPERATING SYSTEM PROCESSES

A program in the execution is called a Process. Process is not the same as program. A process is more than a program code. A process is an 'active' entity as opposed to program which is considered to be a 'passive' entity.

Process memory is divided into four sections for efficient working :

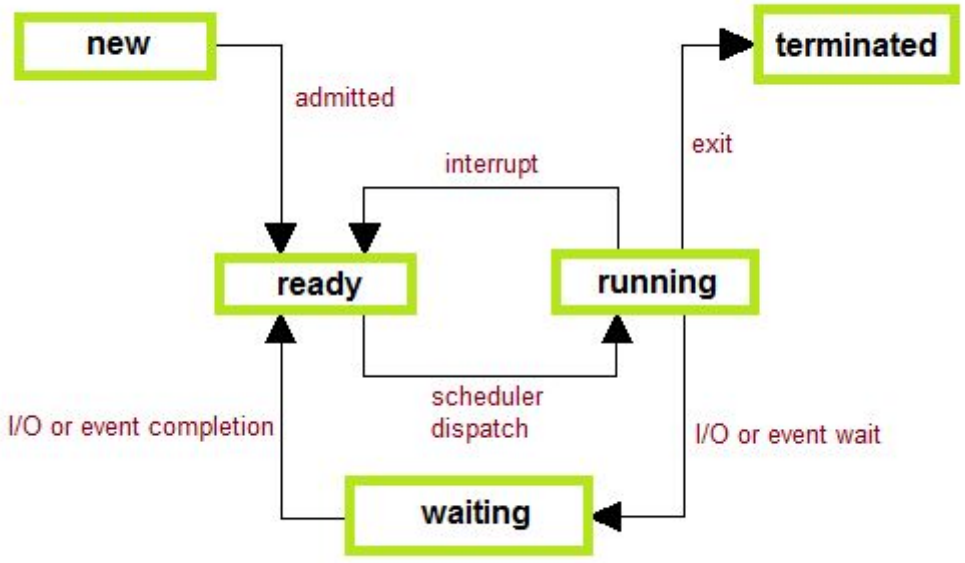
- The text section is made up of the compiled program code when the program is launched.
- The data section is made up the global and static variables, allocated and initialized prior to executing the main.
- The heap is used for the dynamic memory allocation, and is managed via calls to new, delete, malloc, free, etc.
- The stack is used for local variables. Space on the stack is reserved for local variables when they are declared.



PROCESS STATE

Processes can be any of the following states :

- **New** - The process is in the stage of being created.
- **Ready** - The process has all the resources available that it needs to run, but the CPU is not currently working on this process's instructions.
- **Running** - The CPU is working on this process's instructions.
- **Waiting** - The process cannot run at the moment, because it is waiting for some resource to become available or for some event to occur.
- **Terminated** - The process has completed.



Every modern operating system supports these two modes.

Kernel Mode

- When CPU is in **kernel mode**, the code being executed can access any memory address and any hardware resource.
- Hence kernel mode is a very privileged and powerful mode.
- If a program crashes in kernel mode, the entire system will be halted.

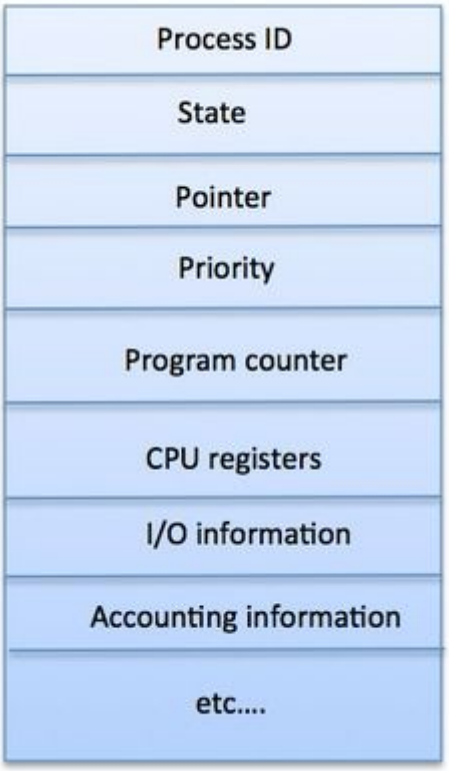
User Mode

- When CPU is in **user mode**, the programs don't have direct access to memory and hardware resources.
- In user mode, if any program crashes, only that particular program is halted.
- That means the system will be in a safe state even if a program in user mode crashes.
- Hence, most programs in an OS run in user mode.

PROCESS CONTROL BLOCK

There is a Process Control Block for each process, enclosing all the information about the process. It is a data structure, which contains the following :

- Process State - It can be running, waiting etc.
- Process ID.
- CPU registers and Program Counter. **Program Counter** holds the address of the next instruction to be executed for that process.
- CPU Scheduling information - Such as priority information and pointers to scheduling queues.
- Memory Management information - Eg. page tables or segment tables.
- Accounting information - user and kernel CPU time consumed, account numbers, limits, etc.
- I/O Status information - Devices allocated, open file tables, etc.



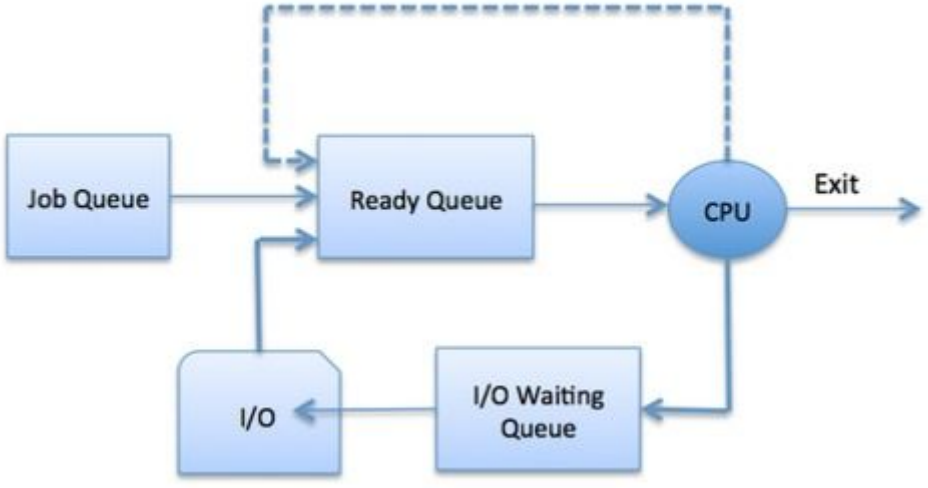
Process Scheduling Queues

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

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

The Operating System maintains the following important process scheduling queues

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



Types of Schedulers

There are three types of schedulers available :

1. **Long Term Scheduler** :

Long term scheduler runs less frequently. Long Term Schedulers decide which program must get into the job queue. From the job queue, the Job Processor, selects processes and loads them into the memory for execution. Primary aim of the Job Scheduler is to maintain a good degree of Multiprogramming. An optimal degree of Multiprogramming means the average rate of process creation is equal to the average departure rate of processes from the execution memory.

2. Short Term Scheduler :

It is responsible for selecting one process from ready state for scheduling it on the running state. Note: Short term scheduler only selects the process to schedule it doesn't load the process on running.

Dispatcher is responsible for loading the selected process by Short Term scheduler on the CPU (Ready to Running State) Context switching is done by dispatcher only. A dispatcher does following:

1) Switching context.

2) Switching to user mode.

3) Jumping to the proper location in the newly loaded program.

3. Medium Term Scheduler :

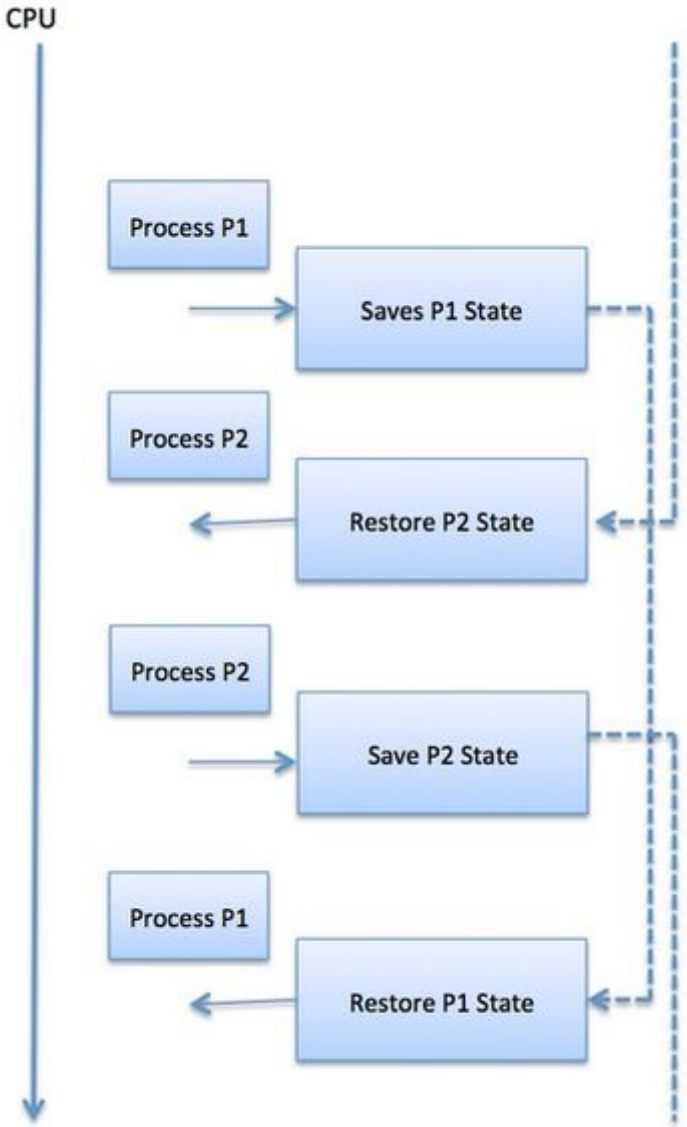
Medium-term scheduling is a part of **swapping**. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium-term scheduler is in-charge of handling the swapped out-processes.

A running process may become suspended if it makes an I/O request. A suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other processes, the suspended process is moved to the secondary storage. This process is called **swapping**, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix.

Context Switch

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

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


CPU
Process P1
Saves P1 State
Process P2
Restore P2 State
Process P2
Save P2 State
Process P1
Restore P1 State

CPU SCHEDULING ALGORITHMS:

Below are different time with respect to a process.

Arrival Time: Time at which the process arrives in the ready queue.

Completion Time: Time at which process completes its execution.

Burst Time: Time required by a process for CPU execution.

Turn Around Time: Time Difference between completion time and arrival time.

Turn Around Time = Completion Time - Arrival Time

Waiting Time(W.T): Time Difference between turn around time and burst time.

Waiting Time = Turn Around Time - Burst Time

Throughput

It is the total number of processes completed per unit time or rather say total amount of work done in a unit of time. This may range from 10/second to 1/hour depending on the specific processes.

These algorithms are either **non-preemptive or preemptive**. Non-preemptive algorithms are designed so that once a process enters the running state, it cannot be preempted until it completes its allotted time, whereas the preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state.

Objectives of Process Scheduling Algorithm

Max CPU utilization [Keep CPU as busy as possible]

Fair allocation of CPU.

Max throughput [Number of processes that complete their execution per time unit]

Min turnaround time [Time taken by a process to finish execution]

`Min waiting time [Time a process waits in ready queue]`

`Min response time [Time when a process produces first response]`

Scheduling Algorithms

We'll discuss four major scheduling algorithms here which are following :

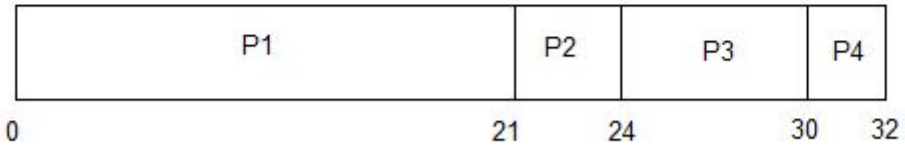
1. First Come First Serve(FCFS) Scheduling
2. Shortest-Job-First(SJF) Scheduling
3. Priority Scheduling
4. Round Robin(RR) Scheduling
5. Multilevel Queue Scheduling

First Come First Serve(FCFS) Scheduling

- Jobs are executed on first come, first serve basis.
- Easy to understand and implement.
- Poor in performance as average wait time is high.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

The average waiting time will be = (0 + 21 + 24 + 30)/4 = 18.75 ms



This is the GANTT chart for the above processes

Shortest-Job-First(SJF) Scheduling

- Best approach to minimize waiting time.
- Actual time taken by the process is already known to processor.
- Impossible to implement.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

In Shortest Job First Scheduling, the shortest Process is executed first.

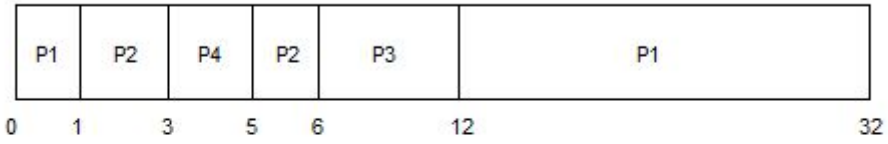
Hence the GANTT chart will be following :

Now, the average waiting time will be = (0 + 2 + 5 + 11)/4 = 4.5 ms

In Preemptive Shortest Job First Scheduling, jobs are put into ready queue as they arrive, but as a process with short burst time arrives, the existing process is preemptied.

PROCESS	BURST TIME	ARRIVAL TIME
P1	21	0
P2	3	1
P3	6	2
P4	2	3

The GANTT chart for Preemptive Shortest Job First Scheduling will be,



The average waiting time will be, ((5-3) + (6-2) + (12-1))/4 = 4.25 ms

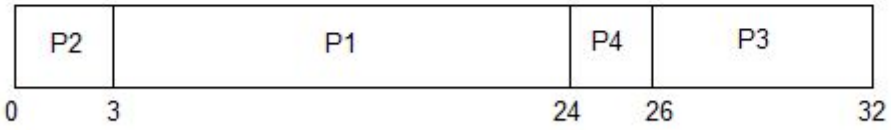
The average waiting time for preemptive shortest job first scheduling is less than both, non-preemptive SJF scheduling and FCFS scheduling.

Priority Scheduling

- Priority is assigned for each process.
- Process with highest priority is executed first and so on.
- Processes with same priority are executed in FCFS manner.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement.

PROCESS	BURST TIME	PRIORITY
P1	21	2
P2	3	1
P3	6	4
P4	2	3

The GANTT chart for following processes based on Priority scheduling will be,



The average waiting time will be, (0 + 3 + 24 + 26)/4 = 13.25 ms

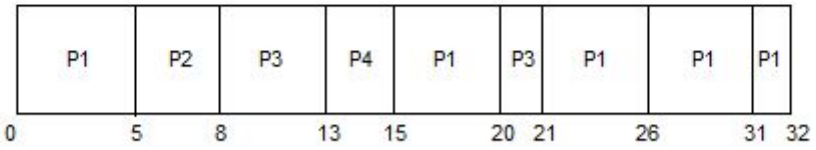
---

Round Robin(RR) Scheduling

- A fixed time is allotted to each process, called **quantum**, for execution.
- Once a process is executed for given time period that process is preemptied and other process executes for given time period.
- Context switching is used to save states of preemptied processes.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

The GANTT chart for round robin scheduling will be,



The average waiting time will be, 11 ms.

Multilevel Queue Scheduling

Multiple-level queues are not an independent scheduling algorithm. They make use of other existing algorithms to group and schedule jobs with common characteristics.

- Multiple queues are maintained for processes with common characteristics.
- Each queue can have its own scheduling algorithms.
- Priorities are assigned to each queue.

For example, CPU-bound jobs can be scheduled in one queue and all I/O-bound jobs in another queue. The Process Scheduler then alternately selects jobs from each queue and assigns them to the CPU based on the algorithm assigned to the queue.

Comparison of Scheduling Algorithms

By now, you must have understood how CPU can apply different scheduling algorithms to schedule processes. Now, let us examine the advantages and disadvantages of each scheduling algorithm.

First Come First Serve (FCFS)

Advantages:

- FCFS algorithm doesn't include any complex logic, it just puts the process requests in a queue and executes it one by one.
- Hence, FCFS is pretty simple and easy to implement.
- Eventually, every process will get a chance to run, so starvation doesn't occur.

Disadvantages:

- There is no option for pre-emption of a process. If a process is started, then CPU executes the process until it ends.
- Because there is no pre-emption, if a process executes for a long time, the processes in the back of the queue will have to wait for a long time before they get a chance to be executed.

Shortest Job First (SJF)

Advantages:

- According to the definition, short processes are executed first and then followed by longer processes.
- The throughput is increased because more processes can be executed in less amount of time.

Disadvantages:

- The time taken by a process must be known by the CPU beforehand, which is not possible.
- Longer processes will have more waiting time, eventually they'll suffer starvation.

Note: Preemptive Shortest Job First scheduling will have the same advantages and disadvantages as those for SJF.

Round Robin (RR)

Advantages:

- Each process is served by the CPU for a fixed time quantum, so all processes are given the same priority.
- Starvation doesn't occur because for each round robin cycle, every process is given a fixed time to execute. No process is left behind.

Disadvantages:

- The throughput in RR largely depends on the choice of the length of the time quantum. If time quantum is longer than needed, it tends to exhibit the same behavior as FCFS.
- If time quantum is shorter than needed, the number of times that CPU switches from one process to another process, increases. This leads to decrease in CPU efficiency.

Priority based Scheduling

Advantages:

- The priority of a process can be selected based on memory requirement, time requirement or user preference. For example, a high end game will have better graphics, that means the process which updates the screen in a game will have higher priority so as to achieve better graphics performance.

Disadvantages:

- A second scheduling algorithm is required to schedule the processes which have same priority.
- In preemptive priority scheduling, a higher priority process can execute ahead of an already executing lower priority process. If lower priority process keeps waiting for higher priority processes, starvation occurs.

Usage of Scheduling Algorithms in Different Situations:

Every scheduling algorithm has a type of a situation where it is the best choice. Let's look at different such situations:

- **Situation 1:** The incoming processes are short and there is no need for the processes to execute in a specific order.
- In this case, FCFS works best when compared to SJF and RR because the processes are short which means that no process will wait for a longer time. When each process is executed one by one, every process will be executed eventually.
- **Situation 2:** The processes are a mix of long and short processes and the task will only be completed if all the processes are executed successfully in a given time.
- Round Robin scheduling works efficiently here because it does not cause starvation and also gives equal time quantum for each process.
- **Situation 3:** The processes are a mix of user based and kernel based processes.

- Priority based scheduling works efficiently in this case because generally kernel based processes have higher priority when compared to user based processes.
- For example, the scheduler itself is a kernel based process, it should run first so that it can schedule other processes.

Synchronization!

On the basis of synchronization, processes are categorized as one of the following two types:

- **Independent Process** : Execution of one process does not affects the execution of other processes.
- **Cooperative Process** : Execution of one process affects the execution of other processes.

Process synchronization problem arises in the case of Cooperative process also because resources are shared in Cooperative processes.

Process Synchronization means sharing system resources by processes in a such a way that, Concurrent access to shared data is handled thereby minimizing the chance of inconsistent data. Maintaining data consistency demands mechanisms to ensure synchronized execution of cooperating processes.

Race Condition: A race condition is where multiple processes concurrently read and write to a shared memory location and the result depends on the order of the execution.

Race Condition Example

- We will implement count++ and count-- and run them concurrently
  - Let us say they are executed by different threads accessing a global variable
  - At the end we expect count's value not to change
- count++ implementation:

```
register1 = count
register1 = register1 + 1
count = register 1
```

- count-- implementation:

```
register2 = count
register2 = register2 - 1
count = register2
```

Race Condition Example ...

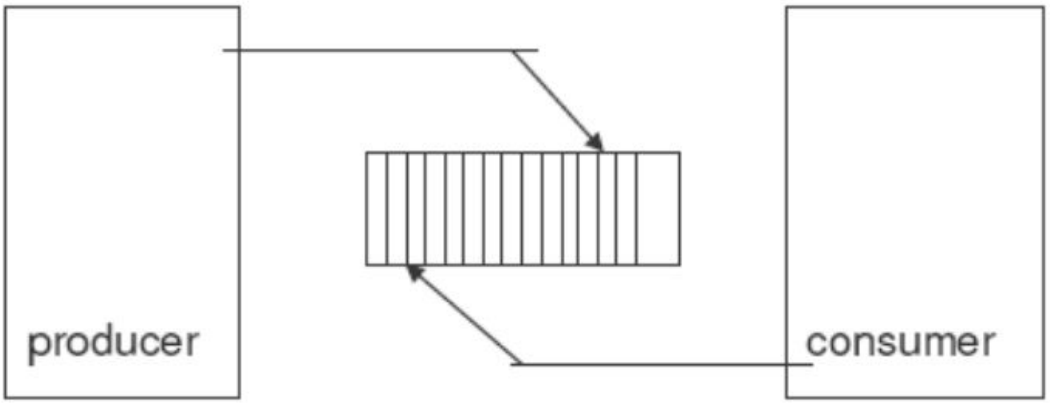
- Let count = 5 initially. One possible concurrent execution of count++ and count-- is

```
register1 = count {register1 = 5}
register1 = register1 + 1 {register1 = 6}
register2 = count {register2 = 5}
register2 = register2 - 1 {register2 = 4}
count = register1 {count = 6}
count = register2 {count = 4}
```

  - count = 4 after count++ and count--, even though we started with count = 5
  - Easy question: what other values can count be from doing this incorrectly?
- Obviously, we would like to have count++ execute, followed by count-- (or vice versa)

Producer Consumer Problem

Producer/consumer problem is more general form of the previous problem.



Producer Consumer Problem

Producer Process

```
while (true) {
        // produce an item in nextProduced
        while (counter == BUFFER_SIZE); // do nothing
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE ;
        counter++;
}
```

Consumer Process

```
while (true) {
        while (counter == 0); // do nothing
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        //consume the item in nextConsumed
}
```

Critical Section

- A **critical section** is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution.
- The goal is to provide a mechanism by which only one instance of a critical section is executing for a particular shared resource.
- Unfortunately, it is often very difficult to detect critical section bugs

Critical Section …

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

Critical Section Solutions

The critical section must **ENFORCE ALL THREE** of the following rules:

- **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
- **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
- **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

Peterson's Solution

Peterson's solution requires the two processes to share two data items:

```
int turn;
boolean flag[2];
```

```
do {
        flag [i] = TRUE;
        turn= j;
        while (flag[j] &&  turn j);

                critical section

        flag [i] = FALSE;

                remainder section

} while (true);
```

Synchronization using Locks

```
do {
        acquire lock

                critical section

        release lock

                remainder section
} while (TRUE);
```

Read about TestandSet() and Swap() method for locking Method!

What are semaphores?

- Semaphore *S* – integer variable
- Can only be accessed via two **indivisible (atomic)** operations wait and signal

```
wait (S) {
        while (S≤ 0);
          // do no-op
        S--;
}

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

Semaphores contd..

- Semaphore could be counting or binary (1 or 0).
- Binary Semaphores are generally called as mutex locks as that provide mutual exclusion
- Counting semaphores can be used to control access to a given resource consisting of a finite number of instance.

Critical Section of N Processes

- Shared data:

  semaphore mutex; //initially *mutex* = 1

- Process *Pi:*

```
do {
  wait(mutex);
        critical section
  signal(mutex);
        remainder section
} while (1);
```

Semaphores …

Current Implementation doesn't solve busy waiting problem.

```
typedef struct {
        int value;
        struct process *list;
} semaphore;

wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}

signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P fron< S->list;
                wakeup(P);
        }
}
```

Producer Consumer – Semaphores

- Shared :
  semaphore full, empty, mutex;
- Initially:
  full = 0, empty = n, mutex = 1

Producer Process

```
do {
                    …
                      produce an item in nextp
                    …
                    wait(empty);
                    wait(mutex);
                    …
                      add nextp to buffer
                     …
                    signal(mutex);
                    signal(full);
  } while (1);
```

Consumer Process

```
do {
                    wait(full);
                    wait(mutex);
                    ...
                      remove an item from buffer
   to nextc         ...
                    signal(mutex);
                    signal(empty);
                     ...
                      consume the item in nextc
                     ...
} while (1);
```

Consider a situation where we have a file shared between many people.

- If one of the people tries editing the file, no other person should be reading or writing at the same time, otherwise changes will not be visible to him/her.
- However if some person is reading the file, then others may read it at the same time.

Precisely in OS we call this situation as the **readers-writers problem**

Problem parameters:

- One set of data is shared among a number of processes
- Once a writer is ready, it performs its write. Only one writer may write at a time
- If a process is writing, no other process can read it
- If at least one reader is reading, no other process can write
- Readers may not write and only read

Readers Writers - Sempahores

- Shared :
  Semaphore mutex, wrt;
- Initially:
  wrt= 1, mutex = 1, readcount = 0

Writer Process

```
wait(wrt)
        …
                writing is performed
        …
signal(wrt)
```

Readers Process

```
wait(mutex);
readcount+;
if (readcount == 1)
                wait(rt);
signal(mutex);
…
                reading is performed
…
wait(mutex);
readcount--;
if (readcount == 0)
                signal(wrt);
signal(mutex);
```

Deadlock Introduction

A process in operating systems uses different resources and uses resources in following way.

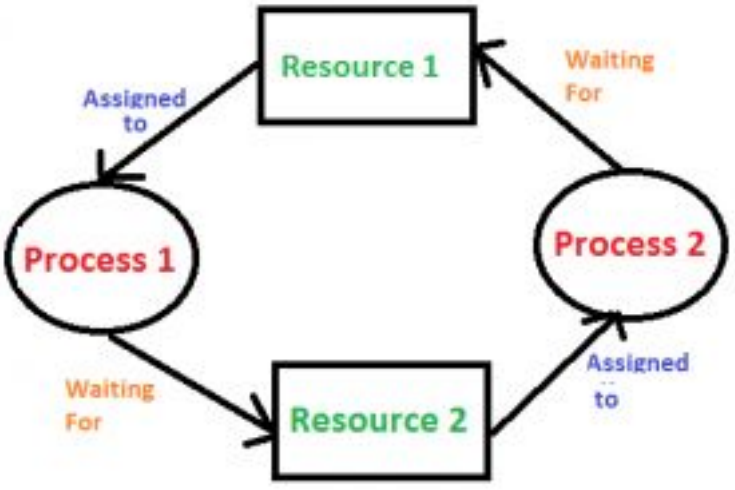
1) Requests a resource

2) Use the resource

2) Releases the resource

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

Consider an example when two trains are coming toward each other on same track and there is only one track, none of the trains can move once they are in front of each other. Similar situation occurs in operating systems when there are two or more processes hold some resources and wait for resources held by other(s). For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.



Deadlock can arise if following four conditions hold simultaneously (Necessary Conditions)

Mutual Exclusion: One or more than one resource are non-sharable (Only one process can use at a time)

Hold and Wait: A process is holding at least one resource and waiting for resources.

No Preemption: A resource cannot be taken from a process unless the process releases the resource.

Circular Wait: A set of processes are waiting for each other in circular form.

Methods for handling deadlock

There are three ways to handle deadlock

1) Deadlock prevention or avoidance: The idea is to not let the system into deadlock state.

2) Deadlock detection and recovery: Let deadlock occur, then do preemption to handle it once occurred.

3) Ignore the problem all together: If deadlock is very rare, then let it happen and reboot the system. This is the approach that both Windows and UNIX take.

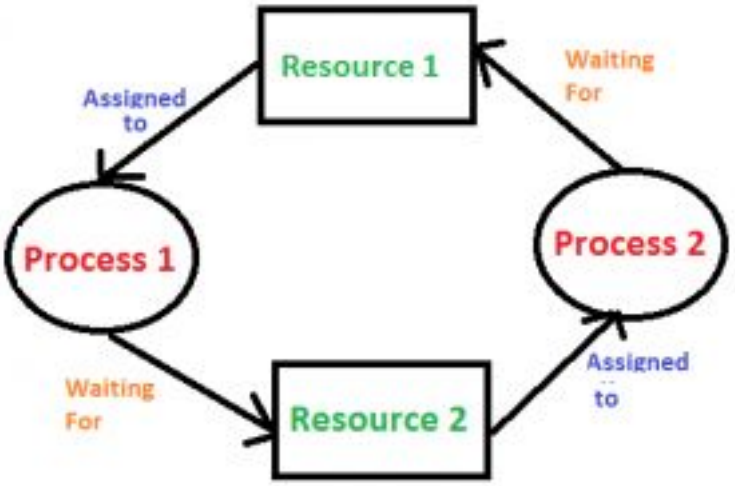
Deadlock Detection And Recovery

In the previous post, we have discussed Deadlock Prevention and Avoidance. In this post, Deadlock Detection and Recovery technique to handle deadlock is discussed.

Deadlock Detection

1. If resources have single instance:

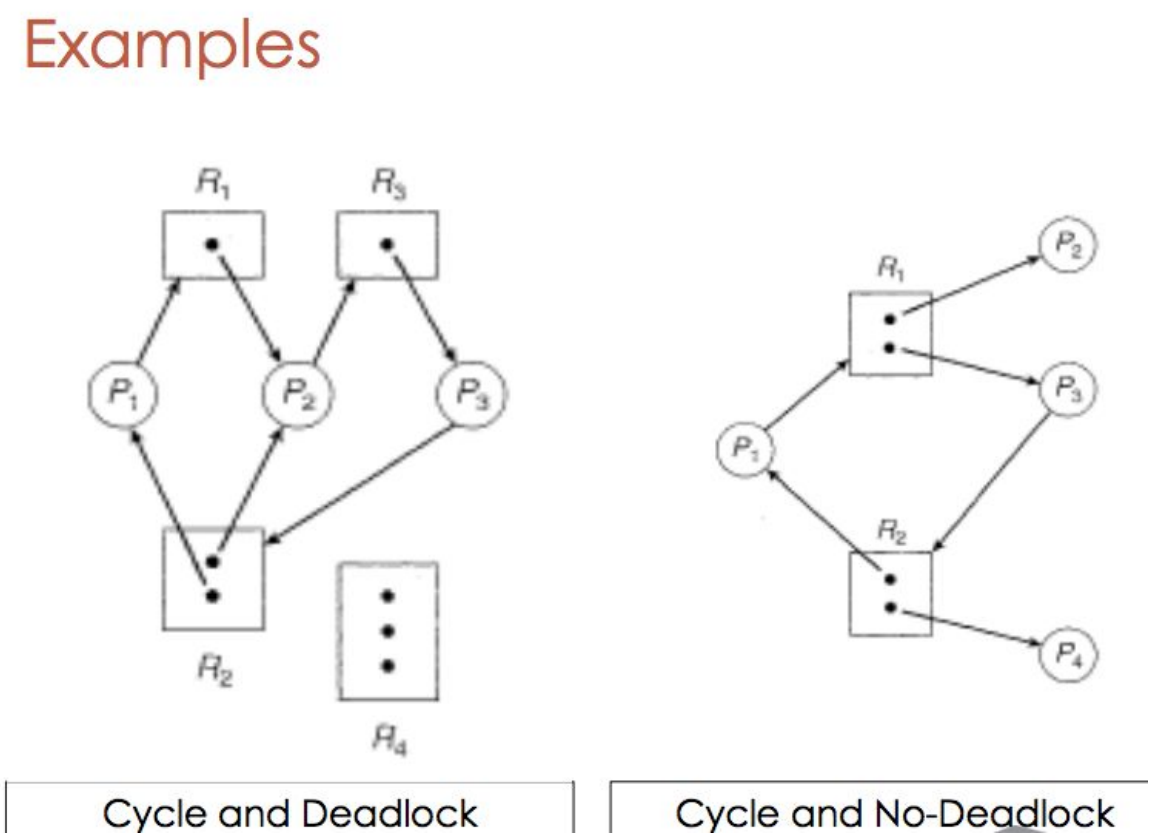
In this case for Deadlock detection we can run an algorithm to check for cycle in the Resource Allocation Graph. Presence of cycle in the graph is the sufficient condition for deadlock.



In the above diagram, resource 1 and resource 2 have single instances. There is a cycle R1–>P1–>R2–>P2. So Deadlock is Confirmed.

2. If there are multiple instances of resources:

Detection of cycle is necessary but not sufficient condition for deadlock detection, in this case system may or may not be in deadlock varies according to different situations.



Deadlock Recovery

Traditional operating system such as Windows doesn't deal with deadlock recovery as it is time and space consuming process. Real time operating systems use Deadlock recovery.

Recovery method

1. Killing the process.

```
    killing all the process involved in deadlock.
```

```
     Killing process one by one. After killing each
     process check for deadlock again keep repeating
     process till system recover from deadlock.
```

2. Resource Preemption

Resources are preempted from the processes involved in deadlock, preempted resources are allocated to other processes, so that their is a possibility of recovering the system from deadlock. In this case system go into starvation.

Deadlock Characteristics

As discussed in the previous post, deadlock has following characteristics.

```
Mutual Exclusion.
Hold and Wait.
No preemption.
Circular wait.
```

Deadlock Prevention

We can prevent Deadlock by eliminating any of the above four condition.

Eliminate Mutual Exclusion

It is not possible to dis-satisfy the mutual exclusion because some resources, such as the tap drive and printer, are inherently non-shareable.

Eliminate Hold and wait

1. Allocate all required resources to the process before start of its execution, this way hold and wait condition is eliminated but it will lead to low device utilization. for example, if a process requires printer at a later time and we have allocated printer before the start of its execution printer will remained blocked till it has completed its execution.

2. Process will make new request for resources after releasing the current set of resources. This solution may lead to starvation.

Eliminate No Preemption

Preempt resources from process when resources required by other high priority process.

Eliminate Circular Wait

Each resource will be assigned with a numerical number. A process can request for the resources only in increasing order of numbering.

For Example, if P1 process is allocated R5 resources, now next time if P1 ask for R4, R3 lesser than R5 such request will not be granted, only request for resources more than R5 will be granted.

Deadlock Avoidance

Deadlock avoidance can be done with Banker's Algorithm.

Banker's Algorithm

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

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

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

Available :

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

Max :

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

Allocation :

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

Need :

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

$Allocation_i$ specifies the resources currently allocated to process P_i and $Need_i$ specifies the additional resources that process P_i may still request to complete its task.

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

Safety Algorithm

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

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

   Initialize: Work = Available

   Finish [i] = false; for i=1, 2, ..., n

2) Find an i such that both

   a) Finish [i] = false

   b) $Need_i \leq work$

      If no such i exists goto step (4)

3) Work = Work + $Allocation_i$

   Finish [i] = true

   goto step (2)

4) If Finish [i] = true for all i,

   then the system is in safe state.

Resource-Request Algorithm

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

1) If $Request_i \leq Need_i$

   Goto step (2) ; otherwise, raise an error condition, since the process has exceeded its maximum claim.

2) If $Request_i \leq Available$

   Goto step (3); otherwise, P_i must wait, since the resources are not available.

3) Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:

   $Available = Available - Request_i$

   $Allocation_i = Allocation_i + Request_i$

   $Need_i = Need_i - Request_i$

Example:

Considering a system with five processes P_0 through P_4 and three resources types A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time $t_0$following snapshot of the system has been taken:

Process	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Question1. What will be the content of the Need matrix?

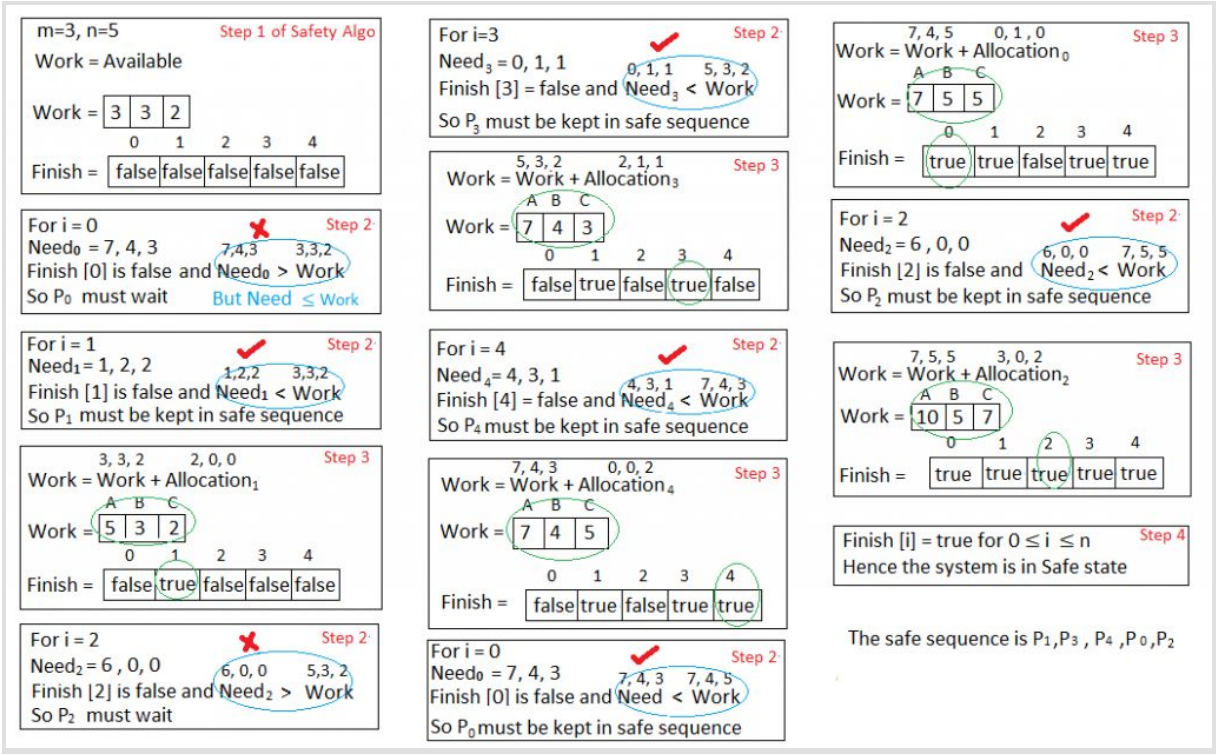
Need [i, j] = Max [i, j] – Allocation [i, j]

So, the content of Need Matrix is:

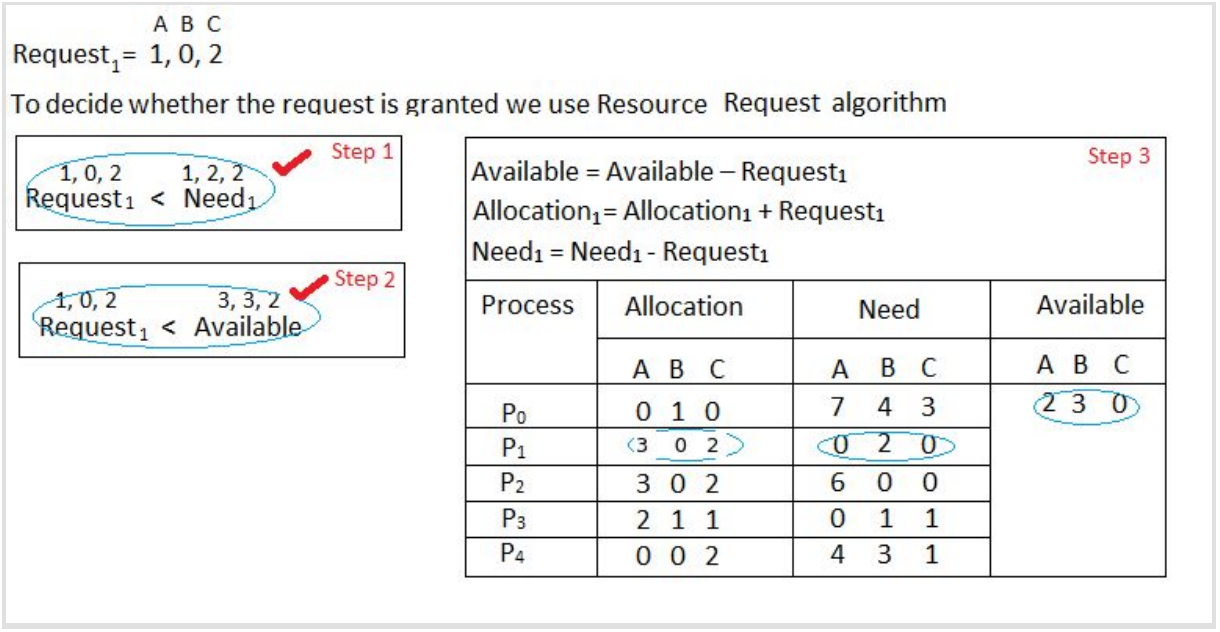
Process	Need		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

Question2. Is the system in safe state? If Yes, then what is the safe sequence?

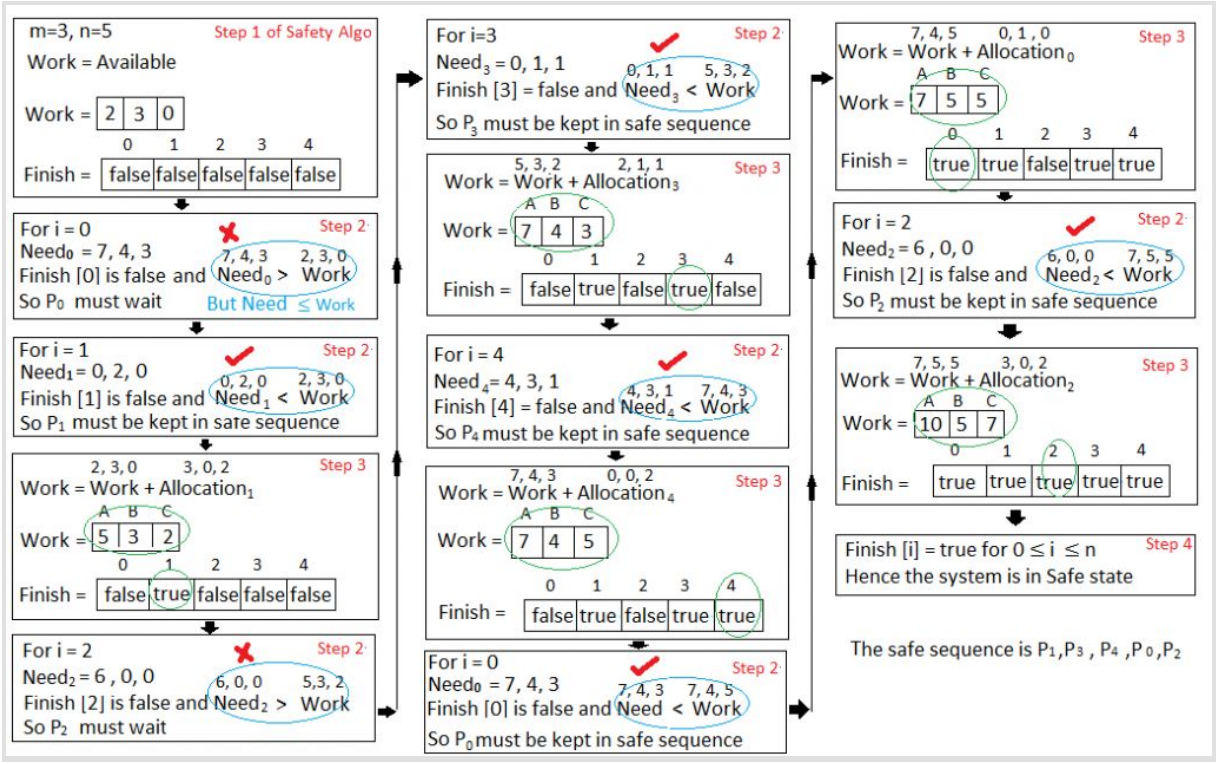
Applying the Safety algorithm on the given system,



Question3. What will happen if process P1 requests one additional instance of resource type A and two instances of resource type C?



We must determine whether this new system state is safe. To do so, we again execute Safety algorithm on the above data structures.



Hence the new system state is safe, so we can immediately grant the request for process **P1** .

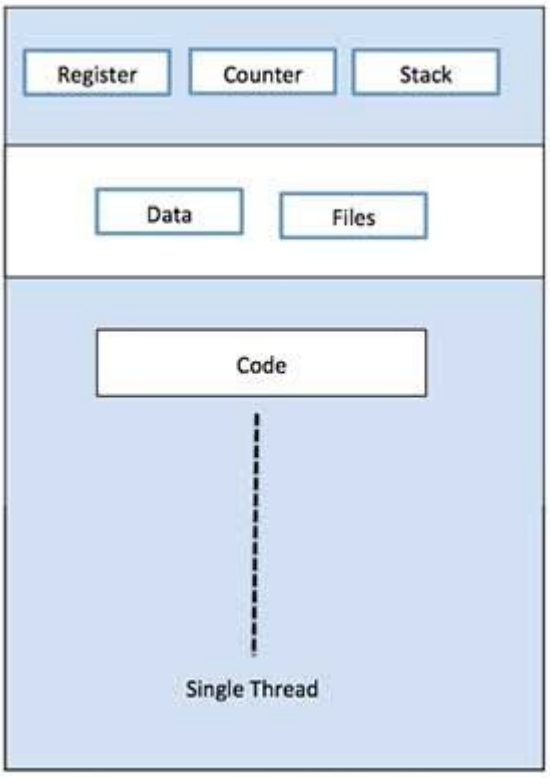
What is Thread?

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

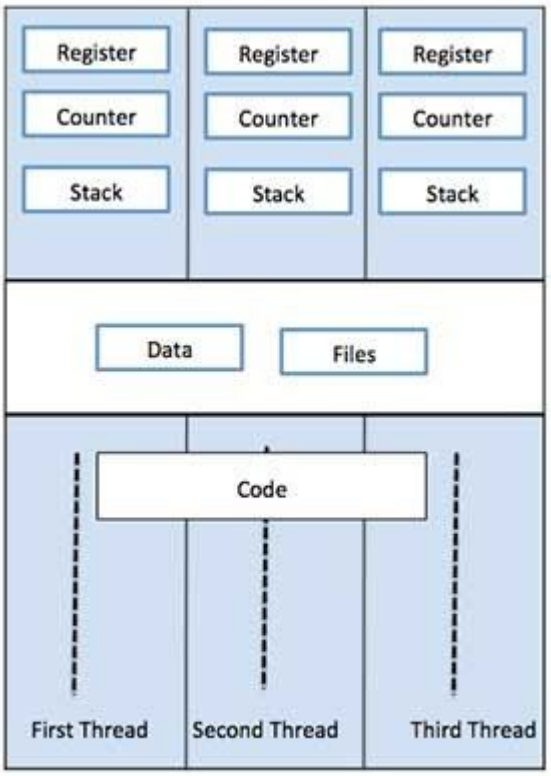
A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

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

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



Single Process P with single thread



Single Process P with three threads

Difference between Process and Thread

S.N.	Process	Thread
1	**Process is heavy weight or resource intensive.**	**Thread is light weight, taking lesser resources than a process.**
2	**Process switching needs interaction with operating system.**	**Thread switching does not need to interact with operating system.**
3	**In multiple processing environments, each process executes the same code but has its own memory and file resources.**	**All threads can share same set of open files, child processes.**
4	**If one process is blocked, then no other process can execute until the first process is unblocked.**	**While one thread is blocked and waiting, a second thread in the same task can run.**
5	**Multiple processes without using threads use more resources.**	**Multiple threaded processes use fewer resources.**
6	**In multiple processes each process operates independently of the others.**	**One thread can read, write or change another thread's data.**

Advantages of Thread

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

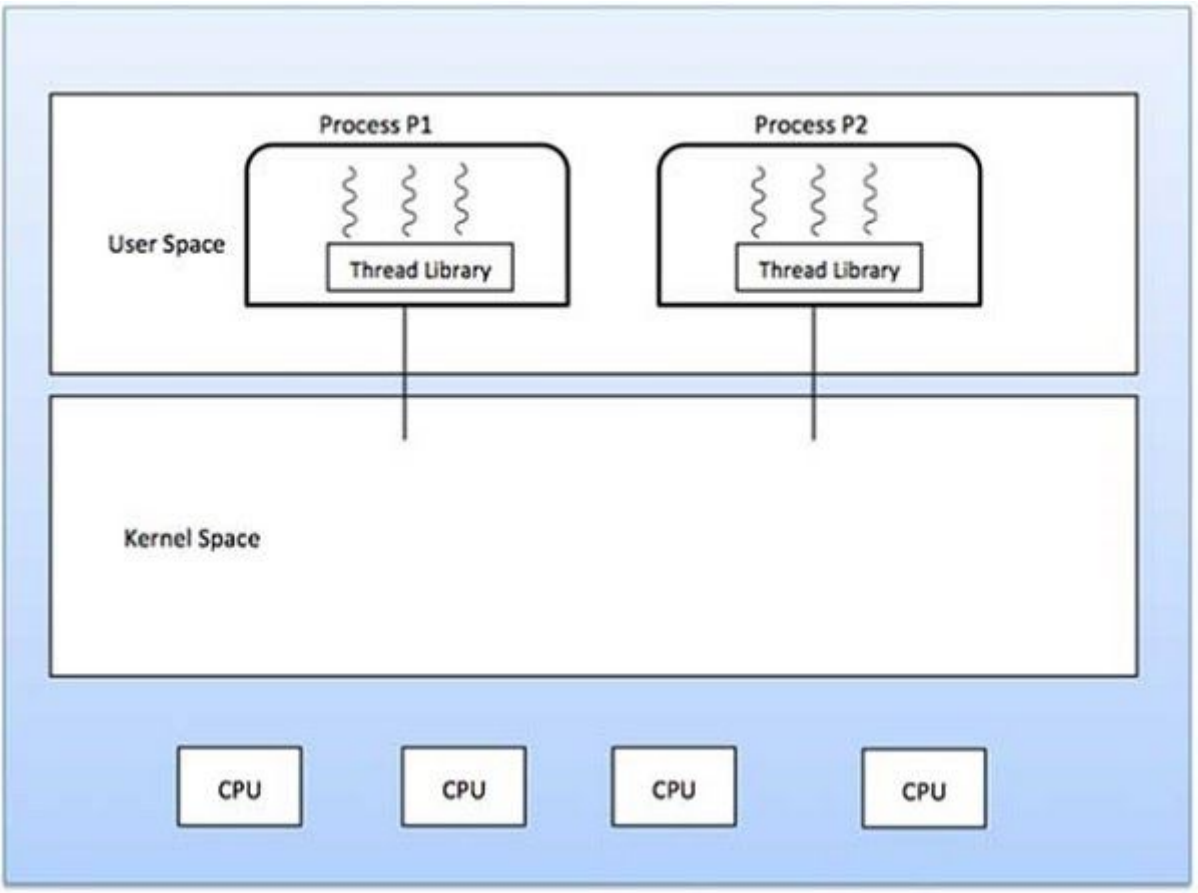
Types of Thread

Threads are implemented in following two ways –

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

User Level Threads

In this case, the thread management kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. The application starts with a single thread.



Advantages

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

Disadvantages

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

Kernel Level Threads

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

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

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

Advantages

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

Disadvantages

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

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

This tutorial will teach you basic concepts related to Memory Management.

Process Address Space

The process address space is the set of logical addresses that a process references in its code. For example, when 32-bit addressing is in use, addresses can range from 0 to 0x7fffffff; that is, 2^31 possible numbers, for a total theoretical size of 2 gigabytes.

The operating system takes care of mapping the logical addresses to physical addresses at the time of memory allocation to the program. There are three types of addresses used in a program before and after memory is allocated –

S.N.	Memory Addresses & Description
1	**Symbolic addresses** **The addresses used in a source code. The variable names, constants, and instruction labels are the basic elements of the symbolic address space.**

2	**Relative addresses** **At the time of compilation, a compiler converts symbolic addresses into relative addresses.**
3	**Physical addresses** **The loader generates these addresses at the time when a program is loaded into main memory.**

Virtual and physical addresses are the same in compile-time and load-time address-binding schemes. Virtual and physical addresses differ in execution-time address-binding scheme.

The set of all logical addresses generated by a program is referred to as a logical address space. The set of all physical addresses corresponding to these logical addresses is referred to as a physical address space.

The runtime mapping from virtual to physical address is done by the memory management unit (MMU) which is a hardware device. MMU uses following mechanism to convert virtual address to physical address.

- **The value in the base register is added to every address generated by a user process, which is treated as offset at the time it is sent to memory. For example, if the base register value is 10000, then an attempt by the user to use address location 100 will be dynamically reallocated to location 10100.**
- **The user program deals with virtual addresses; it never sees the real physical addresses.**

Static vs Dynamic Loading

The choice between Static or Dynamic Loading is to be made at the time of computer program being developed. If you have to load your program statically, then at the time of compilation, the complete programs will be compiled and linked without leaving any external program or module dependency. The linker combines the object program with other necessary object modules into an absolute program, which also includes logical addresses.

If you are writing a Dynamically loaded program, then your compiler will compile the program and for all the modules which you want to include dynamically, only references will be provided and rest of the work will be done at the time of execution.

At the time of loading, with static loading, the absolute program (and data) is loaded into memory in order for execution to start.

If you are using dynamic loading, dynamic routines of the library are stored on a disk in relocatable form and are loaded into memory only when they are needed by the program.

Static vs Dynamic Linking

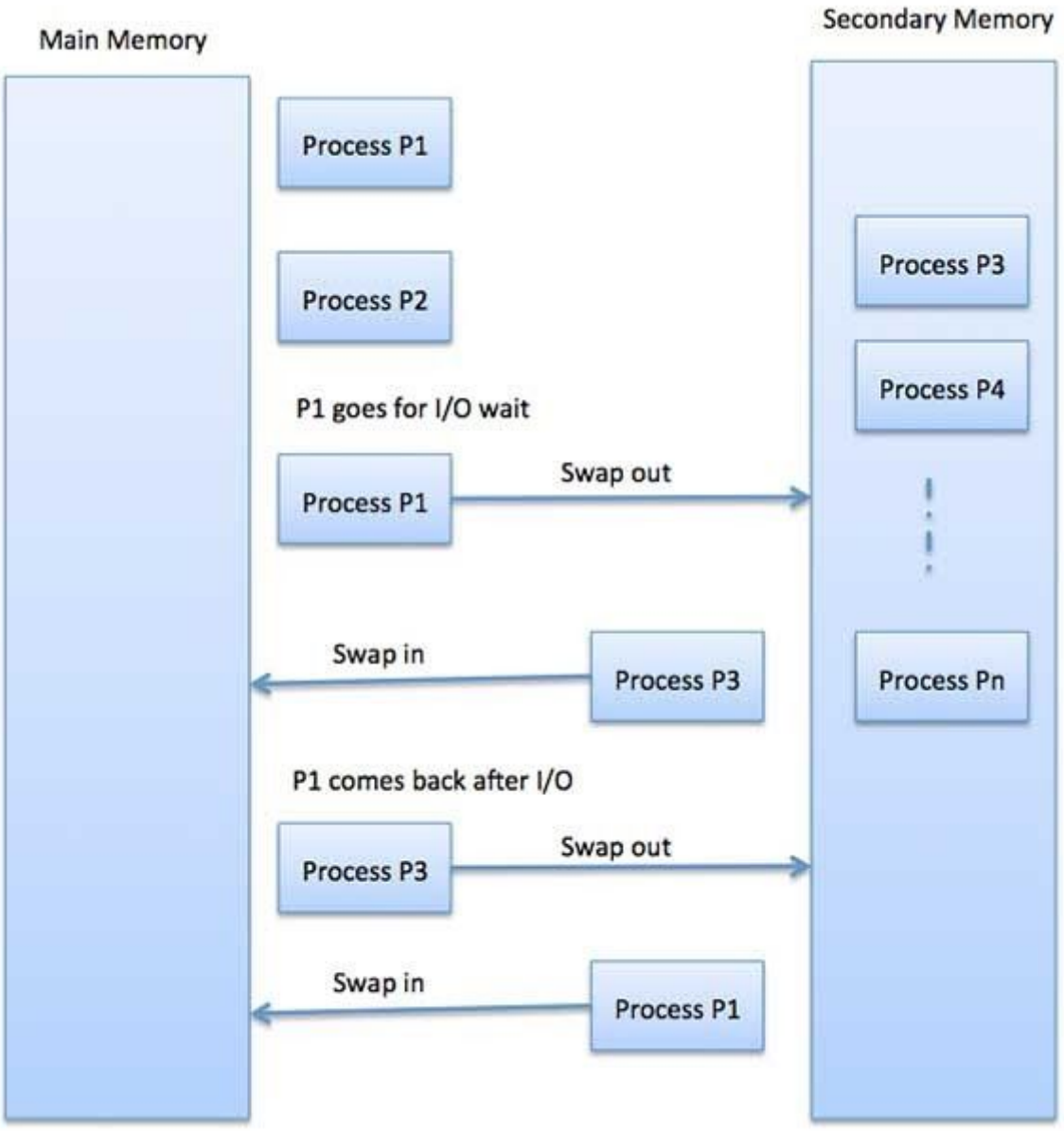
As explained above, when static linking is used, the linker combines all other modules needed by a program into a single executable program to avoid any runtime dependency.

When dynamic linking is used, it is not required to link the actual module or library with the program, rather a reference to the dynamic module is provided at the time of compilation and linking. Dynamic Link Libraries (DLL) in Windows and Shared Objects in Unix are good examples of dynamic libraries.

Swapping

Swapping is a mechanism in which a process can be swapped temporarily out of main memory (or move) to secondary storage (disk) and make that memory available to other processes. At some later time, the system swaps back the process from the secondary storage to main memory.

Though performance is usually affected by swapping process but it helps in running multiple and big processes in parallel and that's the reason Swapping is also known as a technique for memory compaction.



The total time taken by swapping process includes the time it takes to move the entire process to a secondary disk and then to copy the process

back to memory, as well as the time the process takes to regain main memory.

Let us assume that the user process is of size 2048KB and on a standard hard disk where swapping will take place has a data transfer rate around 1 MB per second. The actual transfer of the 1000K process to or from memory will take

```
2048KB / 1024KB per second
= 2 seconds
= 2000 milliseconds
```

Now considering in and out time, it will take complete 4000 milliseconds plus other overhead where the process competes to regain main memory.

Fragmentation

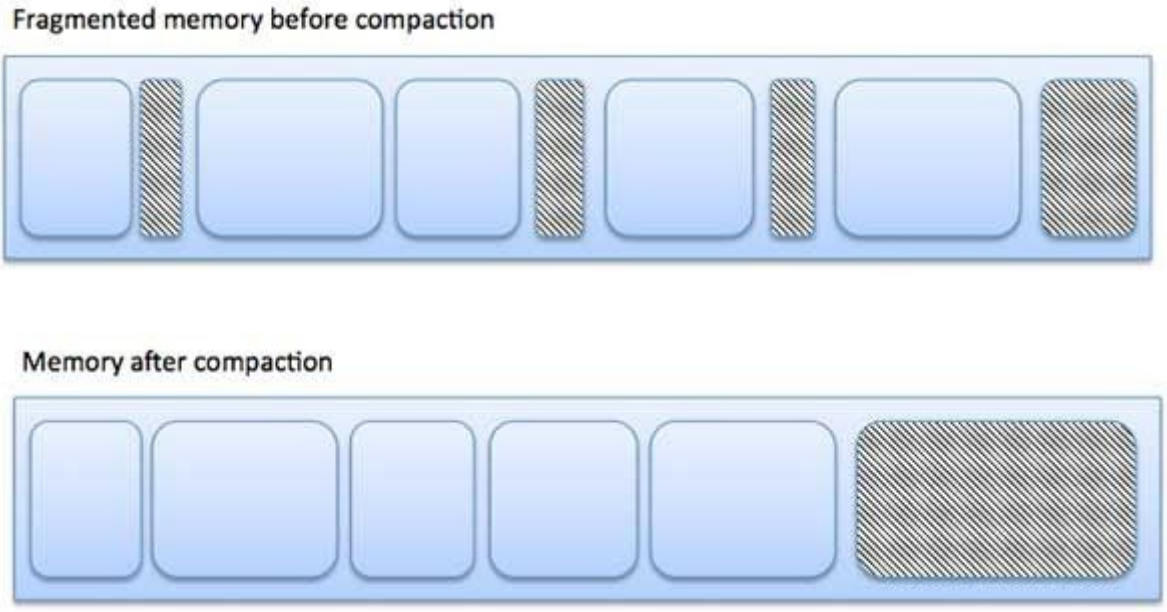
As processes are loaded and removed from memory, the free memory space is broken into little pieces. It happens after sometimes that processes cannot be allocated to memory blocks considering their small size and memory blocks remains unused. This problem is known as Fragmentation.

Fragmentation is of two types –

S.N.	Fragmentation & Description
1	**External fragmentation** **Total memory space is enough to satisfy a request or to reside a process in it, but it is not contiguous, so it cannot be used.**

2	**Internal fragmentation** **Memory block assigned to process is bigger. Some portion of memory is left unused, as it cannot be used by another process.**

The following diagram shows how fragmentation can cause waste of memory and a compaction technique can be used to create more free memory out of fragmented memory –



External fragmentation can be reduced by compaction or shuffle memory contents to place all free memory together in one large block. To make compaction feasible, relocation should be dynamic.

The internal fragmentation can be reduced by effectively assigning the smallest partition but large enough for the process.

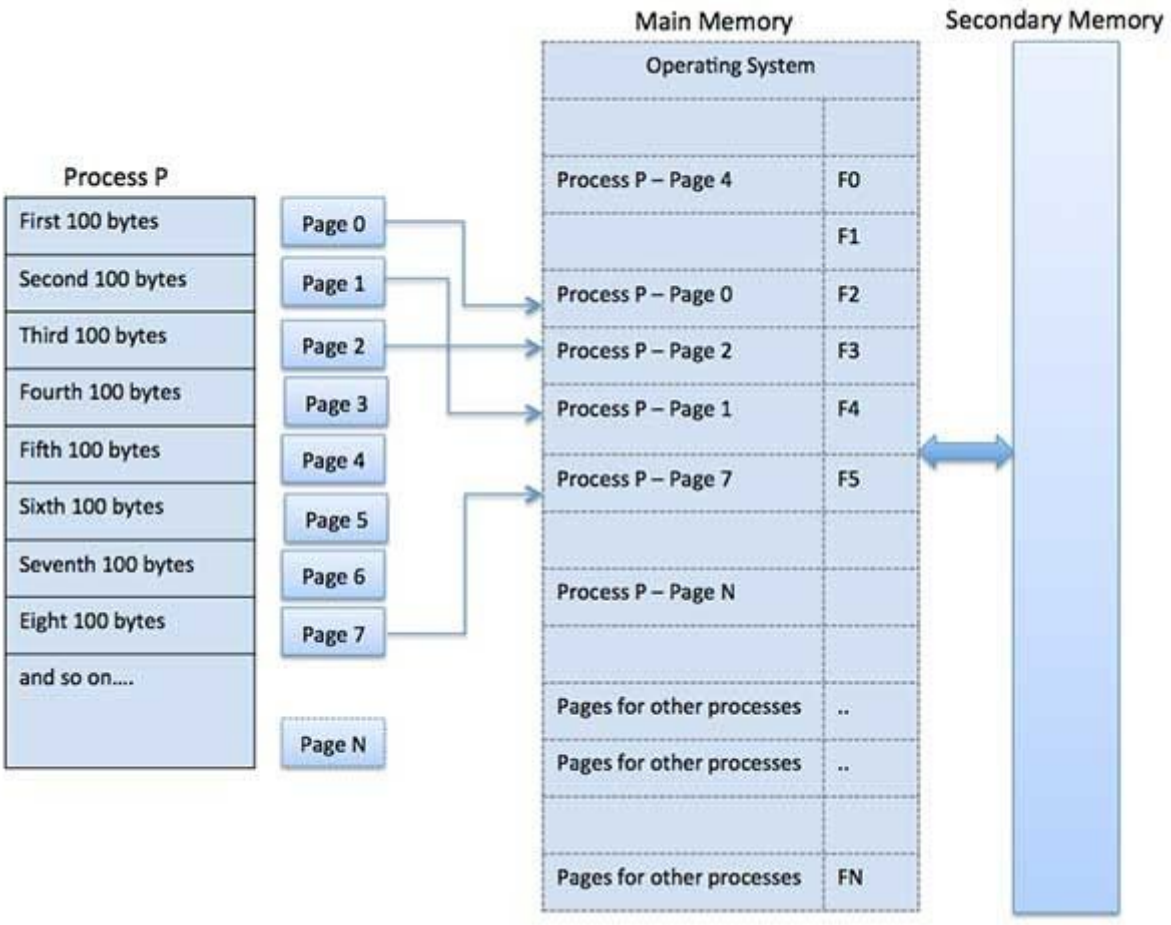
Paging

A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it

is a section of a hard that's set up to emulate the computer's RAM. Paging technique plays an important role in implementing virtual memory.

Paging is a memory management technique in which process address space is broken into blocks of the same size called pages (size is power of 2, between 512 bytes and 8192 bytes). The size of the process is measured in the number of pages.

Similarly, main memory is divided into small fixed-sized blocks of (physical) memory called frames and the size of a frame is kept the same as that of a page to have optimum utilization of the main memory and to avoid external fragmentation.



Address Translation

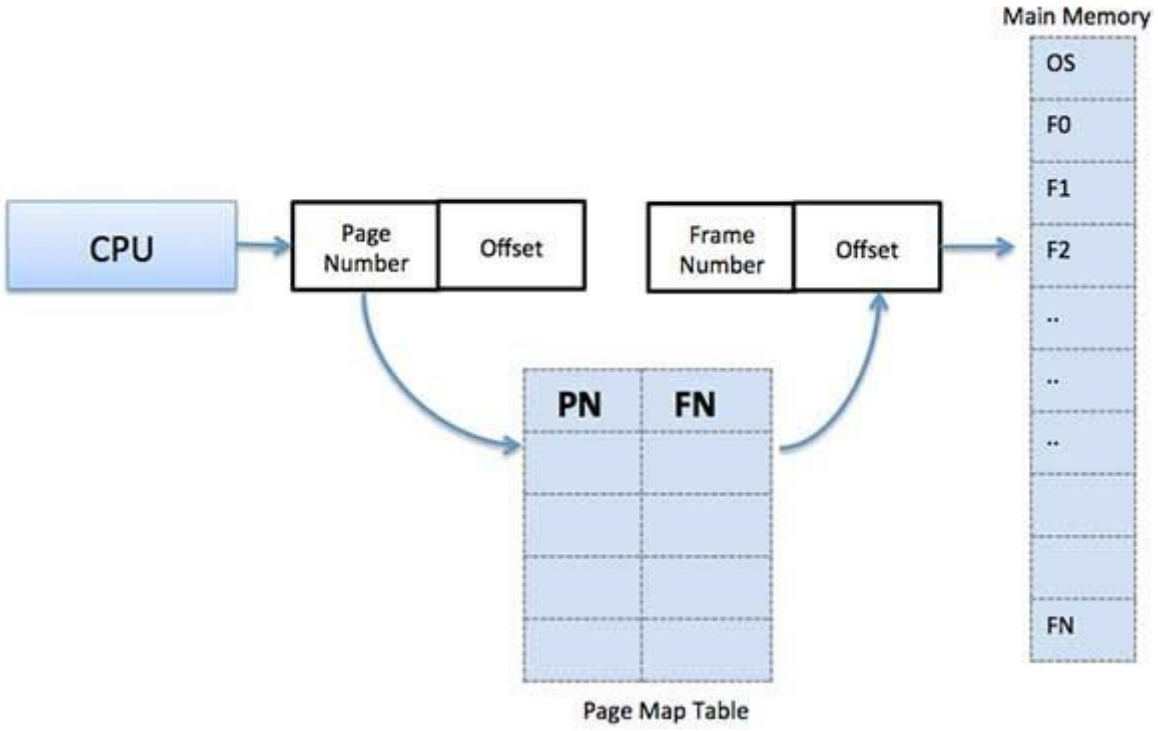
Page address is called logical address and represented by page numberand the offset.

```
Logical Address = Page number + page offset
```

Frame address is called physical address and represented by a frame number and the offset.

```
Physical Address = Frame number + page offset
```

A data structure called page map table is used to keep track of the relation between a page of a process to a frame in physical memory.



When the system allocates a frame to any page, it translates this logical address into a physical address and create entry into the page table to be used throughout execution of the program.

When a process is to be executed, its corresponding pages are loaded into any available memory frames. Suppose you have a program of 8Kb but your memory can accommodate only 5Kb at a given point in time, then the paging concept will come into picture. When a computer runs out of RAM, the operating system (OS) will move idle or unwanted pages of memory to secondary memory to free up RAM for other processes and brings them back when needed by the program.

This process continues during the whole execution of the program where the OS keeps removing idle pages from the main memory and write them onto the secondary memory and bring them back when required by the program.

Advantages and Disadvantages of Paging

Here is a list of advantages and disadvantages of paging –

- **Paging reduces external fragmentation, but still suffer from internal fragmentation.**
- **Paging is simple to implement and assumed as an efficient memory management technique.**
- **Due to equal size of the pages and frames, swapping becomes very easy.**
- **Page table requires extra memory space, so may not be good for a system having small RAM.**

Segmentation

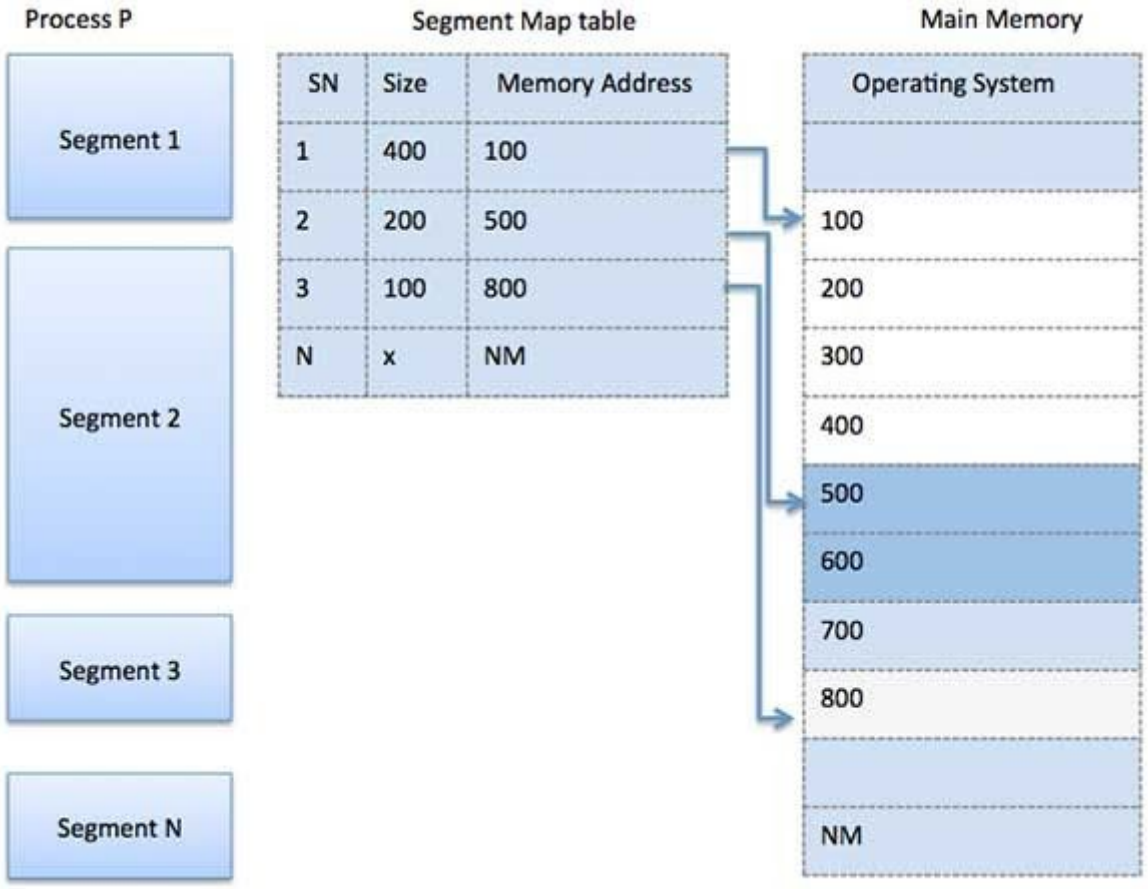
Segmentation is a memory management technique in which each job is divided into several segments of different sizes, one for each module that contains pieces that perform related functions. Each segment is actually a different logical address space of the program.

When a process is to be executed, its corresponding segmentation are loaded into non-contiguous memory though every segment is loaded into a contiguous block of available memory.

Segmentation memory management works very similar to paging but here segments are of variable-length where as in paging pages are of fixed size.

A program segment contains the program's main function, utility functions, data structures, and so on. The operating system maintains a segment map table for every process and a list of free memory blocks along with

segment numbers, their size and corresponding memory locations in main memory. For each segment, the table stores the starting address of the segment and the length of the segment. A reference to a memory location includes a value that identifies a segment and an offset.



Virtual Memory | Operating System

Virtual Memory is a storage allocation scheme in which secondary memory can be addressed as though it were part of main memory. The addresses a program may use to reference memory are distinguished from the addresses the memory system uses to identify physical storage sites, and program generated addresses are translated automatically to the corresponding machine addresses.

The size of virtual storage is limited by the addressing scheme of the computer system and amount of secondary memory is available not by the actual number of the main storage locations.

It is a technique that is implemented using both hardware and software. It maps memory addresses used by a program, called virtual addresses, into physical addresses in computer memory.

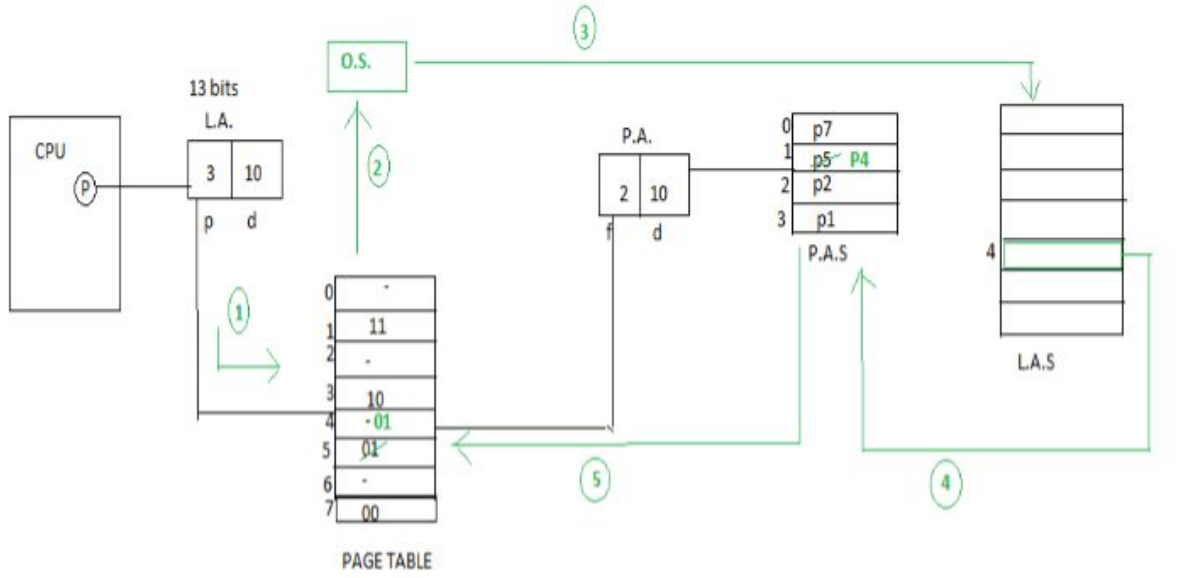
1. **All memory references within a process are logical addresses that are dynamically translated into physical addresses at run time. This means that a process can be swapped in and out of main memory such that it occupies different places in main memory at different times during the course of execution.**
2. **A process may be broken into number of pieces and these pieces need not be continuously located in the main memory during execution. The combination of dynamic run-time addres translation and use of page or segment table permits this.**

If these characteristics are present then, it is not necessary that all the pages or segments are present in the main memory during execution. This means that the required pages need to be loaded into memory whenever required. Virtual memory is implemented using Demand Paging or Demand Segmentation.

Demand Paging :

The process of loading the page into memory on demand (whenever page fault occurs) is known as demand paging.

The process includes the following steps:



1. **If CPU try to refer a page that is currently not available in the main memory, it generates an interrupt indicating memory access fault.**
2. **The OS puts the interrupted process in a blocking state. For the execution to proceed the OS must bring the required page into the memory.**
3. **The OS will search for the required page in the logical address space.**
4. **The required page will be brought from logical address space to physical address space. The page replacement algorithms are used for the decision making of replacing the page in physical address space.**
5. **The page table will updated accordingly.**
6. **The signal will be sent to the CPU to continue the program execution and it will place the process back into ready state.**

Hence whenever a page fault occurs these steps are followed by the operating system and the required page is brought into memory.

Advantages :

- **More processes may be maintained in the main memory: Because we are going to load only some of the pages of any particular process, there is room for more processes. This leads to more efficient utilization of the processor because it is more likely that at least one of the more numerous processes will be in the ready state at any particular time.**

- **A process may be larger than all of main memory: One of the most fundamental restrictions in programming is lifted. A process larger than the main memory can be executed because of demand paging. The OS itself loads pages of a process in main memory as required.**
- **It allows greater multiprogramming levels by using less of the available (primary) memory for each process.**

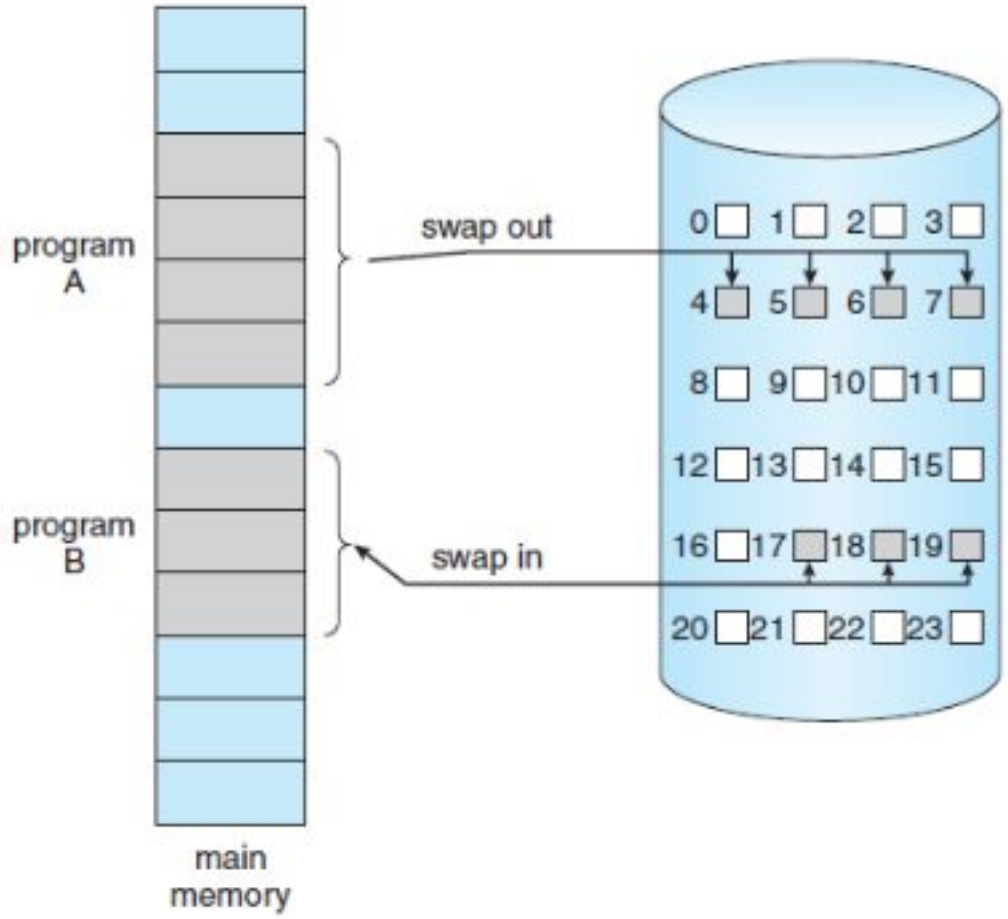
Page Fault Service Time :

The time taken to service the page fault is called as page fault service time. The page fault service time includes the time taken to perform all the above six steps.

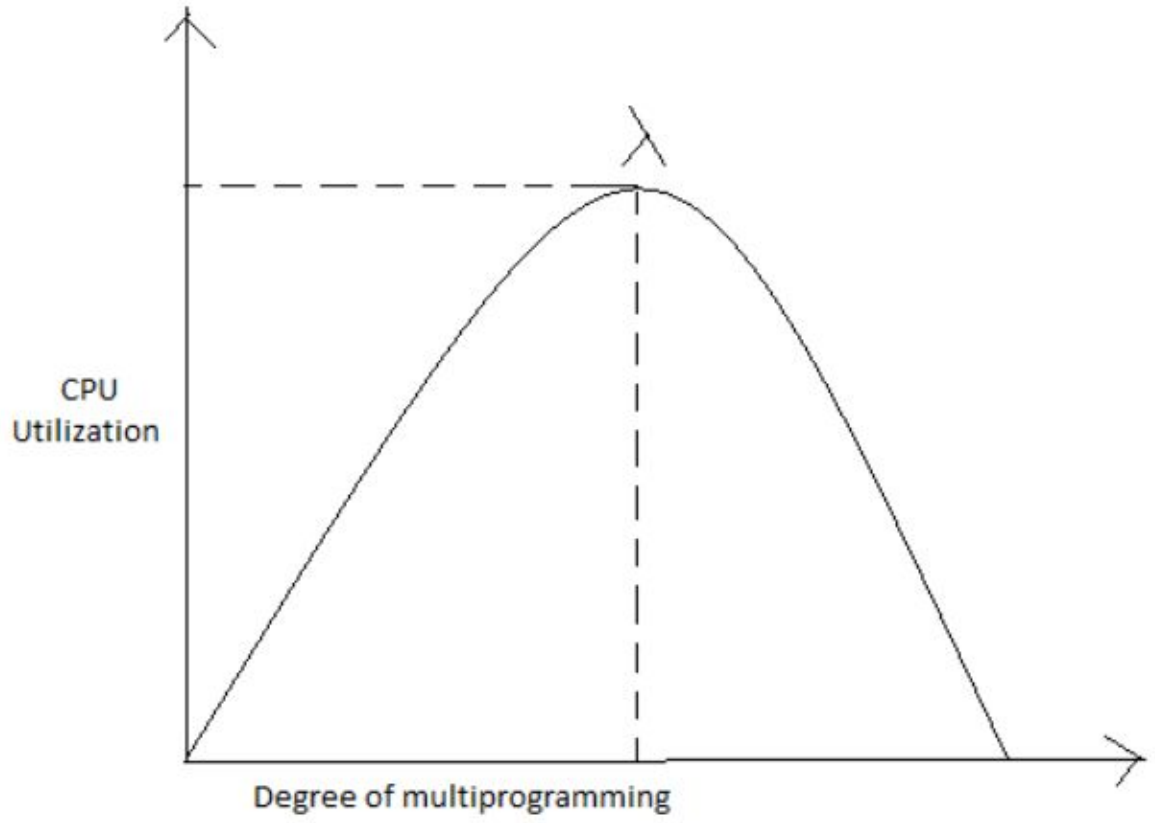
```
Let Main memory access time is: m
Page fault service time is: s
Page fault rate is : p
Then, Effective memory access time = (p*s) + (1-p)*m
```

Swapping:

Swapping a process out means removing all of its pages from memory, or marking them so that they will be removed by the normal page replacement process. Suspending a process ensures that it is not runnable while it is swapped out. At some later time, the system swaps back the process from the secondary storage to main memory. When a process is busy swapping pages in and out then this situation is called thrashing.


program
A
program
B
swap out
swap in
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
16 17 18 19
20 21 22 23
main
memory

Thrashing :



At any given time, only few pages of any process are in main memory and therefore more processes can be maintained in memory. Furthermore time is saved because unused pages are not swapped in and out of memory. However, the OS must be clever about how it manages this scheme. In the steady state practically, all of main memory will be occupied with process's pages, so that the processor and OS has direct access to as many processes as possible. Thus when the OS brings one page in, it must throw another out. If it throws out a page just before it is used, then it will just have to get that page again almost immediately. Too much of this leads to a condition called Thrashing. The system spends most of its time swapping pages rather than executing instructions. So a good page replacement algorithm is required.

In the given diagram, initial degree of multi programming upto some extent of point(lamda), the CPU utilization is very high and the system resources are utilized 100%. But if we further increase the degree of multi programming the CPU utilization will drastically fall down and the

system will spent more time only in the page replacement and the time taken to complete the execution of the process will increase. This situation in the system is called as thrashing.

Causes of Thrashing :

1. **High degree of multiprogramming : If the number of processes keeps on increasing in the memory than number of frames allocated to each process will be decreased. So, less number of frames will be available to each process. Due to this, page fault will occur more frequently and more CPU time will be wasted in just swapping in and out of pages and the utilization will keep on decreasing.**
2. **For example:**
3. **Let free frames = 400**
4. **Case 1: Number of process = 100**
5. **Then, each process will get 4 frames.**
6. **Case 2: Number of process = 400**
7. **Each process will get 1 frame.**
8. **Case 2 is a condition of thrashing, as the number of processes are increased,frames per process are decreased. Hence CPU time will be consumed in just swapping pages.**
9. **Lacks of Frames:If a process has less number of frames then less pages of that process will be able to reside in memory and hence more frequent swapping in and out will be required. This may lead to thrashing. Hence sufficient amount of frames must be allocated to each process in order to prevent thrashing.**

Recovery of Thrashing :

- **Do not allow the system to go into thrashing by instructing the long term scheduler not to bring the processes into memory after the threshold.**
- **If the system is already in thrashing then instruct the mid term schedular to suspend some of the processes so that we can recover the system from thrashing.**

Operating System | Page Replacement Algorithms

In a operating systems that use paging for memory management, page replacement algorithm are needed to decide which page needed to be replaced when new page comes in. Whenever a new page is referred and not present in memory, page fault occurs and Operating System replaces one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.

Page Fault

A page fault is a type of interrupt, raised by the hardware when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory.

Page Replacement Algorithms

First In First Out

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

For example, consider page reference string 1, 3, 0, 3, 5, 6 and 3 page slots.

Initially all slots are empty, so when 1, 3, 0 came they are allocated to the empty slots —> 3 Page Faults.

when 3 comes, it is already in memory so —> 0 Page Faults.

Then 5 comes, it is not available in memory so it replaces the oldest page slot i.e 1. —>1 Page Fault.

Finally 6 comes, it is also not available in memory so it replaces the oldest page slot i.e 3 —>1 Page Fault.

Belady's anomaly

Belady's anomaly proves that it is possible to have more page faults when increasing the number of page frames while using the First in First Out (FIFO) page replacement algorithm. For example, if we consider reference string 3 2 1 0 3 2 4 3 2 1 0 4 and 3 slots, we get 9 total page faults, but if we increase slots to 4, we get 10 page faults.

Optimal Page replacement

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

Let us consider page reference string 7 0 1 2 0 3 0 4 2 3 0 3 2 and 4 page slots.

Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> 4 Page faults

0 is already there so —> 0 Page fault.

when 3 came it will take the place of 7 because it is not used for the longest duration of time in the future.—>1 Page fault.

0 is already there so —> 0 Page fault..

4 will takes place of 1 —> 1 Page Fault.

Now for the further page reference string —> 0 Page fault because they are already available in the memory.

Optimal page replacement is perfect, but not possible in practice as operating system cannot know future requests. The use of Optimal Page replacement is to set up a benchmark so that other replacement algorithms can be analyzed against it.

Least Recently Used

In this algorithm page will be replaced which is least recently used.

Let say the page reference string 7 0 1 2 0 3 0 4 2 3 0 3 2 . Initially we have 4 page slots empty.

Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> 4 Page faults

0 is already their so —> 0 Page fault.

when 3 came it will take the place of 7 because it is least recently used —>1 Page fault

0 is already in memory so —> 0 Page fault.

4 will takes place of 1 —> 1 Page Fault

Now for the further page reference string —> 0 Page fault because they are already available in the memory.