Operating System and its Types

What is an operating system? An operating system acts as an intermediary between the user of a computer and computer hardware. The purpose of an operating system is to provide an environment in which a user can execute programs in a convenient and efficient manner. Besides that, an OS is responsible for the proper management of the system's resources (Memory, CPU, I/O devices, Files, etc.) to the various user tasks.
Based on various use-cases, different types of OS have been developed:

- Batch OS
- Time-sharing OS
- Distributed OS
- Network OS
- Real-time OS

Each of the above mentioned is discussed in brief below:

Batch Operating System In a batch OS, there is an *operator* program which groups the jobs into batches based on similar requirements. There is no manual intervention or pre-emption between batches. Usage: Payroll System, Bank statements etc.

Some of the key points of a Batch OS are:

- Multiple users can assign jobs (operator program manages them into batches).
- Completion Time of a Job is unknown.
- No idle time for the CPU
- Convoy Effect can occur (Bottleneck for waiting jobs if current job is taking too much time).

Time-Sharing Operating System Multiple users can concurrently interact with the system. Each user is allocated a time quantum (chunk) for execution, after which the OS context-switches to another user. This maintains responsiveness of the OS. e.g. Multix, Unix etc.

Some of the key points of a Time-Sharing OS are:

- Each user gets equal opportunity
- No user has to wait indefinitely if some other is stuck (due to context-switch)
- Switching Time is an extra overhead

Distributed Operating System A distributed OS consists of multiple hardware units (independent CPU, Memory, I/O Units), with a single OS managing execution of processes in those various independent hardware systems. They are thus also known as **loosely coupled systems**. The major benefit of working with these types of an operating system is that it is always possible that one user can access the files or software which are not actually present on his system but on some other system connected within this network i.e., remote access is enabled within the devices connected in that network.

Some of the key points of a distributed OS are:

- Failure of one will not affect the other.
- Computation is fast (as resources are shared & parallel execution is possible)
- Easily Scalable (add more hardware units)

Network Operating System These systems run on a server and provide the capability to manage data, users, groups, security, applications, and other networking functions. All the users should be well aware of the underlying configuration, of all other users within the network, their individual connections etc. and that's why these computers are popularly known as **tightly coupled systems**. Usage e.g. Microsoft Windows Server, Ubuntu Server, CentOS etc.

Some of the key points of a Network OS are:

- Centralized System (vulnerable to failure)
- Remotely Accessible from different location and types of systems
- New upgradations are easily done (1 central system)

Real-time Operating System They are OS meant to handle mission-critical tasks. They have high responsiveness, and fast processing time. The functionalities are limited and specific to the domain. e.g. Defense Systems, Robots, WSNs, Air-Traffic Control, Space Rovers systems.

Some of the key points of a Realtime OS are:

- Maximum Utilization & 0 downtime (can't afford to go down)
- Application Specific
- Error-free and automatic recoverable systems

Multiprogramming, Multiprocessing & Multithreading

Although the 3 words seem similar (and they do in some respect ~ concurrency), however, there are subtle differences amongst all 3 of them:
Multiprocessing: True parallelism is achieved via Multiprocessing as there physically exists multiple hardware units (CPU, cache, and even Memory) amongst which programs are shared. Thus, at a time, parallel execution is possible.

Multiprocessing aided systems have an added advantage of failure-tolerance. i.e. Even if one CPU fails, the system can keep running relying upon the other processors. Also, one can gain significant gains in computation by horizontally scaling the system (using multiple mediocre CPUs instead of upgrading the single one). I/O and peripheral devices can be shared amongst the units thus saving total expense.

Multiprogramming: Multiple programs share single hardware resources. However, fake parallelism is achieved by context-switching between the processes. Responsiveness between multiple programs is achieved in such a manner. The number of programs that can simultaneously run thus depends upon the size of the main memory.

In such a system, all the programs are placed in a queue for execution (also called job pool). The scheduler picks jobs one-by-one and executes till the time quantum expires, or the process is pre-empted by some external factors (I/O requirement, interrupt, etc.). In such a case, the process is context-switched with the next one, thus preventing the CPU from being idle.

Multithreading: Multiple Threads running as part of the same process. Threads have common shared resources (memory, file descriptors, code segment, etc). However, it has independent stack-space, program-counters, etc.

Threads are said to be lightweight processes, which run together in the same context. They also get context-switched, however, the delay is much lesser compared to the whole process being switched.

The reason being threads share the same PCB (Process-control-block, which will be covered later). e.g. Tabs in a web-browser is a good example of threads (Each tab shares the same browser process, but has its own separate identity).

Process management - Introduction to process

Introduction to process: A process is a program in execution or a process is an ‘active’ entity, as opposed to a program, which is considered to be a ‘passive’ entity.
A single program can create many processes when running multiple times; when we open a .exe or binary file multiple times, multiple instances begin (multiple processes are created).
For example When we write a program in C or C++ and compile it, the compiler creates binary code. The original code and binary code are both programs. When we actually run the binary code, it becomes a process.

What does a process look like in memory?

Text Section*:* A Process, sometimes known as the Text Section, also includes the current activity represented by the value of the ***Program Counter***.
Stack*:* The Stack contains the temporary data, such as function parameters, returns addresses, and local variables.
Data Section*:* Contains the global variable.
Heap Section*:* Dynamically allocated memory to process during its run time.
Refer this for more details on sections.

Process States

- **Single Tasking Systems(MS-DOS):** In this kind of system, the second process begins only when the first process ends. By the time one process completes there might be other I/O devices, waiting for the first process to complete its task. This might lead to a delay in the process of the operating system, which is not feasible from the user's point of view. Therefore, the need arose for a multiprogramming system which can execute multiple processes at a given time. Given below is the process state diagram of the following:

- **Multiple programming System:** In this system, multiple programs share single hardware resources. However, fake parallelism is achieved by context-switching between the processes. Responsiveness between multiple programs is achieved in such a manner. The number of programs that can simultaneously run thus depends upon the size of the main memory.
  There are various states through which a processor passes to complete a particular or multiple executions. This is explained below using the process state diagram.
  Here is the basic diagram showing a 5-state model and an advanced diagram showing the 7-state model.

5-State Model

7-State Model

States of a process are as following:

  - **New (Create) -** In this step, the process is about to be created but not yet created, it is the program which is present in secondary memory that will be picked up by OS to create the process.

- **Ready -** New -> Ready to run. After the creation of a process, the process enters the ready state i.e. the process is loaded into the main memory. The process here is ready to run and is waiting to get the CPU time for its execution. Processes that are ready for execution by the CPU are maintained in a queue for ready processes.
- **Run -** The process is chosen by CPU for execution and the instructions within the process are executed by anyone of the available CPU cores.
- **Blocked or wait -** Whenever the process requests access to I/O or needs input from the user or needs access to a critical region(the lock for which is already acquired) it enters the blocked or waits for the state. The process continues to wait in the main memory and does not require CPU. Once the I/O operation is completed the process goes to the ready state.
- **Terminated or completed -** Process is killed as well as PCB is deleted.
- **Suspend ready -** Process that was initially in the ready state but were swapped out of main memory(refer Virtual Memory topic) and placed onto external storage by scheduler are said to be in suspending ready state. The process will transition back to ready state whenever the process is again brought onto the main memory. Linux uses swap space and partition to do this task.
- **Suspend wait or suspend blocked -** Similar to suspend ready but uses the process which was performing I/O operation and lack of main memory caused them to move to secondary memory.
  When work is finished it may go to suspend ready.

Process Control Block or PCB Various Attributes or Characteristics of a Process are:

```
1. Process Id: A unique identifier assigned by the operating system

2. Process State: Can be ready, running, etc.

3. CPU registers: Like the Program Counter (CPU registers must be saved and
                  restored when a process is swapped in and out of CPU)
4. Accounts information: This includes the amount of CPU used for process
                  execution, time limits, execution ID etc.
5. I/O status information: For example, devices allocated to the process,
                  open files, etc.
6. CPU scheduling information: For example, Priority (Different processes
                  may have different priorities, for example, a short process may be
                  assigned a low priority in the shortest job first scheduling)
```

All of the above attributes of a process are also known as the ***context of the process***.
Every process has its own program control block(PCB), i.e each process will have a unique PCB. All of the above attributes are part of the PCB.
A process control block (PCB) contains information about the process, i.e. registers, quantum, priority, etc. The process table is an array of PCB's, which means logically contains a PCB for all of the current processes in the system.

- **Pointer -** It is a stack pointer which is required to be saved when the process is switched from one state to another to retain the current position of the process.
- **Process state -** It stores the respective state of the process.
- **Process number -** Every process is assigned with a unique id known as process ID or PID which stores the process identifier.
- **Program counter -** It stores the counter which contains the address of the next instruction that is to be executed for the process.
- **Register -** These are the CPU registers which includes: accumulator, base, registers and general purpose registers.
- **Memory limits -** This field contains the information about memory management system used by operating system. This may include the page tables, segment tables etc.
- **Open files list -** This information includes the list of files opened for a process.

Context Switching The process of saving the context of one process and loading the context of another process is known as Context Switching. In simple terms, it is like loading and unloading the process from running state to ready state.

When does context switching happen? 1. When a high-priority process comes to ready state (i.e. with higher priority than the running process)
2. An Interrupt occurs
3. User and kernel mode switch (It is not necessary though)
4. Preemptive CPU scheduling used.

Context Switch vs Mode Switch A mode switch occurs when CPU privilege level is changed, for example when a system call is made or a fault occurs. The kernel works in more a privileged mode than a standard user task. If a user process wants to access things that are only accessible to the kernel, a mode switch must occur. The currently executing process need not be changed during a mode switch.
A mode switch typically occurs for a process context switch to occur. Only the kernel can cause a context switch.

Process Types (Zombie & Orphan)

When the system boots, it begins with a single process. In UNIX-based systems, it is called the ***init*** process. Thereafter, other processes are spawned by the init process until the final loading screen of the OS is displayed. Process **spawning** is a fundamental concept as it allows to create and destroy multiple processes during the lifetime of the running of the OS. A process 'A' if creates a new process 'B', then we refer to 'A' as being the **parent** and 'B' as the **child**.
Each process in the OS needs to be uniquely identifiable, hence they are assigned a unique **process ID**, and the information about the state of the process is kept in a data structure called the **process-table**:

When a process terminates, the entry corresponding to it gets deleted. Depending upon the state of execution of a child process and termination, 2 categories of processes is defined:

- **Zombie Process**: A process which has finished the execution but still has an entry in the process table to report to its parent process is known as a zombie process. A child process always first becomes a zombie before being removed from the process table. The parent process reads the exit status of the child process which reaps off the child process entry from the process table. A child process undergoes a zombie state when it has finished termination but the parent is yet to reap (collect exit status and delete from process-table) it off. The reason being the parent waiting for I/O or in sleep etc. Till the time it gets removed, it is deemed a Zombie process.
- **Orphan Process**: A child process currently in execution whose parent no longer exists. Such a case is possible when parent terminates unconditionally (due to an error etc.) or simply forgets to reap its child of. i.e. It doesn't wait for the child to terminate first. In such a case, the orphan process is adopted by the *init* process and then reaped off (once it finishes).

Process management - Process scheduling (Algorithms and Criteria)

CPU and IO Bound Processes If the process is intensive in terms of CPU operations then it is called the CPU bound process. Similarly, If the process is intensive in terms of I/O operations then it is called IO bound process.
There are three types of process scheduler.

1. **Long Term or job scheduler** It brings the new process to the 'Ready State'. It controls ***Degree of Multi-programming***, i.e., number of process present in ready state at any point of time. It is important that the long-term scheduler make a careful selection of both IO and CPU bound processes.

2. **Short term or CPU 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 process selected by Short-term scheduler on the CPU (Ready to Running State) Context switching is done by dispatcher only. A dispatcher does the following:

   1. Switching context.
   2. Switching to user mode.
   3. Jumping to the proper location in the newly loaded program.

3. **Medium-term scheduler** It is responsible for suspending and resuming the process. It mainly does swapping (moving processes from main memory to disk and vice versa). Swapping may be necessary to improve the process mix or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up.

Multiprogramming - We have many processes ready to run. There are two types of multiprogramming:

1. **Pre-emption -** Process is forcefully removed from CPU. Pre-emption is also called as time sharing or multitasking.
2. **Non pre-emption -** Processes are not removed until they complete the execution.

Degree of multiprogramming -
The number of processes that can reside in the ready state at maximum decides the degree of multiprogramming, e.g., if the degree of programming = 100 means 100 processes can reside in the ready state at maximum.

There are different queues of processes (in an operating system):

- **Ready Queue:** The set of all processes that are in main memory and are waiting for CPU time is kept in the ready queue.

- **Job Queue:** Each new process goes into the job queue. Processes in the job queue reside on mass storage and await the allocation of main memory.

- **Waiting (Device) Queues:** The set of processes waiting for allocation of certain I/O devices is kept in the waiting (device) queue. The short-term scheduler (also known as CPU scheduling) selects a process from the ready queue and yields control of the CPU to the process

Short-Term Scheduler and Dispatcher - Consider a situation, where various processes residing in the ready queue and waiting for execution. But CPU can't execute all the processes of ready queue simultaneously, the operating system has to choose a particular process on the basis of scheduling algorithm used. So, this procedure of selecting a process among various processes is done by **scheduler**. Now here the task of scheduler completed. Now **dispatcher** comes into the picture as scheduler has decided a process for execution, it is dispatcher who takes that process from ready queue to the running status, or you can say that providing CPU to that process is the task of the dispatcher.

Example - There are 4 processes in ready queue, i.e., P1, P2, P3, P4; They all have arrived at t0, t1, t2, t3 respectively. First in First out scheduling algorithm is used. So, scheduler decided that first of all P1 has come, so this is to be executed first. Now dispatcher takes P1 to the running state.

Difference between the Scheduler and Dispatcher:

Properties	DISPATCHER	SCHEDULER
Definition:	Dispatcher is a module that gives control of CPU to the process selected by short term scheduler	Scheduler is something which selects a process among various processes
Types:	There are no diifrent types in dispatcher.It is just a code segment.	There are 3 types of scheduler i.e. Long-term, Short-term, Medium-term
Dependency:	Working of dispatcher is dependednt on scheduler.Means dispatcher have to wait untill scheduler selects a process.	Scheduler works idependently.It works immediately when needed
Algorithm:	Dispatcher has no specific algorithm for its implementation	Scheduler works on various algorithm such as FCFS, SJF, RR etc.
Time Taken:	The time taken by dispatcher is called dispatch latency.	TIme taken by scheduler is usually negligible.Hence we neglect it.
Functions:	Dispatcher is also responsible for:Context Switching, Switch to user mode, Jumping to proper location when process again restarted	The only work of scheduler is selection of processes.

Below are different time with respect to a process.

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

2. **Completion Time:** Time at which process completes its execution.

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

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

```
Turn Around Time = Completion Time - Arrival Time
```

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

```
Waiting Time = Turn Around Time - Burst Time
```

6. **Response Time:** Time difference between arrival time and the first time the process gets CPU.

Example: Suppose we are given to calculate each type of time for the process P0 for the Gantt chart:

In the diagram from time 0 to 1, the CPU is free. Time slot 1 to 3 has been scheduled for P0, 3 to 5 has been scheduled for P1 and 5 to 6 are again scheduled to P0.
So for P0:

- Arrival Time (time of arrival into PC) = 1

- Completion Time (time of completion at PC) = 6

- Burst Time (Time taken by P0) = From 1 to 3 P0 takes 2 units of time and from 5 to 6 P0 takes 1 unit of time. So, in all Burst Time for P0 is 3.

- Turn Around Time = Completion Time - Arrival Time = 6 - 1 = 5.

- Waiting Time = Turn Around Time - Burst Time = 5 - 3 = 2.

- Response Time = 0 since the process got the CPU immediately.

Goals of a Scheduling Algorithm:

1. Maximum CPU Utilization: CPU must not be left idle and should be used to the full potential

2. Maximum Throughput: Throughput refers to the number of jobs completed per unit time. So the CPU must try to execute maximum number of jobs per unit time.

3. Minimum Turnaround time: Time between arrival and completion must be minimum.

4. Minimum Waiting Time: Total amount of time in the ready queue must be minimum.

5. Minimum Response Time: Difference between the arrival time and the time at which the process gets the CPU must be mininmum.

6. Fair CPU Allocation: No job must be starved.

Different Scheduling Algorithms

- ***First Come First Serve (FCFS)*** - Simplest scheduling algorithm that schedules according to arrival times of processes. First come first serve scheduling algorithm process that requests the CPU first is allocated the CPU first. It is implemented by using the FIFO queue. FCFS is a non-preemptive scheduling algorithm.

- ***Shortest Job First (SJF)*** - Process which have the shortest burst time are scheduled first.If two processes have the same bust time then FCFS is used to break the tie. It is a non-preemptive scheduling algorithm.

- ***Longest Job First (LJF)*** - It is similar to SJF scheduling algorithm. But, in this scheduling algorithm, we give priority to the process having the longest burst time. This is non-preemptive in nature i.e., when any process starts executing, can't be interrupted before complete execution.

- ***Shortest Remaining Time First (SRTF)*** - It is preemptive mode of SJF algorithm in which jobs are schedule according to shortest remaining time.

- ***Longest Remaining Time First (LRTF)*** - It is preemptive mode of LJF algorithm in which we give priority to the process having largest burst time remaining.

- ***Round Robin Scheduling (RR)*** - Each process is assigned a fixed time(Time Quantum/Time Slice) in cyclic way. It is designed especially for the time-sharing system. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1-time quantum.

- ***Priority Based scheduling (Non-Preemptive)*** - In this scheduling, processes are scheduled according to their priorities, i.e., highest priority process is scheduled first. If priorities of two processes match, then schedule according to arrival time. Here starvation of process is possible.

- ***Highest Response Ratio Next (HRRN)*** - In this scheduling, processes with highest response ratio is scheduled. This algorithm avoids starvation.

```
Response Ratio = (Waiting Time + Burst time) / Burst time
```

- ***Multilevel Queue Scheduling*** - According to the priority of process, processes are placed in the different queues. Generally high priority process are placed in the top level queue. Only after completion of processes from top level queue, lower level queued processes are scheduled. It can suuffer from starvation.

- ***Multi level Feedback Queue Scheduling*** - It allows the process to move in between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it is moved to a lower-priority queue.

Some useful facts about Scheduling Algorithms:

1. FCFS can cause long waiting times, especially when the first job takes too much CPU time.

2. Both SJF and Shortest Remaining time first algorithms may cause starvation. Consider a situation when the long process is there in the ready queue and shorter processes keep coming.

3. If time quantum for Round Robin scheduling is very large, then it behaves same as FCFS scheduling.

4. SJF is optimal in terms of average waiting time for a given set of processes,i.e., average waiting time is minimum with this scheduling, but problems are, how to know/predict the time of next job.

FCFS Scheduling

FCFS (First-come-First-serve) follows the principle of FIFO (First-in-First-out). It is the simplest scheduling algorithm. FCFS simply queues processes in the order that they arrive in the ready queue. In this, the process which comes first will be executed first and the next process starts only after the previous gets fully executed. It is a non-preemptive algorithm. Let's look at this problem:

Only the processes in the ready queue are to be considered. Since there is no preemption, once the job starts it is going to run until its completion. Now let's look at the Gantt chart:

Now, let's compute each type of time using concepts and formulae, that we studied in the previous lecture. Now the chart looks like this:

So, Average waiting time = (0+1+6+9+17)/5 = 33/5
and Average turn-around time = (2+7+10+18+29)/5 = 74/5

Example:

Implementation:

```
1-  Input the processes along with their burst time (bt).
2-  Find the waiting time (wt) for all processes.
3-  As the first process that comes need not to wait so
    waiting time for process 1 will be 0 i.e. wt[0] = 0.
4-  Find waiting time for all other processes i.e. for
     process i ->
       wt[i] = bt[i-1] + wt[i-1] .
5-  Find turnaround time = waiting_time + burst_time
    for all processes.
6-  Find average waiting time =
                total_waiting_time / no_of_processes.
7-  Similarly, find average turnaround time =
                total_turn_around_time / no_of_processes.
```

Important Points:

1. Simple and Easy to implement

2. Non-preemptive
3. Average Waiting Time is not optimal
4. Can not utilize resources in parallel: Results in Convoy effect. Convoy Effect is a phenomenon associated with the First Come First Serve (FCFS) algorithm, in which the whole Operating System slows down due to a few slow processes. Consider a situation when many IO-bound processes are there and one CPU bound process. The IO bound processes have to wait for CPU bound process when the CPU bound process acquires CPU. The IO-bound process could have taken CPU for some time, then used IO devices. The diagram below is a good analogy to the problem:

SJF Scheduling

The Shortest-Job-First (SJF) or shortest job next, is a scheduling policy that selects the waiting process with the smallest execution time to execute next. It is a non-preemptive algorithm. Being a non-preemptive algorithm, the first process assigned to the CPU is executed till completion, then the process whose burst time is minimum is assigned next to the CPU and hence it continues.

Algorithm:

```
1- Sort all the processes in increasing order
   according to burst time.
2- Then simply, apply FCFS.
```

Let's look at the following non preemptive SJFS problem and understand the stepwise execution:

Now lets draw the Gantt chart to the following problem:

How to compute below times in SJF using a program?

1. Completion Time: Time at which process completes its execution.
2. Turn Around Time: Time Difference between completion time and arrival time. Turn Around Time = Completion Time - Arrival Time
3. Waiting Time(W.T): Time Difference between turn around time and burst time.
   Waiting Time = Turn Around Time - Burst Time

Therefore the final chart showing all type of times are as follows:

So, Average turn-around time = (5+11+6+17)/4 = 39/4
and Average waiting time = (0+7+3+9)/4 = 19/4

Example:

Some of the key points are as follows:

- Shortest Job first has the advantage of having a minimum average waiting time among all scheduling algorithms.
- It is a Greedy Algorithm.
- It may cause starvation if shorter processes keep coming. This problem can be solved using the concept of ageing.

- It is practically infeasible as Operating System may not know burst time and therefore may not sort them. While it is not possible to predict execution time, several methods can be used to estimate the execution time for a job, such as a weighted average of previous execution times. SJF can be used in specialized environments where accurate estimates of running time are available.

SRTF Scheduling

In the previous post, we have discussed SJF which is a non-preemptive scheduling algorithm. In this post, we will discuss the preemptive version of SJF known as Shortest Remaining Time First (SRTF).

In this scheduling algorithm, the process with the smallest amount of time remaining until completion is selected to execute. Since the currently executing process is the one with the shortest amount of time remaining by definition, and since that time should only reduce as execution progresses, processes will always run until they complete or a new process is added that requires a smaller amount of time.

Implementation:

```
1- Traverse until all process gets completely executed.
   a) Find process with minimum remaining time at every single time lap.
   b) Reduce its time by 1.
   c) Check if its remaining time becomes 0.
   d) Increment the counter of process completion.
   e) Completion time of current process = current_time +1;
   f) Calculate the waiting time for each completed process.
         wt[i]= Completion time - arrival_time-burst_time
   g) Increment time lap by one.

2- Find turnaround time (waiting_time+burst_time).

   Let's look at the below problem.
```

At first, the P0 is scheduled and it runs for 1 unit of time. P1 also around at 1 unit. So, now the remaining time of P0(i.e., 7) is compared to the burst time of P1(i.e., 4). The lesser is scheduled, here P1 is scheduled. Again it runs for 1 unit when P2 arrives. The same comparison is done. Since P1 has 3 remaining unit time which is less than P2, it is continued to be scheduled. At 3 P3 also arrives whose burst time is 5. Being more than the remaining time of P2 that is 2, P2 is continued to be scheduled until completion. Then P3 is scheduled whose burst time is 5. This runs till completion followed by P2. All the breakdowns can be seen in the below and the Gantt chart:

Gantt chart for the process:

Now let's calculate all other time using the basic formulae.

So, Average turn around time = (17+4+24+7)/4 = 52/4
and Average waiting time = (9+0+15+2)/4 = 26/4

Example:

Process	Duration	Order	Arrival Time
P1	9	1	0
P2	2	2	2

P1 waiting time: 4-2 = 2
P2 waiting time: 0
The average waiting time(AWT): (0 + 2) / 2 = 1

Advantage:

1. Short processes are handled very quickly.
2. The system also requires very little overhead since it only makes a decision when a process completes or a new process is added.
3. When a new process is added the algorithm only needs to compare the currently executing process with the new process, ignoring all other processes currently waiting to execute.

Disadvantage:

1. Like shortest job first, it has the potential for process starvation.
2. Long processes may be held off indefinitely if short processes are continually added.
3. It is impractical since it is not possible to know the burst time of every process in ready queue in advance.

LRTF Scheduling

LRTF (Longest-Remaining-Time-First) is the pre-emptive version of the LJF algorithm. In this scheduling algorithm, we find the process with the maximum remaining time and then process it. We check for the maximum remaining time after some interval of time (say 1 unit each) to check if another process having more Burst Time arrived up to that time.
The algorithm stands as:

```
1) Sort the processes in increasing order of their arrival time.
2) Choose the process having the least arrival-time but the most burst-time.
   Execute it for 1 unit/quantum.
   Check if any other process arrives upto that point of execution.
3) Repeat above steps until all processes have been executed.
```

As an example, consider the following 4 processes (P1, P2, P3, P4):

```
Process   Arrival time   Burst Time

P1            1 ms           2 ms

P2            2 ms           4 ms

P3            3 ms           6 ms

P4            4 ms           8 ms
```

Execution:

1. At t = 1, Available Process : P1. So, select P1 and execute 1 ms.
2. At t = 2, Available Process : P1, P2. So, select P2 and execute 1 ms (since BT(P1)=1 which is less than BT(P2) = 4)
3. At t = 3, Available Process : P1, P2, P3. So, select P3 and execute 1 ms (since, BT(P1) = 1 , BT(P2) = 3 , BT(P3) = 6).
4. Repeat the above steps until the execution of all processes.

Note that CPU will be idle for 0 to 1 unit time since there is no process available in the given interval.

Gantt chart will be as following below,

Output:

```
Total Turn Around Time = 68 ms

So, Average Turn Around Time = 68/4 = 17.00 ms



And, Total Waiting Time = 48 ms

So Average Waiting Time = 48/4 = 12.00 ms
```

Priority Scheduling

Non-Preemptive Priority scheduling: It is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems. Each process is assigned a priority. The process with the highest priority is to be executed first and so on.
Processes with the same priority are executed on the FCFS scheme. Priority can be decided based on memory requirements, time requirements or any other resource requirement.

Algorithm:

```
1- First input the processes with their burst time
   and priority.
2- Sort the processes, burst time and priority
   according to the priority.
3- Now simply apply FCFS algorithm.
```

Let's look at the following problem with P2 set as the highest priority.

At first, P0 enters into scheduling since it arrives first. After its completion, the other 3 processes arrive. Now, priority comes into the picture. The process with the highest priority is scheduled. Here it is P2, followed by P3 and P1.

Now let's look at the Gantt chart:

The final chart looks like this:

Example:

Pre-emptive Priority Scheduling: This follows preemption where one process can be pre-empted and replaced by another process. Let's look at a similar problem as taken above and understand the working of this algorithm:

At first, P0 is scheduled and it runs for 1 unit followed by the arrival of P1. Since P1 has less priority than P0, P0 continues for 1 more unit, Now P2 arrives which has a greater priority and hence is scheduled. It runs until completion and then P3 follows which has a greater priority among the present processes. It completes and then the priority of the remaining processes are compared i.e., P0 and P1. P0 having greater priority finishes its remaining 1 unit of time and then P1 follows.

Below is the Gantt chart of the following process:

Here is the final chart showing all the time:

Note: A major problem with priority scheduling is indefinite blocking or **Starvation**. A solution to the problem of indefinite blockage of the low-priority process is ageing. **Aging** is a technique of gradually increasing the priority of processes that wait in the system for a long period of time.

Round-Robin Scheduling

Round-Robin is a CPU scheduling algorithm where each process is assigned a fixed time slot (also called quantum). Once that chunk of time is completed, the process is context-switched with the next in the queue. It is the most common and practically usable scheduling algorithm, as it doesn't require the system to estimate the burst-time of a process. Calculation of burst-time is practically not possible as no one can predict how long a process will take to execute. It although has a minor disadvantage of the overhead of context-switching (some time gets wasted in the process). Let's try to understand the working using the following problem, with given Time Quantum of 2 units:

Each process runs for 2 unit of time, followed by the next process. The whole process runs in a circular manner.

Here is the Gantt chart of the above problem:

Here is the final chart showing all type of times:

The algorithm to calculate the various time statistics are as follows:

```
1) Create an array rem_bt[] to keep track of remaining
   burst time of processes. This array is initially a
   copy of bt[] (burst times array)
2) Create another array wt[] to store waiting times
   of processes. Initialize this array as 0.
3) Initialize time : t = 0
4) Keep traversing all processes while all processes
   are not done. Do following for the ith process if it is not done yet.
    a- If rem_bt[i] > quantum
       (i)  t = t + quantum
       (ii) bt_rem[i] -= quantum;
    c- Else // Last cycle for this process
       (i)  t = t + bt_rem[i];
```

```
        (ii) wt[i] = t - bt[i]
        (ii) bt_rem[i] = 0; // This process is over
```

Once we have waiting times, we can compute turn around time tat[i] of a process as sum of waiting and burst times, i.e., wt[i] + bt[i]

Multilevel Queue Scheduling

Multi-level Queue Scheduling is required when we group processes into some specific categories. e.g. **Foreground (interactive)** and **Background (batch)** processes. These two classes have different scheduling needs. *System* processes are of the highest priority (extremely critical), then comes *Interactive* processes followed by *Batch* and *Student* processes.

Ready Queue is divided into separate queues for each class of processes. For example, let us take three different types of process System processes, Interactive processes, and Batch Processes. All three process has its own queue. Now, look at the below figure.

All three different types of processes have their own queue. Each queue has its own Scheduling algorithm. For example, queue 1 and queue 2 uses **Round Robin** while queue 3 can use **FCFS** to schedule there processes.

Scheduling among the queues : What will happen if all the queues have some processes? Which process should get the CPU? To determine this Scheduling among the queues is necessary. There are two ways to do so -

1. **Fixed priority preemptive scheduling method -** Each queue has absolute priority over lower priority queue. Let us consider following priority order **queue 1 > queue 2 > queue 3**.According to this algorithm no process in the batch queue(queue 3) can run unless queue 1 and 2 are empty. If any batch process (queue 3) is running and any system (queue 1) or Interactive process(queue 2) entered the ready queue the batch process is preempted.
2. **Time slicing** - In this method each queue gets certain portion of CPU time and can use it to schedule its own processes.For instance, queue 1 takes 50 percent of CPU time queue 2 takes 30 percent and queue 3 gets 20 percent of CPU time.

Example Consider the below table of four processes under Multilevel queue scheduling. Queue number denotes the queue of the process.

Priority of queue 1 is greater than queue 2. queue 1 uses Round Robin (Time Quantum = 2) and queue 2 uses FCFS.

Below is the **gantt chart** of the problem :

At starting both queues have process so process in queue 1 (P1, P2) runs first (because of higher priority) in the round robin fashion and completes after 7 units then process in queue 2 (P3) starts running (as there is no process in queue 1) but while it is running P4 comes in queue 1 and interrupts P3 and start running for 5 second and after its completion P3 takes the CPU and completes its execution.

Multilevel Feedback Queue Scheduling

MLFQ (Multilevel-Feedback-Queue) scheduling is a modification to the MLQ (Multilevel-Queue) which allows processes to move between queues. It keeps analyzing the behavior (time of execution) of processes, according to which it changes its priority.

Now let us suppose that queues 1 and 2 follow round robin with time quantum 4 and 8 respectively and queue 3 follow FCFS. One implementation of MFQS can be:

1. When a process starts executing then it first enters queue 1.

2. In queue 1 process executes for 4 unit and if it completes in this 4 unit or it gives CPU for I/O operation in this 4 unit than the priority of this process does not change and if it again comes in the ready queue than it again starts its execution in Queue 1.
3. If a process in queue 1 does not complete in 4 unit then its priority gets reduced and it shifted to queue 2.
4. Above points 2 and 3 are also true for queue 2 processes but the time quantum is 8 unit. In a general case if a process does not complete in a time quantum than it is shifted to the lower priority queue.
5. In the last queue, processes are scheduled in FCFS manner.
6. A process in lower priority queue can only execute only when higher priority queues are empty.
7. A process running in the lower priority queue is interrupted by a process arriving in the higher priority queue.

NOTE: A process in the lower priority queue can suffer from starvation due to some short processes taking all the CPU time. A simple solution can be to boost the priority of all the process after regular intervals and place them all in the highest priority queue.

What is the need for such a complex scheduling algorithm?

- Firstly, it is more flexible than the multilevel queue scheduling.
- To optimize turnaround time algorithms like SJF is needed which require the running time of processes to schedule them. But the running time of the process is not known in advance. MFQS runs a process for a time quantum and then it can change its priority(if it is a long process). Thus it learns from past behavior of the process and then predicts its future behavior. This way it tries to run a shorter process first thus optimizing turnaround time.
- MFQS also reduces the response time.

Deadlock - Characteristics

A process in operating systems uses different resources and uses resources in the 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 the same track and there is only one track, none of the trains can move once they are in front of each other. A similar situation occurs in operating systems when there are two or more processes that 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 the resource

This can also occur for more than two processes as shown in the below diagram:

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

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

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

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

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

Resource allocation graph: As Banker's algorithm using some kind of table like allocation, request, available all that thing to understand what is the state of the system. Similarly, if you want to understand the state of the system instead of using those tables, actually tables are very easy to represent and understand it, but then still you could even represent the same information in the graph. That graph is called **Resource Allocation Graph (RAG)**.

So, the resource allocation graph is explained to us what is the state of the system in terms of **processes and resources**. Like how many resources are available, how many are allocated and what is the request of each process. Everything can be represented in terms of the diagram. One of the advantages of having a diagram is, sometimes it is possible to see a deadlock directly by using RAG, but then you might not be able to know that by looking at the table. But the tables are better if the system contains lots of process and resource and Graph is better if the system contains less number of process and resource.
We know that any graph contains vertices and edges. So RAG also contains vertices and edges. In RAG vertices are two type -

1. **Process vertex -** Every process will be represented as a process vertex.Generally, the process will be represented with a circle.

2. **Resource vertex -** Every resource will be represented as a resource vertex. It is also two type -

    - **Single instance type resource -** It represents as a box, inside the box, there will be one dot.So the number of dots indicate how many instances are present of each resource type.
    - **Multi-resource instance type resource -** It also represents as a box, inside the box, there will be many dots present.

Now coming to the edges of RAG. There are two types of edges in RAG -

1. Assign Edge - If you already assign a resource to a process then it is called Assign edge.
2. Request Edge - It means in future the process might want some resource to complete the execution, that is called request edge.

So, if a process is using a resource, an arrow is drawn from the resource node to the process node. If a process is requesting a resource, an arrow is drawn from the process node to the resource node.
Example 1 (Single instances RAG) -

If there is a cycle in the Resource Allocation Graph and each resource in the cycle provides only one instance, then the processes will be in a deadlock. For example, if process P1 holds resource R1, process P2 holds resource R2 and process P1 is waiting for R2 and process P2 is waiting for R1, then process P1 and process P2 will be in a deadlock.

Here's another example, that shows Processes P1 and P2 acquiring resources R1 and R2 while process P3 is waiting to acquire both resources. In this example, there is no deadlock because there is no circular dependency.
So cycle in single-instance resource type is the sufficient condition for deadlock.
Example 2 (Multi-instances RAG) -

From the above example, it is not possible to say the RAG is in a safe state or in an unsafe state. So to see the state of this RAG, let's construct the allocation matrix and request matrix.

- The total number of processes are three; P1, P2 & P3 and the total number of resources are two; R1 & R2.

Allocation matrix -

- For constructing the allocation matrix, just go to the resources and see to which process it is allocated.
- R1 is allocated to P1, therefore write 1 in allocation matrix and similarly, R2 is allocated to P2 as well as P3 and for the remaining element just write 0.

Request matrix -

- In order to find out the request matrix, you have to go to the process and see the outgoing edges.
- P1 is requesting resource R2, so write 1 in the matrix and similarly, P2 requesting R1 and for the remaining element write 0.

So now available resource is = (0, 0).

Checking deadlock (safe or not) -

So, there is no deadlock in this RAG. Even though there is a cycle, still there is no deadlock. Therefore in multi-instance resource cycle is not sufficient condition for deadlock.

The above example is the same as the previous example except that, the process P3 requesting for resource R1.
So the table becomes as shown in below.

So,the Available resource is = (0, 0), but requirement are (0, 1), (1, 0) and (1, 0).So you can't fulfill any one requirement.Therefore, it is in deadlock.

Therefore, every cycle in a multi-instance resource type graph is not a deadlock, if there has to be a deadlock, there has to be a cycle. So, in case of RAG with a multi-instance resource type, the cycle is a necessary condition for deadlock, but not sufficient.

Deadlock - Handling Deadlock

Methods for handling deadlock There are three ways to handle deadlock

1. **Deadlock Prevention:** The OS accepts all the sent requests. The idea is to not to send a request that might lead to a deadlock condition.

2. **Deadlock Avoidance:** The OS very carefully accepts requests and checks whether if any request can cause deadlock and if the process leads to deadlock, the process is avoided.

3. **Deadlock Detection and Recovery:** Let deadlock occur, then do preemption to handle it once occurred.

4. **Ignore the problem altogether:** If the deadlock is very rare, then let it happen and reboot the system. This is the approach that both Windows and UNIX take.

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

1. Mutual Exclusion
2. Hold and Wait
3. No preemption
4. Circular wait

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

- **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. It can be avoided to some extent using Spooling.

- **Eliminate Hold and wait:**
    1. Allocate all required resources to the process before the 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 remain blocked till it has completed its execution.
    2. The process will make a new request for resources after releasing the current set of resources. This solution may lead to starvation.

- **Eliminate No Preemption:** Preempt resources from the process when resources required by other high priority processes.

- **Eliminate Circular Wait:** Each resource will be assigned with a numerical number. A process can request the resources only in increasing order of numbering.
  For Example, if the P1 process is allocated R1, P2 has R2 and P3 has R3, then P3 cannot request R1 which is less than R3.

Deadlock Avoidance
Deadlock avoidance can be done with Banker's Algorithm.

If a system does not employ either a deadlock prevention or deadlock avoidance algorithm then a deadlock situation may occur. In this case-

- Apply an algorithm to examine state of system to determine whether deadlock has has occurred or not.
- Apply an algorithm to recover from the deadlock. For more refer- Deadlock Recovery

Deadlock Detection Algorithm/Bankers Algorithm: It 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.
The algorithm employs several time varying data structures:

- **Available-** A vector of length m indicates the number of available resources of each type.
- **Allocation-** An n*m matrix defines the number of resources of each type currently allocated to a process. Column represents resource and resource represent process.
- **Request-** An n*m matrix indicates the current request of each process. If request[i][j] equals k then process P_i is requesting k more instances of resource type R_j.

We treat rows in the matrices Allocation and Request as vectors, we refer them as $Allocation_i$ and $Request_i$.

Steps of Algorithm:

1. Let *Work* and *Finish* be vectors of length m and n respectively. Initialize *Work= Available*. For *i=0, 1, ..., n-1*, if $Allocation_i$ = 0, then *Finish[i]* = true; otherwise, *Finish[i]*= false.
2. Find an index i such that both
   a) *Finish[i] == false* b) $Request_i$ <= *Work* If no such *i* exists go to step 4.
3. *Work= Work+* $Allocation_i$ *Finish[i]= true* Go to Step 2.
4. If *Finish[i]== false* for some i, 0<=i<n, then the system is in a deadlocked state. Moreover, if *Finish[i]==false* the process P_i is deadlocked.

Example 1:

1. In this, Work = [0, 0, 0] &

2. Finish = [false, false, false, false, false]

3.
4. *i=0* is selected as both Finish[0] = false and [0, 0, 0]<=[0, 0, 0].

5.
6. Work =[0, 0, 0]+[0, 1, 0] =>[0, 1, 0] &

7. Finish = [true, false, false, false, false].

8.
9. *i=2* is selected as both Finish[2] = false and [0, 0, 0]<=[0, 1, 0].

10.
11. Work =[0, 1, 0]+[3, 0, 3] =>[3, 1, 3] &

12. Finish = [true, false, true, false, false].

13.
14. *i=1* is selected as both Finish[1] = false and [2, 0, 2]<=[3, 1, 3].

15.
16. Work =[3, 1, 3]+[2, 0, 0] =>[5, 1, 3] &

17. Finish = [true, true, true, false, false].

18.
19. *i=3* is selected as both Finish[3] = false and [1, 0, 0]<=[5, 1, 3].

20.
21. Work =[5, 1, 3]+[2, 1, 1] =>[7, 2, 4] &

22. Finish = [true, true, true, true, false].

23.
24. *i=4* is selected as both Finish[4] = false and [0, 0, 2]<=[7, 2, 4].

25.
26. Work =[7, 2, 4]+[0, 0, 2] =>[7, 2, 6] &

27. Finish = [true, true, true, true, true].

28.
29. Since Finish is a vector of all true it means **there is no deadlock** in this example.

Example 2:
Considering a system with five processes P_0 through P_4 and three resources of type 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:

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:

Question2. Is the system in a safe state? If Yes, then what is the safe sequence?
Applying the Safety algorithm on the given system,

Question3. What will happen if process P_1 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 **P_1**.

Recovery From Deadlock

There are three basic approaches to recover from deadlock:

1. Allow users to take manual intervention and inform the system operator.
2. Terminate one or more than one process which is involved in the deadlock.
3. Preempt resources.

How to terminate the process:

- There are two basic approaches to terminate processes and recover the resources allocated to those processes.
    - Terminate all processes involved in deadlock. Deadlock will be recovered but at the expense of more processes termination.
    - Terminate processes one by one until the deadlock is broken, this is a conservative approach and requires deadlock detection after each step.
- In another approach there are many factors that will decide which processes to terminate next:
    - Process priorities.
    - Running time of processes and remaining time to finish.
    - What type of resources are held by process and how many processes are held by the process.
    - How many requests for resources are made by the process.
    - How many processes need to be terminated.
    - Is the process interactive or batch?
    - Is there any non-restorable changes has been made by any process?

Resources preemption: There are three main issues to be addressed while relieving deadlock

- **Selecting a victim** It is a decision making the call to decide which resource must be released by which process. Decision criteria are discussed in the above section.
- **Rollback** Once the resources are taken back from a process, it is hard to decide the safe state in rollback or what is safe rollback. So, the safest rollback is the starting or beginning of the process.
- **Statrvation:** Once the preemption of resources started there is higher chance of starvation of process. Starvation can be avoided or can be decreased by the priority system.

Process Synchronization and Critical Section

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

1. **Independent Process:** Execution of one process does not affect the execution of other processes.

2. **Cooperative Process:** Execution of one process affects the execution of other processes.

So far we have discussed Independent processes where each process had there own address space, and they did not require to communicate with other processes. But the problem arises in the case of Cooperative process because resources are shared in Cooperative processes and various processes communicate with each other.
In this respect, we here will be discussing the problems in process synchronization, in the case of single PC inter-process communication, due to concurrent execution.

Let's look at the below example

List of Global Variable:

```
1
2
3
4
5
6
7
8
```

```
int SIZE = 10;
char buffer[SIZE];
int in = 0, out = 0;
int count = 0
```

The Producer:

```
1
2
```

```
void producer()
{
    while(true)
    {
        while(count == size)
        {
            ;
        }
        buffer[in] = produceItem();
        in =(in+1) % SIZE;
        count++;
    }
}
```

The Consumer:

```
11
12
13
14
15
16
17
```

```
void consumer()
{
    while(true)
    {
        while(count == 0)
        {
            ;
        }
        consumeItem(buffer[out]);
        out =(out+1) % SIZE;
        count--;
    }
}
```

Here we have written codes for Producer and Consumer. We have a few global variables which are shared among both the Producer and Consumer The Producer keeps on adding elements to by increasing the count by 1, each time it adds an element. The consumer, on the other hand, decreases the value of count by 1 each time an item is taken out. This process goes on and is enitled to be preempted at any point of time when the Consumer and producer are needed to be switched between.

Let's delete the unnecessary part of the code and focus on the locus of understanding, count++ and count--. These are not any simple instructions but are converted to the following internal assembly instructions.

count in Producer:

```
1
2
3
4
5
6
7
```

```
reg = count;
reg = reg + 1;
count = reg;
```

count in Consumer:

```
1
2
3
4
5
6
7
```

```
reg = count;
reg = reg - 1;
count = reg;
```

Let's analyze it: Let the value of count be 8 in the Producer part. If in the producer part, just after the register is incremented and before the count gets updated, the process is preempted to some other process, the problem arises. Now if the control flows to Consumer, the reg of the consumer will get value 8 instead of 9. On completion the count = 7 in Consumer part. When the control flows back to the Producer, the process resumes from where it had stopped and count in Producer gets 9. Therefore we have inconsistent values for Producer and Consumer.
This inconsistency among the interdependent stream of execution where each step is related to one another leads to the Race Condition.

Race Condition: Several processes access and process the manipulations over the same data concurrently, then the outcome depends on the particular order in which the access takes place.

Let's look at another example:

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

```
// Global Variable
x = 2
// first function
void fun1()
{
    .
    .
    if (x == 2) // Preemption occurs
        y = x + 5;
    .
    .
    .
}
// second function
void fun2()
{
    .
    .
    .
    x++;
    .
    .
    .
}
```

If there is no preemption in this program, then the execution goes smoothly and y becomes 7. But suppose just after the if condition in fun1(), preemption occurs, then the control flows to fun2(), where x is incremented to 3. When the flows go back to fun1() the if condition will never satisfy and we won't get the desired value of y as 7. This happens because of the Race Condition.

Critical Section: Critical section is a code segment that can be accessed by only one process at a time. The critical section contains shared variables which need to be synchronized to maintain consistency of data variables.
Let's look at the following example:

```
// Global variable
int balance = 100;
void Deposit(int x)
{
    // Preemption occurs
    balance = balance + x;  // Critical section
}

void Withdraw(int x)
{
    balance = balance - x;  // Critical section
}
```

Due to preemption, we get the value of balance as 100 in Deposit() and as 90 in Withdraw(). This inconsistency is called the Race Condition and it occurs due to the preemption along the critical section of the problem. Therefore, we need a mechanism to deal with such cases. The mechanism includes putting up conditions in the entry or exit section.

Entry Section: It is part of the process which decides the entry of a particular process in the Critical Section, out of many other processes.
Exit Section: This process allows the other process that is waiting in the Entry Section, to enter into the Critical Sections. It checks that a process that after a process has finished execution in Critical

Section can be removed through this Exit Section.
Remainder Section: The other parts of the Code other than Entry Section, Critical Section and Exit Section is known as Remainder Section.

Any solution to the critical section problem must satisfy these four requirements or we can say that there are four goals of synchronization algorithm:

1. **Mutual Exclusion:** If a process is executing in its critical section, then no other process is allowed to execute in the critical section.

2. **Progress:** If no process is executing in the critical section and other processes are waiting outside the critical section, then only those processes that are not executing in their remainder section can participate in deciding which will enter in the critical section next, and the selection can not be postponed indefinitely.

3. **Bounded Waiting(Fair):** Abound 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.

4. **Performance:** The mechanism that allows only one process to enter the critical section should be fast. This mechanism is referred to as the locking mechanism and can be implemented in two ways, hardware or software mechanism. The hardware-based mechanism is generally faster since it only involves the registers.

Note: Mutual Exclusion and Progress are mandatory goals for synchronization mechanism.
Overview of Process Synchronization
Overview of Synchronization Mechanism:

1. **Disabling Interrupts:** In this, a process tells the processor that it is undergoing a critical section job and it must not be interrupted before its completion. Now, this might sound good but it comes with various serious problems. This sort of mechanism works only with a single processor system and not in multiprocessor systems. The second and more serious problem is that allowing user processes to stop interruption might lead to system failure or security issues. Therefore, this is not feasible.

2. **Locks or Mutex:** A mutex is a binary variable whose purpose is to provide a locking mechanism. It is used to provide mutual exclusion to a section of code, which means only one process can work on a particular code section at a time. This can be implemented in both software and hardware. It is the building block of all other mechanisms.

3. **Semaphores:** A semaphore is simply a variable. This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment. The two most common kinds of semaphores are counting semaphores and binary semaphores. Counting semaphore can take non-negative integer values and Binary semaphore can take the value 0 & 1. only.

4. **Monitors:** Monitor is one of the ways to achieve Process synchronization. The monitor is supported by programming languages to achieve mutual exclusion between processes. For example Java Synchronized methods. Java provides wait() and notify() constructs.

Few applications of Synchronization are:

- Producer-Consumer Problem
- Dining Philosopher Problem
- Bounded Buffer Problem
- Reader-Writer Problem

Inter-process Communication

A process can be of two types:

- Independent process.
- Co-operating process.

An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes.
Though one can think that those processes, which are running independently, will execute very efficiently but in practice, there are many situations when co-operative nature can be utilized for increasing computational speed, convenience, and modularity.

Interprocess communication (IPC) is a set of interfaces, which is usually programmed in order for the programs to communicate between a series of processes. This allows running f program concurrently in an Operating System. These are methods in IPC:

1. **Pipes (Same Process) -** This allows the flow of data in one direction only. Analogous to simplex systems (Keyboard). Data from the output is usually buffered until the input process receives it which must have a common origin.

2. **Names Pipes (Different Processes) -** This is a pipe with a specific name it can be used in processes that don't have a shared common process origin. E.g. in FIFO, where the data is written to a pipe, is first named.

3. **Message Queuing -** This allows messages to be passed between processes using either a single queue or several message queue. This is managed by system kernel these messages are coordinated using an API.
4. **Semaphores -** This is used in solving problems associated with synchronization and to avoid race condition. These are integer values which are greater than or equal to 0.
5. **Shared memory -** This allows the interchange of data through a defined area of memory. Semaphore values have to be obtained before data can get access to shared memory.
6. **Sockets -** This method is mostly used to communicate over a network between a client and a server. It allows for a standard connection which is computer and OS independent.

Inter-Process Communication through shared memory is a concept where two or more processes can access the common memory. And communication is done via this shared memory where changes made by one process can be viewed by another process.
The problem with pipes, FIFO and message queue – is that for two processes to exchange information. The information has to go through the kernel.

- Server reads from the input file.
- The server writes this data in a message using either a pipe, FIFO or message queue.

- The client reads the data from the IPC channel, again requiring the data to be copied from kernel’s IPC buffer to the client’s buffer.

- Finally the data is copied from the client’s buffer.

A total of four copies of data are required (2 reads and 2 writes). So, shared memory provides a way by letting two or more processes share a memory segment. With Shared Memory the data is only copied twice – from input file into shared memory and from shared memory to the output file.

SYSTEM CALLS USED ARE:

ftok(): is use to generate a unique key.

shmget(): int shmget(key_t,size_tsize,intshmflg); upon successful completion,
shmget() returns an identifier for the shared memory segment.

shmat(): Before you can use a shared memory segment, you have to attach yourself to it using shmat(). void *shmat(int shmid ,void *shmaddr ,int shmflg);
shmid is a shared memory id. shmaddr specifies a specific address to use but we should set
it to zero and OS will automatically choose the address.

shmdt(): When you're done with the shared memory segment, your program should
detach itself from it using shmdt(). int shmdt(void *shmaddr);

shmctl(): when you detach from shared memory,it is not destroyed. So, to destroy
shmctl() is used. shmctl(int shmid,IPC_RMID,NULL);

Interprocess Communication (IPC) is a mechanism that allows processes to communicate with each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them. Processes can communicate with each other using these two ways:

- **Shared Memory** Communication between processes using shared memory requires processes to share some variable and it completely depends on how the programmer will implement it. The Figure 1 below shows a basic structure of communication between processes via shared memory method and via message passing.

  Example of shared memory: Producer consumer problem

- **Message passing:** In message passing method processes communicate with each other without using any kind of shared memory. If two processes p1 and p2 want to communicate with each other, they proceed as follow:

  - Establish a communication link (if a link already exists, no need to establish it again.)
  - Start exchanging messages using basic primitives.
    We need at least two primitives:
    – **send**(message, destinaion) or **send**(message)
    – **receive**(message, host) or **receive**(message)

The message size can be of fixed size or of variable size. if it is of fixed size, it is easy for OS designer but complicated for the programmer and if it is of variable size then it is easy for the

programmer but complicated for the OS designer. A standard message can have two parts: **header and body.** The **header part** is used for storing Message type, destination id, source id, message length and control information. The control information contains information like what to do if runs out of buffer space, sequence number, the priority. Generally, the message is sent using the FIFO style.
A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. A new queue is created or an existing queue opened by **msgget()**.
New messages are added to the end of a queue by **msgsnd()**. Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to msgsnd() when the message is added to a queue. Messages are fetched from a queue by **msgrcv()**. We don't have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type of field.

All processes can exchange information through access to a common system message queue. The sending process places a message (via some (OS) message-passing module) onto a queue that can be read by another process. Each message is given an identification or type so that processes can select the appropriate message. The process must share a common key in order to gain access to the queue in the first place.

System calls used for message queues:

```
ftok(): is use to generate a unique key.

msgget(): either returns the message queue identifier for a newly created message
queue or returns the identifiers for a queue that exists with the same key value.

msgsnd(): Data is placed on to a message queue by calling msgsnd().

msgrcv(): messages are retrieved from a queue.

msgctl(): It performs various operations on a queue. Generally it is use to
destroy message queue.
```

Locks for Synchronization

Let's understand the working of Lock mechanism using the following example:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
```

```
15
16
17
18
```

```
// Global variables
int amount = 100;
bool lock = false;
void Deposit(int x)
{
    // Critical section
    amount = amount + x;
}

void Withdraw(int x)
{
    // Critical section
    amount = amount - x;
}
```

Previously we saw that this problem provided inconsistent results. Now, let's add the lock mechanism into this piece of code:

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

```
// Global variables
int amount = 100;
bool lock = false;
void Deposit(int x)
{
    // entry section
    while(lock == true)
    {
        ;
    }
    // Preempted
    lock = true;
    // Critical section
    amount = amount + x;
    // exit section
    lock = false;
}

void Withdraw(int x)
{
    // entry section
    while(lock == true)
    {
        ;
    }
    lock = true;
```

Let's analyze the mechanism. Although we have applied the locking mechanism, still the code doesn't guarantee mutual exclusion that is a mandatory requirement. Suppose in the deposit part, the process gets preempted just after the while statement and the control flows to the Withdraw part. After the execution, when the control flows back to the Deposit part, it enters the critical section.

Therefore, the principle of mutual exclusion is violated, as both the processes might enter the critical section and cause a race condition.
To avoid this we use a hardware-based implementation called Test and Set or the TSL lock mechanism.
Deposit and Withdraw mechanism using TSL lock:

```
// Global variables
int amount = 100;
bool lock = false;
void Deposit(int x)
{
```

```
    while(test_and_set(lock))
    {
        ;
    }
    // Critical section
    amount = amount + x;
    lock = false;
}

void Withdraw(int x)
{
    // entry section
    while(test_and_set(lock))
    {
        ;
    }
    // Preempted
    lock = true;

    // Critical section
    amount = amount - x;
    // exit section
```

Run

This TSL lock work both ways, it waits while the lock is true and also the lock is updated as true if it is false. This function doing both the things has to be atomic. So this is an atomic operation to acquire the lock and once it is acquired nobody else could acquire the lock. This solves the basic problem of Process Synchronization.

Process synchronization - Semaphore and MUTEX

Problem with Locking Mechanism: In the previous lectures, we have seen how the locking mechanism facilitates process execution by applying locks when a process enters the critical section and releasing it when the job is done. While doing this other processes had to wait and they went into a waiting loop until the process inside completes its execution. This is a major problem and can be avoided using Semaphores.

Semaphore: Semaphore is simply a variable. This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment. The two most common kinds of semaphores are counting semaphores and binary semaphores. Counting semaphore can take non-negative integer values and Binary semaphore can take the value 0 & 1 only.

1. **Counting Semaphore:** Let's see how this works using a restroom analogy. Suppose there is a restroom(critical section) which is being used by a person(process). When another process comes, he had to wait until the previous process comes out and releases the lock. All the upcoming processes had to go into a waiting loop. This problem can be avoided using a semaphore. Consider a semaphore is like a guard who maintains a record and as long as a process is inside the restroom, it tells all other processes to sleep or do any other task and not to go into a waiting loop. When the process inside completes its execution, the guard informs the process in the queue and it can enter the restroom.
   To perform this complex process, the guard or the semaphore maintains a record. It maintains two functions namely, wait and signal.

   wait(): The wait keeps a record of the count. The count is initially assigned a value equal to the number of available resources. Every time a process is allocated to a resource, the value of count is

decreased by one. Once the value of count is negative, it indicates that all the resource is underuse and the address of the PCB of the upcoming processes are stored in a queue. This way, the processes are not kept waiting. The negative value now indicates the number of processes outside who need to use the resource.

signal(): The signal function increments the value of count by one every time a process finish using a resource which can be made available to some other process. The guard wakes or signals a process in the queue when the count is decremented, that a resource is free to use and releases a process in the FCFS basis.

Let's write a pseudo code for the above process. For start we assume there are 3 available resources. So count will be initialized as 3. We have a semaphore containing a count and queue, to keep track of the processes.

PseudoCode:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
```

```
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
```

```
// A semaphore structure having count and queue
struct Sem{
    int count;
    Queue q;
}
```

```
// Creating variable s of semaphore
// where count = 3 and q is empty
Sem s(3, empty);
// Used while entering into the washroom
// or the critical section
void wait(){
    // process before entering CS
    s.count--;
    // No resource is available
    if(s.count < 0){
    // No resource is available
        1. Add the caller process P to q;
        2. sleep(P);
    }
}
// Used while exiting from the washroom
// or the critical section
void signal(){
    // process after exiting CS
```

Now let's look at two operations that can be used to access and change the value of the semaphore variable and how this was originally implemented by Dijkstra:

Some point regarding P and V operation

1. P operation is also called wait, sleep or down operation and V operation is also called signal, wake-up or up operation.
2. Both operations are atomic and semaphore(s) is always initialized to one.
3. A critical section is surrounded by both operations to implement process synchronization. See below image critical section of Process P is in between P and V operation.

This PV operation comes with few problems like busy waiting, as the while loop keeps running when the process does not get a resource or critical section. This way the CPU cycle gets wasted. Another problem is that of bounded waiting when a process because of preemption might not get a resource for a very long time. These problems have been solved in the previous discussion where we have used count, queue and sleep and wake up concept to make efficient use of CPU ans see that no CPU cycle is wasted.

2. **Binary Semaphore:** In a binary semaphore, the counter logically goes between 0 and 1. You can think of it as being similar to a lock with two values: open/closed. In binary semaphore, only two processes can enter into the critical section at any time and mutual exclusion is also guaranteed. Let's look a the pseudo-code below to understand the logic behind the implementation of binary semaphore. This Binary Semaphore can be used as mutex too with the additional functionalities like sleep and wake concept etc. Let's look at the pseudo-code:

```
1
2
3
4
5
6
7
8
9
10
```

```
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
```

```
// Binary Semaphore containing
```

```
// a boolean value and queue q
// to keep track of processes
// entering critical section
struct BinSem
{
    bool val;
    Queue q;
};
// Global initialization
BinSem s(true, empty);
// wait() is called before critical section
// during entry
void wait()
{
    // Checking if critical section is
    // available or not
    if (s.val == 1)
        // acquiring the critical section
        s.val = 0;

    // if not available
    else
```

```
{
    1. Put this process P in q;
    2. sleep(P);
}
```

Implementation of Mutual Exclusion: Now, let us see how it implements mutual exclusion. Let there be two processes P1 and P2 and a semaphore s is initialized as 1. Now if suppose P1 enters in its critical section then the value of semaphore s becomes 0. Now if P2 wants to enter its critical section then it will wait until s > 0, this can only happen when P1 finishes its critical section and calls V operation on semaphore s. This way mutual exclusion is achieved. Look at the below image for details.

The description above is for binary semaphore which can take only two values 0 and 1. There is one other type of semaphore called counting semaphore which can take values greater than one.

Now suppose there is a resource whose number of instances is 4. Now we initialize S = 4 and rest is the same as for binary semaphore. Whenever the process wants that resource it calls P or wait() function and when it is done it calls V or signal function. If the value of S becomes zero then a process has to wait until S becomes positive. For example, Suppose there are 4 processes P1, P2, P3, P4 and they all call wait operation on S(initialized with 4). If another process P5 wants the resource then it should wait until one of the four processes calls signal function and the value of semaphore becomes positive.

Problem in this implementation of semaphore
Whenever any process waits then it continuously checks for semaphore value (look at this line while (s==0); in P operation) and waste CPU cycle. To avoid this another implementation is provided below.

```
1
2
3
4
5
6
7
8
9
10
11
12
```

```
P(Semaphore s)
{
    s = s - 1;
    if (s <= 0) {
        // add process to queue
        block();
    }
}
V(Semaphore s)
{
    s = s + 1;
    if (s <= 0) {
        // remove process p from queue
        wakeup(p);
    }
}
```

- In this implementation whenever process waits it is added to a waiting queue of processes associated with that semaphore. This is done through the system call block() on that process. When a process is completed it calls signal function and one process in the queue is resumed. It uses wakeup() system call.

    - A mutex is a binary variable whose purpose is to provide a locking mechanism. It is used to provide mutual exclusion to a section of code, which means only one process can work on a particular code section at a time.

    - There is misconception that binary semaphore is same as mutex variable but both are different in the sense that binary semaphore apart from providing locking mechanism also provides two atomic operation signal and wait, means after releasing resource semaphore will provide signaling mechanism for the processes who are waiting for the resource.

What are the differences between Mutex vs Semaphore? When to use mutex and when to use semaphore?

As per operating system terminology, mutex and semaphore are kernel resources that provide synchronization services (also called as *synchronization primitives*). *Why do we need such synchronization*

primitives? Won't be only one sufficient? To answer these questions, we need to understand a few keywords. Please read the posts on the critical section. We will illustrate with examples to understand these concepts well, rather than following the usual OS textual description.

The producer-consumer problem:

Consider the standard producer-consumer problem. Assume, we have a buffer of 4096-byte length. A producer thread collects the data and writes it to the buffer. A consumer thread processes the collected data from the buffer. The objective is, both the threads should not run at the same time.

- **Using Mutex:**
  A mutex provides mutual exclusion, either producer or consumer can have the key (mutex) and proceed with their work. As long as the buffer is filled by the producer, the consumer needs to wait, and vice versa.

  At any point of time, only one thread can work with the *entire* buffer. The concept can be generalized using a semaphore.

- **Using Semaphore:**
  A semaphore is a generalized mutex. In lieu of single buffer, we can split the 4 KB buffer into four 1 KB buffers (identical resources). A semaphore can be associated with these four buffers. The consumer and producer can work on different buffers at the same time.

Misconception:

- There is an ambiguity between *binary semaphore* and *mutex*. We might have come across that a mutex is a binary semaphore. *But they are not*! The purpose of mutex and semaphore is different. Maybe, due to the similarity in their implementation a mutex would be referred to as a binary semaphore.

- Strictly speaking, a mutex is **locking mechanism** used to synchronize access to a resource. Only one task (can be a thread or process based on OS abstraction) can acquire the mutex. It means there is ownership associated with a mutex, and only the owner can release the lock (mutex).

- Semaphore is **signaling mechanism** ("I am done, you can carry on" kind of signal). For example, if you are listening songs (assume it as one task) on your mobile and at the same time your friend calls you, an interrupt is triggered upon which an interrupt service routine (ISR) signals the call processing task to wakeup.

Note: The content is generalized explanation. Practical details vary with implementation.

General Questions:

1. *Can a thread acquire more than one lock (Mutex)?*
   Yes, it is possible that a thread is in need of more than one resource, hence the locks. If any lock is not available the thread will wait (block) on the lock.

2. *Can a mutex be locked more than once?*
   A mutex is a lock. Only one state (locked/unlocked) is associated with it. However, a *recursive mutex* can be locked more than once (POSIX complaint systems), in which a count is associated with it, yet retains only one state (locked/unlocked). The programmer must unlock the mutex as many number times as it was locked.

3. *What happens if a non-recursive mutex is locked more than once.*
   Deadlock. If a thread which had already locked a mutex, tries to lock the mutex again, it will enter into the waiting list of that mutex, which results in a deadlock. It is because no other thread can unlock the mutex. An operating system implementer can exercise care in identifying the owner of the mutex and return if it is already locked by the same thread to prevent deadlocks.

4. *Are binary semaphore and mutex same?*
   No. We suggest to treat them separately, as it is explained by signaling vs locking mechanisms. But a binary semaphore may experience the same critical issues (e.g. priority inversion) associated with a mutex. We will cover these in a later article.

   A programmer can prefer mutex rather than creating a semaphore with count 1.

5. *What are a mutex and critical section?*
   Some operating systems use the same word *critical section* in the API. Usually, a mutex is a costly operation due to protection protocols associated with it. At last, the objective of mutex is atomic access. There are other ways to achieve atomic access like disabling interrupts which can be much faster but ruins responsiveness. The alternate API makes use of disabling interrupts.

6. *What are events?*
   The semantics of mutex, semaphore, event, critical section, etc... are the same. All are synchronization primitives. Based on their cost in using them they are different. We should consult the OS documentation for the exact details.

7. *Can we acquire mutex/semaphore in an Interrupt Service Routine?*
   An ISR will run asynchronously in the context of the current running thread. It is **not recommended** to query (blocking call) the availability of synchronization primitives in an ISR. The ISR is meant to be short, the call to mutex/semaphore may block the current running thread. However, an ISR can signal a semaphore or unlock a mutex.

8. *What we mean by "thread blocking on mutex/semaphore" when they are not available?*
   Every synchronization primitive has a waiting list associated with it. When the resource is not available, the requesting thread will be moved from the running list of the processor to the waiting list of the synchronization primitive. When the resource is available, the higher priority thread on the waiting list gets the resource (more precisely, it depends on the scheduling policies).

9. *Is it necessary that a thread must block always when the resource is not available?*
   Not necessary. If the design is sure '*what has to be done when the resource is not available*', the thread can take up that work (a different code branch). To support application requirements the OS provides non-blocking API.

   For example POSIX pthread_mutex_trylock() API. When the mutex is not available the function returns immediately whereas the API pthread_mutex_lock() blocks the thread till resource is available.

Monitors

The monitor is one of the ways to achieve Process synchronization. The monitor is supported by programming languages to achieve mutual exclusion between processes. For example Java Synchronized methods. Java provides wait() and notify() constructs.

1. It is the collection of condition variables and procedures combined together in a special kind of module or a package.

2. The processes running outside the monitor can't access the internal variable of the monitor but can call procedures of the monitor.

3. Only one process at a time can execute code inside monitors.

Syntax of Monitor

The idea of implementing monitors is that instead of going into the complexities of acquiring, lease, wait for the signal, we use a class and put various functions inside it and synchronize the whole mechanism.

Example:

```
// Illustrating Monitor concept in Java
// higher level of synchronization
class AccountUpdate
{
    // shared variable
    private int bal;
```

```
    // synchronized method
    void synhronized deposit(int x)
    {
        bal = bal + x;
    }

    // synchronized method
    void synhronized withdraw(int x)
    {
        bal = bal - x;
    }
}
```
Run

Condition Variables
Two different operations are performed on the condition variables of the monitor.

```
Wait.

signal.
```
let say we have 2 condition variables
condition x, y; //Declaring variable

Wait operation x.wait() : Process performing wait operation on any condition variable are suspended. The suspended processes are placed in block queue of that condition variable.

Note: Each condition variable has its unique block queue.

Signal operation x.signal(): When a process performs signal operation on condition variable, one of the blocked processes is given chance.

```
If (x block queue empty)

  // Ignore signal

else

  // Resume a process from block queue.
```

Priority Inversion
In the Operating System, one of the important concepts is Task Scheduling. There are several Scheduling methods such as First Come First Serve, Round Robin, Priority-based scheduling, etc. Each scheduling method has its pros and cons. As you might have guessed, Priority Inversion comes under Priority-based Scheduling. Basically, it's a problem which arises sometimes when Priority-based scheduling is used by OS. In Priority-based scheduling, different tasks are given different priority so that higher priority tasks can intervene in lower priority tasks if possible.

So, in priority-based scheduling, if lower priority task (L) is running and if a higher priority task (H) also needs to run, the lower priority task (L) would be preempted by higher priority task (H). Now, suppose

both lower and higher priority tasks need to share a common resource (say access to the same file or device) to achieve their respective work. In this case, since there are resource sharing and task synchronization is needed, several methods/techniques can be used for handling such scenarios. For sake of our topic on Priority Inversion, let us mention a synchronization method say mutex. Just to recap on the mutex, a task acquires mutex before entering critical section (CS) and releases mutex after exiting critical section (CS). While running in CS, task access this common resource. Now, say both L and H shares a common Critical Section (CS) i.e. the same mutex is needed for this CS.
Coming to our discussion of priority inversion, let us examine some scenarios.

1. L is running but not in CS; H needs to run; H preempts L; H starts running; H relinquishes or releases control; L resumes and starts running

2. L is running in CS; H needs to run but not in CS; H preempts L; H starts running; H relinquishes control; L resumes and starts running.

3. L is running in CS; H also needs to run in CS; H waits for L to come out of CS; L comes out of CS; H enters CS and starts running

Please note that the above scenarios don't show the problem of any Priority Inversion (not even scenario 3). Basically, so long as lower priority task isn't running in shared CS, higher priority task can preempt it. But if L is running in shared CS and H also needs to run in CS, H waits until L comes out of CS. The idea is that CS should be small enough so that it doesn't result in H waiting for a long time while L was in CS. That's why writing a CS code requires careful consideration. In any of the above scenarios, priority inversion (i.e. reversal of priority) didn't occur because the tasks are running as per the design.
Now let us add another task of middle priority say M. Now the task priorities are in the order of L < M < H. In our example, M doesn't share the same Critical Section (CS). In this case, the following sequence of task running would result in 'Priority Inversion' problem.
4) L is running in CS; H also needs to run in CS; H waits for L to come out of CS; M interrupts L and starts running; M runs till completion and relinquishes control; L resumes and starts running till the end of CS; H enters CS and starts running.
Note that neither L nor H share CS with M.
Here, we can see that running of M has delayed the running of both L and H. Precisely speaking, H is of higher priority and doesn't share CS with M; but H had to wait for M. This is where Priority-based scheduling didn't work as expected because priorities of M and H got inverted in spite of not sharing any CS. This problem is called Priority Inversion. This is what Priority Inversion is. In a system with priority-based scheduling, higher priority tasks can face this problem and it can result in unexpected behaviour/result. In general-purpose OS, it can result in slower performance. In RTOS, it can result in more severe outcomes. The most famous 'Priority Inversion' problem was what happened at Mars Pathfinder.
If we have a problem, there has to be a solution for this. For Priority Inversion as well, there're different solutions such as Priority Inheritance, etc.

Priority Inheritance: In Priority Inversion, higher priority task (H) ends up waiting for middle priority task (M) when H is sharing a critical section with lower priority task (L) and L is already in the critical section. Effectively, H waiting for M results in inverted priority i.e. Priority Inversion. One of the solutions to this problem is Priority Inheritance. In Priority Inheritance, when L is in the critical section, L inherits the priority of H at the time when H starts pending for critical section. By doing so, M doesn’t interrupt L and H doesn’t wait for M to finish. Please note that inheriting of priority is done temporarily i.e. L goes back to its old priority when L comes out of the critical section.

Memory Mangement - Type of memory

Memories are made up of registers. Each register in the memory is one storage location. The storage location is also called a memory location. Memory locations are identified using **Address**. The total number of bit a memory can store is its **capacity**.

A storage element is called a **Cell**. Each register is made up of storage element in which one bit of data is stored. The data in a memory are stored and retrieved by the process called **writing** and **reading** respectively.

A **word** is a group of bits where a memory unit stores binary information. A word with a group of 8 bits is called a **byte**.
A memory unit consists of data lines, address selection lines, and control lines that specify the direction of transfer. The block diagram of a memory unit is shown below:

Data lines provide the information to be stored in memory. The control inputs specify the direct transfer. The k-address lines specify the word chosen.

When there are k address lines, 2^k memory word can be accessed.

In the Computer System Design, Memory Hierarchy is an enhancement to organize the memory such that it can minimize the access time. The Memory Hierarchy was developed based on a program behavior known as locality of references. The figure below clearly demonstrates the different levels of memory hierarchy :

This Memory Hierarchy Design is divided into 2 main types:

1. **External Memory or Secondary Memory -** Comprising of Magnetic Disk, Optical Disk, Magnetic Tape i.e. peripheral storage devices which are accessible by the processor via I/O Module.
2. **Internal Memory or Primary Memory -** Comprising of Main Memory, Cache Memory & CPU registers. This is directly accessible by the processor.

We can infer the following characteristics of Memory Hierarchy Design from the above figure:

1. **Capacity:**
   It is the global volume of information the memory can store. As we move from top to bottom in the Hierarchy, the capacity increases.

2. **Access Time:**
   It is the time interval between the read/write request and the availability of the data. As we move from top to bottom in the Hierarchy, the access time increases.

3. **Performance:**
   Earlier when the computer system was designed without Memory Hierarchy design, the speed gap increases between the CPU registers and Main Memory due to the large difference in access time. This results in lower performance of the system and thus, enhancement was required. This enhancement was made in the form of Memory Hierarchy Design because of which the performance of the system increases. One of the most significant ways to increase system performance is minimizing how far down the memory hierarchy one has to go to manipulate data.

4. **Cost per bit:**
   As we move from bottom to top in the Hierarchy, the cost per bit increases i.e. Internal Memory is costlier than External Memory.

Static and dynamic memeory to be framed Memory is the most essential element of a computing system because without it computers can’t perform simple tasks. Computer memory is of two basic types - Primary memory / Volatile memory and Secondary memory / non-volatile memory. Random Access Memory (RAM) is a volatile memory and Read Only Memory (ROM) is a non-volatile memory.

1. Random Access Memory (RAM) -

- It is also called as *read write memory* or the *main memory* or the *primary memory*.
- The programs and data that the CPU requires during execution of a program are stored in this memory.
- It is a volatile memory as the data loses when the power is turned off.
- RAM is further classified into two types- *SRAM (Static Random Access Memory)* and *DRAM (Dynamic Random Access Memory)*.

2. Read Only Memory (ROM) -

- Stores crucial information essential to operate the system, like the program essential to boot the computer.
- It is not volatile.
- Always retains its data.
- Used in embedded systems or where the programming needs no change.
- Used in calculators and peripheral devices.
- ROM is further classified into 4 types- *ROM*, *PROM*, *EPROM*, and *EEPROM*.

Types of Read Only Memory (ROM) -

1. **PROM (Programmable read-only memory)** - It can be programmed by user. Once programmed, the data and instructions in it cannot be changed.
2. **EPROM (Erasable Programmable read only memory)** - It can be reprogrammed. To erase data from it, expose it to ultra violet light. To reprogram it, erase all the previous data.
3. **EEPROM (Electrically erasable programmable read only memory)** - The data can be erased by applying electric field, no need of ultra violet light. We can erase only portions of the chip.

Programs Compiled and Run

Example:

```
1
2
3
4
5
6
7
8
9
```

```c
#include <stdio.h>
int main() {
    printf("Hello");
    return 0;
}
```

Let's take this sample code to understand the various steps.
Step 1: The above code is the source code which is passed to the compiler.

Step 2: The compiler converts this source code to machine-executable object code.

Step 3: Then Linker is used to linking the functions mentioned inside the main function. Here "printf()" is a library function, whose code is attached to the main function using the linker. The linking can be done statically or dynamically.

- **Static Linking:** In this, the code of the "printf" function is copied into the object file and then one executable file is generated and the execution is done line by line.

- **Dynamic Linking:** This refers to the linking that is done during load or run-time and not when the file is created. When one program is dependent on some other program rather than loading all the dependent programs, CPU links the dependent programs to the main executing program when it's required.

Step 4: After the creation of the executable file the loader copies the executable code into the main memory.

Step 5: Finally the code is run in the main memory and while running dynamic linking can be made with other system libraries.

Address Binding

Address binding refers to the mapping of relocatable or relative address to physical addresses. The binary of code contains relative or relocatable addresses which are used to commute between various instructions. There can be various binary instructions ranging from goto, load, add etc. To execute this sort of binary files, they are needed to be loaded into the main memory. This main memory has physical addresses, and any processes stored in the main memory acquires a slot. The physical addresses of the main memory influence the binary addresses accordingly by readjusting them accordingly. For eg., when the binary file is loaded into the slot after 50k, they are readjusted to 50k, 50k+1, 50k+2, 50k+3 etc.

These bindings can happen at various stages of execution:

1. **Compile Time:** Its work is to generate a logical address(also known as virtual address). If you know that during compile time where the process will reside in memory then the absolute address is generated.
    - Instruction are translated into an absolute address.

- It works with the logical address.
- It is static.
- Code is compiled here.

2. **Load time:** It will generate a physical address. If at the compile-time it is not known where the process will reside then relocatable address will be generated. In this, if address changes then we need to reload the user code.
    - Absolute address is converted to relocatable address
    - It works with a physical address
    - It is static
    - Instructions are loaded in memory

Problem with Load Time Binding is that once it is loaded into the main memory, it cannot be relocated. Let's look at this example:

Suppose there is this process P1 which is loaded at 50k of the main memory. If it is waiting for some I/O, then the process is swapped out to make the memory available to some other processes. Now, when P1 completes its I/O work and it is swapped back into the main memory, the process is stored at some different address, suppose 150k. This cannot be implemented in Load Time binding.

3. **Run Time:** It is the most preferred binding technique in modern OS. The addresses generated by the CPU during the run time are logical addresses and not the physical addresses. This logical address is then converted into physical addresses using a memory-management unit(MMU). This is used in the process can be moved from one memory to another during execution(dynamic linking-Linking that is done during load or run time). Hardware support is required during run time-binding.

    - The dynamic absolute address is generated.

    - It works in between and helps in execution.

    - It is dynamic.

    - From memory, instructions are executed by CPU.

Runtime Binding

Runtime Binding happens through hardware. The memory management unit in CPU has two registers namely, Limit Register and the Relocation Register. When a process is loaded into the memory, during context switching, the OS automatically loads these registers into the memory. The registers are loaded according to the slots in the memory.

The process begins with the CPU generating the logical addresses. Then the process is sent to the memory management unit where the limit register first checks whether the generated logical address is within or beyond the memory allocation. If the address is beyond the limit, a trap is sent to the OS else the control flows to the Relocation register which relocates or adjusts the address and converts the logical address to physical address.

Why is the whole process implemented in hardware?
If a software implements this runtime binding, then during the context switching, when the logical address is generated, an OS system call will happen, which will convert the logical address to physical address. This will require Mode switching from process to OS context which will be done for every instruction, generated by the CPU. This will be a very slow process and hence the hardware is implemented to do the binding.

Evolution of Memory Management

The memory management evolved in two phases:
Single Tasking Systems: This sort of system has a part loaded with OS. When one process arrives it is assigned to a slot and only after its completion can a second process be allowed a slot in the memory. Suppose in the diagram only after the execution of P1 completes can a process P2 be loaded into the memory.

The problem with this system was that the context switching could not take place in a single-tasking system when multiple processes are sent into the memory. This results in a low degree of multiprogramming leading to the less efficient use of CPU.

Multitasking Systems: In this multiple processes can be taken by the memory at a single time and each process are run one by one using efficient memory management techniques, leading to the higher

degree of multiprogramming. In the below diagram we can see that P1, P2 and P3 are stored inside the memory at the same time and can be run simultaneously.

There can be various types of memory management allocation technique in Multitasking.

Static or Fixed Partitioning Technique: This is the oldest and simplest technique used to put more than one processes in the main memory. In this partitioning, a number of partitions (non-overlapping) in RAM is fixed but the size of each partition may or may not be same. As it is contiguous allocation, hence no spanning is allowed. Here partition is made before execution or during system configure. This can be of two types:

1. **Equal Size Memory Allocation:** In this technique, fixed memory size is allocated to each process. Let's look at the below example:

   In this case, each process P1, P2 and P3 are allocated 10MB each irrelevant of how much memory each process requires. Suppose P1 had a demand of 8MB, P2 7MB and P3 6Mb. After each allocation, we can see that 2MB, 3MB, and 4Mb of memory is wasted from each slot. Although it adds up to 9 Mb, when a process P4 demands 5Mb of memory, it cannot be allocated, since they are available in different chunks. This phenomenon is referred to as external fragmentation. Let's understand fragmentation:
   Fragmentation occurs in a dynamic memory allocation system when many of the free blocks are too small to satisfy any request.

   - **External Fragmentation:** External Fragmentation happens when a dynamic memory allocation algorithm allocates some memory and a small piece is left over that cannot be effectively used. If too much external fragmentation occurs, the amount of usable memory is drastically reduced. Total memory space exists to satisfy a request, but it is not contiguous.
   - **Internal Fragmentation:** Internal fragmentation is the space wasted inside of allocated memory blocks because of restriction on the allowed sizes of allocated blocks. Allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.

   In the case of equal size memory allocation, internal fragmentation leads to external fragmentation. This results in high memory loss since we can observe that 9 MB of memory goes wasted.

2. **Unequal Size Memory allocation:** This technique is used to reduce memory loss in comparison to the equal size memory allocation. Let's look at the below example.

   Suppose we want to allocate a process of 3 MB to the memory. Instead of allocating it to a fixed size of memory, a minimum MB slot is found and P1 is allocated to it. In our case, the 4MB slot is allocated to P1 consuming 3MB memory. If another process P2 arrives with a memory requirement of less than 2 Mb, then the first slot is allocated to it. Similarly, another process of 7Mb can take the 8Mb slot. So, we can see that although here also there is memory loss due to internal fragmentation and ultimately leading to external fragmentation, it is less in comparison to the Equal size memory allocation. Here there is a memory loss of total 3 MBs.

Advantages of Fixed Partitioning:

1. **Easy to implement:** Algorithms needed to implement Fixed Partitioning are easy to implement. It simply requires putting a process into certain partition without focussing on the emergence of Internal and External Fragmentation.

2. **Little OS overhead:** Processing of Fixed Partitioning require lesser excess and indirect computational power.

Disadvantages of Fixed Partitioning -

1. **Internal Fragmentation:** Main memory use is inefficient. Any program, no matter how small, occupies an entire partition. This can cause internal fragmentation.

2. **External Fragmentation:** The total unused space (as stated above) of various partitions cannot be used to load the processes even though there is space available but not in the contiguous form (as spanning is not allowed).

3. **Limit process size:** Process of size greater than size of partition in Main Memory cannot be accommodated. Partition size cannot be varied according to the size of incoming process's size. Hence, process size of 32MB in above stated example is invalid.

4. **Limitation on Degree of Multiprogramming:** Partition in Main Memory are made before execution or during system configure. Main Memory is divided into fixed number of partition. Suppose if there are $n1$n1 partitions in RAM and $n2$n2 are the number of processes, then $n2 \leq n1$n2≤n1 condition must be fulfilled. Number of processes greater than number of partitions in RAM is invalid in Fixed Partitioning.

Dynamic or Variable Partitioning Technique:
It is a part of Contiguous allocation technique. It is used to alleviate the problem faced by Fixed Partitioning. In contrast with fixed partitioning, partitions are not made before the execution or during system configure. Various **features** associated with variable Partitioning-

1. Initially RAM is empty and partitions are made during the run-time according to process's need instead of partitioning during system configure.

2. The size of partition will be equal to incoming process.

3. The partition size varies according to the need of the process so that the internal fragmentation can be avoided to ensure efficient utilisation of RAM.

4. Number of partitions in RAM is not fixed and depends on the number of incoming process and Main Memory's size.

There are some advantages and disadvantages of variable partitioning over fixed partitioning as given below.

Advantages of Variable Partitioning -

1. **No Internal Fragmentation:** In variable Partitioning, space in main memory is allocated strictly according to the need of process, hence there is no case of internal fragmentation. There will be no unused space left in the partition.

2. **No restriction on Degree of Multiprogramming:** More number of processes can be accommodated due to absence of internal fragmentation. A process can be loaded until the memory is not empty.

3. **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 loaded and process can not be divided as it is invalid in contiguous allocation technique. Here, In variable partitioning, the process size can't be restricted since the partition size is decided according to the process size.

Disadvantages of Variable Partitioning -

1. **Difficult Implementation:** Implementing variable Partitioning is difficult as compared to Fixed Partitioning as it involves allocation of memory during run-time rather than during system configure.

2. **External Fragmentation:** There will be external fragmentation inspite of absence of internal fragmentation.

   For example, suppose in above example- process P1(2MB) and process P3(1MB) completed their execution. Hence two spaces are left i.e. 2MB and 1MB. Let's suppose process P5 of size 3MB comes. The empty space in memory cannot be allocated as no spanning is allowed in contiguous allocation. The rule says that the process must be contiguously present in main memory to get executed. Hence it results in External Fragmentation. These spaces are called holes.

   Now P5 of size 3 MB cannot be accommodated in spite of required available space because in contiguous no spanning is allowed.

Dynamic Partitioning

The major disadvantage of dynamic partitioning is that when a process completes its execution, it leaves the memory leaving a hole behind. Thus when multiple processes leave, similar holes are created and thus even if there is no internal fragmentation, external fragmentation is observed.

There are two methods to avoid such holes:

1. **Bitmap:** These are used to keep a track of the unit of memories that are occupied as 1 and the unallocated memories are assigned 0. So when a new process comes, it looks for the 0's to get a track of unallocated spaces and thus the problem is solved. This way memory management is made simpler and easier.

   It also has a major problem. Bitmap requires a lot of space. Since for every unit of memory, a bit is needed in the bitmap to store the value. Eg., Suppose we have a memory unit of size 32 bit. So for every 32 unit, 1 bit of memory is needed to store the bit in the bitmap. Therefore for 32 bits, one bit is wasted. So 1 by 33 of the memory is wasted in the process.
   If a larger size of the unit is taken, then this might lead to internal fragmentation. So both the processes are not feasible.

2. **Linked List:** In this the processes and the holes are connected using a doubly-linked list. So if a process ends in between, then the holes get connected to form a bigger hole. In the below diagram if P2 completes its execution, then both the H combines and a greater hole is generated, which can accommodate any incoming process of relevant size. But, there is a catch of how to allocate memory as in what process should get which holes. We have various strategies to do so. Let's discuss:

   In Partition Allocation, when there is more than one partition freely available to accommodate a process's request, a partition must be selected. To choose a particular partition, a partition allocation method is needed. A partition allocation method is considered better if it avoids internal fragmentation.

   Below are the various partition allocation schemes :

   1. **First Fit:** In the first fit, the partition is allocated which is first sufficient block from the top of Main Memory. Let's look at the example below:

      Suppose in this setup a process P4 of size 5 MB makes a request. In First Fit, the linked list is traversed from the top of the memory and wherever, a size of 5 MB or more is available, the process gets allocated. In this case, the P4 gets allocated in the 10 MB hole leaving another hole of 5 MB. Now if another request of 3 MB is requested then it gets stored in this 5 MB hole. Again if a 10 MB request arrives, then the linked list is traversed from the top and since only the 12 MB hole can accommodate the request, it gets allocated here.

2. **Best Fit:** Allocate the process to the partition which is the first smallest sufficient partition among the free available partition. Let's look at the example below:

   Suppose in this setup a process P4 of size 3 MB makes a request. In Best Fit, the linked list is traversed from the top of the memory and instead of assigning the process to wherever the sufficient memory is available, it keeps a track of the minimum block which can accommodate the process. In this case the P4 with 3 MB is assigned to the 4 MB block even if 10 MB block is first encountered. It has a few disadvantages:

   - Every time a process makes a request, the whole linked list is needed to be traversed, thus this approach has greater time complexity.
   - Various small memory holes are created which remains unused as this could not be assigned to any processes.
3. **Next Fit:** Next fit is similar to the first fit but it will search for the first sufficient partition from the last allocation point. Let's look at the example below:

   Suppose in this setup a process P4 of size 8 MB and a process P5 of size 3 MB makes a request. At first, the linked list is traversed from the top of the memory and wherever, a size of 8 MB or more is available, the process gets allocated. In this case, the P4 gets allocated in the 10 MB hole leaving another hole of 2 MB. When the next request P5 of 3 MB arrives then, instead of traversing the whole list from the top, it is traversed from the last stop i.e., from P4 and process P5 gets allocated into the 4 MB block.

4. **Worst Fit:** Allocate the process to the partition which is the largest sufficient among the freely available partitions available in the main memory.

After doing some analysis, it has been found out that the First fit is the best approach since the best-fit algorithm needs to traverse the whole list every time a request is made.

Paging in Operating System

Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory. This scheme permits the physical address space of a process to be non - contiguous. Let's revise various types of address and spaces:

- **Logical Address or Virtual Address (represented in bits):** An address generated by the CPU
- **Logical Address Space or Virtual Address Space(represented in words or bytes):** The set of all logical addresses generated by a program
- **Physical Address (represented in bits):** An address actually available on memory unit
- **Physical Address Space (represented in words or bytes):** The set of all physical addresses corresponding to the logical addresses

Example:

- If Logical Address = 31 bit, then Logical Address Space = 2^{31} words = 2 G words (1 G = 2^{30})
- If Logical Address Space = 128 M words = $2^{7} * 2^{20}$ words, then Logical Address = $\log_2 2^{27}$ = 27 bits
- If Physical Address = 22 bit, then Physical Address Space = 2^{22} words = 4 M words (1 M = 2^{20})
- If Physical Address Space = 16 M words = $2^{4} * 2^{20}$ words, then Physical Address = $\log_2 2^{24}$ = 24 bits

The mapping from virtual to physical address is done by the memory management unit (MMU) which is a hardware device and this mapping is known as paging technique.

- **Frames:** The Physical Address Space is conceptually divided into a number of fixed-size blocks.
- **Pages:** The Logical address Space is also split into fixed-size blocks, called .
  Note: Page Size is always equal to Frame Size

Let us consider an example:

- Physical Address = 12 bits, then Physical Address Space = 4 K words
- Logical Address = 13 bits, then Logical Address Space = 8 K words
- Page size = frame size = 1 K words (assumption)

Let's see how Paging helps in Memory Management through this diagram:

We can see that when P1 completes and goes out of the memory the remaining 4 blocks of memory is taken by P3 in a non-contiguous manner. No more memory is wasted and the problem of fragmentation is almost eliminated.

Runtime binding is used in Paging to facilitate swapping during run time. To do this logical address is needed.
Since the logical addresses are contiguous and we might need a non-contiguous address in paging, these logical addresses are converted to physical addresses. This conversion is quite complex during paging since non-contiguous memory address is available. This brings us to the concept of Page Table, which stores the mapping of logical address spaces to memory frames. So when a process is loaded into the memory, a page table is created for the same. To facilitate the task, the hardware provides us with two registers- Page Table Base Register and Page Table Length Register. These registers help during context switching. The former is used to store the starting address of the created Page Table when context switching occurs and the later is used for security purposes. For every frame in the main memory, we have a corresponding page address in the page table.

This diagram shows the control flow diagram of Paging. As we can see that the logical address stores the page number as 2 and page offset as 3. 2 signifies the page number in the page table. From the page table, we get the frame number to the main memory. Here it is 5 and from the logical address, we get the offset value within frame number. There can be many offsets within a frame. This is how we can access a physical address located inside main memory using CPU generated logical addresses.

There might arise a situation when the address in a page table points to an invalid location of main memory. This is called the Page Fault. A page fault occurs when a program attempts to access data or code that is in its address space but is not currently located in the system RAM. Page table entry has the following information.

1. **Frame Number -** It gives the frame number in which the current page you are looking for is present. The number of bits required depends on the number of frames.Frame bit is also known as address translation bit.

```
Number of bits for frame = Size of physical memory/frame size
```

2. **Present/Absent bit -** Present or absent bit says whether a particular page you are looking for is present or absent. In case if it is not present, that is called Page Fault. It is set to 0 if the corresponding page is not in memory. Used to control page fault by the operating system to support virtual memory. Sometimes this bit is also known as **valid/invalid** bits.

3. **Protection bit -** Protection bit says that what kind of protection you want on that page. So, these bit for the protection of the page frame (read, write etc).

4. **Referenced bit -** Referenced bit will say whether this page has been referred in the last clock cycle or not. It is set to 1 by hardware when the page is accessed.

5. **Caching enabled/disabled -** Some times we need the fresh data. Let us say the user is typing some information from the keyboard and your program should run according to the input given by the user. In that case, the information will come into the main memory. Therefore main memory contains the latest information which is typed by the user. Now if you try to put that page in the cache, that cache will show the old information. So whenever freshness is required, we don't want to go for caching or many levels of the memory.The information present in the closest level to the CPU and the information present in the closest level to the user might be different. So we want the information has to be consistency, which means whatever information user has given, CPU should be able to see it as first as possible. That is the reason we want to disable caching. So, this bit **enables or disable** caching of the page.

6. **Modified bit -** Modified bit says whether the page has been modified or not. Modified means sometimes you might try to write something on to the page. If a page is modified, then whenever you should replace that page with some other page, then the modified information should be kept on the hard disk or it has to be written back or it has to be saved back. It is set to 1 by hardware on write-access to page which is used to avoid writing when swapped out. Sometimes this modified bit is also called as the **Dirty bit**.

Memory Management – Fragmentation

In operating systems, Memory Management is the function responsible for allocating and managing a computer's main memory. Memory Management function keeps track of the status of each memory location, either allocated or free to ensure effective and efficient use of Primary Memory.

There are two Memory Management Techniques: **Contiguous**, and **Non-Contiguous**. In Contiguous Technique, executing process must be loaded entirely in main-memory. Contiguous Technique can be divided into:

1. Fixed (or static) partitioning
2. Variable (or dynamic) partitioning

Fixed Partitioning: This is the oldest and simplest technique used to put more than one process in the main memory. In this partitioning, number of partitions (non-overlapping) in RAM are **fixed but size** of each partition may or **may not be same**. As it is **contiguous** allocation, hence no spanning is allowed. Here partitions are made before execution or during system configure.

As illustrated in the above figure, the first process is only consuming 1MB out of 4MB in the main memory.
Hence, Internal Fragmentation in first block is (4-1) = 3MB.
Sum of Internal Fragmentation in every block = (4-1)+(8-7)+(8-7)+(16-14)= 3+1+1+2 = 7MB.

Suppose process P5 of size 7MB comes. But this process cannot be accommodated in spite of available free space because of contiguous allocation (as spanning is not allowed). Hence, 7MB becomes part of External Fragmentation.

There are some advantages and disadvantages of fixed partitioning.

Advantages of Fixed Partitioning -

1. **Easy to implement:** Algorithms needed to implement Fixed Partitioning are easy to implement. It simply requires putting a process into certain partition without focussing on the emergence of Internal and External Fragmentation.

2. **Little OS overhead:** Processing of Fixed Partitioning require lesser excess and indirect computational power.

Disadvantages of Fixed Partitioning -

1. **Internal Fragmentation:** Main memory use is inefficient. Any program, no matter how small, occupies an entire partition. This can cause internal fragmentation.

2. **External Fragmentation:** The total unused space (as stated above) of various partitions cannot be used to load the processes even though there is space available but not in the continuous form (as spanning is not allowed).

3. **Limit process size:** The process of size greater than the size of the partition in Main Memory cannot be accommodated. The partition size cannot be varied according to the size of the incoming process's size. Hence, the process size of 32MB in the above-stated example is invalid.

4. **Limitation on Degree of Multiprogramming:** Partition in Main Memory is made before execution or during system configure. Main Memory is divided into a fixed number of the partition. Suppose if there are $n1$n1 partitions in RAM and $n2$n2 are the number of processes, then $n2 \leq n1$n2≤n1 condition must be fulfilled. Number of processes greater than the number of partitions in RAM is invalid in Fixed Partitioning.

Fragmentation occurs in a dynamic memory allocation system when many of the free blocks are too small to satisfy any request.
External Fragmentation: External Fragmentation happens when a dynamic memory allocation algorithm allocates some memory and a small piece is left over that cannot be effectively used. If too much external fragmentation occurs, the amount of usable memory is drastically reduced. Total memory space exists to satisfy a request, but it is not contiguous.

Internal Fragmentation: Internal fragmentation is the space wasted inside of allocated memory blocks because of restriction on the allowed sizes of allocated blocks. Allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.

Virtual Memory Management - Description of virtual memory

Virtual Memory is a storage allocation scheme in which secondary memory can be addressed as though it were part of the 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 the 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 a number of pieces and these pieces need not be continuously located in the main memory during execution. The combination of dynamic run-time address 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.

In main memory management it was discussed how to avoid memory fragmentation by breaking process memory requirements down into smaller bites (pages), and storing the pages non-contiguously in memory. But the entire process still had to be stored in memory somewhere.

- In practice, most real processes do not need all their pages, or at least not all at once, for several reasons:
    1. There is no need of Error handling code unless that specific error occurs, some of which are quite rare.
    2. Arrays are often over-sized for worst-case scenarios, and only a small fraction of the arrays are actually used in practice.
    3. Certain features of certain programs are rarely used, such as the routine to balance the federal budget.
- There are some benefits of the ability to load only the portions of processes that were actually needed (and only when they were needed):
    - Programs could be written for much larger address space (virtual memory space) than physically exists on the computer.
    - Since every process is only using a fraction of their total address space that's why there is more memory left for other programs this will improvise CPU utilization and system throughput.
    - For swapping processes less I/O is needed in and out of RAM this will speed things up.

Let's try to understand the impact of Page Fault:
Suppose we have a 99% hit ratio of the RAM and the Ram access time is 10 ns. During page fault time taken is 5000000 ns. So let's find out the average access time.
The average access time for the above process is = 0.99 x 10 ns + 0.01 x 5000000 ns = 50,009.9 ns which is approximately equal to 50 micro-second.
So, we see how a 1% page fault can increase the ram access time from 10 ns to 50 micro-second.

Figure 1 given below shows the general layout of virtual memory, which can be much larger than the physical memory:

- Figure 2 given below shows virtual address space, which is the programmer's logical view of process memory storage. The actual physical layout is controlled by the process's page table.
- Note that the address space shown in Figure 9.2 is sparse - A great hole in the middle of the address space is never used unless the stack and/or the heap grow to fill the hole.

- By multiple processes, virtual memory also allows the sharing of files and memory, with some benefits:

    - If System libraries are mapped with virtual address space of more than one process these libraries can be shared.
    - Virtual memory can also be shared with processes if it is mapped with the same block of memory to more than one process.
    - During a fork( ) system call process pages can also be shared, eliminating the need to copy all of the pages of the original (parent)process.

Memory Management – TLB

In Operating System (Memory Management Technique: Paging), for each process page table will be created, which will contain Page Table Entry (PTE). This PTE will contain information like frame number (The address of main memory where we want to refer), and some other useful bits (e.g., valid/invalid bit, dirty bit, protection bit, etc). This page table entry (PTE) will tell where in the main memory the actual page is residing.

Now the question is where to place the page table, such that overall access time (or reference time) will be less.

The problem initially was to fast access the main memory content based on an address generated by CPU (i.e logical/virtual address). Initially, some people thought of using registers to store page table, as they are high-speed memory so access time will be less.

The idea used here is, place the page table entries in registers, for each request generated from CPU (virtual address), it will be matched to the appropriate page number of the page table, which will now tell where in the main memory that corresponding page resides. Everything seems right here, but the problem lies in the small register size (in practical, it can accommodate maximum of 0.5k to 1k page table entries) and process size may be big hence the required page table will also be big (lets say this page table contains 1M entries), so registers may not hold all the PTE's of Page table. So this is not a practical approach.

To overcome this size issue, the entire page table was kept in the main memory. but the problem here is two main memory references are required:

```
1. To find the frame number.

2.
3. To go to the address specified by frame number.

4.
```

To overcome this problem a high-speed cache is set up for page table entries called a Translation Lookaside Buffer (TLB). Translation Lookaside Buffer (TLB) is nothing but a special cache used to keep track of recently used transactions. TLB contains page table entries that have been most recently used. Given a virtual address, the processor examines the TLB if a page table entry is present (TLB hit), the frame number is retrieved and the real address is formed. If a page table entry is not found in the TLB (TLB miss), the page number is used to index the process page table. TLB first checks if the page is already in main memory, if not in main memory a page fault is issued then the TLB is updated to include the new page entry.

Steps in TLB hit:

1. CPU generates virtual address.
2. It is checked in TLB (present).
3. Corresponding frame number is retrieved, which now tells where in the main memory page lies.

Steps in Page miss:

1. CPU generates virtual address.
2. It is checked in TLB (not present).
3. Now the page number is matched to page table residing in main memory (assuming page table contains all PTE).
4. Corresponding frame number is retrieved, which now tells where in the main memory page lies.
5. The TLB is updated with new PTE (if space is not there, one of the replacement technique comes into picture i.e either FIFO, LRU or MFU etc).

Effective memory acess time(EMAT) : TLB is used to reduce effective memory access time as it is a high speed associative cache.
EMAT = h*(c+m) + (1-h)*(c+2m)
where, h = hit ratio of TLB
m = Memory access time
c = TLB access time
Virtual Memory Management - Demand Paging
When a process is swapped in, all its pages are not swapped in at once. Rather they are swapped in only when the process demands them or needs them this is the basic idea behind demand paging. This is also termed as a lazy swapper.

Basic Concept:

- The basic idea behind paging is that when a process is swapped in, the pager only loads into memory those pages that it expects the process to need (right away.)
- Those pages which are not loaded into memory are marked as invalid pages in the page table using the invalid bit. The remaining page table entry may either be blank or have some information about the swapped-out page on the hard drive.
- If the process only ever access pages that are loaded in memory (memory resident pages), then the process runs exactly as if all the pages were loaded into memory.

On the other hand, if a page is needed that was not originally loaded up, then a page fault trap is generated, which must be handled in a series of steps:

1. The memory address requested is first checked, to make sure it was a valid memory request.
2. A process is terminated if the reference was invalid. Else, the page must be paged in.
3. A free frame is located from a free-frame list.
4. To bring the necessary page from a disk operation is scheduled. It will allow some other process to use the CPU and block the process on an I/O wait.
5. The process's page table is updated when the I/O operation is completed, with the new frame number, and the invalid bit is changed its status to valid indicating that this is now a valid page reference.
6. The instruction will be restarted that caused the page fault.(as soon as this process gets another turn on the CPU.)

- NO pages are swapped in for a process until they are requested by page faults in the extreme case, This is termed as pure demand paging.
- Theoratically each instruction could generate multiple page faults. In practice, this is very rare, due to the locality of reference.
- A page table and secondary is necessary to support virtual memory and it is the same as for paging and swapping.
- Once the demanded page is available in memory, the instruction must be restarted from scratch(a crucial part of the demand paging). For the basic instructions, it is not a difficulty. But in some architectures, a large block of data is modified by a single instruction. And if some of the data get modified before the page fault occurs, this could cause some problems. One solution is to access both ends of the block before executing the instruction so that the necessary pages already paged before the instruction begins.

Performance of Demand Paging

- Obviously there is some slowdown and performance hit whenever a page fault occurs and the system has to go get it from memory, but just how big a hit is it exactly?
- A number of steps take place while servicing a page fault (see the book for full details), and some of the steps are optional or variable. But just for the sake of discussion, suppose that normal memory access requires 300 nanoseconds and that servicing a page fault takes 8 milliseconds. (9,000,000 nanoseconds, or 30,000 times a normal memory access.) Where p is the page fault rate, (ranges from 0 to 1), the effective access time is now:

- `= ( 1 - p ) * ( 300 ) + p * 9000000`

`= 300 + 8,999,700 * p`

- which clearly depends heavily on p! Even if only one access in 1000 causes a page fault, the effective access time drops from 200 nanoseconds to 8.2 microseconds, a slowdown of a factor of 40 times. In order to keep the slowdown less than 10%, the page fault rate must be less than 0.0000025, or one in 399,990 accesses.

- A subtlety is that swap space is faster to access than the regular file system because it does not have to go through the whole directory structure. For this reason, some systems will transfer an entire process from the file system to swap space before starting up the process so that future paging all occurs from the (relatively) faster swap space.
- Some systems use demand paging directly from the file system for binary code (which never changes and hence does not have to be stored on a page operation), and to reserve the swap space for data segments that must be stored. Both Solaris and BSD Unix use this approach.

Virtual Memory Management - Thrashing

Thrashing is a condition or a situation when the system is spending a major portion of its time in servicing the page faults, but the actual processing done is very negligible.

The basic concept involved is that if a process is allocated too few frames, then there will be too many and too frequent page faults. As a result, no useful work would be done by the CPU and the CPU utilization would fall drastically. The long-term scheduler would then try to improve the CPU utilization by loading some more processes into the memory thereby increasing the degree of multiprogramming. This would result in a further decrease in the CPU utilization triggering a chained reaction of higher page faults followed by an increase in the degree of multiprogramming, called Thrashing.

Locality Model - A locality is a set of pages that are actively used together. The locality model states that as a process executes, it moves from one locality to another. A program is generally composed of several different localities that may overlap.

For example, when a function is called, it defines a new locality where memory references are made to the instructions of the function call, it's local and global variables, etc. Similarly, when the function is exited, the process leaves this locality.

Techniques to handle:

1. **Working Set Model -**
   This model is based on the above-stated concept of the Locality Model.
   The basic principle states that if we allocate enough frames to a process to accommodate its current locality, it will only fault whenever it moves to some new locality. But if the allocated frames are lesser than the size of the current locality, the process is bound to thrash.

   According to this model, based on a parameter A, the working set is defined as the set of pages in the most recent 'A' page references. Hence, all the actively used pages would always end up being a part of the working set.

   The accuracy of the working set is dependant on the value of parameter A. If A is too large, then working sets may overlap. On the other hand, for smaller values of A, the locality might not be covered entirely.

   If D is the total demand for frames and WSS_iWSSi is the working set size for a process I,

   $D=\sum nolimits\{WSS_i\}$D=∑nolimits{WSSi}

   Now, if 'm' is the number of frames available in the memory, there are 2 possibilities:

   - (i) D>m i.e. total demand exceeds the number of frames, then thrashing will occur as some processes would not get enough frames.
   - (ii) D<=m, then there would be no thrashing.

   If there are enough extra frames, then some more processes can be loaded in memory. On the other hand, if the summation of working set sizes exceeds the availability of frames, then some of the processes have to be suspended(swapped out of memory).

   This technique prevents thrashing along with ensuring the highest degree of multiprogramming possible. Thus, it optimizes CPU utilization.

2. **Page Fault Frequency -**
   A more direct approach to handle thrashing is the one that uses the Page-Fault Frequency concept.

   The problem associated with Thrashing is the high page fault rate and thus, the concept here is to control the page fault rate.

   If the page fault rate is too high, it indicates that the process has too few frames allocated to it. On the contrary, a low page fault rate indicates that the process has too many frames.

   Upper and lower limits can be established on the desired page fault rate as shown in the diagram.
   If the page fault rate falls below the lower limit, frames can be removed from the process. Similarly, if the page fault rate exceeds the upper limit, the number of frames can be allocated to the process.
   In other words, the graphical state of the system should be kept limited to the rectangular region formed in the given diagram.

   Here too, if the page fault rate is high with no free frames, then some of the processes can be suspended and frames allocated to them can be reallocated to other processes. The suspended processes can then be restarted later.

   Causes of Thrashing :

   1. **High degree of multiprogramming** : If the number of processes keeps on increasing in the memory then the number of frames allocated to each process will be decreased. So, less number of frames will be available to each process. Due to this, a 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.
      For example:
      Let free frames = 400
      Case 1: Number of process = 100
      Then, each process will get 4 frames.

      Case 2: Number of process = 400
      Each process will get 1 frame.
      Case 2 is a condition of thrashing, as the number of processes is increased, frames per process are decreased. Hence CPU time will be consumed in just swapping pages.
   2. **Lacks Frames**: If a process has less number of frames then fewer 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 midterm scheduler to suspend some of the processes so that we can recover the system from thrashing.

Virtual Memory Management - Page Replacement
In an operating system that uses paging for memory management, a page replacement algorithm is needed to decide which page needs to be replaced when a new page comes in.

Page Fault - A page fault happens when a running program accesses a memory page that is mapped into the virtual address space but not loaded in physical memory.

Since actual physical memory is much smaller than virtual memory, page faults happen. In case of a page fault, the Operating System might have to replace one of the existing pages with the newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce the number of page faults.

Page Replacement Algorithms :

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

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

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

  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 would not be used for the longest duration of time in the future.

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

  Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots → **4 Page faults** 0 is already there so → **0 Page fault.** when 3 came it will take the place of 7 because it is not used for the longest duration of time in the future. →**1 Page fault.** 0 is already there so → **0 Page fault.**.
  4 will takes place of 1 → **1 Page Fault.**
  Now for the further page reference string → **0 Page fault** because they are already available in the memory.

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

- **Least Recently Used -** In this algorithm page will be replaced which is least recently used.

  Example-3 Consider the page reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2 with 4 page frames.Find number of page faults.

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

Note: You can also refer LFU and Second chance page replacement policy.

Memory Management - Segmentation

A process is divided into Segments. The chunks that a program is divided into which are not necessarily all of the same sizes are called segments. Segmentation gives the user's view of the process that paging does not give. Here the user's view is mapped to physical memory.
There are types of segmentation:

1. **Virtual memory segmentation -** Each process is divided into a number of segments, not all of which are resident at any one point in time.
2. **Simple segmentation -** Each process is divided into a number of segments, all of which are loaded into memory at run time, though not necessarily contiguously.

There is no simple relationship between logical addresses and physical addresses in segmentation. A table stores the information about all such segments and is called Segment Table.

Segment Table - It maps two-dimensional Logical address into one-dimensional Physical address. It's each table entry has:

- **Base Address:** It contains the starting physical address where the segments reside in memory.
- **Limit:** It specifies the length of the segment.

Translation of Two-dimensional Logical Address to one-dimensional Physical Address.

The address generated by the CPU is divided into:

- **Segment number (s):** Number of bits required to represent the segment.

- **Segment offset (d):** Number of bits required to represent the size of the segment.

Advantages of Segmentation -

- No Internal fragmentation.
- Segment Table consumes less space in comparison to Page table in paging.

Disadvantage of Segmentation -

- As processes are loaded and removed from the memory, the free memory space is broken into little pieces, causing External fragmentation.

Memory Management - Swapping

A computer has a sufficient amount of physical memory but most of the time we need more so we swap some memory on disk. Swap space is a space on the hard disk which is a substitute of physical memory. It is used as a virtual memory that contains the process memory image. Whenever our computer runs short of physical memory it uses it's virtual memory and stores information in memory on disk. Swap space helps the computer's operating system in pretending that it has more RAM than it actually has. It is also called a swap file. This interchange of data between virtual memory and real memory is called as swapping and space on disk as "swap space".

Virtual memory is a combination of RAM and disk space that running processes can use. Swap space is the portion of virtual memory that is on the hard disk, used when RAM is full.

Swap space can be useful to the computer in various ways:

1. It can be used as a single contiguous memory which reduces i/o operations to read or write a file.
2. Applications that are not used or are used less can be kept in the swap file.
3. Having sufficient swap file helps the system keep some physical memory free all the time.
4. The space in physical memory which has been freed due to swap space can be used by OS for some other important tasks.

In operating systems such as Windows, Linux, etc systems provide a certain amount of swap space by default which can be changed by users according to their needs. If you don't want to use virtual memory you can easily disable it all together but in case if you run out of memory then the kernel will kill some of the processes in order to create a sufficient amount of space in physical memory. So it totally depends upon the user whether he wants to use swap space or not.

What is standard swapping?

- If compile-time or load-time address binding is used, the processes must be swapped back into the same memory location from which they were swapped out. The processes can be swapped back into any available location if execution time binding is used.
- Comparing with other processes in a system swapping is a very slow process. Swapping involves moving old data out as well as new data in, for efficient processor scheduling, the CPU time slice should be significantly longer than this lost transfer time.
- Swapping transfer overhead can be reduced by applying limitation over information to be transferred, which requires that the system know how much memory a process is using, as opposed to how much it might use. If programmers freeing up dynamic memory that is no longer in use, it will be helpful.

- When processes are idle(No I/O operation are pending), it can be swapped out. There are two options either swap only totally idle processes or perform pending I/O operations only into and out of OS buffers, which are then transferred to or from the process's main memory as a second step.
- Most modern OSes no longer use swapping, because it is too slow and there are faster alternatives available. (e.g. Paging.) But if the system gets extremely full, some UNIX systems will still invoke swapping, and then discontinue swapping when the load reduces again.

Figure:
How swapping occurs on the mobile system?

- There are several reasons why swapping is not supported on mobile platforms:
    - On Mobile devices there is typically flash memory in place of more spacious hard drives, so there is not as much space available.
    - Flash memory has a limit to be written to a limited number of times before it becomes unreliable.
    - There is low bandwidth to flash memory.
- Apple's IOS asks applications to voluntarily free up memory
    - Read-only data.
    - Modified data.
    - OS must uninstall the apps which are failed to free up memeory space
- Android follows a similar strategy.
    - In android, it write application state to flash memory for quick restarting, instead to terminate a process

Virtual Memory Management - Copy-on -write

- The idea behind a copy-on-write fork is that the pages for a parent process do not have to be actually copied for the child until one or the other of the processes changes the page. They can be simply shared between the two processes in the meantime, with a bit set that the page needs to be copied if it ever gets written to. This is a reasonable approach since the child process usually issues an exec( ) system call immediately after the fork.

- Pages that can be modified also need to be labeled as copy-on-write. Code segments can simply be shared.
- Pages used to satisfy copy-on-write duplications are typically allocated using zero-fill-on-demand, meaning that their previous contents are zeroed out before the copy proceeds.
- An alternative to the fork() system call is provided by some systems is called a virtual memory fork, vfork( ). In this case, the parent is suspended, and the child uses the parent's memory pages. This is very fast for process creation but requires that the child not modify any of the shared memory pages before performing the exec( ) system call. (In essence, this addresses the question of which process executes first after a call to fork, the parent or the child. With vfork, the parent is suspended, allowing the child to execute first until it calls exec( ), sharing pages with the parent in the meantime.

Multi-threaded programming - Threads and its types

Thread is a single sequence stream within a process. Threads have same properties as of the process so they are called as light weight processes. Threads are executed one after another but gives the illusion as if they are executing in parallel. Each thread has different states. Each thread has

1. A program counter

2. A register set
3. A stack space

Threads are not independent of each other as they share the code, data, OS resources, etc.

Similarity between Threads and Processes -

- Only one thread or process is active at a time
- Within process both execute sequentiall
- Both can create children

Differences between Threads and Processes -

- Threads are not independent, processes are.
- Threads are designed to assist each other, processes may or may not do it

Types of Threads:

1. **User Level thread (ULT) -** It is implemented in the user level library, they are not created using the system calls. Thread switching does not need to call OS and to cause an interrupt to Kernel. The kernel doesn't know about the user level thread and manages them as if they were single-threaded processes.

   Advantages of ULT -
   - Can be implemented on an OS that does't support multithreading.
   - Simple representation since thread has only program counter, register set, stack space.
   - Simple to create since no intervention of kernel.
   - Thread switching is fast since no OS calls need to be made.

   Disadvantages of ULT -

   - No or less co-ordination among the threads and Kernel.
   - If one thread causes a page fault, the entire process blocks.

2. **Kernel Level Thread (KLT) -** Kernel knows and manages the threads. Instead of thread table in each process, the kernel itself has thread table (a master one) that keeps track of all the threads in the system. In addition kernel also maintains the traditional process table to keep track of the processes. OS kernel provides system call to create and manage threads.

   Advantages of KLT -
   - Since kernel has full knowledge about the threads in the system, scheduler may decide to give more time to processes having large number of threads.
   - Good for applications that frequently block.

Disadvantages of KLT -

- Slow and inefficient.
- It requires thread control block so it is an overhead.

Summary:

1. Each ULT has a process that keeps track of the thread using the Thread table.
2. Each KLT has Thread Table (TCB) as well as the Process Table (PCB).

Multithreaded programming - Multithreading.

A **thread** is a path which is followed during a program's execution. Majority of programs written nowadays run as a single thread. Let's say, for example, a program is not capable of reading keystrokes while making drawings. These tasks cannot be executed by the program at the same time. This problem can be solved through multitasking so that two or more tasks can be executed simultaneously.

Multitasking is of two types: Processor based and thread based. Processor-based multitasking is totally managed by the OS, however multitasking through multithreading can be controlled by the programmer to some extent.

The concept of **multi-threading** needs proper understanding of these two terms - **a process and a thread**. A process is a program being executed. A process can be further divided into independent units known as threads.

A thread is like a small light-weight process within a process. Or we can say a collection of threads is what is known as a process.

Applications -

- Threading is used widely in almost every field. Most widely it is seen over the internet nowadays where we are using transaction processing of every type like recharges, online transfer, banking, etc. Threading is a segment that divides the code into small parts that are of very lightweight and has less burden on CPU memory so that it can be easily worked out and can achieve the goal in the desired field.
- The concept of threading is designed due to the problem of fast and regular changes in technology and less work in different areas due to less application. Then as says "need is the generation of creation or innovation" hence by following this approach human mind develop the concept of thread to enhance the capability of programming.

Many operating systems support kernel thread and user thread in a combined way. An example of such a system is Solaris. The multithreading model is of three types.

```
Many to many models.

Many to one model.

one to one model.
```

Many to Many Model
In this model, we have multiple user threads multiplex to the same or lesser number of kernel-level threads. The number of kernel-level threads is specific to the machine, advantage of this model is if a user thread is blocked we can schedule others user thread to other kernel thread. Thus, the System doesn't block if a particular thread is blocked.

Many to One Model
In this model, we have multiple user threads mapped to one kernel thread. In this model when a user thread makes a blocking system call entire process blocks. As we have only one kernel thread and only one user thread can access kernel at a time, so multiple threads are not able to access multiprocessor at the same time.

One to One Model In this model, one to one relationship between the kernel and user thread. In this model, multiple threads can run on multiple processors. The problem with this model is that creating a user thread requires the corresponding kernel thread.

Benefits of Multithreading:

1. Resource Sharing
2. Responsiveness
3. Economy
4. Scalability

Multi-threaded programming - Thread libraries(Linux)
Three main types of libraries nowadays are used:
An API is provided by thread libraries to the programmers to create and handle the thread. It can be implemented in either kernel mode or in user mode. Former API was implemented solely within user mode, without any kernel support. The latter libraries involve system call and to do this kernel thread library support is required.
POSIX pthreads:

1. It is provided either as a kernel level of a user-level library. It extends the POSIX standards.
2. **Win32 threads:** It is a kernel level library on windows system.
3. **JAVA threads:** JAVA is not a platform independent, JAVA virtual machine (JVM) provide the atmosphere to run a JAVA file. Implementation is depending upon the Operating system and hardware on which JVM is running on; either POSIX threads or Win32 threads

- **POSIX:** The POSIX standard defines the specification of thread but implementation is not there.
- Solaris, Mac OSX, Tru 64 provide thread and for windows, it is available through public domain shareware.
- Global variables are shared among all threads.
- One thread can wait for others to rejoin before continuing.
- pThread begin execution in a specified function.
  For example:

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

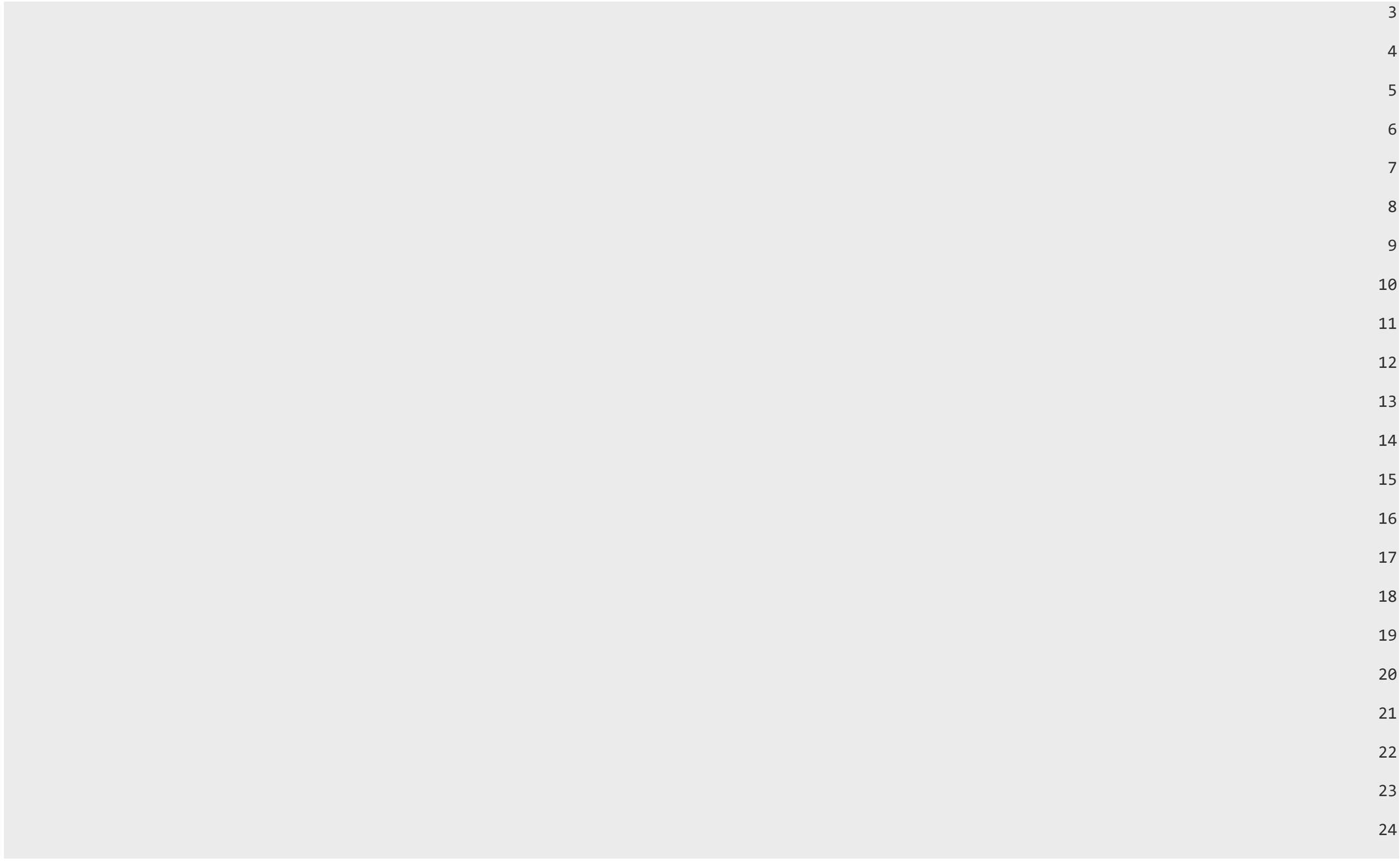
```
#include<pthread.h>
#include<stdio.h>
int add;
void *run(void *super);
int main(int argc, char *argv[])
{
pthread_p pid;
pthread_att_p att;
if (argc != 2)
{
fprintf(stderr, "usage.aout <integer value>\n");
```

```
return -1;
}
if(atoi(argv[1]) < 0)
fprintf(stderr, "%d must be >= 0\n", atoi(argv[1]));
return -1;
pthread_att_init(&att);
pthread_create(&pid, &att, super, argv[1]);
pthread_join(pid, NULL);
printf("add = %d\n" add);
void *super()
{
  int u, upper = atoi(super);
  add = 0;
  for(u = 1; u <= upper; u++)
  add += u;
```

Run

Windows thread: The only difference between POSIX and windows thread is syntactic and nomenclature.

```
1
2
```



```
25
26
27
28
29
30
31
```

```
#include<windows.h>
#include<stdio.h>
DWORD Add;
DWORD WINAPI Summation (LPVOID Super)
{
DWORD Upper = *(DWORD *)Super;
for(DWORD i = 0; i <= Upper; i++)
 add += i;
return 0;
}
int main (int argc, char *argv[])
{
DWORD ThreadId;
HANDLE ThreadHandle;
int Super;
```

```
if(argc != 2)

{

fprintf(stderr,"An integer parameter is required\n");

return -1;

}

Super = atoi(argv[1]);

if(Super < 0)

fprintf(stderr,"An integer >= 0 is required\n");

return -1;

}

ThreadHandle = CreateThread(Null, 0, addition, &Super, 0, &ThreadId);

if(ThreadHandle != NULL)
```

Run

JAVA Thread:

- Threads are used in all JAVA programs, even in common single-threaded programs.
- The creation of new Threads requires Objects that implement the Runnable Interface, which means they contain a method "public void run( )". Any descendant of the Thread class will naturally contain such a method. (In practice the run( ) method must be overridden/provided for the thread to have any practical functionality.)
- By Creating a Thread Object does not mean that it starts the thread running - To do that the program must call the Thread's "start( )" method. Start( ) allocates and initializes memory for the Thread, and then calls the run( ) method. (No Programmers call run( ) directly.)
- It is well known that Java does not support global variables, A reference must be passed by threads to a Shared Object in order to share data, in this example the "Add" Object.
- It is noted that the JVM runs on top of a native OS and that the JVM specification does not specify what model to use for mapping Java threads to kernel threads. This decision depends on JVM implementation and may be one-to-one, many-to-many, or many to one.. (On a UNIX system the JVM normally uses PThreads and on a Windows system it normally uses windows threads.)

```
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
```

```
Class Add
{
private int Add;
public int get add()
{
return addition;
}

public void setadd(int add)
{
this.add = add;
}}
Class Addition implements Runnable()
{
```

```
private int add;

private Add addValue;

public addition(int upper  add addValue)

{

int add = 0;

for(int i = o; i <= upper; i++)

add += i;

addvalue.setAdd(add);

}

}

public class Driver

{

public static void main(String args[])

{
```

Run

Multi-threaded programming - Thread scheduling

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

Approaches to Multiple-Processor Scheduling -

One approach is when all the scheduling decisions and I/O processing are handled by a single processor which is called the **Master Server** and the other processors execute only the **user code**. This is simple and reduces the need for data sharing. This entire scenario is called **Asymmetric Multiprocessing**.

A second approach uses **Symmetric Multiprocessing** where each processor is **self scheduling**. All processes may be in a common ready queue or each processor may have its own private queue for ready processes. The scheduling proceeds further by having the scheduler for each processor examine the ready queue and select a process to execute.

Processor Affinity -

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

There are two types of processor affinity:

1. **Soft Affinity -** When an operating system has a policy of attempting to keep a process running on the same processor but not guaranteeing it will do so, this situation is called soft affinity.
2. **Hard Affinity -** Some systems such as Linux also provide some system calls that support Hard Affinity which allows a process to migrate between processors.

The figure below is showing the architecture featuring non-uniform memory access which is depicting faster access to some parts than to other parts. This is the scenario of the system containing combined CPU and memory boards

Load Balancing -

Load Balancing is the **phenomena** which keeps the **workload** evenly **distributed** across all processors in an SMP system. Load balancing is necessary only on systems where each processor has its own private queue of the process which is eligible to execute. Load balancing is unnecessary because once a processor becomes idle it immediately extracts a runnable process from the common run queue. On SMP(symmetric multiprocessing), it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor else one or more processor will sit idle while other processors have high workloads along with lists of processors awaiting the CPU.

There are two general approaches to load balancing:

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

Multicore Processors -

In multicore processors **multiple processors** cores are placed on the same physical chip. Each core has a register set to maintain its architectural state and thus appears to the operating system as a separate physical processor. **SMP systems** that use multicore processors are faster and consume **less power** than systems in which each processor has its own physical chip.

However multicore processors may **complicate** the scheduling problems. When a processor accesses memory then it spends a significant amount of time waiting for the data to become available. This

situation is called **MEMORY STALL**. It occurs for various reasons such as cache miss, which is accessing the data that is not in the cache memory. In such cases, the processor can spend up to fifty percent of its time waiting for data to become available from the memory. To solve this problem recent hardware designs have implemented multithreaded processor cores in which two or more hardware threads are assigned to each core. Therefore if one thread stalls while waiting for the memory, the core can switch to another thread.

The figure below is depicting the memory stall:

The figure below is depicting the multithread multicore system:

There are two ways to multithread a processor:

1. **Coarse-Grained Multithreading -** In coarse-grained multithreading a thread executes on a processor until a long latency event such as a memory stall occurs, because of the delay caused by the long latency event, the processor must switch to another thread to begin execution. The cost of switching between threads is high as the instruction pipeline must be terminated before the other thread can begin execution on the processor core. Once this new thread begins execution it begins filling the pipeline with its instructions.

2. **Fine-Grained Multithreading -** This multithreading switches between threads at a much finer level mainly at the boundary of an instruction cycle. The architectural design of fine-grained systems include logic for thread switching and as a result, the cost of switching between threads is small.

Virtualization and Threading -

In this type of **multiple-processor** scheduling, even a single CPU system acts like a multiple-processor system. In a system with Virtualization, the virtualization presents one or more virtual CPU's to each of the virtual machines running on the system and then schedules the use of physical CPU'S among the virtual machines. Most virtualized environments have one host operating system and many guest operating systems. The host operating system creates and manages the virtual machines and each virtual machine has a guest operating system installed and applications running within that guest. Each guest operating system may be assigned for specific use cases, applications, and users, including time-sharing or even real-time operation. Any guest operating-system scheduling algorithm that assumes a certain amount of progress in a given amount of time will be negatively impacted by the virtualization. In a time-sharing operating system that tries to allow 100 milliseconds to each time slice to give users a reasonable response time. A given 100 millisecond time slice may take much more than 100 milliseconds of virtual CPU time. Depending on how busy the system is, the time slice may take a second or more which results in very poor response time for users logged into that virtual machine. The net effect of such scheduling layering is that individual virtualized operating systems receive only a portion of the available CPU cycles, even though they believe they are receiving all cycles and that they are scheduling all of those cycles. Commonly, the time-of-day clocks in virtual machines are incorrect because timers take no longer to trigger than they would on dedicated CPU's.

Virtualizations can thus undo the good scheduling-algorithm efforts of the operating systems within virtual machines.

Difference between multithreading and multi-tasking:

1. The main difference between multitasking and multithreading is that in multitasking the system allows executing multiple programs and tasks at the same time, whereas, in multithreading, the system executes multiple threads of the different or same processes at the same time.
2. Multi-threading is more granular than multi-tasking. In multitasking, CPU switches between multiple programs to complete their execution in real time, while in multi-threading CPU switches between multiple threads of the same program. There is more context switching cost in multiple processes than switching between multiple threads of the same program.
3. Processes are heavyweight as compared to threads. They require their own address space, which means multi-tasking is heavy compared to multithreading.
4. For each process/program multitasking allocates separate memory and resources whereas, in multithreading, threads belonging to the same process share the same memory and resources like that of the process.

System Calls

To understand **System Calls** in an UNIX-based system, one is required to know about the modes of operation of OS. There are basically 2 modes of operation namely, **Kernel** & **User** mode:
Kernel Mode The mode of operation in which critical OS procedures run. Only privileged instructions such as:

- Handling Interrupts
- Process Management
- I/O Management

are allowed to run. This is to ensure that no user application is able to tamper or cause a glitch in core-OS tasks, thus preventing any critical error which may cause the system to crash. When the OS boots, it begins execution in kernel-mode only. All user applications which are loaded afterwards are run on User Mode.

User Mode All user applications and high-level programs are run in User Mode so that they don't mess up with the critical OS procedures. However, sometimes a user application may require access to low-level utilities such as I/O peripherals, File System (Hard-drives), etc. For that, it requires the OS to switch from the User Mode of execution to Kernel Mode. The way of switching is done via System Calls:

We shall cover some basic system calls related to process management (creation, reaping etc.) as given below:

fork Fork system call is used to create a child process. It is a special function in the fact that this system call has a conditional return value:

- < 0: Unsuccessful Child creation
- 0: Value returned to the child process
- > 0: Value returned to the parent process. It is equal to the PID (Process ID) of the child process

If we count the parent as 1 single processes, then after each fork call, 2 processes are said to exist now (the original parent and the newly created child).
Execution continues from the point of fork() call in both child and parent processes. i.e. The same code is executed in both the processes. However, everything else (variables, stack, heap etc.) is duplicated.
Example:

```
1
2
3
4
5
6
7
8
9
10
```

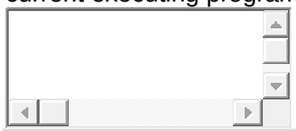
```c
#include <stdio.h>
#include <unistd.h>
int g_var;
int main()
{
    int l_var = 0;
    char *str;

    int pid = fork();
    if (pid < 0)
        printf ("Unsuccessful Child creation");
    else if (pid == 0) {    //Inside Child Process Only
        g_var++;
        l_var++;
        str = "From Child Process";
    }
    else {    //Inside Parent Process Only
        str = "From Parent Process";
    }

    printf ("%s: Global Var = %d, Local Var = %d\n", str, g_var, l_var);
    return 0;
}
```

Output:

```
From Parent Process: Global Var = 0, Local Var = 0
From Child Process: Global Var = 1, Local Var = 1
```

As can be seen, everything (global, local segment etc.) was duplicated in both child and parent. Only the variables corresponding to the child were incremented.

exec family Fork creates a new child process but has the same code executing afterwards. If we want to load a new program into the child process, we use the *exec family* calls. Exec calls replaces the current executing program with a new one. We demonstrate the usage of **execvp()** call:



```c
#include <stdio.h>
#include <unistd.h>
int main()
{
    int pid = fork();
    if (pid < 0)
        printf ("Unsuccessful Child creation");
    else if (pid == 0) {    //Inside Child Process
        printf ("Still executing Same Code\n");
        execvp("ls");
        printf ("Unreachable Code as new Program loaded already\n");
    }
```

```
    else {    //Parent Process
        printf ("Inside Parent Process\n");
    }

    return 0;
}
```

Output:

```
Inside Parent Process
Still Executing Same Code
bin   dev  home        lib    lost+found  mnt  proc  run   srv  tmp  var
boot  etc  initrd.img  lib64  media       opt  root  sbin  sys  usr  vmlinuz
```

NOTE: Since, we loaded "ls" program into our child process, we never saw "Unreachable Code ..." string getting printed.

getpid Returns the Process ID of the currently executing process (program):

```
1
2
3
4
5
6
7
8
9
10
```

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    printf ("Process ID: %d", getpid());
    return 0;
}
```

Output:

```
Process ID: 10013
```

wait and exit Wait is a blocking-call which prevents the parent process from exiting without properly reaping any child process. Exit is used to terminate the current process. It takes the exit code as an argument, which is returned to the parent. This blocking effect is demonstrated below:

```c
#include <stdio.h>
#include <unistd.h>
#include <time.h>

int main()
{
    int pid = fork();

    if (pid < 0)
        printf ("Unsuccessful Child creation");
    else if (pid == 0) {    //Inside Child Process
        sleep(5);
        exit(0);
```

```
    }
    else {    //Parent Process
        printf ("Start: %ld\n", time(NULL));
        wait(NULL);
        printf ("End: %ld\n", time(NULL));

    }

    return 0;
}
```

Output:

```
Start: 1557231700
End: 1557231705
```

As we can see there is a difference of 5 secs. (because Child process was sleeping for 5secs). The parent waited for the child to exit.