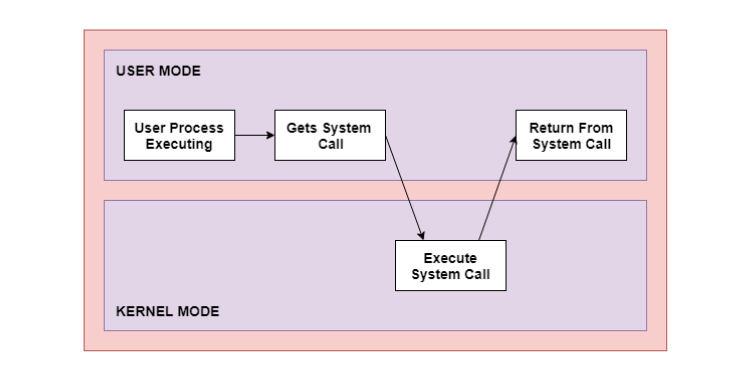
**UNIT – 3**

**Implementing Process**

**Topic 1 : System Calls**

The interface between a process and an operating system is provided by system calls. In general, system calls are available as assembly language instructions. They are also included in the manuals used by the assembly level programmers. System calls are usually made when a process in user mode requires access to a resource. Then it requests the kernel to provide the resource via a system call.

A figure representing the execution of the system call is given as follows −



As can be seen from this diagram, the processes execute normally in the user mode until a system call interrupts this. Then the system call is executed on a priority basis in the kernel mode. After the execution of the system call, the control returns to the user mode and execution of user processes can be resumed.

In general, system calls are required in the following situations −

* If a file system requires the creation or deletion of files. Reading and writing from files also require a system call.
* Creation and management of new processes.
* Network connections also require system calls. This includes sending and receiving packets.
* Access to a hardware device such as a printer, scanner etc. requires a system call.

**Types of System Calls**

There are mainly five types of system calls. These are explained in detail as follows −

1. **Process Control**

These system calls deal with processes such as process creation, process termination etc.

1. **File Management**

These system calls are responsible for file manipulation such as creating a file, reading a file, writing into a file etc.

1. **Device Management**

These system calls are responsible for device manipulation such as reading from device buffers, writing into device buffers etc.

These system calls handle information and its transfer between the operating system and the user program. These system calls are useful for interprocess communication. They also deal with creating and deleting a communication connection.

Some of the examples of all the above types of system calls in Windows and Unix are given as follows −

| **Types of System Calls** | **Windows** | **Linux** |
| --- | --- | --- |
| Process Control | CreateProcess() ExitProcess() WaitForSingleObject() | fork() exit() wait() |
| File Management | CreateFile() ReadFile() WriteFile() CloseHandle() | open() read() write() close() |
| Device Management | SetConsoleMode() ReadConsole() WriteConsole() | ioctl() read() write() |
| Information Maintenance | GetCurrentProcessID() SetTimer() Sleep() | getpid() alarm() sleep() |
| Communication | CreatePipe() CreateFileMapping() MapViewOfFile() | pipe() shmget() mmap() |

There are many different system calls as shown above. Details of some of those system calls are as follows −

**open ()**

The open () system call is used to provide access to a file in a file system. This system call allocates resources to the file and provides a handle that the process uses to refer to the file. A file can be opened by multiple processes at the same time or be restricted to one process. It all depends on the file organisation and file system.

**read ()**

The read () system call is used to access data from a file that is stored in the file system. The file to read can be identified by its file descriptor and it should be opened using open () before it can be read. In general, the read () system calls take three arguments i.e., the file descriptor, buffer which stores read data and number of bytes to be read from the file.

**write ()**

The write () system calls write the data from a user buffer into a device such as a file. This system call is one of the ways to output data from a program. In general, the write system calls take three arguments i.e., file descriptor, pointer to the buffer where data is stored and number of bytes to write from the buffer.

**close ()**

The close () system call is used to terminate access to a file system. Using this system call means that the file is no longer required by the program and so the buffers are flushed, the file metadata is updated and the file resources are de-allocated.

**Topic 2: System Call Interface**

The operating system provides a set of operations which are called system calls. All services that the operating system provides can be requested through these system calls. From the user point of view, the entire functionality of the operating system is defined by the system calls. A system call interface is the description of the set of system calls implemented by the operating system.

**An Example System Call Interface:** To explain the system call interface we are considering the system call interface of the UNIX operating system

1. **System Call Overview:** The overview of system calls used in UNIX sub system will be as follows:
2. **File and I/o System Calls**
   1. **Open** – get ready to read or write a file
   2. **Create** – create a new file and open it
   3. **Read -** read bytes from an open file
   4. **Write –** write bytes to an open file
   5. **Lseek -** change the location in the file of the next read or write
   6. **Close -** indicate that reading or writing is complete
   7. **Stat -** get information about a file
   8. **Unlink -** remove a file name from a directory
3. **Process Management System Calls**
   1. **CreateProcess -** create a new process
   2. **Exit -** terminate the process making the system call
   3. **Wait -** wait for another process to exit
   4. **Fork -** create a duplicate of the process making the system call
   5. **Execv -** run a new program in the process making the system call
4. **Interprocess Communication System calls**
   1. **CreateMessageQueue -** create a queue to hold messages
   2. **SendMessage –** send a message to a message queue
   3. **ReceiveMessage –** receive a message from message queue
   4. **DestroyMessageQueue** – destroy a message queue
5. Every system call is defined as a function when called returns an integer which is used to determine the status of a file or a process. Some of the system calls in UNIX sub system are explained below with their function structure and return values.
6. For example, The open () system call when called with parameters such as file name and flag returns a positive integer if the file is opened successfully , otherwise returns a negative integer for different errors that are occurred while opening the file.

|  |  |
| --- | --- |
| Flag value | Specification |
| 0 | Open the file for reading |
| 1 | Open the file for writing |
| 2 | Open the file for both reading and writing |

|  |  |
| --- | --- |
| Return Value | Fact |
| -1 | The file path name was invalid |
| -2 | The open flag argument is out of range |
| -3 | The file is not readable |
| -4 | The file is not writable |

1. **Hierarchical File Naming Systems:**  All operating systems need a way to name files. A basic concept in a hierarchical file naming system is a directory. A directory is a collection of names, each referring to a file or another directory. This creates a tree structure where leaf nodes are files and interior nodes are directories. There is a special directory that is the root of the tree is called the root directory. A file name represents the path of the file from the root directory to the file being named. A special character is used to separate the names in the path.

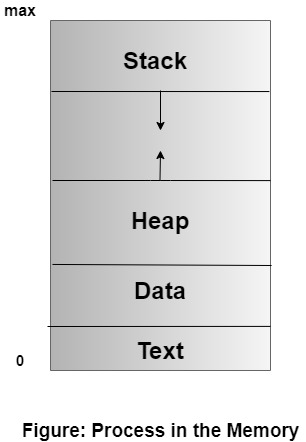
See the source image

**Topic 3: Process Architecture and Implementation**

A process is a program in execution which then forms the basis of all computation. The process is not as same as program code but a lot more than it. A process is an 'active' entity as opposed to the program which is considered to be a 'passive' entity. Attributes held by the process include hardware state, memory, CPU, etc.

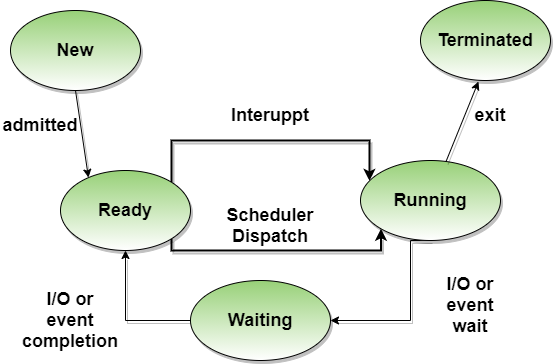
**Process memory** is divided into four sections for efficient working:

* The **Text section** is made up of the compiled program code, read in from non-volatile storage when the program is launched.
* The **Data section** is made up of the global and static variables, allocated and initialized prior to executing the main.
* The **Heap** is used for the dynamic memory allocation and is managed via calls to new, delete, malloc, free, etc.
* The **Stack** is used for local variables. Space on the stack is reserved for local variables when they are declared.



The different Process States: Processes in the operating system can be in any of the following states:

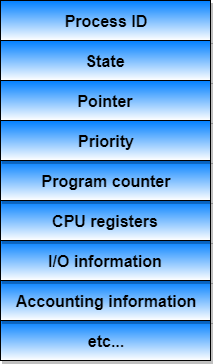
* NEW- The process is being created.
* READY- The process is waiting to be assigned to a processor.
* RUNNING- Instructions are being executed.
* WAITING- The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
* TERMINATED- The process has finished execution.



**Process Control Block**

There is a Process Control Block for each process, enclosing all the information about the process. It is also known as the task control block. It is a data structure, which contains the following:

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



**Process vs Program**

| **Process** | **Program** |
| --- | --- |
| The process is basically an instance of the computer program that is being executed. | A Program is basically a collection of instructions that mainly performs a specific task when executed by the computer. |
| A process has a **shorter lifetime**. | A Program has a **longer lifetime**. |
| A Process requires resources such as memory, CPU, Input-Output devices. | A Program is stored by hard-disk and does not require any resources. |
| A process has a dynamic instance of code and data | A Program has static code and static data. |
| Basically, a process is the **running instance** of the code. | On the other hand, the program is the **executable code**. |

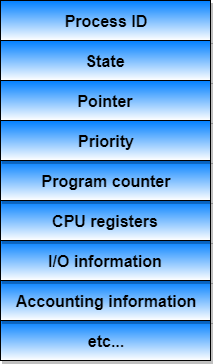
**Process Scheduling**

When there are two or more runnable processes then it is decided by the Operating system which one to run first then it is referred to as Process Scheduling.

A scheduler is used to make decisions by using some scheduling algorithm.

Given below are the properties of a **Good Scheduling Algorithm**:

* Response time should be minimum for the users.
* The number of jobs processed per hour should be maximum i.e good scheduling algorithm should give maximum throughput.
* The utilization of the CPU should be 100%.
* Each process should get a fair share of the CPU.



**Topic 4: Process Schedulers**

A new process is initially put in the Ready queue. It waits in the ready queue until it is selected for execution (or dispatched). Once the process is assigned to the CPU and is executing, one of the following several events can occur:

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

There are three types of schedulers available to determine the process state and control the execution flow of process:

1. Long Term Scheduler
2. Short Term Scheduler
3. Medium Term Scheduler

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

### Short Term Scheduler: This is also known as CPU Scheduler and runs very frequently. The primary aim of this scheduler is to enhance CPU performance and increase process execution rate.

### Medium Term Scheduler: This scheduler removes the processes from memory (and from active contention for the CPU), and thus reduces the degree of multiprogramming. At some later time, the process can be reintroduced into memory and its execution van be continued where it left off. This scheme is called swapping. The process is swapped out, and is later swapped in, by the medium-term scheduler. 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.

### Topic 5: Scheduling Algorithms

A **Scheduling Algorithm** is the algorithm which tells us how much CPU time we can allocate to the processes. These scheduling algorithms are either pre-emptive or non-preemptive. **Pre-emptive Scheduling Algorithms** are those which are based on the priority of the processes. By preference, when a high priority process enters, it pre-empts a low priority process in between and executes the high priority process first. **Non-preemptive Scheduling Algorithms** are those who can’t be pre-empted in between, i.e., we cannot take control of CPU in between until the current process completes its execution. The goals of scheduling algorithms are as follows:

• Maximum utilization of CPU so that we can keep the CPU as busy as possible.

• **Throughput** means the number of processes which are completing their execution in per unit time. There must be maximum throughput.

• **Turnaround time** means that the time taken by the processes to finish their implementation. It must be a minimum.

• **Waiting time** is that time for which the process remains in the ready queue. It must be a minimum.

• There must be fare allocation of CPU.  
• **Response time** is the time when the process gives its first response. It must be a minimum.

### First Come First Serve (FCFS) Scheduling

In this scheduling, the process which arrives first in front of CPU will be executed first by the CPU. It is a non-preemptive type of scheduling algorithm, i.e., in this scheduling algorithm priority of processes does not matter, the process will be executed in the manner they arrived in front of the CPU. FCFS scheduling is also called as FIFO, i.e. first in first out.

**Example**: if we consider the real-life example of FCFS scheduling, it will be taken as buying the ticket on the ticket counter. As we know while purchasing the ticket, every passenger is served in the queue manner. Similarly, happen in the FCFS scheduling. Thus, the process which arrives first in a queue will first be served by the CPU and the other process will later on served by CPU.

#### Calculating Average Waiting Time for Scheduling

For seeing the performance, we will calculate the average waiting time of the processes. We will explain this with the help of an example.

**Example**: consider the three processes P1, P2 and P3 whose arrival time and burst time are adequately illustrated in the table given below. The arrival of the processes for execution is in the same manner as they are shown in the table.

|  |  |  |
| --- | --- | --- |
| **Process** | **Burst time** | **Arrival** |
| P1 | 21 | 1st |
| P2 | 3 | 2nd |
| P3 | 5 | 3rd |



This is the GANTT chart of the above process. The average waiting time (0+21+24)/3=15ms

Firstly the process P1 will be executed by the CPU as its arrival time is 0.Hence, there is no waiting time for the process P1 as CPU sits idle during this time. The burst time of the process P1 is 21ms. So, it will take 21 ms to complete its execution. Thus, this time will be counted as waiting time for process P2. In the same way, the waiting time for the process P3 can be calculated when we add the execution time of both the processes P1 and P2 i.e. waiting time for the process P3= 21+3=24 ms . Following the same procedure, the waiting time for the process P4= execution time of P1+P2+P3= 21+3+5= 29ms  
So, we can completely explained this process through the GANTT chart which is illustrated above.

## Shortest Job First (SJF) scheduling algorithm

It is also called the Shortest Job Next (SJN) scheduling. It is both preemptive and non-preemptive. In this scheduling algorithm, the process which has the shortest burst time will be processed first by the CPU. In this, the arrival time of all the processes must be the same. Also, the processor must have to aware about the burst time of all the processes in advance.  
It is the non-preemptive mode of scheduling. The preemptive method of SJF scheduling is known as Shortest Remaining Time First scheduling algorithm.  
Consider the processes which are given below in the table having arrival time is 0 and burst time is given.

|  |  |
| --- | --- |
| **Process** | **Burst Time** |
| P1 | 10 |
| P2 | 20 |
| P3 | 6 |
| P4 | 4 |
| P5 | 2 |

As we know in this scheduling, the shortest job will be executed by the CPU in the starting. This can be easily understood through GANTT chart which is given below:



Average waiting time= (0+2+6+12+22)/5= 42/5= 8.4 ms

## Round Robin scheduling algorithm

Round robin scheduling is the preemptive scheduling algorithm. In this scheduling, every process gets executed cyclically or you can say that a particular time slice is allocated to each process to which we can say as time quantum. Every process, which wants to execute itself, is present in the queue. CPU is assigned to the process for that time quantum. Now, if the process completed its execution in that quantum of time, then the process will get terminated, and if the process does not achieve its implementation, then the process will again be added to the ready queue, and the previous process will wait for its turn to complete its execution. The round-robin scheduling drives its name from the principle which is known as a round robin.

**Example of round robin scheduling:** In this example, we will take six processes P1, P2, P3, P4, P5 and P6 whose arrival and burst time are given in the table. The time quantum is 4 units.

|  |  |  |
| --- | --- | --- |
| **Processes** | **Arrival Time** | **Burst Time** |
| P1 | 0 | 5 |
| P2 | 1 | 6 |
| P3 | 2 | 3 |
| P4 | 3 | 1 |
| P5 | 4 | 5 |
| P6 | 5 | 4 |

As the structures of ready queue and Gantt chart will changes after every scheduling, so we have to maintain it in the algorithm again and again.

**Ready Queue:** Initially, in the starting, the process P1 will be executed for the given time quantum. So, there will be only one process P1 in the ready queue when we start the scheduling.

|  |
| --- |
| P1 |
| 5 |

P1 will be executed for 4 units.

**Ready Queue** Mid while, during the execution of process P1, some other processes like P2, P3, P4 and P5 arises for the execution in the ready queue. As we know P1 has not completed since its time is 5 and we have 4 units of time slice. So, 1 unit is still left. So, it will be again added at the back in ready queue.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| P2 | P3 | P4 | P5 | P1 |
| 6 | 3 | 1 | 5 | 1 |

Now the gantt chart will be like this.

**Ready queue** When the process P2 is executing, another process P6 arrives in the ready queue. As we know that the process P2 has not terminated its execution as its 2 units of burst time is still left. So, for now we will add the process P2 in the ready queue at the back so that it can complete its execution.

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

Now the gantt chart will be like this.

Now, P3 will be executed for 3 units of time slice as its bursts time is 3 units.

Now the process P3 is completed in the time slice of 4 units. So, we will not add P3 in the ready queue and now we will execute the next process P4.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| P4 | P5 | P6 | P1 | P2 |
| 1 | 5 | 1 | 4 | 2 |

Now the gantt chart will be like this.

The next process which is in the ready queue is P5 and it has 5 units of bursts time. Also, the process P4 gets completed or you can say that has terminated its execution as it has only 1 unit of bursts time.

|  |  |  |  |
| --- | --- | --- | --- |
| P5 | P6 | P1 | P2 |
| 5 | 1 | 4 | 2 |

Now the gantt chart will be like this.

Now the process P5 has not completed its execution as 1 unit of burst time is left. So we add it in the ready queue in the back so that it can be further executed by the CPU.

|  |  |  |  |
| --- | --- | --- | --- |
| P1 | P6 | P2 | P5 |
| 1 | 4 | 2 | 1 |

Now the gantt chart will be like this.

Now the turn of the process P1 has come according to the ready queue. So, for now the process P1 will complete its execution. As we know it requires only 1 unit of bursts time, so it will get completed in this chance of execution.

As the process P1 also get completed. So, we will not add it into the ready queue. Now, overall, only three processes P6, P2 and P5 are left.

|  |  |  |
| --- | --- | --- |
| P6 | P2 | P5 |
| 4 | 2 | 1 |

Now the gantt chart will be like this.

Now P6 has 4 units of bursts time. So, it will get executed in 4 units of time slice

Since the process P6 is executed completely. So, we will not add it to the ready queue. Now only 2 processes left in the ready queue which are P2 and P5.

|  |  |
| --- | --- |
| P2 | P5 |
| 2 | 1 |

Now the gantt chart will be like this.

Now P2 will be executed again and it requires only 2 units of bursts time. So now it will be completed.

Only process left in the ready queue is P5 which requires only 1 unit of bursts time. As we know our time slice is of 4 units. So, it will get completed in the next burst.

|  |
| --- |
| P5 |
| 1 |

Now the gantt chart will be like this.

The process P5 will get executed until it gets completed as it is the only process left in the ready queue.

Now we will calculate turnaround time, completion time and average waiting time which is shown in the below table.

Turnaround time= completion time- arrival time  
Waiting time= turnaround time – burst time

Average waiting time= (12+16+6+8+15+11)/6= 76/6 = 12.66 units