**MODULE 5: Real-time Operating System (RTOS) based Embedded System Design:** Operating System basics, Types of Operating Systems, Tasks, Process and Threads, Multiprocessing and Multitasking, Task Scheduling

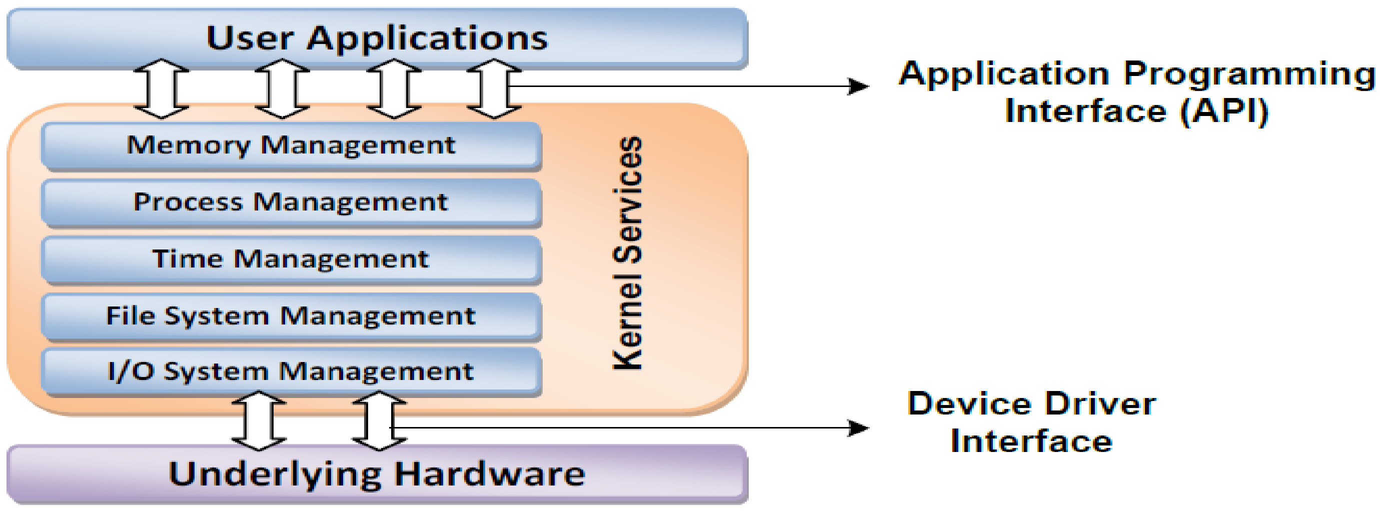
## OPERATING SYSTEM BASICS

Operating System (OS) acts as a bridge between the user applications/ tasks and the underlying system resources through a set of system functionalities and services. Operating system manages the system resources and makes them available to the user applications/tasks on a need basis.

The primary functions of operating systems are:

- Make the system convenient to use
- Organize and manage the system resources efficiently and correctly.

Figure below gives an insight into the basic components of an operating system and their interfaces with rest of the world.



**Kernel:** The kernel is the core of the operating system. It is responsible for managing the system resources and the communication among the hardware and other system services. Kernel acts as the abstraction layer between system resources and user applications. Kernel contains a set of system libraries and services. For a general purpose OS, the kernel contains different services like:

- Process management
- Memory management
- Time management
- File system management
- I/O system management.

**1. Process management:** Process management deals with managing the process/ tasks. Process management includes:

- setting up a memory for the process
- loading process code into memory
- allocating system resources
- scheduling and managing the execution of the process
- setting up and managing Process Control Block (PCB)
- inter process communication and synchronization
- process termination/ deletion, etc.

**2. Primary Memory Management:** Primary memory refers to a volatile memory (RAM), where processes are loaded and variables & shared data are stored. The Memory Management Unit (MMU) of the kernel is responsible for:

- Keeping a track of which part of the memory area is currently used by which process.
- Allocating and De-allocating memory space on a need basis.

**3. File System Management:** File is a collection of related information. A file could be a program (source code or executable), text files, image files, word documents, audio/ video files, etc. A file system management service of kernel is responsible for:

- The creation, deletion and alteration of files
- Creation, deletion, and alteration of directories
- Saving of files in the secondary storage memory
- Providing automatic allocation of file space based on the amount of free running space available
- Providing flexible naming conversion for the files.

**4. I/O System (Device) Management:** Kernel is responsible for routing the I/O requests coming from different user applications to the appropriate I/O devices of the system. The direct access to I/O devices is not allowed; access to them is establish through Application Programming Interface (API). The kernel maintains list of all the I/O devices of the system.

The service "*Device Manager*" of the kernel is responsible for handling all I/O related operations. The Device Manager is responsible for:

- Loading and unloading of device drivers
- Exchanging information and the system specific control signals to and from the device.

**5. Secondary Storage Management:** The secondary storage management deals with managing the secondary storage memory devices (if any) connected to the system. Secondary memory is used as backup medium for programs and data, as main memory is volatile. In most of the systems secondary storage is kept in disks (hard disks). The secondary storage management service of kernel deals with:

- Disk storage allocation
- Disk scheduling
- Free disk space management

**6. Protection Systems:** Modern operating systems are designed in such way to support multiple users with different levels of access permissions. The *protection* deals with implementing the security policies to restrict the access of system resources and particular user by different application or processes and different user.

**7. Interrupt Handler:** Kernel provides interrupt handler mechanism for all external/ internal interrupt generated by the system.

The important services offered by the kernel of an OS:

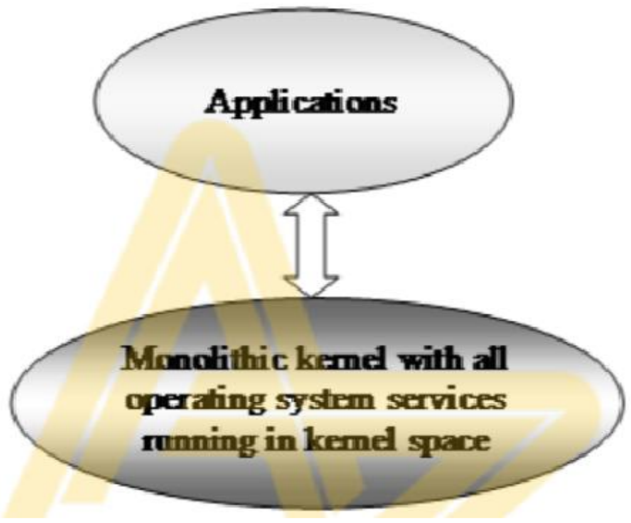
**1. Kernel Space and User Space:** The program code corresponding to the kernel applications/ services are kept in a contiguous area of primary (working) memory and is protected from the un-authorized access by user programs/ applications.

- The memory space at which the kernel code is located is known as "*Kernel Space*". All user applications are loaded to a specific area of primary memory and this memory area is referred as "*User Space*".
- The partitioning of memory into kernel and user space is purely OS dependent.
- Most of the operating systems keep the kernel application code in main memory and it is not swapped out into the secondary memory.

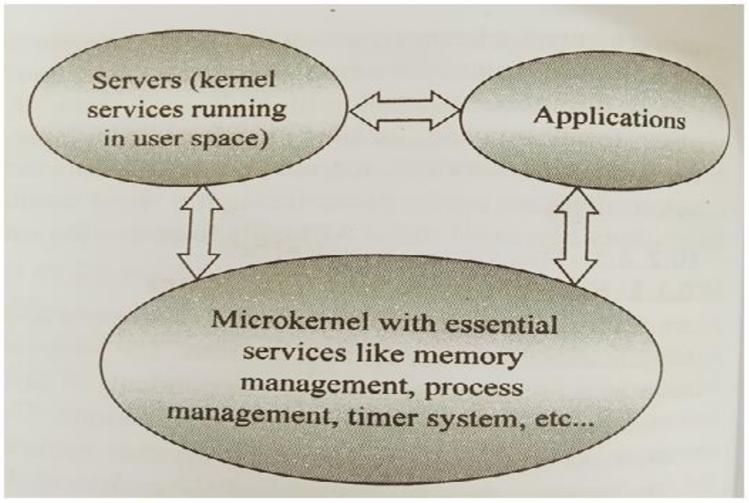
**2. Monolithic Kernel and Microkernel:** Kernel forms the heart of OS. Different approaches are adopted for building an operating system kernel. Based on the kernel design, kernels can be classified into "*Monolithic*" and "*Micro*".

**Monolithic Kernel:** In monolithic kernel architecture, all kernel services run in the kernel space. All kernel modules run within the same memory space under a single kernel thread. Allows effective utilization of the low-level features of the underlying system. The major drawback is that any error or failure in any one of the kernel modules leads to the crashing of the entire kernel application. LINUX, SOLARIS, MS-DOS kernels are the examples.

Monolithic Kernel


Applications
Monolithic kernel with all operating system services running in kernel space

Micro Kernel


Servers (kernel services running in user space)
Applications
Microkernel with essential services like memory management, process management, timer system, etc...

**Microkernel:** The microkernel design iincorporates only essential set of OS services into the kernel. The rest of the OS services are implemented in program known as "*Servers*" which runs in user space. The memory management, timer systems and interrupt handlers are the essential services, which forms the part of the microkernel. The benefits of micro kernel based designs are:

- **Robustness:** If a problem is encountered in any of the services, which runs as a server can be reconfigured and restarted without the restarting the entire OS. Here chances of corruption of kernel services are ideally zero.
- **Configurability:** Any services, which runs as a server application can be changed without the need to restart the whole system.

## TYPES OF OPERATING SYSTEMS

Depending on the type of kernel and kernel services, purpose and type of computing systems where the OS is deployed and the responsiveness to applications, Operating Systems are classified into:

- General Purpose Operating System (GPOS)
- Real Time Purpose Operating System (RTOS)

**1. General Purpose Operating System (GPOS):** Operating systems, which are deployed in general computing systems. The kernel is more generalized and contains all the required services to execute generic applications. Need not be deterministic in execution behaviour. May inject random delays into application software and thus cause slow responsiveness of an application at unexpected times. Personal Computer/Desktop system is typical example for a system where GPOSs are deployed. Windows XP/MS-DOS are examples of GPOS.

**2. Real Time Operating System (RTOS):** Operating Systems, which are deployed in embedded systems demanding real-time response. *Real Time* implies deterministic in timing behavior. RTOS services consumes only known and expected amounts of time regardless the number of services. RTOS implements policies and rules concerning time-critical allocation of a system's resources. RTOS decides which applications should run in which order and how much time needs to be allocated for each application. Windows Embedded Compact, QNX, VxWorks MicroC/OS-II, etc., are examples of RTOSs.

**Real-Time kernel**: The kernel of a Real-Time OS is referred as Real-Time kernel. The Real-Time kernel is highly specialized and it contains only the minimal set of services required for running user applications/ tasks. The basic functions of a Real-Time kernel are listed below:

- Task/ Process management
- Task/ Process scheduling
- Task/ Process synchronization
- Error/ Exception handling
- Memory management
- Interrupt handling
- Time management.

**i) Task/ Process management:** Deals with setting up the memory space for the tasks, loading the tasks code into the memory space, allocating system resources and setting up a Task Control Block (TCB) for the task and task/process termination/deletion. A TCB is used for holding the information corresponding to a task. TCB usually contains the following set of information:

- Task ID: Task Identification Number
- Task State: The current state of the task.
- Task Type: Task type. Indicates what is the type for this task.
- Task Priority: Task priority
- Task Context Pointer: Context pointer. Pointer for context saving
- Task Memory Pointers: Pointers to the code memory, data memory and stack memory
- Task System Resource Pointers: Pointers to system resources
- Task Pointers: Pointers to other TCBs

**ii) Task/ Process Scheduling:** Deals with sharing the CPU among various tasks/ processes. A kernel application called "*Scheduler*" handles the task scheduling. Scheduler is an algorithm implementation, which performs the efficient and optimal scheduling of tasks to provide a deterministic behavior.

**iii) Task/ Process Synchronization:** Deals with synchronizing the concurrent access of a resource, which is shared across multiple tasks and the communication between various tasks.

**iv) Error/ Exception Handling:** Deals with registering and handling the errors occurred/ exceptions raised during the execution of tasks. Insufficient memory, timeouts, deadlocks, deadline missing, bus error, divide by zero, unknown instruction execution etc., are examples of errors/exceptions. Errors/ Exceptions can happen at the kernel level services or at task level. *Deadlock* is an example for kernel level exception, whereas *timeout* is an example for a task level exception.

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. Timeouts and retry are two techniques used together. The tasks retries an event/ message certain number of times; if no response is received after exhausting the limit, the feature might be aborted.

**v) Memory Management:** The memory allocation time increases depending on the size of the block of memory need to be allocated and the state of the allocated memory block. RTOS achieves predictable timing and deterministic behavior, by compromising the effectiveness of memory allocation. RTOS generally uses *"block"* based memory allocation technique, instead of the usual dynamic memory allocation techniques used by the GPOS.

RTOS kernel uses blocks of fixed size of dynamic memory and the block is allocated for a task on a need basis. The blocks are stored in a **"*Free buffer Queue*".** The memory management function a block of fixed memory is always allocated for tasks on need basis and it is taken as a unit. Hence, there will not be any memory fragmentation issues.

**vi) Interrupt Handling:** Interrupts inform the processor that an external device or an associated task requires immediate attention of the CPU. Interrupts can be either Synchronous or Asynchronous.

Interrupts which occurs in sync with the currently executing task is known as Synchronous interrupts. Eg: Divide by zero, memory segmentation error etc. Interrupts which occurs at any point of execution of any task, and are not in sync with the currently executing task are Asynchronous interrupts. Eg: Timer overflow interrupts, serial data reception/ transmission interrupts etc. Priority levels can be assigned to the interrupts and each interrupts can be enabled or disabled individually.

**vii) Time Management:** The time reference to kernel is provided by a high-resolution Real Time Clock (RTC) hardware chip (hardware timer). The hardware timer is programmed to interrupt the processor/ controller at a fixed rate. This timer interrupt is referred as "Timer tick". The Timer tick is taken as the timing reference by the kernel. Usually, the Timer tick varies in the microseconds range. The System time is updated based on the Timer tick.

**Types of Real Time System**

**Hard Real-Time:** A Real Time Operating Systems which strictly adheres to the timing constraints for a task is referred as hard real-time systems. A Hard RTS must meet the deadlines for a task without any slippage. Missing any deadline may produce catastrophic results for Hard RTS, including permanent data lose & irrecoverable damages to the system/users. Most of the Hard Real Time Systems are automatic.

Eg: Air bag control systems and Anti-lock Brake Systems (ABS) of vehicles are typical examples of Hard Real Time Systems.

**Soft Real-Time:** Real Time Operating Systems that does not guarantee meeting deadlines, but, offer the best effort to meet the deadline are referred as soft real-time systems. Missing deadlines for tasks are acceptable if the frequency of deadline missing is within the compliance limit of the Quality of Service (QoS).

Automatic Teller Machine (ATM) is a typical example of Soft Real Time System. If the ATM takes a few seconds more than the ideal operation time, nothing fatal happens.

## TASKS, PROCESSES AND THREADS

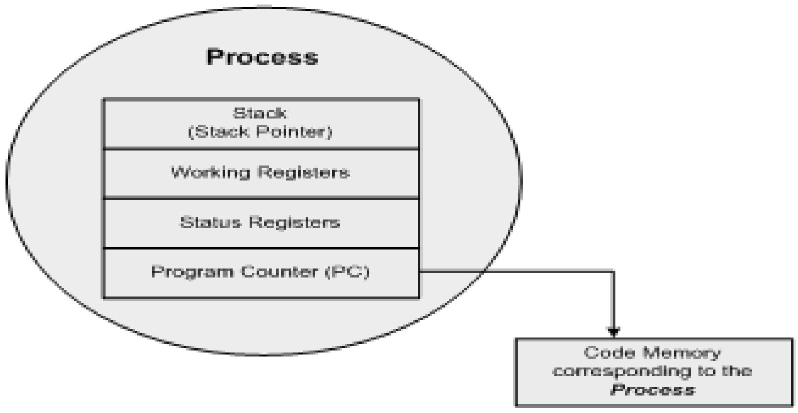
The term "***task***" refers to something that needs to be done. In the Operating System context, a ***task*** is defined as the program in execution and the related information maintained by the Operating system for the program. Task is also known as "*Job*" in the operating system context.

A program or part of it in execution is also called a "*Process*". The terms "*Task*", "*Job*" and "*Process*" refer to the same entity in the Operating System context and most often they are used interchangeably.

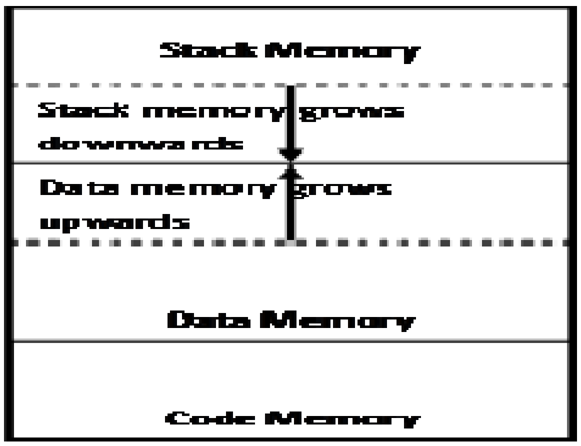
**Process:** A Process is a program, or part of it, in execution. Process is also known as an instance of a program in execution. A process requires various system resources like CPU for executing the process, memory for storing the code corresponding to the process and associated variables, I/O devices for information exchange etc.

**Structure of a Processes:** The concept of Process leads to concurrent execution of tasks and thereby, efficient utilization of the CPU and other system resources. Concurrent execution is achieved through the sharing of CPU among the processes.

A process mimics a processor in properties and holds a set of registers, process status, a Program Counter (PC) to point to the next executable instruction of the process, a stack for holding the local variables associated with the process and the code corresponding to the process. This can be visualized as shown in below figure.



Structure of a Process



Memory Oraganization of a Process

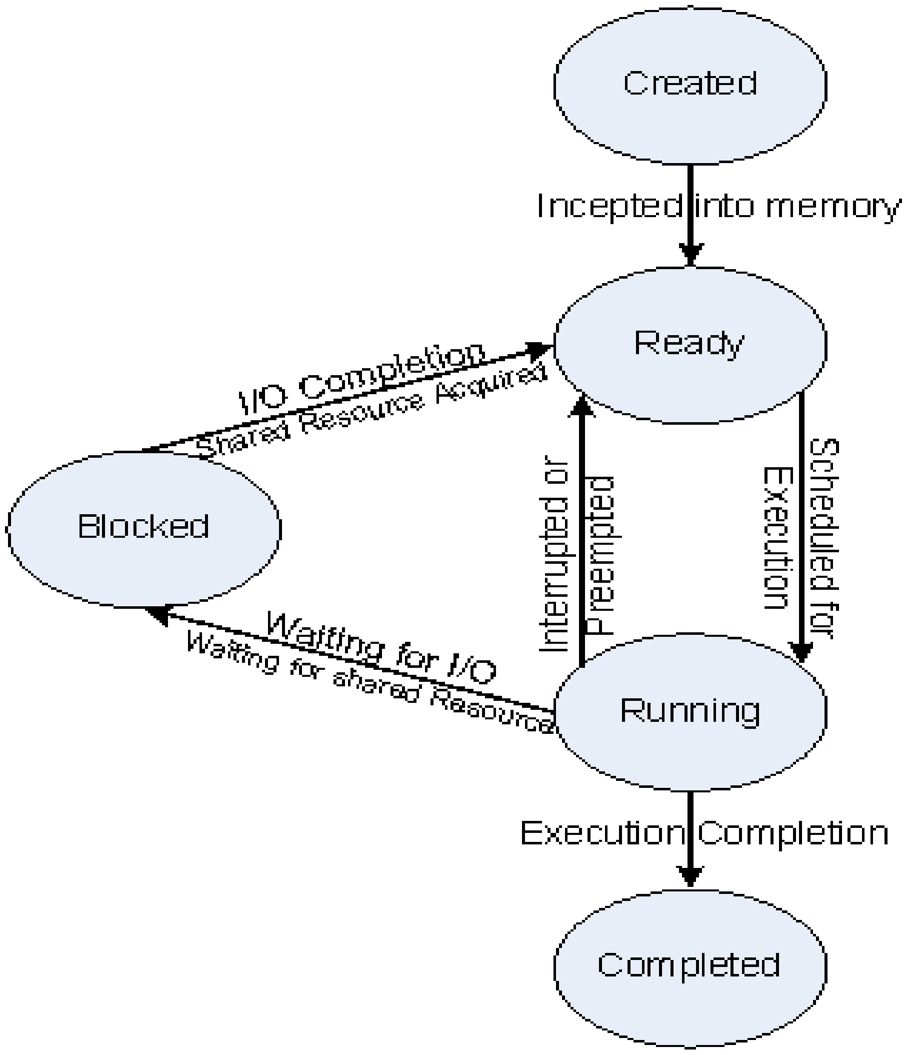
**Memory Organization of Process:** The memory occupied by the process is segregated into three regions namely: Stack memory, Data memory and Code memory.

- "Stack" memory holds all temporary data such as variables local to the process.
- "Data" memory holds all global data for the process.
- "Code" memory contains the program code corresponding to the process.

On loading a process into the main memory, a specific area of memory is allocated for the process. The stack memory usually starts at the highest memory address from the memory area allocated for the process.

**Process States & State Transition**

The process traverses through a series of states during its transition from the newly created state to the terminated state. The cycle through which a process changes its state from "*newly created*" to "*execution completed*" is known as "*Process Life Cycle*". The various states through which a process traverses through during a Process Life Cycle indicates the current status of the process with respect to time & also provides information on what it is allowed to do next. The transition of a process from one state to another is known as "*State transition*". The Process states & state transition representation are shown in below figure.



**Created State:** The state at which a process is being created. The OS recognizes a process in the "Created State" but no resources are allocated to the process.

**Ready State:** The state, where a process is incepted into the memory and awaiting the processor time for execution. At this stage, the process is placed in the "Ready list" queue maintained by the OS.

**Running State:** The state where in the source code instructions corresponding to the process is being executed. Running state is the state at which the process execution happens.

**Blocked State/ Wait State:** Refers to a state where a running process is temporarily suspended from execution & does not have immediate access to resources.

**Completed State:** A state where the process completes its execution.

**Thread:** A thread is the primitive that can execute code. A thread is a single sequential flow of control within a process. A thread is also known as lightweight process.

A process can have many threads of execution. Different threads, which are part of a process, share the same address space; meaning they share the data memory, code memory and heap memory area. Threads maintain their own thread status (CPU register values), Program Counter (PC) and stack. The memory model for a process and its associated threads are given in figure.

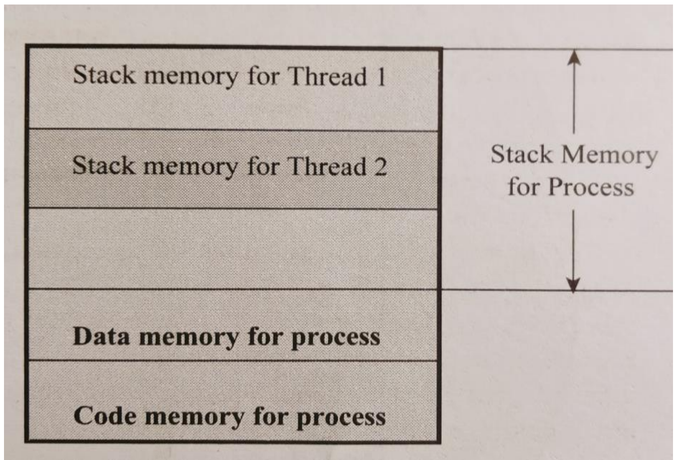
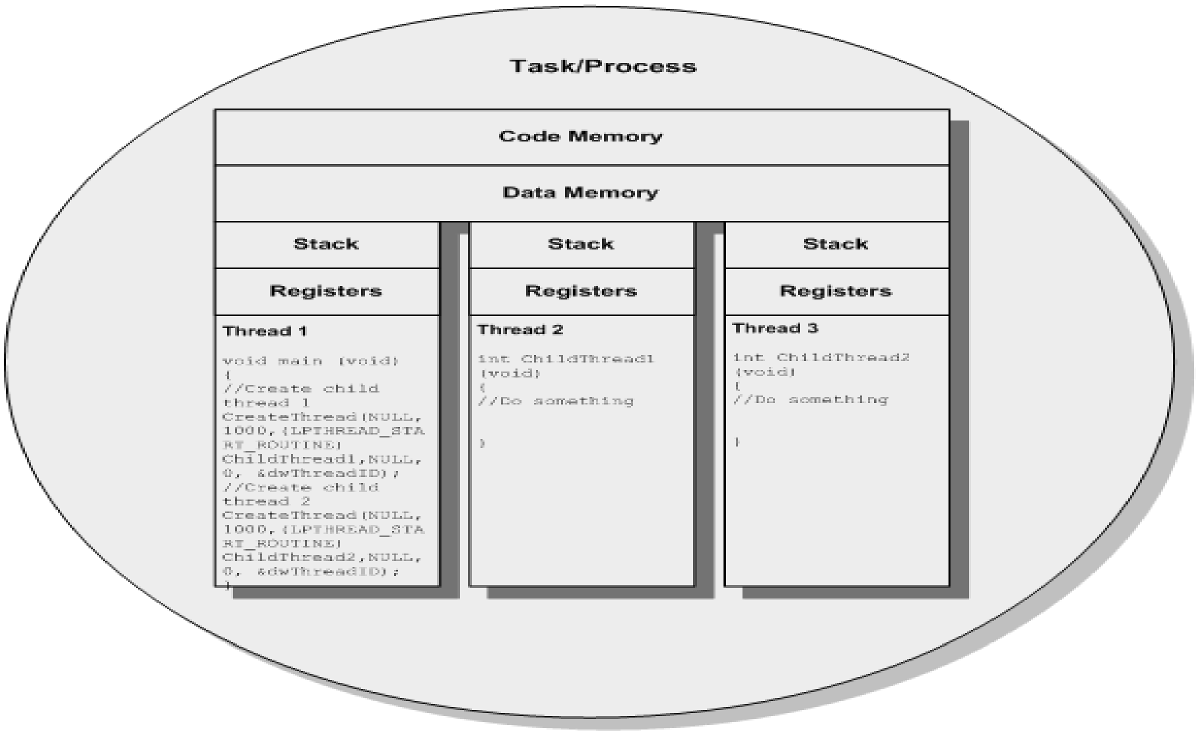


Fig. 10.7 Memory organisation of a Process and its associated Threads

**The Concept of Multithreading:** The process is split into multiple threads, which executes a portion of the process; there will be a main thread and rest of the threads will be created within the main thread. The multithreaded architecture of a process can be visualized with the thread-process diagram, shown.



```
void main (void)
{
//Create child
thread 1
CreateThread(NULL,
1000,(LPTHREAD_STA
RT_ROUTINE)
ChildThread1,NULL,
0, &dwThreadID);
//Create child
thread 2
CreateThread(NULL,
1000,(LPTHREAD_STA
RT_ROUTINE)
ChildThread2,NULL,
0, &dwThreadID);
}
```

```
int ChildThread1
(void)
{
//Do something


}
```

```
int ChildThread2
(void)
{
//Do something


}
```

**Advantages of Multiple threads:** Use of multiple threads to execute a process brings the following advantage:

- Better memory utilization: Multiple threads of the same process share the address space for data memory. This also reduces the complexity of inter thread communication.

- Speed: Since the process is split into different threads, when one thread enters a wait state, the CPU can be utilized by other threads of the process. This speeds up the execution of the process.
- Efficient CPU utilization: The CPU is engaged all time.

**Differences between Threads and Processes**

| Thread | Process |
|---|---|
| Thread is a single unit of execution and is part of process. | Process is a program in execution and contains one or more threads. |
| A thread does not have its own data memory and heap memory. | Process has its own code memory, data memory, and stack memory. |
| A thread cannot live independently; it lives within the process. | A process contains at least one thread. |
| There can be multiple threads in a process; the first (main) thread calls the main function and occupies the start of the stack memory of the process. | Threads within a process share the code, data and heap memory; each thread holds separate memory area for stack. |
| Threads are very inexpensive to create. | Processes are very expensive to create; involves many OS overhead. |
| Context switching is inexpensive and fast. | Context switching is complex and involves lots of OS overhead and comparatively slow. |
| If a thread expires, its stack is reclaimed by the process. | If a process dies, the resource allocated to it are reclaimed by the OS and all associated threads of the process also dies. |

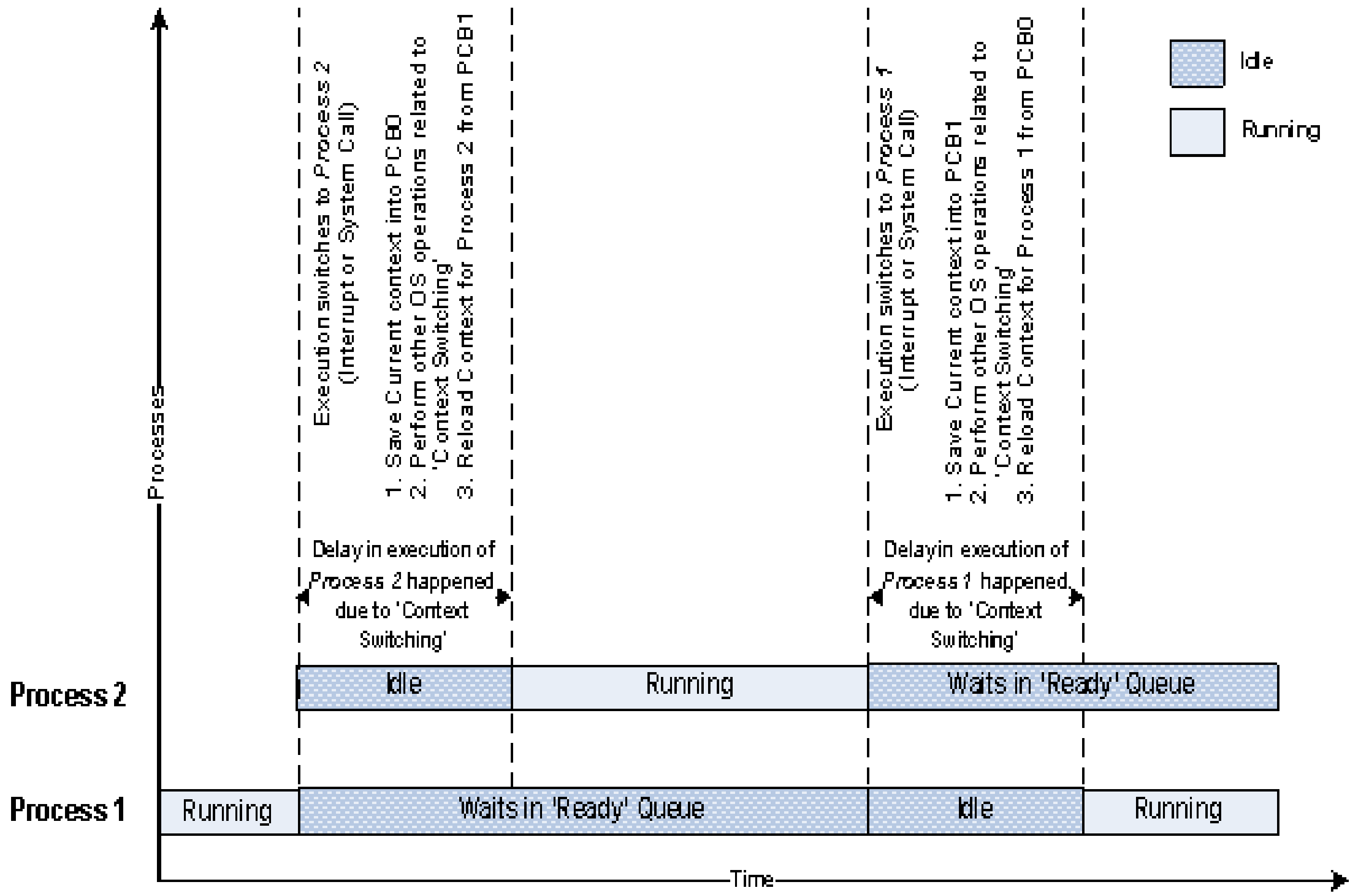
## MULTIPROCESSING AND MULTITASKING

The ability to execute multiple processes simultaneously is referred as *multiprocessing*. Systems which are capable of performing multiprocessing are known as *multiprocessor systems*. Multiprocessor systems possess multiple CPUs and can execute multiple processes simultaneously.

The ability of the Operating System to have multiple programs in memory, which are ready for execution, is referred as *multiprogramming*. In a Uni-processor system, it is not possible to execute multiple processes simultaneously. *Multitasking* refers to the ability of an operating system to hold multiple processes in memory and switch the processor (CPU) from executing one process to another process.

**Multitasking:** Multitasking involves "*Context switching*", "*Context saving*" and "*Context retrieval*". (shown in figure)

- The act of switching CPU among the processes or changing the current execution context is known as "***Context switching***".
- The act of saving the current context (details like Register details, Memory details, System Resource Usage details, Execution details, etc.) for the currently running processes at the time of CPU switching is known as **"*Context saving*".**
- The process of retrieving the saved context details for a process, which is going to be executed due to CPU switching, is known as **"*Context retrieval*".**


Execution switches to *Process 2* (Interrupt or System Call)
1. Save Current context into PCB0
2. Perform other OS operations related to 'Context Switching'
3. Reload Context for Process 2 from PCB1
Execution switches to *Process 1* (Interrupt or System Call)
1. Save Current context into PCB1
2. Perform other OS operations related to 'Context Switching'
3. Reload Context for Process 1 from PCB0
Idle
Running
Delay in execution of *Process 2* happened due to 'Context Switching'
Delay in execution of *Process 1* happened due to 'Context Switching'
Processes
Process 2
Idle
Running
Waits in 'Ready' Queue
Process 1
Running
Waits in 'Ready' Queue
Idle
Running
Time

**Types of Multitasking:** Depending on how the task/ process execution switching act is implemented, multitasking can is classified into –

- **Co-operative Multitasking**
- **Preemptive Multitasking**
- **Non-preemptive Multitasking**

**1. Co-operative Multitasking:** Co-operative Multitasking is the most primitive form of multitasking in which a task/ process gets a chance to execute only when the currently executing task/ process voluntarily relinquishes the CPU.

In this method, any task/ process can avail the CPU as much time as it wants. Since this type of implementation involves the mercy of the tasks each other for getting the CPU time for execution, it is known as co-operative multitasking. If the currently executing task is non-cooperative, the other tasks may have to wait for a long time to get the CPU.

**2. Preemptive Multitasking:** Preemptive multitasking ensures that every task/ process gets a chance to execute. When and how much time a process gets is dependent on the implementation of the preemptive scheduling.

As the name indicates, in preemptive multitasking, the currently running task/process is preempted to give a chance to other tasks/process to execute. The preemption of task may be based on time slots or task/ process priority.

**3. Non-preemptive Multitasking:** The process/ task, which is currently given the CPU time, is allowed to execute until it terminates (enters Completed state) or enters Blocked/ Wait state, waiting for an I/O. The co-operative and non-preemptive multitasking differs in their behaviour when they are in the Blocked/Wait state.

In co-operative multitasking, the currently executing process/task need not relinquish the CPU when it enters the Blocked/ Wait sate, waiting for an I/O, or a shared resource access or an event to occur whereas in non-preemptive multitasking the currently executing task relinquishes the CPU when it waits for an I/O.

## TASK SCHEDULING

Determining which task/process is to be executed at a given point of time is known as task/process scheduling. Scheduling policies forms the guidelines for determining which task is to be executed when. The kernel service/application which implements the scheduling algorithm, is known as 'Scheduler. Process scheduling decision may take place when a process switches its state to:

1. Ready state from Running state
2. Blocked/Wait state from Running state
3. Ready state from Blocked/Wait state
4. Completed state

A process switches to **Ready** state from the **Running** state when it is preempted. Hence, the type of scheduling in scenario 1 is pre-emptive. When a high priority process in the **Blocked/Wait** state completes its I/O and switches to the **Ready** state, the scheduler picks it for execution if the scheduling policy used is priority based preemptive. This is indicated by scenario 3.

In preemptive/non-preemptive multitasking, the process relinquishes the CPU when it enters the **Blocked/Wait** state or the **Completed** state and switching of the CPU happens at this stage. Scheduling under scenario 2 can be either preemptive or non-preemptive. Scheduling under scenario 4 can be preemptive, non-preemptive or co-operative.

The selection of a scheduling criterion/algorithm should consider the following factors:

- CPU Utilisation: The scheduling algorithm should always make CPU utilisation high.
- Throughput: Gives an indication of the number of processes executed per unit of time
- Turnaround Time: Amount of time taken by a process for completing its execution.
- Waiting Time: It is the amount of time spent by a process in the 'Ready' queue waiting to get the CPU time for execution.
- Response Time: It is the time elapsed between the submission of a process and the first response.

The Operating System maintains various queues in connection with the CPU scheduling, and a process passes through these queues during the course of its admittance to execution completion. The various queues maintained by OS in association with CPU scheduling are:

**Job Queue:** Job queue contains all the processes in the system

**Ready Queue:** Contains all the processes, which are ready for execution and waiting for CPU to get their turn for execution. The Ready queue is empty when there is no process ready for running.

**Device Queue:** Contains the set of processes, which are waiting for an I/O device. A process migrates through all these queues during its journey from 'Admitted to 'Completed stage.

Figure below illustrates the transition of a process through the various queues.

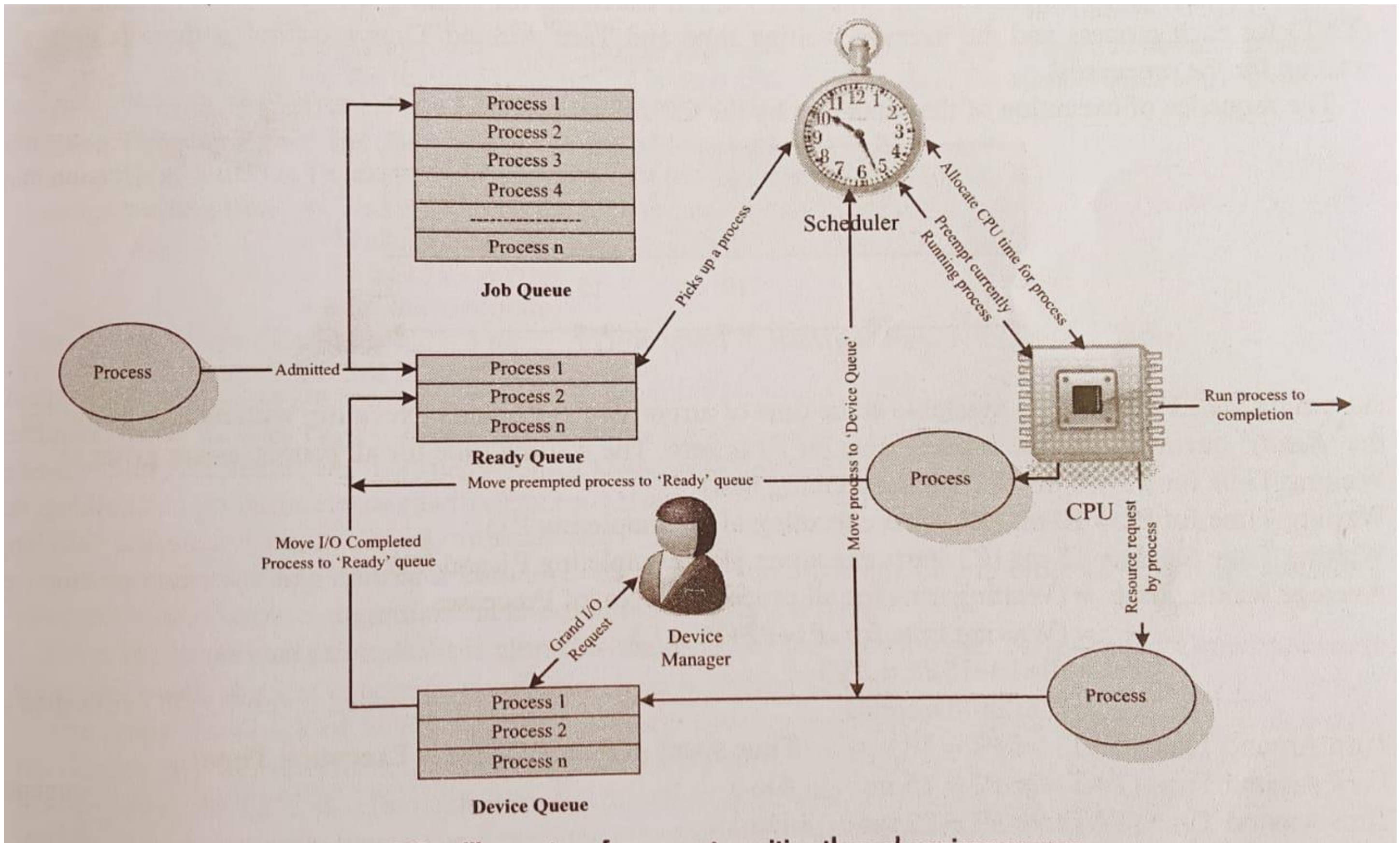

Process 1
Process 2
Process 3
Process 4
Process n
Job Queue
Process
Admitted
Ready Queue
Move preempted process to 'Ready' queue
Picks up a process
Scheduler
Allocate CPU time for process
Preempt currently Running process
Move process to 'Device Queue'
CPU
Run process to completion
Resource request by process
Move I/O Completed Process to 'Ready' queue
Grand I/O Request
Device Manager
Device Queue

Fig. 10.12 Illustration of process transition through various queues

## Non-preemptive Scheduling

Non-preemptive scheduling is employed in systems, which implement non-preemptive multitasking model. In this scheduling type, the currently executing task/process is allowed to run until it terminates or enters the 'Wait' state waiting for an I/O or system resource. The various types of non-preemptive scheduling adopted in task/process scheduling are listed below:

**1. First-Come-First-Served (FCFS)/ FIFO Scheduling**

As the name indicates, the First-Come-First-Served (FCFS) scheduling algorithm allocates CPU time to the processes based on the order in which they enter the 'Ready' queue. The first entered process is serviced first. FCFS scheduling is also known as First In First Out (FIFO) where the process which is put first into the 'Ready queue is serviced first.

**Example 1:** Three processes with process ID's P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds respectively enters the ready queue together in the order P1, P2, P3. Calculate the waiting time and Turn Around Time (TAT) for each process and the average waiting time and Turn Around Time (Assuming there is no 1/0 waiting for the processes).

**Solution:** The sequence of execution of the processes by the CPU is represented as

| P1 | P2 | P3 |
|---|---|---|

Assuming the CPU is readily available at the time of arrival of P1, P1 starts executing without any waiting in the 'Ready' queue. Hence the waiting time for P1 is zero. Thus, the waiting time for all processes are given as:

Waiting Time for P1 = 0

Waiting Time for P2 = 10ms (P2 starts executing after completing P1)

Waiting Time for P3 = 15ms (P3 starts executing after completing P1 and P2)

Average waiting time = (Waiting time for all processes) / No. of Processes

= (0 + 10 +15) / 4 = 8.33 milliseconds.

Turn Around Time (TAT) = Time spent in Ready Queue (Waiting time) + Execution Time

Turn Around Time (TAT) for P1 = 10ms (0 + 10)

Turn Around Time (TAT) for P2 = 15ms (10 + 5)

Turn Around Time (TAT) for P3 = 22ms (15 + 7)

Average Turn Around Time (TAT) = Average waiting time + Average execution time.

Average Execution Time = (Execution time for all processes)/No. of processes

= 10 + 5 + 7 / 3 = 7.33 milliseconds.

Average Turn Around Time (TAT) = 8.33 + 7.33 = 15.66 milliseconds.

**NOTE:** Average Turn Around Time (TAT) = TAT of all processes / No. of Processes

= (10+15+22)/3 = 15.66 milliseconds.

**2. Last-Come-First Served (LCFS)/LIFO Scheduling**

The Last-Come-First Served (LCFS) scheduling algorithm also allocates CPU time to the processes based on the order in which they are entered in the 'Ready' queue. The last entered process is serviced first. LCFS scheduling is also known as Last In First Out (LIFO) where the process, which is put last into the 'Ready' queue, is serviced first.

**Example 1:** Three processes with process ID's P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds respectively enters the ready queue together in the order P1, P2, P3. Now a new process P4 with estimated completion time 6ms enters the 'Ready' queue after 5ms of scheduling Pl. Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time.

**Solution:** Initially there is only P1 available in the Ready queue and the scheduling sequence will be P1, P3, P2. P4 enters the queue during the execution of P1 and becomes the last process entered the 'Ready' queue. Now the order of execution changes to P1, P4, P3, and P2 as given below:

| P1 | P4 | P3 | P2 |
|---|---|---|---|

The waiting time for all the processes is given as:

Waiting Time for P1 = 0ms (P1 starts executing first)

Waiting Time for P4 = 5ms (P4 starts executing after completing Pl. But P4 arrived after 5ms of execution of P1. Hence it’s waiting time = Execution start time - Arrival Time = 5)

Waiting Time for P3 = 16ms (P3 starts executing after completing Pl and P4)

Waiting Time for P2 = 23ms (P2 starts executing after completing P1, P4 and P3)

Average waiting time = (Waiting time for all processes) / No. of Processes

= 0+5+16+23 / 4 = 11 milliseconds.

Turn Around Time (TAT) = Time spent in Ready Queue (Waiting time) + Execution Time

Turn Around Time (TAT) for P1 = 10ms (0 + 10)

Turn Around Time (TAT) for P4 = 11ms ((10-5) + 6)

Turn Around Time (TAT) for P3 = 23ms (16 + 7)

Turn Around Time (TAT) for P3 = 28ms (23 + 5)

Average Turn Around Time (TAT) = TAT of all processes / No. of Processes

= (10+11+23+28) / 4 = 18 milliseconds.

**3. Shortest Job First (SJF) Scheduling**

Shortest Job First (SJF) scheduling algorithm 'sorts the 'Ready' queue' each time a process relinquishes the CPU to pick the process with shortest (least) estimated completion/run time. In SJF, the process with the shortest estimated run time is scheduled first, followed by the next shortest process, and so on.

**Example 1:** Three processes with process IDs P1, P2, P3 with estimated completion time 12, 6, 8 milliseconds respectively enters the ready queue together. Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time (Assuming there is no I/O waiting for the processes) in SJF algorithm.

**Solution:** The scheduler sorts the 'Ready' queue based on the shortest estimated completion time and schedules the process with the least estimated completion time first and the next least one as second, and so on. The order in which the processes are scheduled for execution is represented as

| P2 | P3 | P1 |
|---|---|---|

The waiting time for all processes are given as:

Waiting Time for P2 = 0ms (P2 starts executing first)

Waiting Time for P3 = 6ms (P3 starts executing after completing P2)

Waiting Time for PI = 14ms (P1 starts executing after completing P2 and P3)

Average waiting time = (Waiting time for all processes) / No. of Processes

= (0+6+14) / 3 = 6.66 milliseconds

Turn Around Time (TAT) = Time spent in Ready Queue (Waiting time) + Execution Time

Turn Around Time (TAT) for P2 = 10ms (0 + 6)

Turn Around Time (TAT) for P3 = 14ms (6 + 8)

Turn Around Time (TAT) for P1 = 24ms (14 + 10)

Average Turn Around Time (TAT) = TAT of all processes / No. of Processes

= (10+14+24) / 3 = 16 milliseconds.

**Example 2:** Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time for the above example if a new process P4 with estimated completion time 2ms enters the 'Ready' queue after 2ms of execution of P2. Assume all the processes contain only CPU operation and no I/O operations are involved.

**Solution:** At the beginning, there are only three processes (P1, P2 and P3) available in the 'Ready' queue and the SJF scheduler picks up the process with the least execution completion time (P2) for scheduling. The execution sequence is P2, P3, P1. Now process P4 with estimated execution completion time 2ms enters the 'Ready' queue after 2ms of start of execution of P2. After 6ms of scheduling, P2 terminates and now the scheduler again sorts the 'Ready' queue for process with least execution completion time. Since the execution completion time for P4 (2ms) is less than that of P3 (8ms), which was supposed to be run after the completion of P2 as per the 'Ready' queue available at the beginning of execution scheduling, P4 is picked up for executing. Due to the arrival of the process P4 with execution time 2ms, the 'Ready' queue is re-sorted in the order P2, P4, P3, Pl. At the beginning it was P2, P3, P1. The execution sequence now changes as per the following diagram

| P2 | P4 | P3 | P1 |
|---|---|---|---|

The waiting time for all the processes are given as:

Waiting time for P2 = 0ms (P2 starts executing first)

Waiting time for P4 = 4ms (P4 starts executing after completing P2. But P4 arrived after

2ms of execution of P2. Hence it's waiting time = Execution start time - Arrival Time = 3)

Waiting time for P3 = 8ms (P3 starts executing after completing P2 and P4)

Waiting time for P1 = 16ms (P1 starts executing after completing P2, P4 and P3)

Average waiting time = (Waiting time for all processes) / No. of Processes

= (0+4+8+16) / 4 = 7 milliseconds.

Turn Around Time (TAT) = Time spent in Ready Queue (Waiting time) + Execution Time

Turn Around Time (TAT) for P2 = 6ms (0 + 6)

Turn Around Time (TAT) for P4 = 6ms ((6-2) + 2)

Turn Around Time (TAT) for P3 = 16ms (8 + 8)

Turn Around Time (TAT) for P1 = 28ms (16 + 12)

Average Turn Around Time (TAT) = TAT of all processes / No. of Processes

= (6+6+16+28) / 4 = 19 milliseconds.

**4. Priority Based Scheduling:** Priority based non-preemptive scheduling algorithm ensures that a process with high priority is serviced at the earliest compared to other low priority processes in the 'Ready' queue. While creating the process/task, the priority can be assigned to it. The priority number associated with a task/process is the direct indication of its priority. The non-preemptive priority based scheduler sorts the 'Ready' queue based on priority and picks the process with the highest level of priority for execution.

**Example 1:** Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds and priorities 0, 3, 2 (0-highest priority, 3-lowest priority) respectively enters the ready queue together. Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time (Assuming there is no I/O waiting for the processes) in priority based scheduling algorithm.

**Solution:** The scheduler sorts the 'Ready' queue based on the priority and schedules the process with the highest priority (P1 with priority number 0) first and the next high priority process (P3 with priority number 2) as second, and so on. The order in which the processes are scheduled for execution is represented as

| | | |
|---|---|---|
| **P1** | **P3** | **P2** |

The waiting time for all the processes are given as:

Waiting time for P1 = 0ms (P1 starts executing first)

Waiting time for P3 = 10ms (P3 starts executing after completing PI)

Waiting time for P2 = 17ms (P2 starts executing after completing P1 and P3)

Average waiting time = (Waiting time for all processes) / No. of Processes

= (0+10+17)/3 = 9 milliseconds

Turn Around Time (TAT) = Time spent in Ready Queue (Waiting time) + Execution Time

Turn Around Time (TAT) for P1 = 10ms (0+10)

Turn Around Time (TAT) for P3 = 17ms (10+7)

Turn Around Time (TAT) for P2 = 22ms (17+5)

Average Turn Around Time (TAT) = TAT of all processes / No. of Processes

= (10+17+22)/3 = 16.33 milliseconds

**Example 2:** Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time a Turn Around Time for the above example if a new process P4 with estimated completion time 6ms priority 1 enters the 'Ready' queue after 5ms of execution of P1. Assume all the processes contain only CP operation and no I/O operations are involved.

**Solution:** At the beginning, there are only three processes (P1, P2 and P3) available in the 'Ready' queue and the scheduler picks up the process with the highest priority (P1) for scheduling. The execution sequence is P1, P3, P2. Now process P4 with estimate execution completion time 6ms and priority 1 enters the 'Ready' queue after 5ms of execution of P1. After 10ms of scheduling, Pl terminates and now the scheduler again sorts the 'Ready queue for process with highest priority. Since the priority for P4 (priority 1) is higher than that of P3 (priority 2 which was supposed to be run after the completion of P1 as per the 'Ready' queue available at the beginning of execution scheduling, P4 is picked up for executing. Due to the arrival of the process P4 with priority the 'Ready' queue is resorted in the order P1, P4, P3, P2. At the beginning it was P1, P3, P2. The execution sequence now changes as per the following diagram

| | | | |
|---|---|---|---|
| **P1** | **P4** | **P3** | **P2** |

The waiting time for all the processes are given as:

Waiting time for P1 = 0ms (P1 starts executing first)

Waiting time for P4 = 5ms (P4 starts executing after completing P1. But P4 arrived after 5ms of execution of P1. Hence it's waiting time=Execution start time - Arrival Time=10-5=5)

Waiting time for P3 = 16ms (P3 starts executing after completing PI and P4)

Waiting time for P2 = 23ms (P2 starts executing after completing P1, P4 and P3)

Average waiting time = (Waiting time for all processes) / No. of Processes

= (0+5+16+23) / 4 = 11 milliseconds

Turn Around Time (TAT) = Time spent in Ready Queue (Waiting time) + Execution Time

Turn Around Time (TAT) for P1 = 10ms (0+10)

Turn Around Time (TAT) for P4 = 11ms (5+6)

Turn Around Time (TAT) for P3 = 23ms (16+7)

Turn Around Time (TAT) for P2 = 28ms (23+5)

Average Turn Around Time (TAT) = TAT of all processes / No. of Processes

= (10+11+23+28)/ 4 = 18 milliseconds

## Preemptive Scheduling

In preemptive scheduling, every task in the 'Ready' queue gets a chance to execute. When and how often each process gets a chance to execute is dependent on the type of preemptive scheduling algorithm used for scheduling the processes. In this kind of scheduling, the scheduler can preempt (stop temporarily) the currently executing task/process and select another task from the 'Ready' queue for execution.

A task which is preempted by the scheduler is moved to the 'Ready queue. The act of moving a 'Running process/task into the "Ready' queue by the scheduler, without the processes requesting for it is known as 'Preemption'.

### 1. Preemptive SJF Scheduling/Shortest Remaining Time (SRT)

The preemptive SJF scheduling algorithm sorts the 'Ready' queue when a new process enters the 'Ready' queue and checks whether the execution time of the new process is shorter than the remaining of the total estimated time for the currently executing process. If the execution time of the new process is less, the currently executing process is preempted and the new process is scheduled for execution Preemptive SJF scheduling is also known as Shortest Remaining Time (SRT) scheduling

**Example 1:** Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds respectively enters the ready queue together. A new process P4 with estimated completion time 2ms enters the 'Ready' queue after 2ms. Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time in preemptive SJF/SRT based scheduling algorithm.

**Solution:** At the beginning, there are only three processes (P1, P2 and P3) available in the 'Ready' queue and the SRT scheduler picks up the process with the shortest remaining time for execution completion for scheduling. Now process P4 with estimated execution completion time 2ms enters the 'Ready' queue after 2ms of start of execution of P2. Since the SRT algorithm is preemptive, the remaining time for completion of process P2 is checked with the remaining time for completion of process P4. The remaining time for completion of P2 is 3ms which is greater than that of the remaining time for completion of the newly entered process P4 (2ms). Hence P2 is preempted and P4 is scheduled for execution. P4 continues its execution to finish since there is no new process entered in the 'Ready' queue during its execution. After 2ms of scheduling P4 terminates and now the scheduler again sorts the 'Ready' queue based on the remaining time for completion of the processes present in the 'Ready' queue. The execution sequence now changes as per the following diagram

| | | | | |
|---|---|---|---|---|
| **P2** | **P4** | **P2** | **P3** | **P1** |

The waiting time for all the processes are given as:

Waiting time for P2 = 0ms + (4-2) ms = 2ms (P2 starts executing first and is interrupted by P4 and has to wait till the completion of P4 and has to wait till the completion of P4)

Waiting time for P4 = 0ms (P4 starts executing after preempting P2)

Waiting time for P3 = 7ms (P3 starts executing after completing P4 and P2)

Waiting time for P1= 14ms (P1 starts executing after completing P2, P4 and P3)

Average waiting time = (Waiting time for all processes) / No. of Processes

= (0+2+7+14) / 4 = 5.75 milliseconds

Turn Around Time (TAT) = Time spent in Ready Queue (Waiting time) + Execution Time

Turn Around Time (TAT) for P2 = 7ms (2+5)

Turn Around Time (TAT) for P4 = 2ms (0+2)

Turn Around Time (TAT) for P3 = 14ms (7+7)

Turn Around Time (TAT) for P2 = 24ms (14+10)

Average Turn Around Time (TAT) = TAT of all processes / No. of Processes

= (7+2+14+24)/ 4 = 11.75 milliseconds

**2. Round Robin (RR) Scheduling:** In Round Robin scheduling, each process in the 'Ready' queue is executed for a pre-defined time slot. The execution starts with picking up the first process in the 'Ready' queue (see Fig.). It is executed for a pre-defined time and when the pre-defined time elapses or the process completes (before the pre-defined time slice), the next process in the 'Ready' queue is selected for execution. This is repeated for all the processes in the 'Ready' queue. Once each process in the 'Ready' queue is executed for the pre-defined time period, the scheduler comes back and picks the first process in the "Ready' queue again for execution. The sequence is repeated.

**Example 1:** Three processes with process IDs P1, P2, P3 with estimated completion time 6, 4, 2 milliseconds respectively enters the ready queue together in the order P1, P2, P3, Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time in RR algorithm with Time slice = 2 ms.

**Solution:** The scheduler sorts the 'Ready' queue and picks up the first process PI from the 'Ready' queue and executes it for the time slice 2ms. When the time slice is expired, PI is preempted and P2 is scheduled for execution. The Time slice expires after 2ms of execution of P2. Now P2 is preempted and P3 is picked up for execution. P3 completes its execution within the time slice and the scheduler picks P1 again for execution for the next time slice. This procedure is repeated till all the processes are serviced. The order in which the processes are scheduled for execution is represented as

| P1 | P2 | P3 | P1 | P2 | P1 |
|---|---|---|---|---|---|

The waiting time for all the processes are given as:

Waiting time for P1 = 0 + (6 - 2) + (10 - 8) = 6ms (P1 starts executing first and waits for two time slices to get execution back and again 1 time slice for getting CPU time)

Waiting time for P2 = (2 - 0) + (8 - 4) = 6ms (P2 starts executing after P1 executes for 1 time slice and waits for two time slices get the CPU time)

Waiting time for P3 = (4 – 0) = 4ms (P3 starts executing after completing the first time slices for P1 and P2 and completes its execution in a single time slice)

Average waiting time = (Waiting time for all processes) / No. of Processes

= (6+6+4) / 3 = 5.33 milliseconds

Turn Around Time (TAT) = Time spent in Ready Queue (Waiting time) + Execution Time

Turn Around Time (TAT) for P1 = 12ms (6+6)

Turn Around Time (TAT) for P2 = 10ms (6+4)

Turn Around Time (TAT) for P3 = 6ms (4+2)

Average Turn Around Time (TAT) = TAT of all processes / No. of Processes

= (12+10+6)/ 3 = 9.33 milliseconds

**3. Priority Based Scheduling:** Priority based preemptive scheduling algorithm is same as that of the non-preemptive priority based scheduling except for the switching of execution between tasks. In preemptive scheduling, any high priority process entering the 'Ready' queue is immediately scheduled for execution whereas in the non-preemptive scheduling any high priority process entering the 'Ready' queue is scheduled only after the currently executing process completes its execution or only when it voluntarily relinquishes the CPU.

**Example 1:** Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds and priorities 1, 3, 2 (0-highest priority, 3 lowest priority) respectively enters the ready queue together. A new process P4 with estimated completion time 6 ms and priority O enters the 'Ready' queue after 5 ms of start of execution of Pl. Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time in preemptive priority based scheduling algorithm.

**Solution:** At the beginning, there are only three processes (P1, P2 and P3) available in the 'Ready' queue and the scheduler picks up the process with the highest priority (P1) for scheduling. Now process P4 with estimated execution completion time 6 ms and priority 0 enters the 'Ready' queue after 5 ms of start of execution of P1. Since the scheduling algorithm is preemptive, P1 is preempted by P4 and P4 runs to completion. After 6 ms of scheduling, P4 terminates and now the scheduler again sorts the 'Ready' queue for process with highest priority. Since the priority for P1 (priority 1), which is preempted by P4 is higher than that of P3 (priority 2) and P2 ((priority 3), Pl is again picked up for execution by the scheduler. The execution sequence is as per the following:

| P1 | P4 | P1 | P3 | P2 |
|---|---|---|---|---|

The waiting time for all the processes are given as:

Waiting time for P1 = 0ms + (11-5) ms = 6ms (P21starts executing first and gets preempted by P4 after 5ms and again gets CPU after completion of P4)

Waiting time for P4 = 0ms (P4 starts executing immediately after preempting P1)

Waiting time for P3 = 16ms (P3 starts executing after completing P1 and P4)

Waiting time for P2= 23ms (P2 starts executing after completing P1, P4 and P3)

Average waiting time = (Waiting time for all processes) / No. of Processes

= (6+0+16+23) / 4 = 111.25 milliseconds

Turn Around Time (TAT) = Time spent in Ready Queue (Waiting time) + Execution Time

Turn Around Time (TAT) for P1 = 16ms (6+10)

Turn Around Time (TAT) for P4 = 6ms (0+6)

Turn Around Time (TAT) for P3 = 23ms (16+7)

Turn Around Time (TAT) for P2 = 28ms (23+5)

Average Turn Around Time (TAT) = TAT of all processes / No. of Processes

= (16+6+23+28)/ 4 = 18.25 milliseconds