- Operating System basics
- Task Process and Threads
- Multiprocessing and Multitasking
- Task Scheduling
- Task Synchronization
- Device Drivers

12 marks

6.1 Operating System basics

The operating system acts as a bridge between the user applications/tasks and the underlying system resources through a set of system functionalities and services. The primary function of an operating system is

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

The following figure shows the basic components of an operating system and their interfaces with rest of the world.

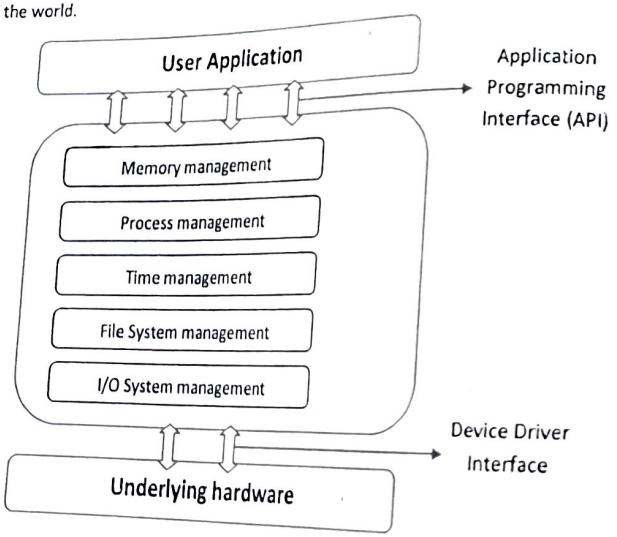


Figure 6.1: Operating System Architecture

Comparison of General Purpose OS (GPOS) with Real Time OS (RTOS)

General Purpose Operating System is software that manages all the system resources and provides common service to all programs running in the system. In case of real time operating system, along with management and services it performs certain function within a specified time constraint. However, both operating systems provide a number of services to application programs and users. Application Programming Interfaces (API) or system calls are the medium through which the services are accessed by the applications.

The differences between GPOS and RTOS can be clarified using following parameters.

- **Deterministic nature**

  RTOS are deterministic in nature; the time required to execute the services is fixed. However, there may not be fixed time defined for any service in case of GPOS.

- **Task Scheduling**

  RTOS uses priority based preemptive scheduling, while scheduling in GPOS is defined so as to achieve high throughput. In RTOS, high priority process execution will override the low priority ones. In GPOS, high priority process may be delayed to perform several low priority tasks.

- **Time Critical systems**

  RTOS is used in time critical systems in which delay in processing can result in undesirable consequences. However, GPOS are implemented in non time critical systems.

- **Preemptive Kernel**

  The kernel of an RTOS is preemptive where as a GPOS kernel is non preemptive. In preemptive kernel, the high priority user process can preempt a kernel call. In other words, the execution of low priority system process can be stopped by high priority user process.

- **Priority Inversion Problem**

  Priority Inversion problem is seen in RTOS in which the high priority task has to wait for the shared resource occupied by low priority task. This results in execution of low priority task first rather than high priority task.

A. The Kernel

The kernel is the core of the operating system and is responsible for managing the system resources and the communication among the hardware and other system services. It acts as the abstraction layer between system resources and user applications. The kernel contains different services for handling the following.

Process Management

It includes setting up the memory space for the process, loading the process's code into the memory space, allocating system resources, scheduling and managing the execution of the process, setting up and managing the process control block (PCB), Inter Process Communication and Synchronization, process termination/deletion, etc.

Primary Memory Management

The term primary memory refers to the volatile memory (RAM) where processes are loaded and variables and shared data associated with each process are stored. The Memory Management Unit (MMU) of the kernel is responsible for

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

File System Management

File is a collection of related information. A file could be a program, text files, word documents, audio/video files, etc. Each of these files differs in the kind of information they hold and the way in which the information is stored. The file operation is a useful service provided by the OS. The file system management service of Kernel is responsible for

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

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. In a well-structured OS, the direct assessing of I/O devices are not allowed and the access to them are provided through a set of Application Programming Interfaces (APIs) exposed by the kernel. The kernel maintains a list of all I/O devices of the system. The list may be available in advance and recent kernel dynamically updates the list of available devices. The service 'Device Manager' of the kernel is responsible for handling all I/O device related operations. The kernel talks to the I/O device through a set of low level system calls, which are implemented in a service, called device drivers. 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

Secondary Storage Management

The secondary storage management deals with managing the secondary storage memory devices that are connected to the system. Secondary memory is used as backup medium for programs and data since the main memory is volatile. In most systems, the secondary storage is kept in disks (Hard Disk). The secondary storage management service of kernel deals with

- Disk storage allocation
- Disk scheduling (time interval at which the disk is activated to backup data)
- Free Disk space management

Protection Systems

Modern operating systems are designed in such a way to support multiple users with different levels of access permissions (For example: Administrator, Standard, Restricted, Guest, etc). Implementing security policies to restrict the access to both user and system resources by different applications or processes or users, one user may not be allowed to view or modify the whole/portions of another user's data or profile details. Some application may not be granted with permission to make use of some of the system resources.

Interrupt Handler

Kernel provides a mechanism to handle all external/internal interrupts generated by the system. Based upon the priority of the interrupt the process either runs in the foreground or background. Depending on the type of operating system, a kernel may contain lesser number of services of more number of services which may include network communication, network management, user-interface graphics, timer services (delays, timeouts, etc.), error handler, database management, etc.

B. Kernel Space and User Space

The program code corresponding to the kernel applications/services are kept in a contiguous area of primary memory and are protected from un-authorized access by user programs/applications. The memory space at which the kernel code is located is known as 'Kernel Space'. Similarly, all user applications are loaded to a specific area of primary memory and this memory area is referred as 'User Space'. User space is the memory area where user applications are loaded and executed. The partitioning of memory into kernel and user space is purely OS dependent.

C. Types of Kernel

Based on the kernel architecture/design, kernels can be classified into Monolithic and Micro.

Monolithic Kernel

In Monolithic Kernel architecture, all kernel services run in the kernel space. Here all kernel modules run within the same memory space under a single kernel thread. It runs all basic system services and provides powerful abstraction of the underlying hardware. Amount of context switches and messaging involved are greatly reduced which makes it run faster than microkernel. The major drawback of monolithic kernel is that any error or failure in any one of the kernel modules leads to the crashing of the entire kernel application. The inclusion of all basic services in kernel space leads to different drawbacks such as requirement of large kernel size, lacking extensibility, poor maintainability. LINUX, SOLARIS, MS-DOS kernels are examples of monolithic kernel.

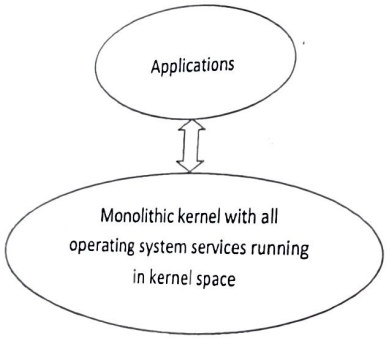


Figure 6.2: The Monolithic Kernel Model

Microkernel

The microkernel design incorporates only the essential set of operating system services such as communication and I/O control into the kernel. The rest of the operating system services are implemented in programs known as 'Servers' which runs in user space. It is more stable than monolithic as the kernel is unaffected even if the server fails. Memory Management, process Management, timer systems and interrupt handlers are the essential services, which forms the part of microkernel. Microkernel based design approach offers the following benefits.

- **Robustness**: If a problem is encountered in any of the service, which runs as 'Server' application, the same can be reconfigured and re-stated without the need for re-starting the entire OS.

- **Configurability**: services can be changed, updated without corrupting the essential services residing within the microkernel.

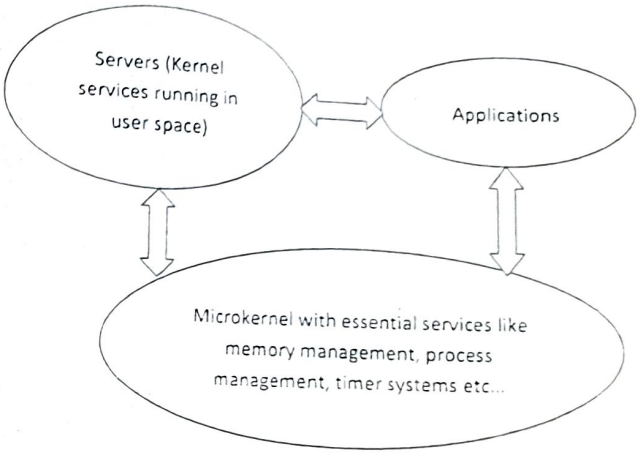


Figure 6.3: The Microkernel Model

6.2 Task Process and Threads

A task is defined as a program in execution and related information maintained by OS for that 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.

A. Process

A process is an instance of a program or part of program in execution. A process requires various system resources such as the CPU for executing the process, memory for storing the code corresponding to the process and associated variables, I/O devices for information exchange etc. A program by itself is not a process; a program is a passive entity, such as a file containing a list of instructions stored on the disk (executable file). A process is an active entity. A program becomes a process when an executable file is loaded into memory.

Structure of a process

A process 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. From a memory perspective, the memory occupied by the process is separated into three regions, stack memory, data memory and code memory.

The stack memory holds all temporary data such as variables local to the process. Data memory holds all global data for the process. The code memory contains the program code (instructions) corresponding to the process.

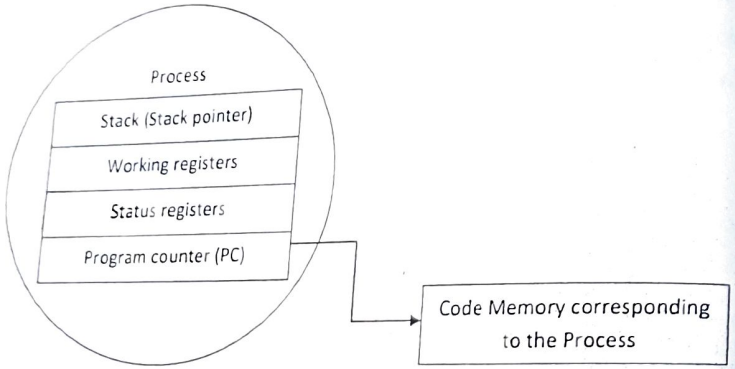


Figure 6.4: Structure of a Process

Process States and 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'.

- Created State: it is the state at which a process is being created. The operating system recognizes a process in the created state but no resources are allocated to the process.
- Ready State: It is the state, where a process is loaded into the memory and awaiting the processor time for execution. The process is placed in the ready list queue maintained by the OS.
- Running State: It is the state where the source code instructions corresponding to the process are being executed. The process execution happens in this state.

- Blocked/Waiting State: it refers to a state at where a running process is temporarily suspended from execution and does not have immediate access to resources. The blocked state might be invoked by various conditions like: the process enters a wait state for an event to occur or waiting for getting access to a shared resource.
- Terminated/Completed State: It is a state where the process completes its execution.

Different OS kernel can have different name for the state associated with a task. Created state may be stated as dormant state, waiting state may be restated as Pending state and so on.

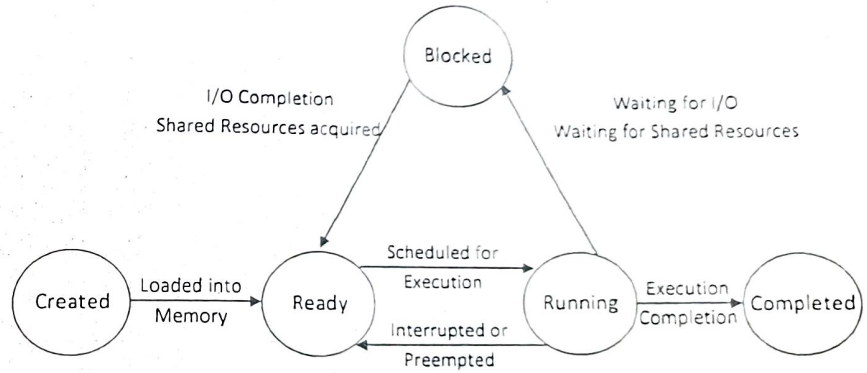


Figure 6.5: Process states and state transition representation

Process Control Block (PCB)

Each process is represented in the OS by a process control block. A PCB serves as a repository for any information that may vary form process to process. A PCB contains many pieces of information associated with a specific process.

- Process state: The state may be new, ready, running, waiting/blocked/pending or completed.
- Program counter: It indicates the address of next instruction to be executed for current process.
- CPU registers: They include accumulators index registers, stack pointers, general purpose registers along with any status registers. The content of PC along with the state information of a process must be saved when an interrupt occurs.

- CPU Scheduling Information: This information includes the process priority and the pointers to the scheduling queues
- Memory management Information: This information includes the value of the base registers, page tables depending upon the memory system used by the OS.
- Accounting information: This information includes the amount of CPU time, time limits and process numbers.
- I/O status information: It includes the list of I/O devices allocated to a process.

B. Threads

A thread, basic unit of CPU utilization, is a single sequential flow of control within a process. A process can have many threads of execution. Different threads which are part of process share the data memory, code memory and the heap memory.

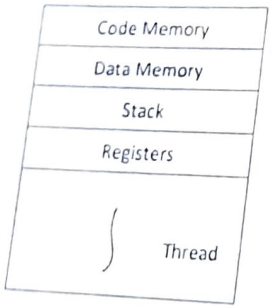


Figure 6.6: Single-Threaded Process

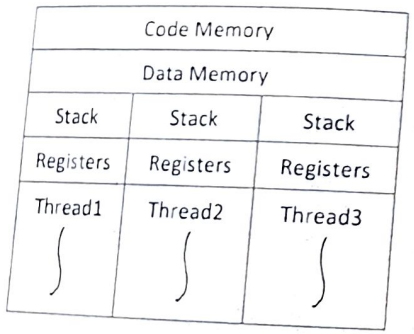


Figure 6.7: Multi-Threaded Process

However, the threads maintain their own thread status (CPU register value), Program Counter (PC) and stack. If a process has multiple threads of control, it can perform more than one task at a time. It is called a multi threaded process. If a process has a single thread of control it can perform a single task and is called single threaded process.

Concept of Multithreading

A process contain various sub-operations like getting input from I/O devices connected to the processor, performing some internal calculations/operations, updating some I/O devices etc. If all the sub-functions of a task are executed in sequence, the CPU utilization may not be efficient. For example, if the process I waiting for a user input, the CPU enters the wait state for the event, and for

the process execution also enters a wait state. If a process is split into different threads carrying out the different sub-functionalities of the process, the CPU can be effectively utilized and when the thread corresponding to the I/O operation enters the wait state, another thread which do not require the I/O event for their operation can be switched into execution. This leads to more speedy execution of the process and the efficient utilization of the processor time and resources.

The benefits of multi-threaded can be broken down into the following major categories:

- **Responsiveness**: Multi-threading on interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.
- **Economical**: Process creation is costly in terms of allocating memory and resources. Multiple thread creation within a process is economical because threads share the resources of the process to which they belong (code, data, heap memory). Creation of threads and context-switch of threads is economical.
- **Utilization of multiprocessor architecture**: The benefits of multithreading can be greatly increased in a multi processor architecture, where threads may be running in parallel in different processors. A single threaded process can only run on one processor, no matter how many processors are available. Multi threading on a multiprocessor machine increases concurrency.
- **Efficient CPU utilization**: CPU is engaged all the time. Since a process is split into different threads, when a thread enters a wait/block state, the CPU can be utilized by other threads of the process. This speeds up the execution of a process.

C. User level & Kernel level threads

User Level Threads

The user level threads don't have kernel/OS support and they exists only in a running process. Even if a process contains multiple user level threads, the OS treats it as a single thread. It is the responsibility of the process to schedule each thread as and when ever required. User level threads of a process are non-preemptive at the thread level from the OS perspective.

Kernel Level Threads

These are individual units of execution, which the OS treats as separate threads. The OS interrupts the execution of the currently running kernel thread and switches the execution of another kernel

thread based on the scheduling policies implemented by the OS. Kernel level threads are pre-emptive.

Relationship between User level thread and Kernel level thread

There are many ways for binding/connecting user level threads with kernel level threads.

Many to One model: Many user level threads are mapped to a single kernel thread. The kernel treats all user level threads as single thread and the execution switching among the user level threads happens when a currently executing user level thread voluntarily blocks itself or relinquishes the CPU.

One to One model: Each user level thread is bonded to a kernel/system level thread. It provides more concurrency than the many to one model by allowing another thread to run when a thread makes a blocking system call. It allows multiple threads to run in parallel on multiprocessor. Creating a user level thread requires creating a corresponding kernel level thread.

Many to many model: It multiplexes many user level threads to a smaller or equal no of kernel level threads. Developers can create as many user level threads as necessary and the corresponding kernel level threads can run in parallel on a multiprocessor. When a thread performs a blocking system call, the kernel can schedule another thread for execution.

D. Thread Libraries

A thread library provides the programmer with an API for creating and managing threads. There are two primary ways of implementing thread library.

The first approach is to provide a library entirely by the user space with no kernel/OS support. All code and data structure for library exists in user space. This means that invoking a function in the library results in a local function call in user space and not a system call.

The second approach is to implement a kernel level library supported directly by the OS. In this case, code and data structure for the library exists in the kernel space. Invoking a function in the API for the library, results in a system call to the kernel.

There are three main thread libraries that are used today.

- **POSIX threads:** POSIX stands for Portable OS Interface. The POSIX standard for defining API, for thread creation and management, is pthreads. Pthreads library defines the set of POSIX thread creation and management functions in C language. Pthread may be provided as either a user level or a kernel level library.

Thread Call	Description
pthread_create()	Creates a new thread
pthread_exit()	Terminates the calling thread
pthread_join()	Blocks the current thread and waits until the completion of the thread pointed by it.
pthread_yield()	Releases the CPU to let another thread run
pthread_attr_init()	Create and initialize a thread's attributes
pthread_attr_destroy()	Releases a thread's attributes

- **Win32 threads:** Win32 threads are supported by various flavors of the windows OS. The win32 API libraries provide a standard set of win32 thread creation and management function. Win32 thread library is a kernel level library.

Thread Call	Description
CreateThread()	Creates a new thread
SuspendThread()	Temporarily suspends thread execution
ResumeThread()	Wakes up a suspended thread
ExitThread()	It terminates a thread and allocates the thread stack resources along with other resources that were held by it.

- **Java threads:** Java threads are the threads supported by Java programming language. The java thread class 'Thread' is defined in the package 'java.lang'. The java thread API allows thread creation and management directly in the java programs. Since a java virtual machine runs on the top of host operating system, the JAVA thread API on the top of a host OS, the JAVA thread API typically implemented using a thread library available on the host system. This means that on windows system, java threads are typically implemented using the win32 API. UNIX and LINUS systems user pthreads.

Thread Call	Description
Start()	Allocates memory and initializes a new thread in JAVA
Yield()	A running thread enters the ready state
Sleep()	A thread enters the suspend state
Wait()	A thread enters a blocked state
Stop()	Terminates a thread and de-allocates resources

E. Difference between Thread and Process

Thread	Process
It is a single unit of execution and is a part of the process	A process is a program in execution and combines one or more threads
A thread shares the code, data, heap memory with other threads of the same process	A process has its own code, data and stack memory
A thread cannot live independently	A process contains at least one thread
Threads are very inexpensive to create	Processes are expensive to create. Involves many OS overhead
Context switching is inexpensive and fast	Context switching is complex and involves lot of OS overhead and is comparatively slower.
If a thread expires, its stack is reclaimed by the process.	If a process dies, the resources allocated to it are reclaimed by the OS and all the associated threads of the process also dies.

6.3 Multiprocessing and Multitasking

Multiprocessing describes the ability to execute multiple processes simultaneously. Systems which are capable of performing multiprocessing are called multiprocessor system. Multiprocessor systems possess multiple CPUs/processors and can execute multiple processes simultaneously. The ability of an OS to have multiple programs in memory, which are ready for execution, is referred as multiprogramming.

In a uniprocessor system, it is not possible to execute multiple processes simultaneously. However, it is possible for a uniprocessor system to achieve some degree of pseudo parallelism in the execution of multiple processes by switching the execution among different processes. The ability of an operating system to hold multiple processes in memory and switch the processor from executing

one process to another process is known as multitasking. Multitasking creates the illusion of multiple tasks executing in parallel. Multitasking involves 'Context switching', 'Context saving' and 'Content retrieval'.

A. Context Switching

Each task may exist in any one of the different states (running, ready, blocked, etc). During the execution of an application program, individual tasks are continuously changing from one state to another. At any point of the execution, only one task is in running mode. During the process of state change, CPU control changes from one task to another, context of the to-be-suspended task will be saved while context of the to-be-executed task will be retrieved.

The process of saving the context of a task being suspended and restoring the context of a task being resumed is called context switching.

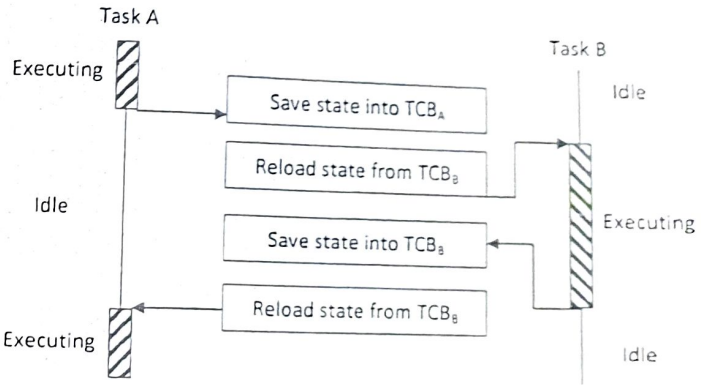


Figure 6.8: Simple Context Switching Diagram

Context Saving is the act of saving the current contents which contains the context details (register details, memory details, system resource usage details, etc for the currently running process at the time of CPU switching. Context retrieval is the process of retrieving the saved context details for a process which is going to be executed due to CPU switching. Context switch time is pure overhead because the system does no useful work while switching.

B. Types of Multitasking

Multitasking involves the switching of execution among multiple tasks. Depending on how the switching act is implemented, multitasking can be classified into different types.

Co-operative Multitasking: It 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. Any task/process can hold the CPU as much time as it wants. If the currently executing task is non-cooperative, the other tasks may have to wait for a long time to get the CPU.

Preemptive Multitasking: It 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. 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.

Non-preemptive Multitasking: In non-preemptive multitasking, the process/task, which is currently given the CPU time, is allowed to execute until it terminates or enters the 'Blocked/Wait' state, waiting for an I/O or system resource. In co-operative multitasking, the currently executing process/task need not relinquish the CPU when it enters the 'Blocked/Wait' state, whereas in non-preemptive multitasking the currently executing task relinquishes the CPU when it waits for an I/O or system resource or an event to occur.

6.4 Task Scheduling

Multitasking involves the execution switching among the different tasks. 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 scheduling policies are implemented in an algorithm and it is run by the kernel as a service. The process scheduling decision may take place when a process switches its state to

- Ready state form Running state
- Blocked/Wait state from Running state
- Ready state from Blocked/Wait state
- Completed state

The selection of a scheduling criterion should consider the following factors

- **CPU utilization**: the scheduling criterion should always make the CPU utilization high. CPU utilization is a direct measure of how much percentage of the CPU is being utilized.
- **Throughput**: This gives an indication of the number of processes executed per unit of time. The throughput for a good scheduler should always be higher.

- **Turnaround time**: It is the amount of time taken by a process for completing its execution. It includes the time spent by the process for waiting for the main memory, time spend in the ready queue, time spent on completing the I/O operations, and the time spend in execution. The turnaround time should be a minimal for a good scheduling algorithm.
- **Waiting Time**: It is the amount of time spent by a process in the 'Ready' queue waiting to get the CPU time for execution. The waiting time should be minimal for a good scheduling algorithm.
- **Response time**: It is the time elapsed between the submission of a process and the first response, for a good scheduling algorithm, the response time should be as least as possible.

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 of the system.
- **Ready Queue**: contains all the processes, which are ready for execution and waiting for CPU to get their turn for execution
- **Device Queue**: contains the set of processes, which are waiting for an I/O device.

The scheduling algorithm can be classified as:

A. Non-preemptive Scheduling

It is employed in systems which implements 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 resources. Various types of non-preemptive scheduling are listed below.

- **First Come First Served (FCFS) / FIFO Scheduling**: The 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. E.g. ticketing reservation system where people need to stand to a queue and the first person standing in the queue is serviced first.
- **Last Come First Served (LCFS) / LIFO scheduling**: The LCFS scheduling algorithm allo allocates CPU time to the Processes based on the order in which they are entered in the ready queue. The last entered process is services first.
- **Shortest Job First (SJF) scheduling**: SJF scheduling algorithm sorts the ready queue eac time a process relinquishes the CPU to pick the process with shortest estimated completio

time. The process with the shortest estimated run time is scheduled first, followed by the nest shortest process, and so on.

- **Priority Based Scheduling**: This scheduling algorithm ensures that a process with high priority is serviced at the earliest compared to other low priority processes in the ready queue. The SJF algorithm can be viewed as a priority based scheduling where each task is prioritized in the order of the time required to complete the task. Another way of priority assigning is associating a priority to the task/process at the time of creation of the task/process. The priority is the number ranging from 0 to the maximum priority supported by the OS. For windows CE operating system a priority number 0 indicates the highest priority.

B. Preemptive Scheduling

Preemptive scheduling is employed in systems, which implements preemptive multitasking model. In this 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. In this scheduling method, the scheduler can preempt (stop temporarily) the currently executing process and select another task from the ready queue for execution. The task which is preempted by the scheduler is moved to the ready queue. The act of moving a running process into a ready queue by the scheduler, without the processes requesting for it is known as preemption. The different types of preemptive scheduling adopted in process scheduling are explained below.

- **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.
- **Round Robin Scheduling**: In this scheduling method, 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. It is executed for a pre defined time slice 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. Once each process in the ready queue is executed for

the pre-defined time period, the scheduler picks the first process in the ready queue again for execution and the sequence is repeated. So, the round robin scheduling is similar to the FCFS scheduling but time slice preemption is added to switch the execution between the processes in the ready queue.

- **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 processes. 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.

6.5 Task Synchronization

In a multitasking environment, multiple processes run concurrently and share the system resources. When two processes try to access display hardware connected to the system or two processes try to access a shared memory area where one process tries to write to a memory location while the other process is trying to read from this. Then, an issue will arise and hence each process must be made aware of the access of the shared resources. The act of making processes aware of the access of the shared resources by each process to avoid conflicts is known as task/process synchronization. Various synchronization issues may arise if processes are not synchronized properly.

Task Communication/Synchronization Issues

A. Racing

Racing or Race condition is the situation in which multiple processes compete each other to access and manipulate shared data concurrently. In a race condition the final value of the shared data depends on the process which acted on the data finally.

Suppose that two processes A and B have access to a shared variable Count:

Process A: Count = Count + 5

Process B: Count = Count + 10

Assume that process A and process B are executing concurrently in a time-shared, multi-programmed system.

Each statement requires several machine level instructions such as

For Count = Count + 5

A1: Load Ra, Count

A2: Add Ra, 05
A3: Store Count, Ra
For Count = Count + 10
B1: Load Rb, Count
B2: Add Rb, 10
B3: Store Count, Rb

In a time-shared or multi-processing system the exact instruction execution order cannot be predicted.

Scenario 1	Scenario 2
A1: Load Ra, Count	A1: Load Ra, Count
A2: Add Ra, 05	A2: Add Ra, 05
A3: Store Count, Ra	Context Switch
Context Switch	B1: Load Rb, Count
B1: Load Rb, Count	B2: Add Rb, 10
B2: Add Rb, 10	B3: Store Count, Rb
B3: Store Count, Rb	Context Switch
	A3: Store Count, Ra
Count is increased by 15	Count is increased by 5

B. Deadlock

A race condition produces incorrect results whereas a deadlock condition creates a situation where none of the processes are able to make any progress in their execution, resulting in a set of deadlocked processes. In its simplest form, 'deadlock' is the condition in which a process is waiting for a resource held by another process which is waiting for a resource held by the first process. For instance, process A holds a resource x and it wants a resource y held by process B. Process B is currently holding resource y and it wants the resources x which is currently held by process A. None of the competing process will be able to access the resources held by other processes since they are locked by the respective processes.

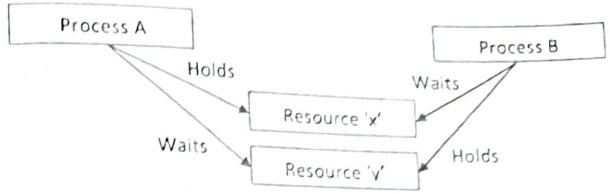


Figure 6.9: Scenarios leading to Deadlock

Coffman conditions: The different conditions favoring a deadlock situation are listed below

- Mutual Exclusion: The criteria that only one process can hold a resource at a time. Processes should access shared resources with mutual exclusion.
- Hold and Wait: The condition in which a process holds a shared resource by acquiring the lock controlling the shared access and waiting for additional resources held by other processes.
- No Resource Preemptive: The criteria that operating system cannot take back a resource from a process which is currently holding it and the resource can only be released voluntarily by the process holding it.
- Circular Wait: A process is waiting for a resource which is currently held by another process which in turn is waiting for a resource held by the first process. In general, there exists a set of waiting process P0, P1 … Pn with P0 is waiting for a resource held by P1 and P1 is waiting for a resource held by P0, …, Pn is waiting for a resource held by P0 and P0 is waiting for a resource held by Pn and so on… This forms a circular wait queue.

Deadlock Handling

A smart OS may foresee the deadlock condition and will act proactively to avoid such a situation. The OS may adopt any of the following techniques to detect and prevent deadlock conditions.

- Ignore Deadlocks: Always assume that the system design is deadlock free. This is acceptable for the reason the cost of removing a deadlock is large compared to the chance of deadlock to occur.
- Detect and Recover: This approach suggests the detection of a deadlock situation and recovery from it. OS keeps a resource graph in their memory. The resource graph is updated on each resource request and release. A deadlock condition can be detected by analyzing the resource graph by graph analyzer algorithms. Once a deadlock condition is detected, the system can terminate a process or preempt the resource to break the deadlocking cycle.

- **Avoid Deadlocks**: Deadlock is avoided by the careful resource allocation techniques by the operating system.
- **Prevent Deadlocks**: Prevent the deadlock condition by negating one of the four conditions favoring the deadlock situation.
  - Ensure that a process does not hold any other resources when it requires a resource
  - Ensure resources preemption.

C. Livelock

In a livelock condition, a process changes its state with time but is unable to make any progress in the execution completion. While in deadlock a process enters a wait state for a response and continues in that state forever without making any progress in the execution. For example, two people attempting to cross each other in a narrow corridor. Both the person moves towards each side of the corridor to allow the opposite person to cross. Since the corridor is narrow, none of them are able to cross each other. Here both of the persons perform some action but still they are unable to achieve their target.

D. Starvation

In the multitasking context, starvation is the condition in which a process does not get the resources required to continue its execution for a long time. As time progresses the process starves on resource. Starvation may arise due to various conditions like byproduct of preventive measures of deadlock, scheduling policies favoring high priority tasks and tasks with shortest execution time, etc.

Task Synchronization techniques

Task synchronization is essential for:

- Avoiding conflicts in resource access in a multitasking environment.
- Ensuring proper sequence of operation across processes.
- Communicating between processes.

The code memory area which holds the program instructions for accessing a shared resource, shared variables is known as critical section. In order to synchronize the access to shared resources, the access to the critical section should be exclusive.

Consider two processes Process A and Process B running on a multitasking system. Process A is currently running and it enters its critical section. Before Process A completes its operation in the critical section, the scheduler preempts process A and schedules Process B for execution. Process B also contains the access to the critical section which is already in use by Process A. if process B continues its execution and enters the critical section which is already in use by Process A, a racing condition will be resulted. A mutual exclusion policy enforces mutually exclusive access of critical sections.

A. Mutual Exclusion through Busy Waiting/Spin Lock

The Busy Waiting technique uses a lock variable for implementing mutual exclusion. Each process/thread checks this lock variable before entering the critical section. The lock is set to 1 by a process/thread if the process/thread is already in its critical section; otherwise the lock is set to 0.

The major challenge in implementing the lock variable based synchronization is the non-availability of a single atomic instruction which combines the reading, comparing and setting of the lock variable. Most often the three different operations related to the locks, the operation of reading the lock variable, checking its present value and setting it are achieved with multiple low level instructions. The low level implementations of these operations are dependent on the underlying processor instruction set and the compiler in user.

Consider a situation where process 1 read the lock variable and tested it and found that the lock is available and it is about to set the lock for acquiring the critical section. But just before process 1 sets the lock variable, process 2 preempts process 1 and starts executing. Process 2 contains a critical section code and it tests the lock variable for its availability. Since process 1 was unable to set the lock variable, its state is still 0 and process 2 sets it and acquires the critical section. Remember, process 1 was preempted at a point just before setting the lock variable. Now process 1 sets the lock variable and enters the critical section. It violates the mutual exclusion policy and may produce unpredicted results.

The above issue can be effectively tackled by combining the actions of reading the lock variable, testing its state and setting the lock into a single step. This can be achieved with the combined hardware and software support. Most of the processors support a single instruction Test and Set Lock (TSL) for testing and setting the lock variable. The TSL instruction call copies the value of the lock variable and sets it to a nonzero value.

The lock based mutual exclusion implementation always checks the state of a lock and waits till the lock is available. This keeps the processes always busy and forces the processes to wait for the availability of the lock for proceeding further. Hence this synchronization mechanism is known as Busy Waiting. This method is useful in handling scenarios where the processes are likely to be blocked for a shorter period of time on waiting the lock, as they avoid OS overheads on context saving and process re-scheduling. The drawback of Spin Lock based synchronization is that if the lock is being held for a long time by a process and if it is preempted by the OS, the other threads waiting for this lock may have to spin a longer time for getting it. The busy waiting mechanism keeps the process always active, performing a task which is not useful and leads to the wastage of processor time and high power consumption.

B. Mutual Exclusion through Sleep & Wakeup

The Busy waiting mutual exclusion enforcement mechanism used by processes makes the CPU always busy by checking the lock to see whether they can proceed. This results in the wastage of CPU time and leads to high power consumption. This is not affordable in embedded systems powered on battery. In sleep and wakeup mechanism, when a process is not allowed to access the critical section, which is currently being locked by another process, the process undergoes Sleep and enters the blocked state. The process which is blocked on waiting for access to the critical section is awakened by the process which currently owns the critical section. The process which owns the critical section sends a wakeup message to the process, which is sleeping as a result of waiting for the access to the critical section, when the process leaves the critical section. The sleep and wakeup policy for mutual exclusion can be implemented in different ways.

- **Semaphore**: It is a sleep and wakeup based mutual exclusion implementation for share resource access. Semaphore is a system resource and the process which wants to access the share resource can first acquire this system object to indicate the other processes which wants the shared resource that the shared resource is currently acquired by it. The resources which are shared among a process can be either for exclusive use by a process or for using by a number of processes in a time. The display device of an embedded system is a typical example for the shared resource which needs exclusive access by a process. The hard disk of a system is a typical example for sharing the resource among a limited number of multiple processes.

Binary Semaphore (Mutex): The binary semaphore provides exclusive access to shared resource by allocating the resource to a single process at a time and not allowing the other processes to access it when it is being owned by a process. Mutex is a synchronization object provided by OS for process synchronization. Any process can create a mutex object and other processes of the system can use this mutex object at a time. The state of a mutex object is set to signaled when it is not owned by any process, and set to non-signaled when it is owned by any process.

Counting Semaphore: The counting semaphore limit the access of resources to fixed number of processes or threads. It maintains a count between zero and a value. It limits the usage of the resource to the maximum value of the count supported by it. The state of the counting semaphore object is set to signaled when the count of the object is greater than zero. The count associated with a semaphore object is decremented by one when a process acquires it and the count is incremented by one when a process releases the semaphore object. The state of the semaphore object is set to non-signaled when the semaphore is acquired by the maximum number of processes that the semaphore can support.

- **Events**: Event object is a synchronization technique which uses the notification mechanism for synchronization. In concurrent execution we may come across situations which demand the processes to wait for a particular sequence for its operations. A thread/process can wait for an event and another thread/process can set this even for processing by the waiting thread/process. The creating and handling event objects for notification is OS kernel dependent.

Priority Inversion

Priority inversion is the byproduct of the combination of blocking based process synchronization and pre-emptive priority scheduling. It is the condition in which a high priority task needs to wait for a low priority task to release a resource which is shared between the high priority task and the low priority task, and a medium priority task which doesn't require the shared resource continue its execution by preempting the low priority task. Priority based preemptive scheduling technique ensures that a high priority task is always executed first, where as the lock based process synchronization mechanism ensures that a process will not access a shared resource, which is currently in use by another process. The synchronization technique is only interested in avoiding

conflicts that may arise due to concurrent access of the shared resources and not at all bother about the priority of the process which tries to access the shared resource.

Consider a three process A, B, C with priorities High, Medium and Low respectively. Process A and C share a variable X and the access to this variable is synchronized through mutual exclusion. Process C is ready and is picked up for execution by the scheduler and process C tries to access the shared variable X and acquires the semaphore to indicate the other processes that it is accessing the shared variable X. At the same time, process B enters the ready state with higher priority compared to C, so Process C gets preempted and B starts executing. Now if Process A enters the ready state at this point. Process B is preempted and process A is scheduled for execution. Process A involves access of shared variable X which is currently being accessed by process C. So process A is put into blocked state and process B gets the CPU and it continues its execution until it relinquishes the CPU voluntarily or enters a wait state or preempted by another high priority task. The high priority A has to wait till Process C gets a chance to execute and release the semaphore. This produces unwanted delay in the execution of the high priority task which is supposed to be executed immediately when it was ready.

The commonly adopted priority inversion workarounds are:

A. Priority Inheritance

A low priority task that is currently accessing a shared resource requested by a high priority task temporarily inherits the priority of that high priority task, from the moment the high priority task raises the request. Boosting the priority of the low priority task to that of the priority of the task which requested the shared resource holding by the low priority task eliminates the preemption of the low priority task by other tasks whose priority are below that of the task requested the shared resource and thereby reduces the delay in waiting to get the resource requested by the high priority task. The priority of the low priority task which is temporarily boosted to high is brought to the original value when it releases the shared resources. Priority inheritance handles priority inversion at the cost of run time overhead at scheduler. It imposes the overhead of checking the priorities of all tasks which tries to access shared resources and adjust the priority dynamically.

B. Priority Ceiling

In Priority Ceiling, a priority is associated with each shared resource. The priority associated to each resource is the priority of the highest priority task which uses this shared resource. This priority level

is called ceiling priority. Whenever a task accesses a shared resource, the scheduler elevates the priority of the task to that of the ceiling priority of the resource. If the task which accesses the shared resource is a low priority task, its priority is temporarily boosted to the priority of the highest priority task to which the resource is also shared. This eliminates the preemption of the task by other medium priority tasks leading to priority inversion. The priority of the task is brought back to the original level once the task completes the accessing of the shared resource. Priority Ceiling brings the added advantage of sharing resources without the need for synchronization techniques like locks. The priority of the task accessing shared resources is boosted to the highest priority of the task among which the resource is shared; the concurrent access of shared resource is automatically handled. Another advantage is that all the overheads are at compile time instead of run-time.

PRIORITY LIST A: HIGH B: MEDIUM C: LOW	Process C acquires shared variables 'X'	Process B preempts C	Process A preempts B	Process A requires shared variable 'X', Priority of C is increased to High	Process C releases the shared resource, so A starts executing with that resource and the priority of C is lowered to its original value.	Process A completes its execution. B starts executing	Process B completes its execution and C starts its execution.
Process A			Running	Waiting	Running		
Process B		Running	Waiting			Running	
Process C	Running	Waiting		Running	Waiting		Running

Figure 6.10: Illustration of Priority Inheritance

6.6 Device Drivers

It is a piece of software that acts as a bridge between the OS and the hardware. The architecture of OS kernel will now allow direct device access from the user application. All devices related access should flow through OS kernel, and the OS kernel routes it to the concerned hardware peripherals. Device drivers are responsible for initiating and managing the communication with hardware

peripherals. They are responsible for establishing connectivity, initializing hardware (setting up various CPU registers) and transferring data.

Device drives which are part of OS are called built in drivers or on-board drivers. These drivers are loaded by OS at the time of booting the device and are kept in RAM. Device drivers which need to be installed for accessing a device are called installable drivers. Whenever the device is connected, the OS loads the corresponding driver into memory. Driver files are usually in the form of '.dll' files. Drivers can run either in user space or in kernel space. Device drivers which run in user space are called user mode driver and the driver which run in kernel space are called kernel mode drivers.

A device driver implements the following:

Device initialization and interrupt configuration: The driver configures the different registers of the device. The interrupt configuration part deals with configuring the interrupts that needs to be associated with the hardware. The basic interrupt configuration involves:

- Set the interrupt type (Edge triggered or Level triggered), enable the interrupts and set the interrupt priorities.
- Bind the interrupt with an interrupt request (IRQ). The processor identifies an interrupt through IRQ. These IRQs are generated by the Interrupt Controller. In order to identify and interrupt the interrupt needs to be bonded to an IRQ.
- Register an Interrupt Service Routine (ISR) with an IRQ. ISR is the handler for an interrupt. In order to service an interrupt, an ISR should be associated with an IRQ.

Interrupt handling and processing: An interrupt is served based on its priority, and the corresponding ISR is invoked. The processing part of an interrupt is handled in an ISR. The whole interrupt processing can be done by the ISR itself or by invoking an Interrupt Service Thread (IST). The IST performs interrupt processing on behalf of the ISR. Since interrupt processing happens at kernel level, user application may not have direct access to the drivers to pass and receive data.

Client Interfacing: The client interfacing implementation makes use of the Inter Process Communication mechanisms supported by the embedded OS for communicating and synchronizing with user applications and drivers. For example, to inform a user application that an interrupt is occurred and the data received from the device is placed in a shared buffer, the client interfacing code can signal an event.

NUMERICAL EXAMPLES

Example 1: Three processes with process IDs P1, P2, P3 with priorities 2, 3, 0 and estimated completion time 10, 5, 7 milliseconds respectively enter the ready queue together in the order P1, P2, P3. Calculate the Waiting Time and Turn Around Time for each process and also the Average Waiting Time and Average Turn Around Time. Assume there is no I/O waiting for the process. Use the following non-preemptive scheduling algorithms.

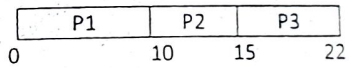
- First Come First Serve Scheduling
- Priority Based Scheduling
- Shortest Job First Scheduling

Solution:

Given information from the question are tabulated as shown below:

Process	Entry Time	Completion Time	Priority	Entered
P1	0	10	2	1st
P2	0	5	3	2nd
P3	0	7	0	3rd

A. First Come First Served Scheduling



Execution Sequence of Processes

Waiting Time = Execution Start Point – Entry Point

Turn Around Time = Completion Point – Entry Point

Waiting Time calculation

P1 = (0-0) = 0ms

P2 = (10-0) = 10ms

P3 = (15-0) = 15ms

Average Waiting Time

= (0+10+15)/3

= 8.33ms

Turn Around Time calculation

P1 = (10-0) = 10ms

P2 = (15-0) = 15ms

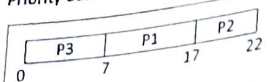
P3 = (22-0) = 22ms

Average Turn Around Time

= (10 + 15 + 22)/3

= 15.67ms

B. **Priority Based Scheduling**



Execution Sequence of Processes

Waiting Time = Execution Start Point – Entry Point

Turn Around Time = Completion Point – Entry Point

Waiting Time calculation

P3 = (0 - 0) = 0ms

P1 = (7 - 0) = 7ms

P2 = (17 - 0) = 17ms

Average Waiting Time

= (0 + 7 + 17)/3

= 8ms

Turn Around Time calculation

P3 = (7 - 0) = 7ms

P1 = (17 - 0) = 17ms

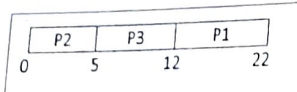
P2 = (22 - 0) = 22ms

Average Turn Around Time

= (7 + 17 + 22)/3

= 15.33ms

C. **Shortest Job First**



Execution Sequence of Processes

Waiting Time = Execution Start Point – Entry Point

Turn Around Time = Completion Point – Entry Point

Waiting Time calculation

P2 = (0 - 0) = 0ms

P3 = (5 - 0) = 5ms

P1 = (12 - 0) = 12ms

Average Waiting Time

= (0 + 5 + 12)/3

= 5.67ms

Turn Around Time calculation

P2 = (5 - 0) = 5ms

P3 = (12 - 0) = 12ms

P1 = (22 - 0) = 22ms

Average Turn Around Time

= (5 + 12 + 22)/3

= 13ms

Example 2: Three processes with process IDs P1, P2, P3 with priorities 0, 1, 3 and estimated completion time 6, 9, 3 milliseconds respectively enter the ready queue together. If a new process P4 (priority 2) with estimated completion time 2ms enters the ready queue after 3ms of execution of P1. Calculate the Waiting Time and Turn around Time for each process and also the Average Waiting Time and Average Turn Around Time. Make use of following non-preemptive scheduling algorithm to solve the problem.

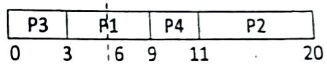
- Shortest Job First (SJF) Scheduling
- Priority Based Scheduling

Solution:

A. **Non - Preemptive SJF Scheduling**

Given information from the question are tabulated as shown below

Process	Entry Time	Completion Time	Priority
P1	0	6	0
P2	0	9	1
P3	0	3	3
P4	3ms after P1 starts	2	2



Execution Sequence of Processes

Waiting Time = (Execution Starting Point – Entry Point)

Turn Around Time = (Completion Point – Entry Point)

Waiting Time calculation

P3 = (0 - 0) = 0ms

P1 = (3 - 0) = 3ms

P4 = (9 - 6) = 3ms

P2 = (11 - 0) = 11ms

Average Waiting Time

= (0 + 3 + 3 + 11)/4

= 4.25ms

Turn Around Time calculation

P3 = (3 - 0) = 3ms

P1 = (9 - 0) = 9ms

P4 = (11 - 6) = 5ms

P2 = (20 - 0) = 20ms

Average Turn Around Time

= (3 + 9 + 5+ 20)/4

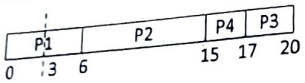
= 9.25ms

Explanation: Entry point for three processes P1, P2 and P3 is same at 0ms but the process P4 enters only after 3ms of execution of P1. So, the entry point for P4 will be at 6ms. Regardless of the shortest completion time of P4, P4 will not halt the execution of P1 as the algorithm is non pre-emptive. However, after the execution of P1, there remain two processes P2 and P4 with completion time 9ms and 2ms respectively. Hence, P4 will start to execute after completion of P1 according to shortest job first scheduling.

B. **Non - Preemptive Priority Based Scheduling**

Given information from the question are tabulated as shown below

Process	Entry Time	Completion Time	Priority
P1	0	6	0
P2	0	9	1
P3	0	3	3
P4	3ms after P1 starts	2	2



Execution Sequence of Processes

Waiting Time = (Execution Starting Point – Entry Point)

Turn Around Time = (Completion Point – Entry Point)

Waiting Time calculation

P1 = (0 - 0) = 0ms

P2 = (6 - 0) = 6ms

P4 = (15 - 3) = 12ms

P3 = (17 - 0) = 17ms

Average Waiting Time

= (0 + 6 + 12 + 17)/4

= 8.75ms

Turn Around Time calculation

P1 = (6 - 0) = 6ms

P2 = (15 - 0) = 15ms

P4 = (17 - 3) = 14ms

P3 = (20 - 0) = 20ms

Average Turn Around Time

= (6 + 15 + 14 + 20)/4

= 13.75ms

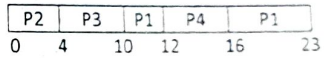
Example 3: Three processes P1, P2, P3 with estimated completion time 9, 4, 6 ms and priorities 1, 3, 2 respectively enters the ready queue together. A new process P4 with estimated completion time 4ms and priority 0 enters the ready queue after 2 ms of start of execution of P1. Calculate the Waiting Time and Turn Around Time for each process. Also Calculate the Average Waiting Time and Average Turn Around Time, using the Preemptive Shortest Job First Scheduling and Priority Based Scheduling.

Solution:

A. **Preemptive SJF Scheduling**

Given information from the question are tabulated as shown below

Process	Entry Time	Completion Time	Priority
P1	0	9	1
P2	0	4	3
P3	0	6	2
P4	2ms after P1 starts	4	0



Execution Sequence of Processes

Waiting Time = (Execution Starting Point – Entry Point) + Halted time

Turn Around Time = (Completion Point – Entry Point)

Waiting Time calculation

P2 = (0 - 0) = 0ms

P3 = (4 - 0) = 4ms

P4 = (12 - 12) = 0ms

P1 = (10 - 0) + (16 - 12) = 14ms

Average Waiting Time

= (0 + 4 + 0 + 14)/4

= 4.5ms

Turn Around Time calculation

P2 = (4 - 0) = 4ms

P3 = (10 - 0) = 10ms

P4 = (16 - 12) = 4ms

P1 = (23 - 0) = 23ms

Average Turn Around Time

= (4 + 10 + 4 + 23)/4

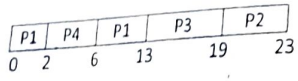
= 10.25ms

Explanation: Entry point for three processes P1, P2 and P3 is same at 0ms but the process P4 enters only after 2ms of execution of P1. So, the entry point for P4 will be at 12ms. At 12ms, there are two processes remaining; P1 with 7ms left to execute and P4 with 4ms. Since P4 is shorter compared to remaining part of P1, P4 will halt the execution of P1 at 12ms and starts its own execution. After P4 completes its execution at 16ms, P1 resumes.

B. **Preemptive Priority based scheduling**

Given information from the question are tabulated as shown below

Process	Entry Time	Completion Time	Priority
P1	0	9	1
P2	0	4	3
P3	0	6	2
P4	2ms after P1 starts	4	0



Execution Sequence of Processes

Waiting Time = (Execution Starting Point – Entry Point) + Halted time

Turn Around Time = (Completion Point – Entry Point)

Waiting Time calculation

P1 = (0 - 0) + (6 - 2) = 4ms

P4 = (2 - 2) = 0ms

P3 = (13 - 0) = 13ms

P2 = (19 - 0) = 19ms

Average Waiting Time

= (4 + 0 + 13 + 19)/4

= 9ms

Turn Around Time calculation

P1 = (13 - 0) = 13ms

P4 = (6 - 2) = 4ms

P3 = (19 - 0) = 19ms

P2 = (23 - 0) = 23ms

Average Turn Around Time

= (13 + 4 + 19 + 23)/4

= 14.75ms

Points to Remember

- When a process entering at the middle of execution does not halt the executing process, then its entry point and start of execution will never be at same point. Hence, its WT is never 0 and TAT is always greater than Completion Time.
- When a process entering at the middle of execution halts the executing process, then its entry point and start of execution will be same. Hence, it's WT = 0 and TAT = Completion Time.