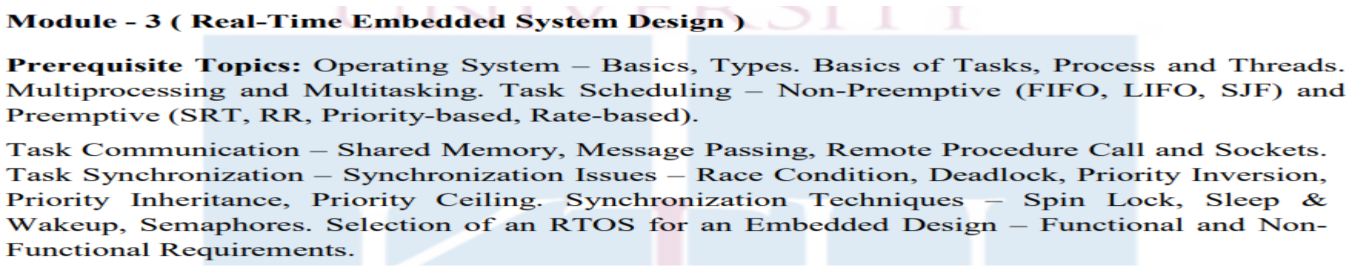


Module - 3 (Real-Time Embedded System Design)

Prerequisite Topics: Operating System – Basics, Types. Basics of Tasks, Process and Threads. Multiprocessing and Multitasking. Task Scheduling – Non-Preemptive (FIFO, LIFO, SJF) and Preemptive (SRT, RR, Priority-based, Rate-based).

Task Communication – Shared Memory, Message Passing, Remote Procedure Call and Sockets. Task Synchronization – Synchronization Issues – Race Condition, Deadlock, Priority Inversion, Priority Inheritance, Priority Ceiling. Synchronization Techniques – Spin Lock, Sleep & Wakeup, Semaphores. Selection of an RTOS for an Embedded Design – Functional and Non-Functional Requirements.

Ref Ch 10 SHIbu KV

REAL TIME OPERATING SYSTEMS

Q .1.What is RTOS and what are its characteristics.(real time operating system).

- A real time is the time, which continuously increments at regular intervals after the start of the system and time for all the activities at difference instances take that time as a reference in the system.
- An RTOS provides running user threads in kernel space so they execute fast.
- An RTOS provides effective handling of interrupt service routine, device drivers
- It provides memory allocation and deallocation.
- Input output management devices,files,mailboxex,pipes and sockets can be made simple with RTOS
- An Operating system is called "Real-Time Operating System" (RTOS) only if it has following ***characteristics:***

i.Deterministic

- An OS is said to be deterministic if the worst case execution time of each of the system calls is calculable.
- The data sheet of an OS should publish the real-time behavior of its RTOS provides average, minimum and maximum number of clock cycles required by each systemcall.

ii.InterruptLatency

Interrupt Latency is the total length of time from an interrupt signal's arrival at the processor to the start of the associated interrupt service routine.

iii.ContextSwitch

Context Switch is important because it represents overhead across your entire system.

Q.2. What are the basics of an operating system and the need for an 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.
- The OS manages the system resources and makes them available to the user applications/tasks on a need basis.
- The primary functions of an operating system is
  - Make the system convenient to use
  - Organise and manage the system resources efficiently and correctly
- Basic components of an operating system and their interfaces are

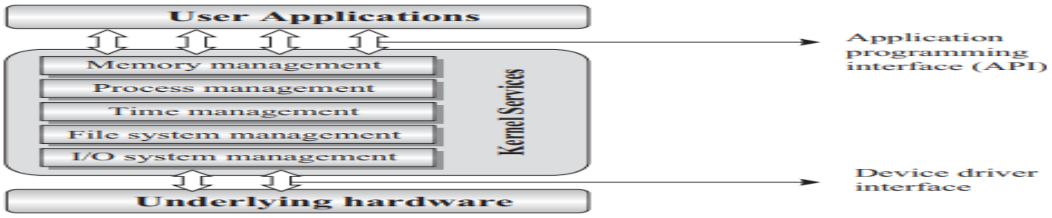


Fig. 10.1 The Operating System Architecture

KERNEL

- Core of the OS and 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 GPOS, the kernel contains different services for handling the following.
    - **Process Management**-Process management deals with managing the processes/tasks. Process management 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 synchronisation, 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.
    - **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. 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*
        - *Creation, deletion and alteration of directories*
        - *Saving of files in the secondary storage memory (e.g. Hard disk storage)*
        - *Providing automatic allocation of file space based on the amount of free space available*
        - *Providing a 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.

  This list may be available in advance, at the time of building the kernel. Some kernels, dynamically updates the list of available devices as and when a new device is installed (e.g. Windows NT kernel keeps the list updated when a new plug 'n' play USB device is attached to the system).

  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**-. Secondary memory is used as backup medium for programs and data since the main memory is volatile.

The secondary storage management service of kernel deals with

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

- **Protecion Systems**- Protection deals with implementing the security policies to restrict the access to both user and system resources by different applications or processes or users.
- **Interrupt Handler**-Kernel provides handler mechanism for all external/internal interrupts generated by the system.

Kernel Space and User Space

- Applications/services are classified into two categories, namely:
  User applications and kernel applications.
- The program code corresponding to the kernel applications/services are kept in a contiguous area (OS dependent) of primary (working) memory and is protected from the unauthorised 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’.
- User space is the memory area where user applications are loaded and executed.
- Entire code for the user application need not be loaded to the main (primary) memory at once; instead the user application code is split into different pages and these pages are loaded into and out of the main memory area on a need basis.
- The act of loading the code into and out of the main memory is termed as ***‘Swapping’***.
- Swapping happens between the main (primary) memory and secondary storage memory.
- Each process run in its own virtual memory space and are not allowed accessing the memory space corresponding to another processes, unless explicitly requested by the process.

Q.3.Difference between monolithic and microkernel

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.
- The tight internal integration of kernel modules in monolithic kernel architecture allows the effective utilisation of the low-level features of the underlying system.
- 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.
- Eg:LINUX, SOLARIS, MS-DOS kernels

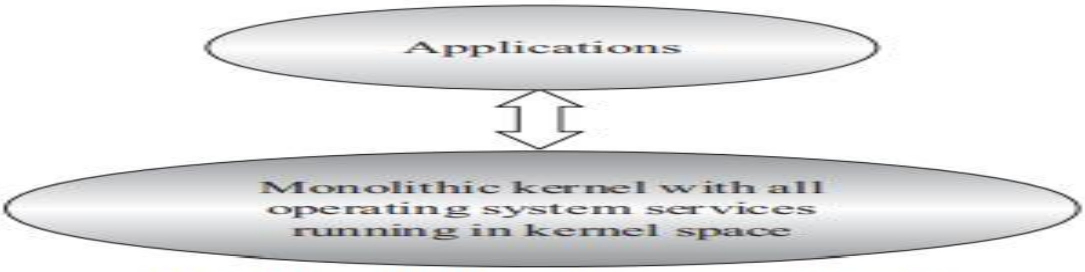

Applications
Monolithic kernel with all operating system services running in kernel space

Fig. 10.2 The Monolithic Kernel Model

MICROKERNEL

- The microkernel design incorporates only the essential set of Operating System services into the kernel.
- The rest of the Operating System services are implemented in programs known as ‘Servers’ which runs in user space.
- This provides a highly modular design and OS-neutral abstraction to the kernel.
- Memory management, process management, timer systems and interrupt handlers are the essential services, which forms the part of the microkernel.

- Eg:Mach, QNX, Minix 3
- Microkernel based design approach offers the following benefits
    - **Robustness:** If a problem is encountered in any of the services, which runs as 'Server' application, the same can be reconfigured and re-started without the need for re-starting the entire OS. Thus, this approach is highly useful for systems, which demands high 'availability'. space, the chances of corruption of kernel services are ideally zero.
    - **Configurability**: Any services, which run as 'Server' application can be changed without the need to restart the whole system. This makes the system dynamically configurable.

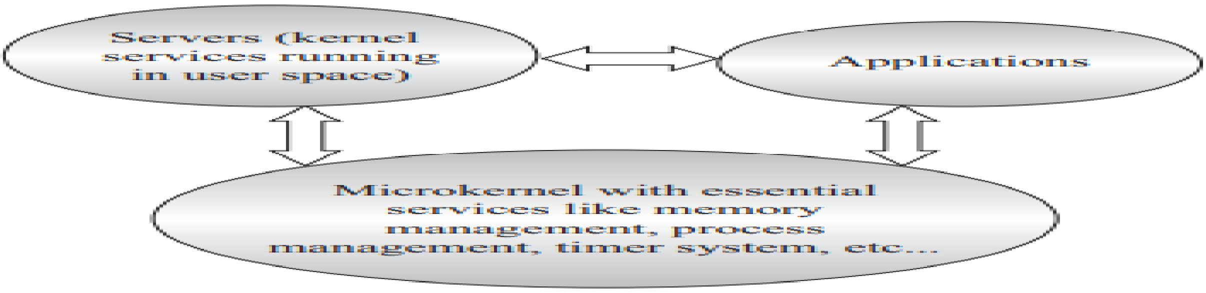

Servers (kernel services running in user space)
Applications
Microkernel with essential services like memory management, process management, timer system, etc...

Fig. 10.3 The Microkernel model

Q.4.Classify the different types of operating systems.

GENERAL PURPOSE OPERATING SYSTEM (GPOS)

- The kernel of such an OS is more generalised and it contains all kinds of services required for executing generic applications.
- Often non-deterministic in behaviour.
- Their services can inject random delays into application software and may cause slow responsiveness of an application at unexpected times.
- Eg:Windows 10/8.x/XP/MS-DOS etc

REAL-TIME OPERATING SYSTEM (RTOS)

- Real-Time' implies deterministic timing behaviour.
- Deterministic timing behaviour in RTOS context means the OS services consumes only known and expected amounts of time
- A Real-Time Operating System or RTOS implements policies and rules concerning time-critical allocation of a system's resources.
- The RTOS decides which applications should run in which order and how much time needs to be allocated for each application.
- Eg: QNX, VxWorks MicroC/OS-II etc

Q.5.Explain the functionalities of a real time kernel

The Real-Time Kernel contains only the minimal set of services required for running the user applications/tasks.

The basic functions of a Real-Time kernel are:

1.*Task/Process management*
2.Task/Process scheduling
3.Task/Process synchronisation
4.Error/Exception handling
5.Memory management
6.Interrupt handling

7.Time management

<u>*1.TASK/ PROCESS MANAGEMENT*</u>

- Deals with setting up the memory space for the tasks, loading the task's code into the memory space, allocating system resources, setting up a Task Control Block (TCB) for the task and task/process termination/deletion.
- A Task Control Block (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: Indicates what is the type for this task. The task can be a hard real time or soft real time or background task.

  Task Priority: (e.g. Task priority = 1 for task with priority = 1)

  Task Context Pointer: Pointer for context saving

  Task Memory Pointers: Pointers to the code memory, data memory and stack memory for the task

  Task System Resource Pointers: Pointers to system resources (semaphores, mutex, etc.) used by the task

  Task Pointers: Pointers to other TCBs (TCBs for preceding, next and waiting tasks)

  - Task management service utilises the TCB of a task in the following way
    - ✓ Creates a TCB for a task on creating a task
    - ✓ Delete/remove the TCB of a task when the task is terminated or deleted
    - ✓ Reads the TCB to get the state of a task
    - ✓ Update the TCB with updated parameters on need basis (e.g. on a context switch)
    - ✓ Modify the TCB to change the priority of the task dynamically

<u>*2. TASK/ PROCESS SCHEDULING*</u>-Deals with sharing the CPU among various tasks/processes. A kernel application called 'Scheduler' handles the task scheduling. Scheduler is nothing but an algorithm implementation, which performs the efficient and optimal scheduling of tasks to provide a deterministic behaviour.

<u>*3.TASK/ PROCESS SYNCHRONISATION*</u> Deals with synchronising the concurrent access of a resource, which is shared across multiple tasks and the communication between various tasks.

<u>*4.ERROR/ EXCEPTION HANDLING*</u>

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,

Watchdog timer is a mechanism for handling the timeouts for tasks.

5.MEMORY MANAGEMENT

- Memory allocation time increases depending on the size of the block of memory needs to be allocated and the state of the allocated memory block
- RTOS makes use of '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'.

6. 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.***
- Usually the software interrupts fall under the Synchronous Interrupt category.
- Eg:Divide by zero, memory segmentation error, etc.
- ***Asynchronous interrupts*** are interrupts, which occurs at any point of execution of any task, and are not in sync with the currently executing task.
- The interrupts generated by external devices (by asserting the interrupt line of the processor/controller to which the interrupt line of the device is connected) connected to the processor/controller, timer overflow interrupts, serial data reception/ transmission interrupts, etc. are examples for asynchronous interrupts.

7.TIME MANAGEMENT

- Accurate time management is essential for providing precise time reference for all
- applications. The time reference to kernel is provided by a high-resolution Real-Time Clock (RTC) hardware
- 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.
- The 'Timer tick' interval may vary depending on the hardware timer.
- Usually the 'Timer tick' varies in the microseconds range.
- The 'Timer tick' interrupt is handled by the 'Timer Interrupt' handler of kernel. The 'Timer tick' interrupt can be utilised for implementing the following actions.

    Save the current context (Context of the currently executing task).

    Increment the System time register by one. Generate timing error and reset

Q.6.List the difference between hard real time OS and soft real time OS

HARD REAL-TIME

- RTOS that strictly adhere to the timing constraints for a task is referred as 'Hard Real-Time' systems.
- It meet the deadlines for a task without any slippage.
- Missing any deadline may produce catastrophic results including permanent data lose and irrecoverable damages to the system/users.
- Eg:Air bag control systems and Anti-lock Brake Systems (ABS) etc

- The Air bag control system should be into action and deploy the air bags when the vehicle meets a severe accident. The time for triggering the air bag deployment task, when an accident is sensed by the Air bag control system, should be zero and the air bags should be deployed exactly within the time frame, which is predefined for the air bag deployment task. Any delay in the deployment of the air bags makes the life of the passengers under threat. When the air bag deployment task is triggered, the currently executing task must be pre-empted, the air bag deployment task should be brought into execution.

SOFT REAL-TIME

- RTOS 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 for a Soft Realtime system if the frequency of deadline missing is within the compliance limit of the Quality of Service
- Eg:Automatic Teller Machine (ATM)
- If the ATM takes a few seconds more than the ideal operation time, nothing fatal happens. An audio-video playback system is another example for Soft Real-Time system. No potential damage arises if a sample comes late by fraction of a second, for playback

Q.7.Discuss tasks, processes and threads in the operating system context

- Task refers to something that needs to be done.
- We will have an order of priority and schedule/timeline for executing these tasks.
- Task is defined as the program in execution

PROCESS

- A 'Process' is a program, or part of it, in execution.
- Multiple instances of the same program can execute simultaneously. 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. A process is sequential in execution.

The Structure of a Process

- The concept of 'Process' leads to concurrent execution (pseudo parallelism) of tasks and thereby the efficient utilisation 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.

- From a memory perspective, the memory occupied by the process is segregated into three regions, namely, 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.
- Code memory contains the program code (instructions) corresponding to the process. On loading a process into the main memory, a specific area of memory is allocated for the process.

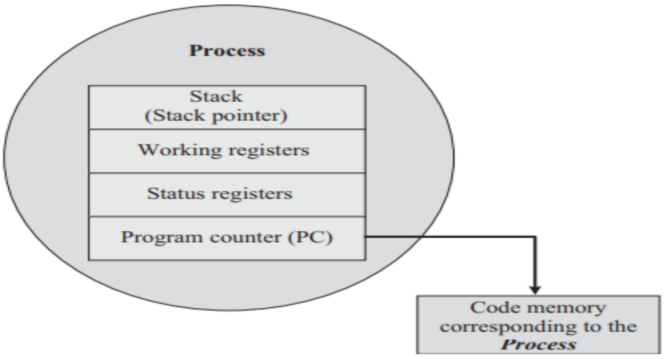


Fig. 10.4 Structure of a Process

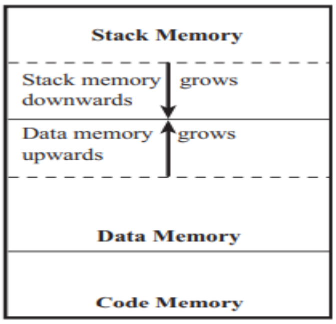


Fig. 10.5 Memory organisation of a Process

PROCESS STATES AND TRANSITION

- 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 and also provides information on what it is allowed to do next.
- The state at which a process is being created is referred as ‘Created State’.
- The Operating System recognises a process in the ‘Created State’ but no resources are allocated to the process.
- The state, where a process is incepted into the memory and awaiting the processor time for execution, is known as ‘Ready State’.
- At this stage, the process is placed in the ‘Ready list’ queue maintained by the OS.
- The state where in the source code instructions corresponding to the process is being executed is called ‘Running State’. 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 and does not have immediate access to resources.
- The transition of a process from one state to another is known as ‘State transition’.
- When a process changes its state from Ready to running or from running to blocked or terminated or from blocked to running, the CPU allocation for the process may also change.

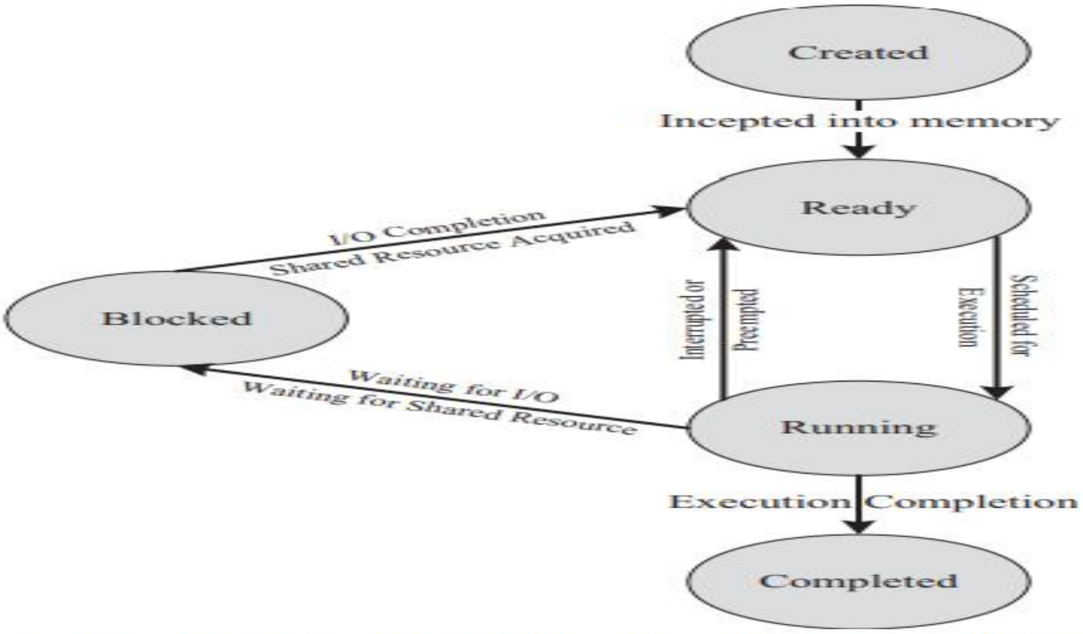


Fig. 10.6 **Process states and state transition representation**

PROCESS MANAGEMENT

Process management deals with the creation of a process, setting up the memory space for the process, loading the process's code into the memory space, allocating system resources, setting up a Process Control Block (PCB) for the process and process termination/deletion.

THREAD

- A thread is the primitive that can execute code.
- A thread is a single sequential flow of control within a process.
- '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. .

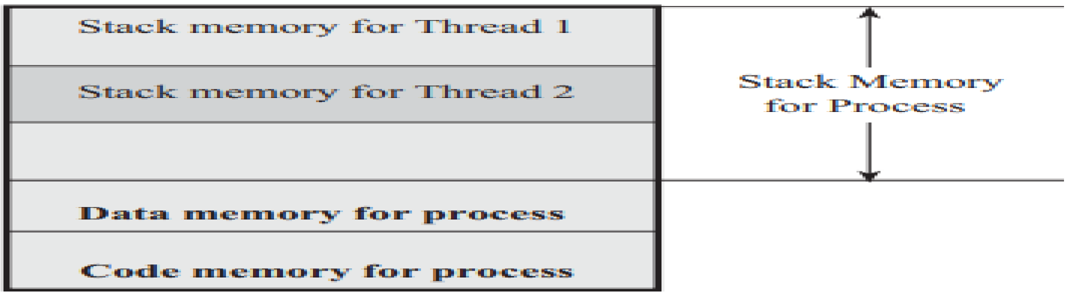
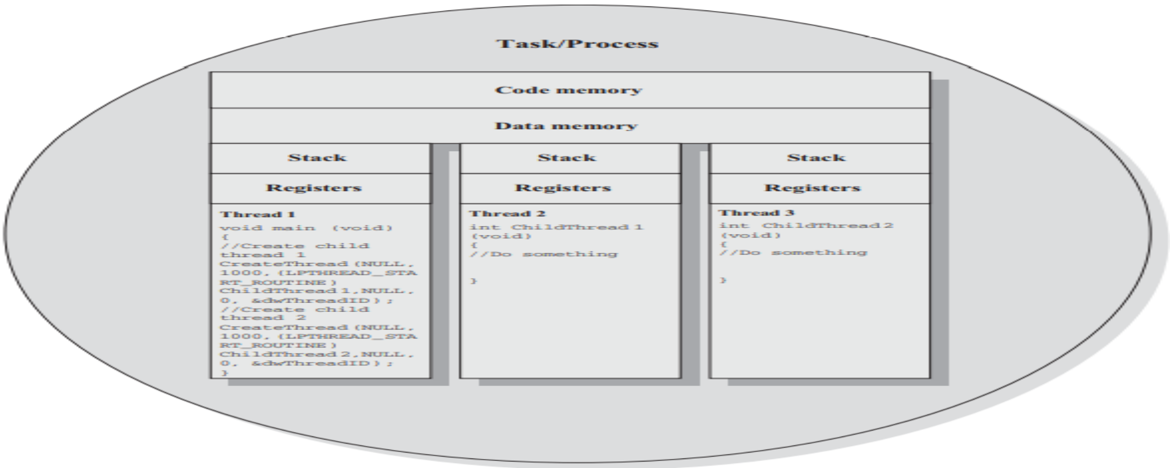


Fig. 10.7 **Memory organisation of a Process and its associated Threads**

MULTITHREADING

- A process/task in embedded application may be a complex or lengthy one and it may contain various suboperations 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 subfunctions of a task are executed in sequence, the CPU utilisation may not be efficient.
- For example, if the process is waiting for a user input, the CPU enters the wait state for the event, and the process execution also enters a wait state.
- Instead of this single sequential execution of the whole process, if the task/process is split into different threads carrying out the different subfunctionalities of the process, the CPU can be effectively utilised and when the thread corresponding to the I/O operation enters the wait state, another threads 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 utilisation of the processor time and resources.
- If 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. Use of multiple threads to execute a process brings the following advantage.
- Better memory utilisation. Multiple threads of the same process share the address space for data memory. This also reduces the complexity of inter thread communication since variables can be shared across the threads.
- Since the process is split into different threads, when one thread enters a wait state, the CPU can be utilised by other threads of the process that do not require the event, which the other thread is waiting, for processing. This speeds up the execution of the process.
- Efficient CPU utilisation. The CPU is engaged all time.



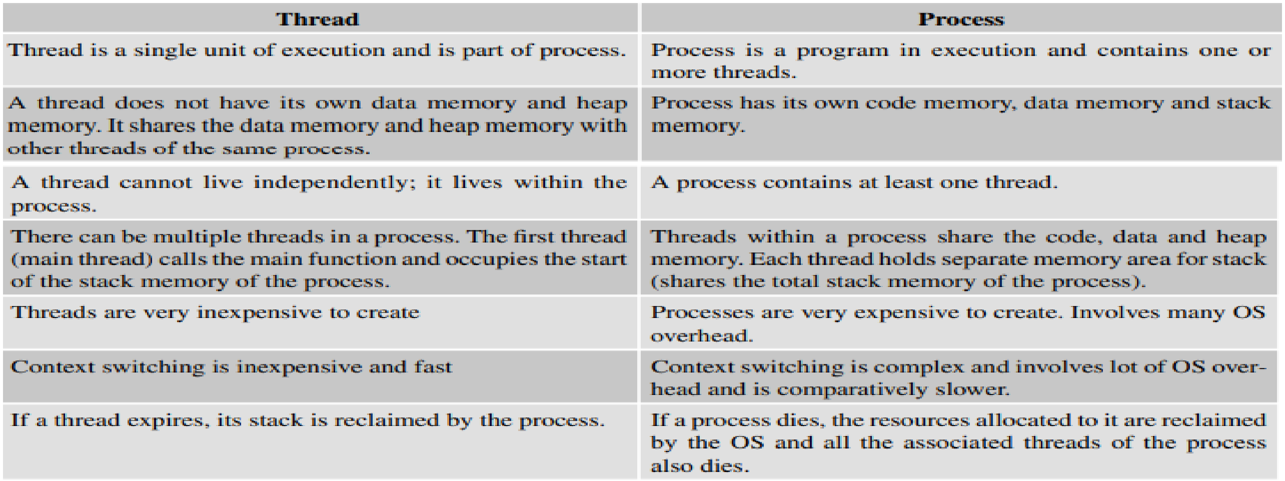
```
void main  (void)
{
//Create child
thread 1
CreateThread(NULL,
1000,(LPTHREAD_STA
RT_ROUTINE)
ChildThread1,NULL,
0, &dwThreadID);
//Create child
thread 2
CreateThread(NULL,
1000,(LPTHREAD_STA
RT_ROUTINE)
ChildThread2,NULL,
0, &dwThreadID);
}
```

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

}
```

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

}
```

Fig. 10.8 Process with multi-threads

Q.8.List difference between process and threads



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. It shares the data memory and heap memory with other threads of the same process.	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 thread (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 (shares the total stack memory of the process).
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 lot of OS over-head 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.

Q.9.Understand the difference between multiprocessing and multitasking

- Multiprocessing describes the ability to 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 uniprocessor system, it is not possible to execute multiple processes simultaneously.
- The ability of an operating system to hold multiple processes in memory and switch the processor (CPU) from executing one process to another process is known as multitasking.
- Multitasking creates the illusion of multiple tasks executing in parallel.
- In a multitasking environment, when task/process switching happens, the virtual processor (task/process) gets its properties converted into that of the physical processor. The switching of the virtual processor to physical processor is controlled by the scheduler of the OS kernel.
- Whenever a CPU switching happens, the current context of execution should be saved to retrieve it at a later point of time when the CPU executes the process, which is interrupted currently due to execution switching.
- The context saving and retrieval is essential for resuming a process exactly from the point where it was interrupted due to CPU switching.
- 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 which contains the context details (Register details, memory details, system resource usage details, execution details, etc.) for the currently running process 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’.
- Multitasking involves ‘ Context switching’, ‘Context saving’ and ‘Context retrieval’.
- Toss Juggling-The skilful object manipulation game is a classic real world example for the multitasking illusion. The juggler uses a number of objects (balls, rings, etc.) and throws them up and catches them. At any point of time, he throws only one ball and catches only one per hand.

- However, the speed at which he is switching the balls for throwing and catching creates the illusion, he is throwing and catching multiple balls or using more than two hands simultaneously, to the spectators.

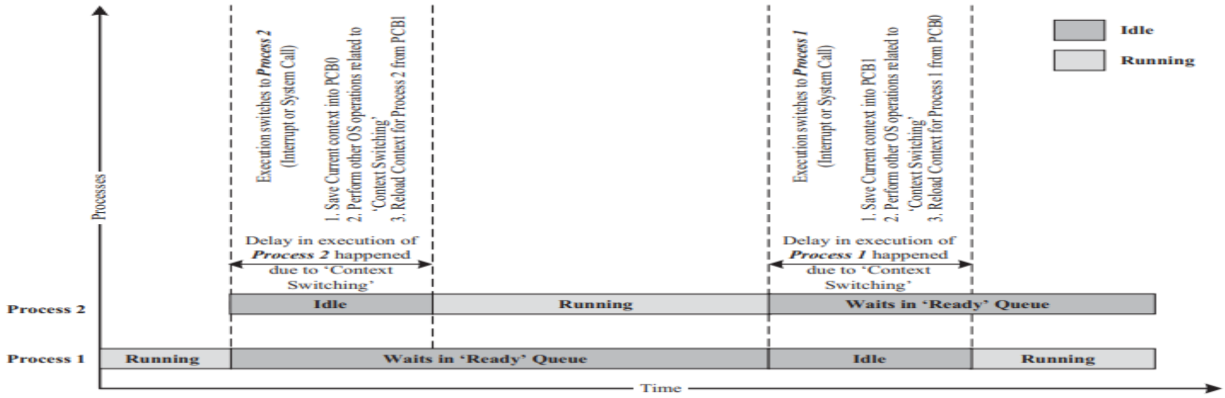

Processes
Execution switches to *Process 2* (Interrupt or System Call)
1. Save Current context into PCB0
2. Perform other OS operations related to 'Context Switching'
3. Reload Context for Process 2 from PCB1
Delay in execution of *Process 2* happened due to 'Context Switching'
Execution switches to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

Fig. 10.11 Context switching

TYPES OF MULTITASKING

Multitasking involves the switching of execution among multiple tasks.

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 hold 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

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. 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' state, 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 or system resource or an event to occur.

Q.10.Explain task scheduling

Multitasking involves the execution switching among the different tasks.

- There should be some mechanism in place to share the CPU among the different tasks and to decide which process/task is to be executed at a given point of time.
- Determining which task/process is to be executed at a given point of time is known as task/process scheduling.
- Task scheduling forms the basis of multitasking.
- 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 kernel service/application, which implements the scheduling algorithm, is known as 'Scheduler'.
- The 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 the CPU utilisation high. CPU utilisation is a direct measure of how much percentage of the CPU is being utilised.
  - ✓ **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 spent in the ready queue, time spent on completing the I/O operations, and the time spent 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 fi rst response. For a good scheduling algorithm, the response time should be as least as possible.

- 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.

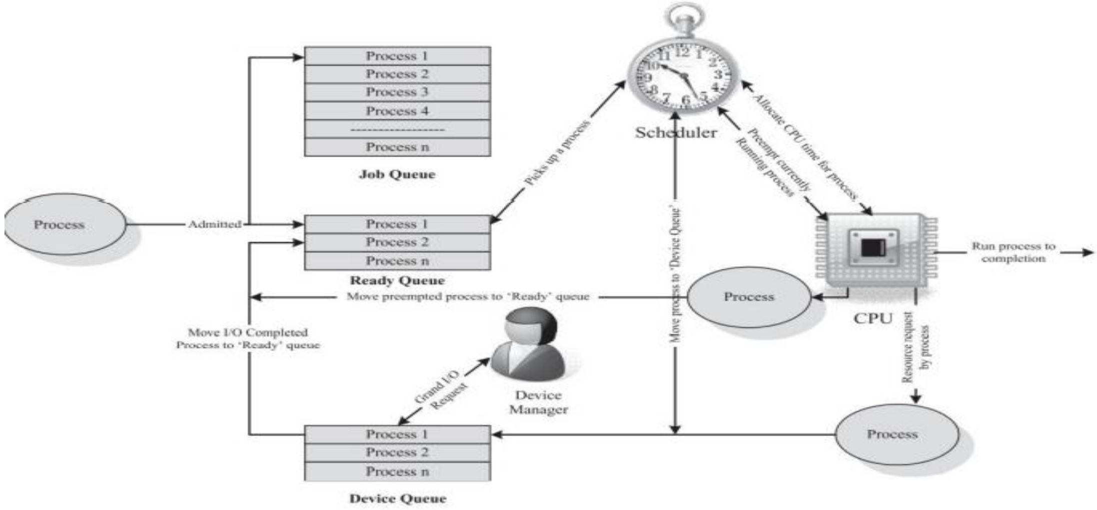


Fig. 10.12 Illustration of process transition through various queues

Pre-emptive scheduling	Non pre-emptive scheduling

SRT(Preemptive SJF Scheduling/ Shortest Remaining Time (SRT)
RR(Round robin)
Priority based
Rate based
	FIFO(First in first out)
	LIFO(Last in first out)
	SJF(Shortest job first)

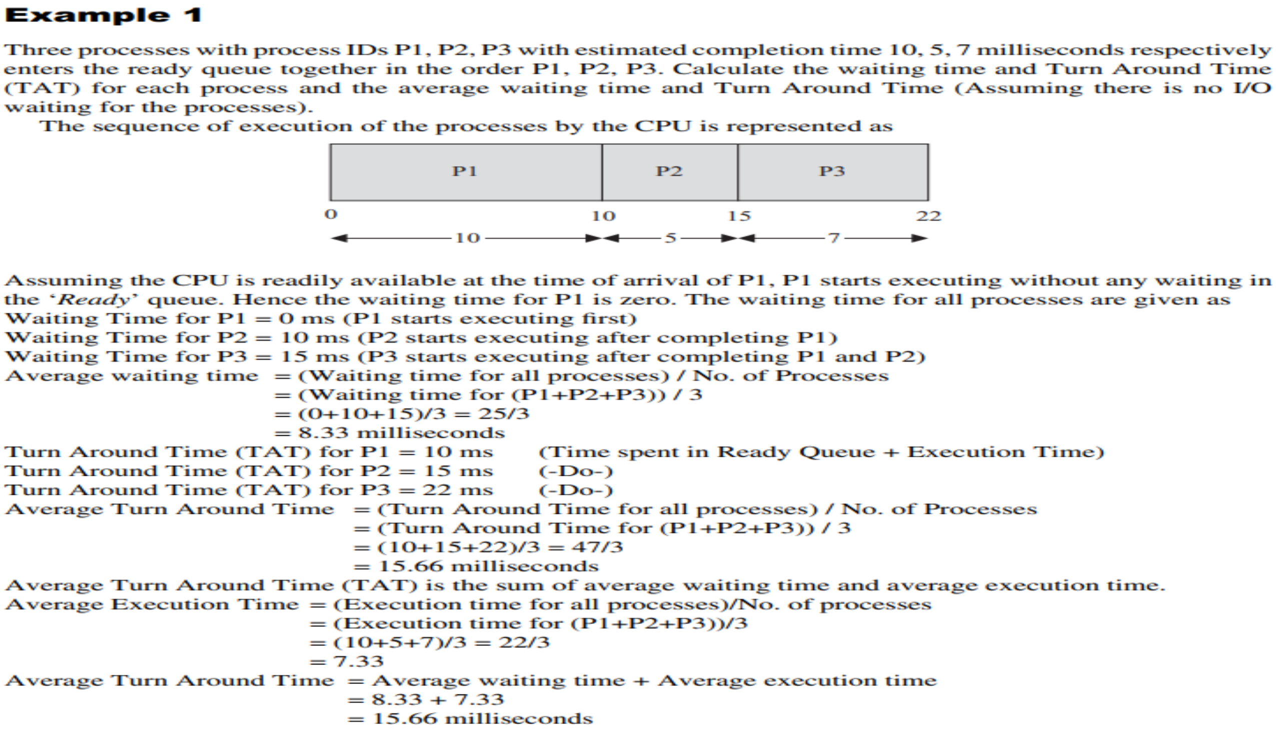
NON PRE-EMPTIVE SCHEDULING

- ✓ 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
  i) First-Come-First-Served (FCFS)/ FIFO Scheduling
  ii) Last-Come-First Served (LCFS)/LIFO Scheduling
  iii) Shortest Job First (SJF) Scheduling
  iv) Priority Based Scheduling

Turn Around time=Waiting Time in queue+Completion time
Waiting time=Execution Time-Arrival Time

i)First Come First Served(FCFS/FIFO) scheduling

- It allocates CPU time to the processes based on the order in which they enter the 'Ready' queue.
- The first entered process is serviced first.
- It is same as any real world application where queue systems are used;
- e.g. Ticketing reservation system where people need to stand in a queue and the first person standing in the queue 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.



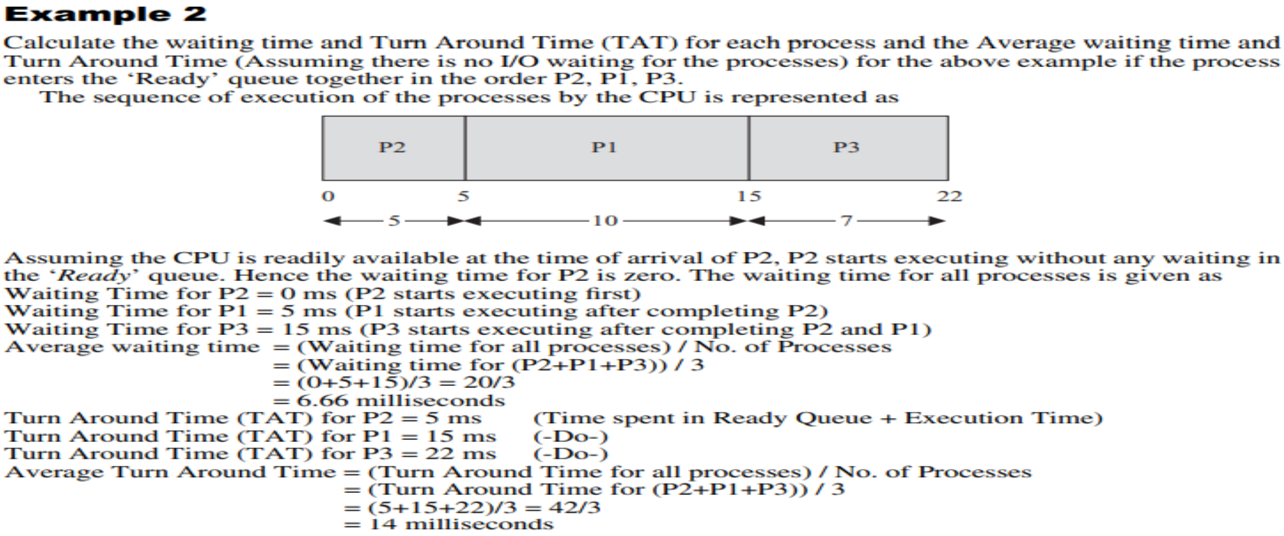
Example 1

Three processes with process IDs 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 I/O waiting for the processes).

The sequence of execution of the processes by the CPU is represented as

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. The waiting time for all processes are given as

Waiting Time for P1 = 0 ms (P1 starts executing first)
Waiting Time for P2 = 10 ms (P2 starts executing after completing P1)
Waiting Time for P3 = 15 ms (P3 starts executing after completing P1 and P2)
Average waiting time = (Waiting time for all processes) / No. of Processes
= (Waiting time for (P1+P2+P3)) / 3
= (0+10+15)/3 = 25/3
= 8.33 milliseconds
Turn Around Time (TAT) for P1 = 10 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P2 = 15 ms (-Do-)
Turn Around Time (TAT) for P3 = 22 ms (-Do-)
Average Turn Around Time = (Turn Around Time for all processes) / No. of Processes
= (Turn Around Time for (P1+P2+P3)) / 3
= (10+15+22)/3 = 47/3
= 15.66 milliseconds
Average Turn Around Time (TAT) is the sum of average waiting time and average execution time.
Average Execution Time = (Execution time for all processes)/No. of processes
= (Execution time for (P1+P2+P3))/3
= (10+5+7)/3 = 22/3
= 7.33
Average Turn Around Time = Average waiting time + Average execution time
= 8.33 + 7.33
= 15.66 milliseconds



Example 2

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) for the above example if the process enters the 'Ready' queue together in the order P2, P1, P3.

The sequence of execution of the processes by the CPU is represented as

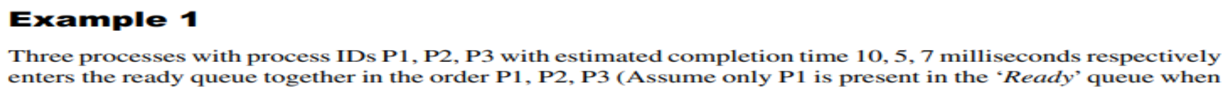
P2
P1
P3
0
5
15
22
5
10
7

Assuming the CPU is readily available at the time of arrival of P2, P2 starts executing without any waiting in the '*Ready*' queue. Hence the waiting time for P2 is zero. The waiting time for all processes is given as

Waiting Time for P2 = 0 ms (P2 starts executing first)
Waiting Time for P1 = 5 ms (P1 starts executing after completing P2)
Waiting Time for P3 = 15 ms (P3 starts executing after completing P2 and P1)
Average waiting time = (Waiting time for all processes) / No. of Processes
= (Waiting time for (P2+P1+P3)) / 3
= (0+5+15)/3 = 20/3
= 6.66 milliseconds
Turn Around Time (TAT) for P2 = 5 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P1 = 15 ms (-Do-)
Turn Around Time (TAT) for P3 = 22 ms (-Do-)
Average Turn Around Time = (Turn Around Time for all processes) / No. of Processes
= (Turn Around Time for (P2+P1+P3)) / 3
= (5+15+22)/3 = 42/3
= 14 milliseconds

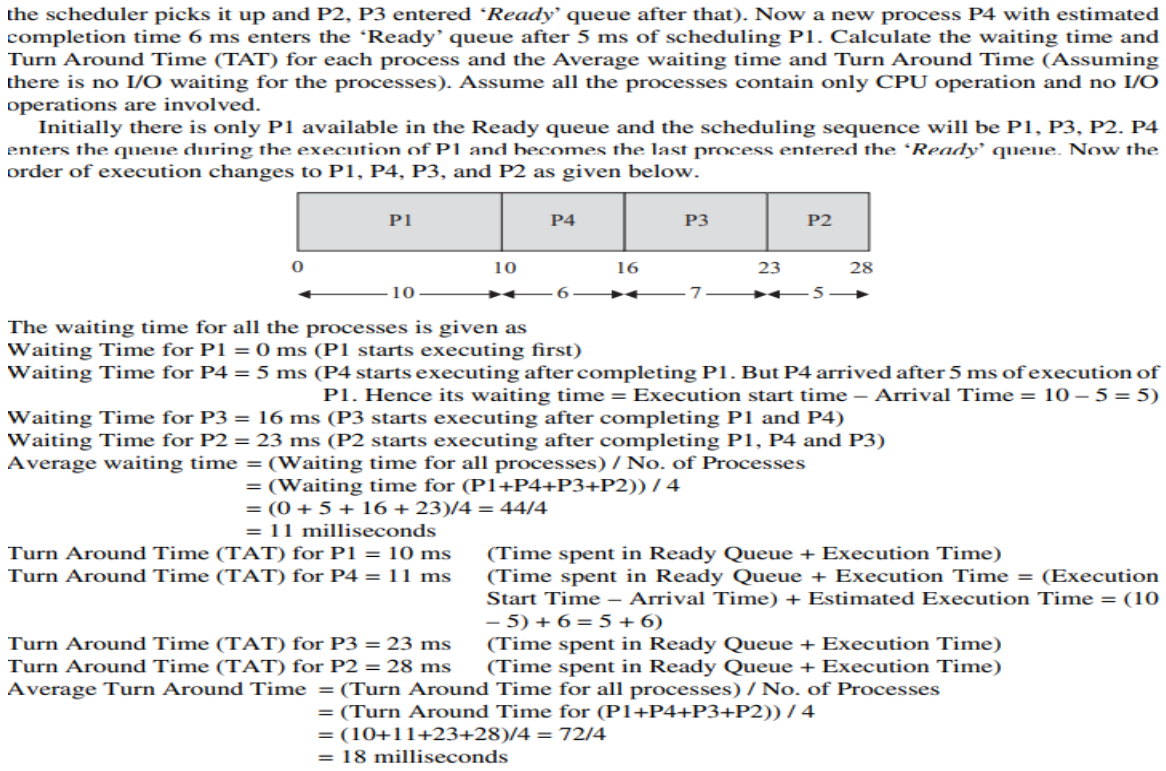
<u>ii)Last come first served scheduling(LCFS)</u>

- 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 fi rst.
- LCFS scheduling is also known as Last In First Out (LIFO) where the process, which is put last into the 'Ready' queue, is serviced fi rst



Example 1

Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds respectively enters the ready queue together in the order P1, P2, P3 (Assume only P1 is present in the '*Ready*' queue when



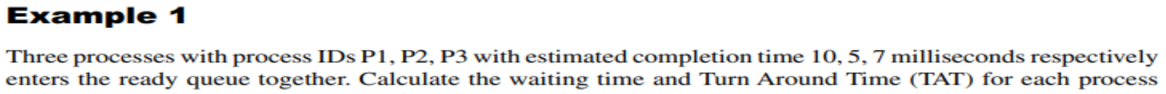
the scheduler picks it up and P2, P3 entered '*Ready*' queue after that). Now a new process P4 with estimated completion time 6 ms enters the 'Ready' queue after 5 ms of scheduling P1. Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time (Assuming there is no I/O waiting for the processes). Assume all the processes contain only CPU operation and no I/O operations are involved.

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.

The waiting time for all the processes is given as
Waiting Time for P1 = 0 ms (P1 starts executing first)
Waiting Time for P4 = 5 ms (P4 starts executing after completing P1. But P4 arrived after 5 ms of execution of P1. Hence its waiting time = Execution start time – Arrival Time = 10 – 5 = 5)
Waiting Time for P3 = 16 ms (P3 starts executing after completing P1 and P4)
Waiting Time for P2 = 23 ms (P2 starts executing after completing P1, P4 and P3)
Average waiting time = (Waiting time for all processes) / No. of Processes
= (Waiting time for (P1+P4+P3+P2)) / 4
= (0 + 5 + 16 + 23)/4 = 44/4
= 11 milliseconds
Turn Around Time (TAT) for P1 = 10 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P4 = 11 ms (Time spent in Ready Queue + Execution Time = (Execution Start Time – Arrival Time) + Estimated Execution Time = (10 – 5) + 6 = 5 + 6)
Turn Around Time (TAT) for P3 = 23 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P2 = 28 ms (Time spent in Ready Queue + Execution Time)
Average Turn Around Time = (Turn Around Time for all processes) / No. of Processes
= (Turn Around Time for (P1+P4+P3+P2)) / 4
= (10+11+23+28)/4 = 72/4
= 18 milliseconds

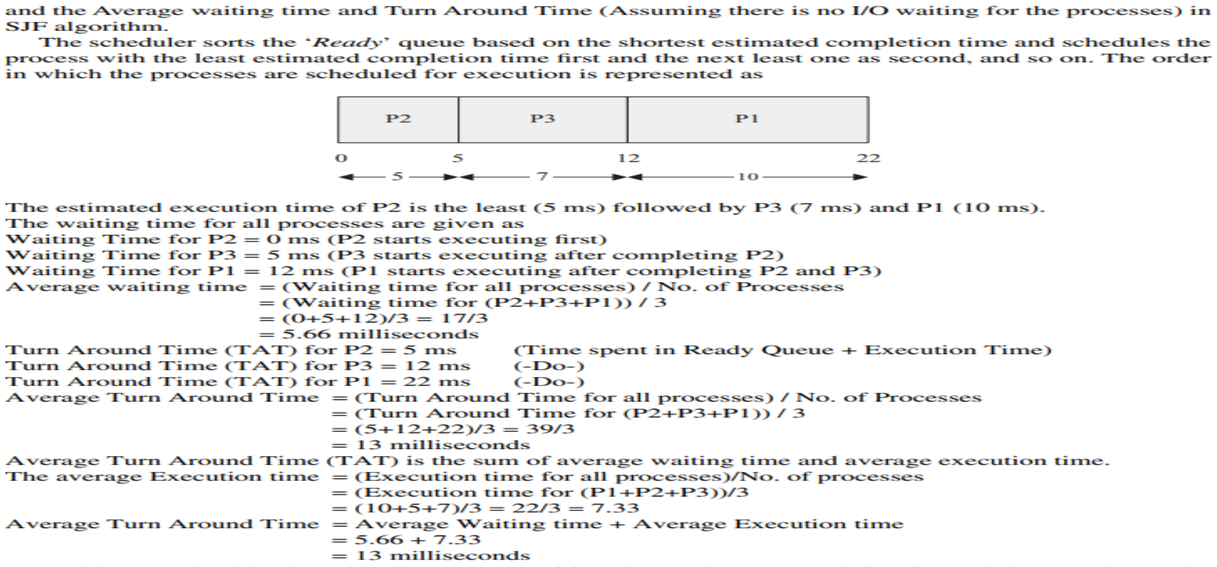
iii)Shortest job first scheduling(SJF)

- It'sorts the 'Ready' queue' each time a process relinquishes the CPU (either the process terminates or enters the 'Wait' state waiting for I/O or system resource) to pick the process with shortest (least) estimated completion/run time.
- Here process with the shortest estimated run time is scheduled first, followed by the next shortest process, and so on.



Example 1

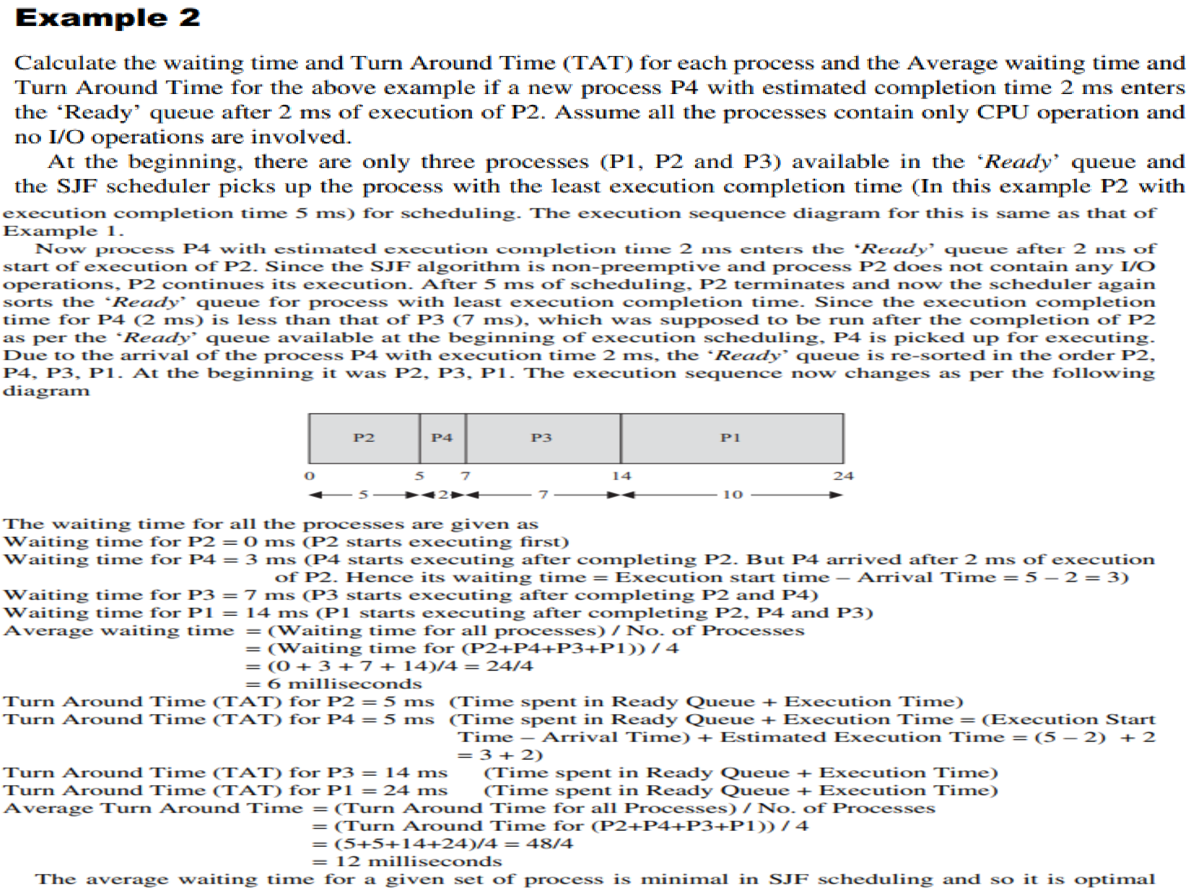
Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds respectively enters the ready queue together. Calculate the waiting time and Turn Around Time (TAT) for each process



and the Average waiting time and Turn Around Time (Assuming there is no I/O waiting for the processes) in SJF algorithm.

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

The estimated execution time of P2 is the least (5 ms) followed by P3 (7 ms) and P1 (10 ms).
The waiting time for all processes are given as
Waiting Time for P2 = 0 ms (P2 starts executing first)
Waiting Time for P3 = 5 ms (P3 starts executing after completing P2)
Waiting Time for P1 = 12 ms (P1 starts executing after completing P2 and P3)
Average waiting time = (Waiting time for all processes) / No. of Processes
= (Waiting time for (P2+P3+P1)) / 3
= (0+5+12)/3 = 17/3
= 5.66 milliseconds
Turn Around Time (TAT) for P2 = 5 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P3 = 12 ms (-Do-)
Turn Around Time (TAT) for P1 = 22 ms (-Do-)
Average Turn Around Time = (Turn Around Time for all processes) / No. of Processes
= (Turn Around Time for (P2+P3+P1)) / 3
= (5+12+22)/3 = 39/3
= 13 milliseconds
Average Turn Around Time (TAT) is the sum of average waiting time and average execution time.
The average Execution time = (Execution time for all processes)/No. of processes
= (Execution time for (P1+P2+P3))/3
= (10+5+7)/3 = 22/3 = 7.33
Average Turn Around Time = Average Waiting time + Average Execution time
= 5.66 + 7.33
= 13 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 2 ms enters the 'Ready' queue after 2 ms of execution of P2. Assume all the processes contain only CPU operation and no I/O operations are involved.

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 (In this example P2 with execution completion time 5 ms) for scheduling. The execution sequence diagram for this is same as that of Example 1.

Now process P4 with estimated execution completion time 2 ms enters the '*Ready*' queue after 2 ms of start of execution of P2. Since the SJF algorithm is non-preemptive and process P2 does not contain any I/O operations, P2 continues its execution. After 5 ms 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 (2 ms) is less than that of P3 (7 ms), 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 2 ms, the '*Ready*' queue is re-sorted in the order P2, P4, P3, P1. At the beginning it was P2, P3, P1. The execution sequence now changes as per the following diagram

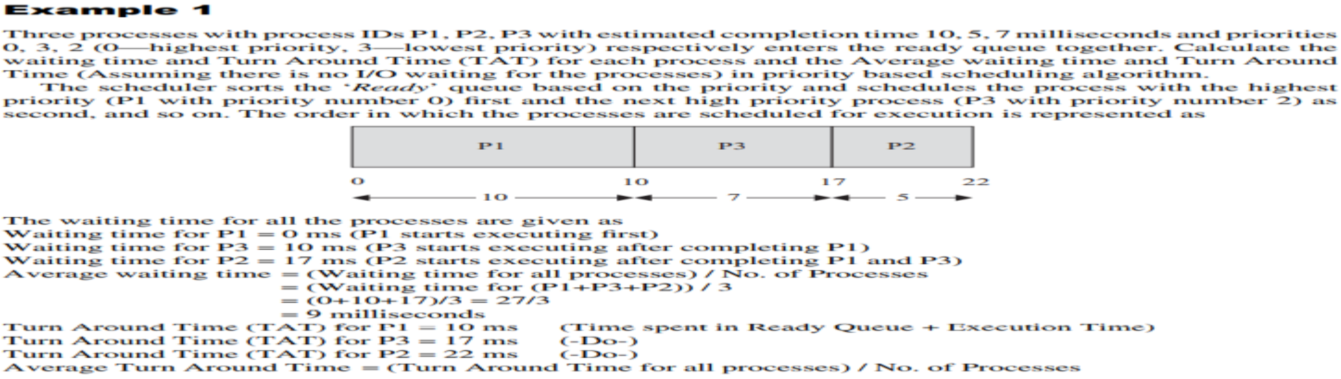
The waiting time for all the processes are given as
Waiting time for P2 = 0 ms (P2 starts executing first)
Waiting time for P4 = 3 ms (P4 starts executing after completing P2. But P4 arrived after 2 ms of execution of P2. Hence its waiting time = Execution start time – Arrival Time = 5 – 2 = 3)
Waiting time for P3 = 7 ms (P3 starts executing after completing P2 and P4)
Waiting time for P1 = 14 ms (P1 starts executing after completing P2, P4 and P3)
Average waiting time = (Waiting time for all processes) / No. of Processes
= (Waiting time for (P2+P4+P3+P1)) / 4
= (0 + 3 + 7 + 14)/4 = 24/4
= 6 milliseconds
Turn Around Time (TAT) for P2 = 5 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P4 = 5 ms (Time spent in Ready Queue + Execution Time = (Execution Start Time – Arrival Time) + Estimated Execution Time = (5 – 2) + 2 = 3 + 2)
Turn Around Time (TAT) for P3 = 14 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P1 = 24 ms (Time spent in Ready Queue + Execution Time)
Average Turn Around Time = (Turn Around Time for all Processes) / No. of Processes
= (Turn Around Time for (P2+P4+P3+P1)) / 4
= (5+5+14+24)/4 = 48/4
= 12 milliseconds

The average waiting time for a given set of process is minimal in SJF scheduling and so it is optimal

- The major drawback of SJF algorithm is that a process whose estimated execution completion time is high may not get a chance to execute if more and more processes with least estimated execution time enters the 'Ready' queue before the process with longest estimated execution time started its execution (In non-preemptive SJF). This condition is known as **'Starvation'.**

  iv)Priority based scheduling

  - It ensures that a process with high priority is serviced at the earliest compared to other low priority processes in the 'Ready' queue.
  - The Shortest Job First (SJF) algorithm can be viewed as a priority based scheduling where each task is prioritised in the order of the time required to complete the task.
  - The lower the time required for completing a process the higher is its priority in SJF algorithm.
  - The priority is a number ranging from 0 to the maximum priority supported by the OS.
  - The maximum level of priority is OS dependent.
  - While creating the process/task, the priority can be assigned to it.

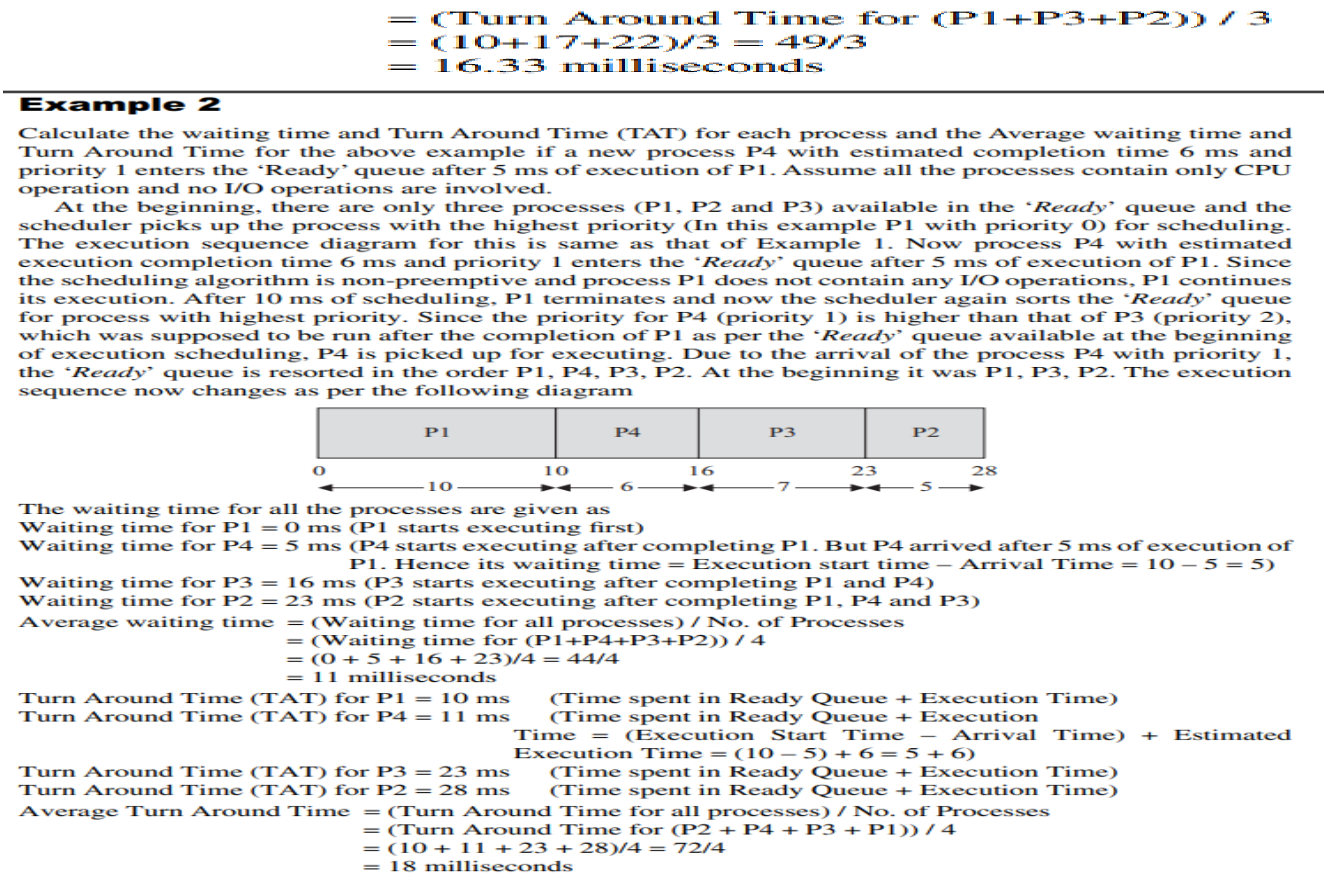


Example 1

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.

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

The waiting time for all the processes are given as
Waiting time for P1 = 0 ms (P1 starts executing first)
Waiting time for P3 = 10 ms (P3 starts executing after completing P1)
Waiting time for P2 = 17 ms (P2 starts executing after completing P1 and P3)
Average waiting time = (Waiting time for all processes) / No. of Processes
= (Waiting time for (P1+P3+P2)) / 3
= (0+10+17)/3 = 27/3
= 9 milliseconds
Turn Around Time (TAT) for P1 = 10 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P3 = 17 ms (-Do-)
Turn Around Time (TAT) for P2 = 22 ms (-Do-)
Average Turn Around Time = (Turn Around Time for all processes) / No. of Processes



= (Turn Around Time for (P1+P3+P2)) / 3
= (10+17+22)/3 = 49/3
= 16.33 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 6 ms and priority 1 enters the 'Ready' queue after 5 ms of execution of P1. Assume all the processes contain only CPU operation and no I/O operations are involved.

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 (In this example P1 with priority 0) for scheduling. The execution sequence diagram for this is same as that of Example 1. Now process P4 with estimated execution completion time 6 ms and priority 1 enters the '*Ready*' queue after 5 ms of execution of P1. Since the scheduling algorithm is non-preemptive and process P1 does not contain any I/O operations, P1 continues its execution. After 10 ms of scheduling, P1 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 1, 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

The waiting time for all the processes are given as
Waiting time for P1 = 0 ms (P1 starts executing first)
Waiting time for P4 = 5 ms (P4 starts executing after completing P1. But P4 arrived after 5 ms of execution of P1. Hence its waiting time = Execution start time – Arrival Time = 10 – 5 = 5)
Waiting time for P3 = 16 ms (P3 starts executing after completing P1 and P4)
Waiting time for P2 = 23 ms (P2 starts executing after completing P1, P4 and P3)

Average waiting time = (Waiting time for all processes) / No. of Processes
= (Waiting time for (P1+P4+P3+P2)) / 4
= (0 + 5 + 16 + 23)/4 = 44/4
= 11 milliseconds

Turn Around Time (TAT) for P1 = 10 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P4 = 11 ms (Time spent in Ready Queue + Execution Time = (Execution Start Time – Arrival Time) + Estimated Execution Time = (10 – 5) + 6 = 5 + 6)
Turn Around Time (TAT) for P3 = 23 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P2 = 28 ms (Time spent in Ready Queue + Execution Time)

Average Turn Around Time = (Turn Around Time for all processes) / No. of Processes
= (Turn Around Time for (P2 + P4 + P3 + P1)) / 4
= (10 + 11 + 23 + 28)/4 = 72/4
= 18 milliseconds

PRE-EMPTIVE SCHEDULING

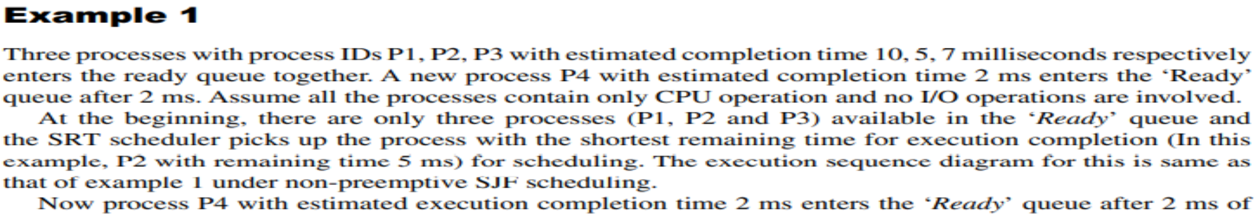
- Every task in the 'Ready' queue gets a chance to execute.
- In this kind of scheduling, the scheduler can preempt (stop temporarily) the currently executing task/process and select another task from the 'Ready' queue for execution. When to pre-empt a task and which task is to be picked up from the 'Ready' queue for execution after preempting the current task is purely dependent on the scheduling algorithm.
- 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'.
- The various types of preemptive scheduling adopted in task/process scheduling are explained below.

  i)Pre-emptive SJF(SRT)
  ii)Round robin scheduling
  iii)Priority based
  iv)Rate based scheduling

i)Pre-emptive SJF(SRT)

- It 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.

- Thus preemptive SJF scheduling always compares the execution completion time (It is same as the remaining time for the new process) of a new process entered the 'Ready' queue with the remaining time for completion of the currently executing process and schedules the process with shortest remaining time for execution. Preemptive SJF scheduling is also known as Shortest Remaining Time (SRT) scheduling.

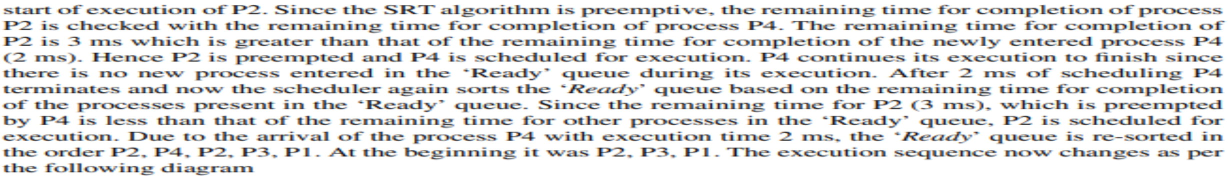


Example 1

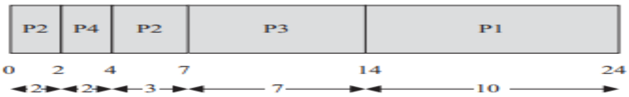
Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds respectively enters the ready queue together. A new process P4 with estimated completion time 2 ms enters the 'Ready' queue after 2 ms. Assume all the processes contain only CPU operation and no I/O operations are involved.

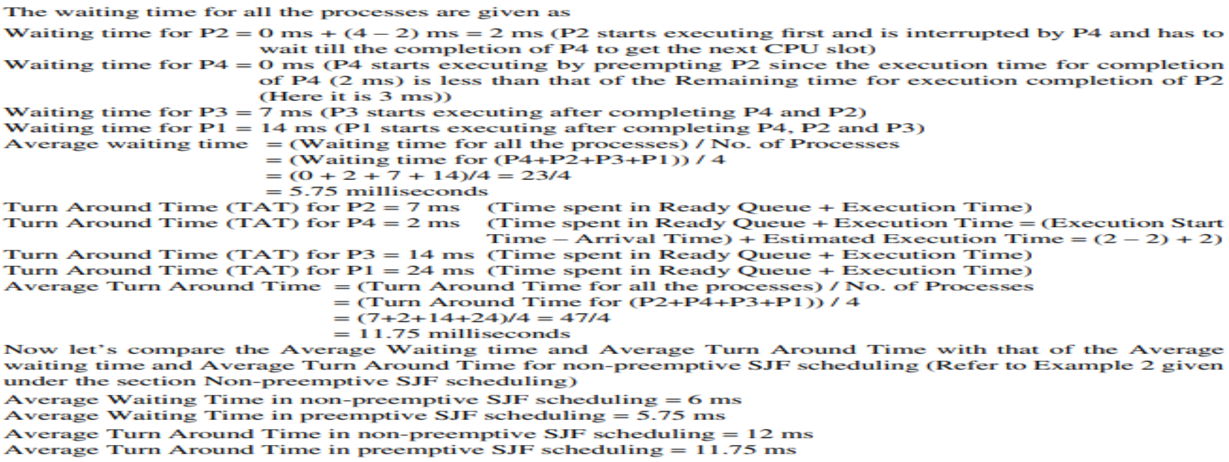
At the beginning, there are only three processes (P1, P2 and P3) available in the '*Ready*' queue and the SRT scheduler picks up the process with the shortest remaining time for execution completion (In this example, P2 with remaining time 5 ms) for scheduling. The execution sequence diagram for this is same as that of example 1 under non-preemptive SJF scheduling.

Now process P4 with estimated execution completion time 2 ms enters the '*Ready*' queue after 2 ms 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 3 ms which is greater than that of the remaining time for completion of the newly entered process P4 (2 ms). 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 2 ms 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. Since the remaining time for P2 (3 ms), which is preempted by P4 is less than that of the remaining time for other processes in the 'Ready' queue, P2 is scheduled for execution. Due to the arrival of the process P4 with execution time 2 ms, the '*Ready*' queue is re-sorted in the order P2, P4, P2, P3, P1. At the beginning it was P2, P3, P1. The execution sequence now changes as per the following diagram





The waiting time for all the processes are given as

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

Waiting time for P4 = 0 ms (P4 starts executing by preempting P2 since the execution time for completion of P4 (2 ms) is less than that of the Remaining time for execution completion of P2 (Here it is 3 ms))

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

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

Average waiting time = (Waiting time for all the processes) / No. of Processes
= (Waiting time for (P4+P2+P3+P1)) / 4
= (0 + 2 + 7 + 14)/4 = 23/4
= 5.75 milliseconds

Turn Around Time (TAT) for P2 = 7 ms (Time spent in Ready Queue + Execution Time)

Turn Around Time (TAT) for P4 = 2 ms (Time spent in Ready Queue + Execution Time = (Execution Start Time – Arrival Time) + Estimated Execution Time = (2 – 2) + 2)

Turn Around Time (TAT) for P3 = 14 ms (Time spent in Ready Queue + Execution Time)

Turn Around Time (TAT) for P1 = 24 ms (Time spent in Ready Queue + Execution Time)

Average Turn Around Time = (Turn Around Time for all the processes) / No. of Processes
= (Turn Around Time for (P2+P4+P3+P1)) / 4
= (7+2+14+24)/4 = 47/4
= 11.75 milliseconds

Now let's compare the Average Waiting time and Average Turn Around Time with that of the Average waiting time and Average Turn Around Time for non-preemptive SJF scheduling (Refer to Example 2 given under the section Non-preemptive SJF scheduling)

Average Waiting Time in non-preemptive SJF scheduling = 6 ms
Average Waiting Time in preemptive SJF scheduling = 5.75 ms
Average Turn Around Time in non-preemptive SJF scheduling = 12 ms
Average Turn Around Time in preemptive SJF scheduling = 11.75 ms

ii)Round robin scheduling

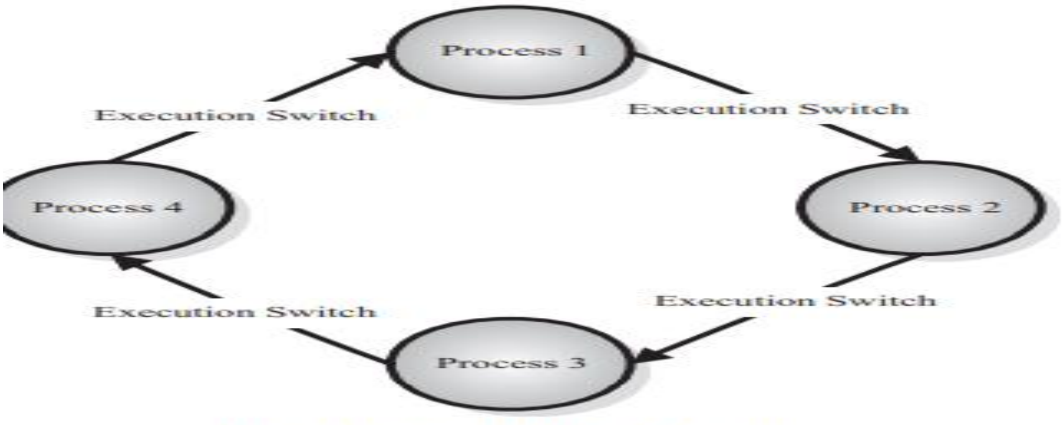
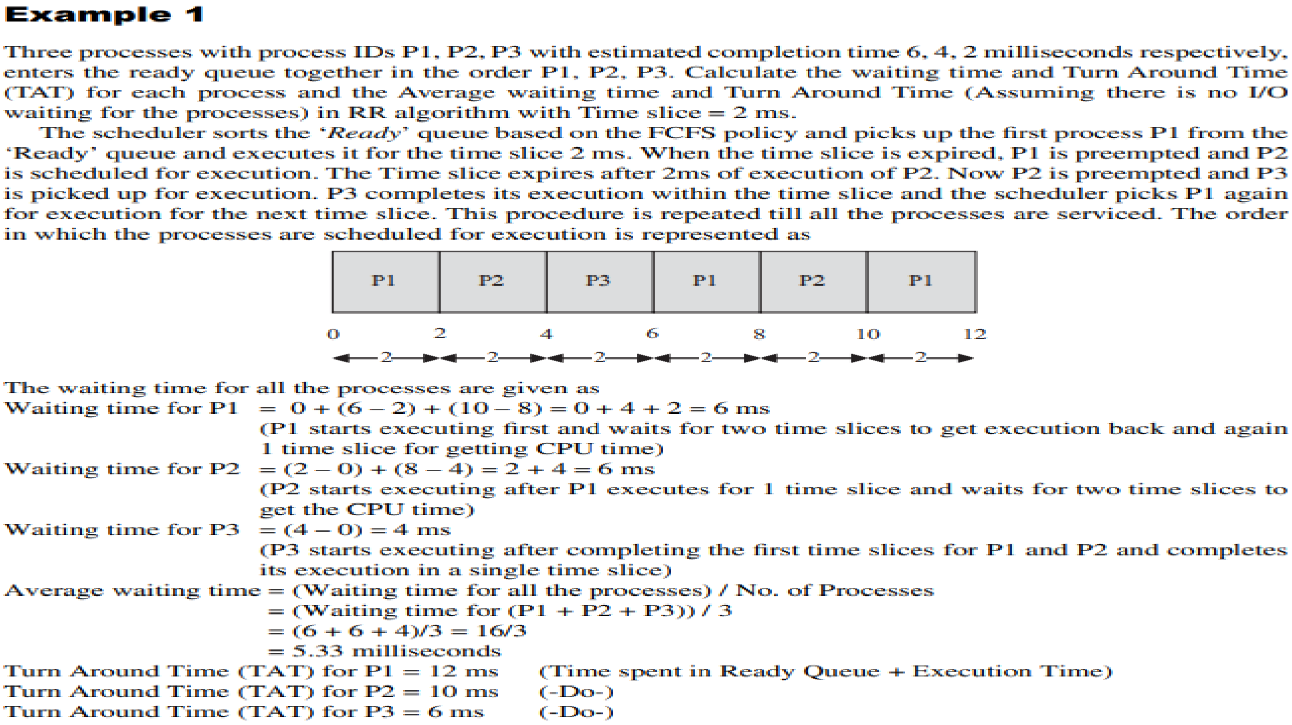


Fig. 10.13 Round Robin Scheduling

- In Round Robin scheduling, each process in the 'Ready' queue is executed for a pre-defi ned time slot.
- The execution starts with picking up the first process in the 'Ready' queue.

- 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 time slice is provided by the timer tick feature of the time management unit of the OS kernel (Refer the
- When the process gets its fixed time for execution is determined by the FCFS policy.
- If a process terminates before the elapse of the time slice, the process releases the CPU voluntarily and the next process in the queue is scheduled for execution by the scheduler. The implementation of RR scheduling is kernel dependent



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 (Assuming there is no I/O waiting for the processes) in RR algorithm with Time slice = 2 ms.

The scheduler sorts the '*Ready*' queue based on the FCFS policy and picks up the first process P1 from the 'Ready' queue and executes it for the time slice 2 ms. When the time slice is expired, P1 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
0–2	2–4	4–6	6–8	8–10	10–12
←2→	←2→	←2→	←2→	←2→	←2→

The waiting time for all the processes are given as

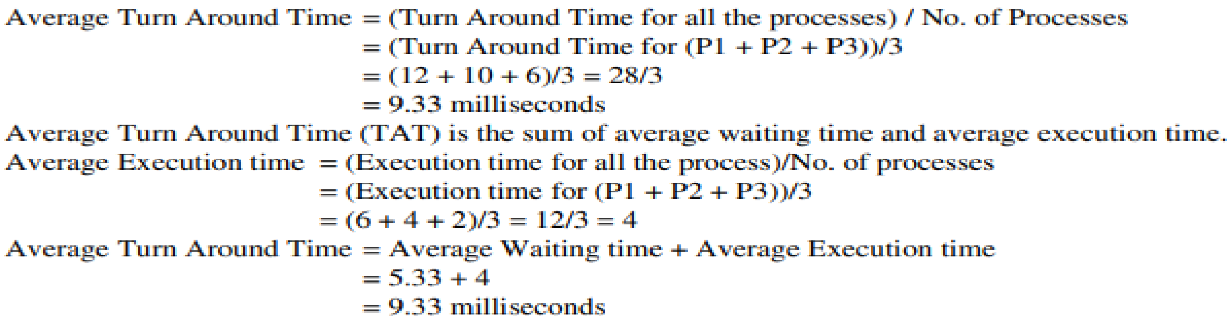
Waiting time for P1 = 0 + (6 – 2) + (10 – 8) = 0 + 4 + 2 = 6 ms
(P1 starts executing first and waits for two time slices to get execution back and again 1 time slice for getting CPU time)

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

Waiting time for P3 = (4 – 0) = 4 ms
(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 the processes) / No. of Processes
= (Waiting time for (P1 + P2 + P3)) / 3
= (6 + 6 + 4)/3 = 16/3
= 5.33 milliseconds

Turn Around Time (TAT) for P1 = 12 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P2 = 10 ms (-Do-)
Turn Around Time (TAT) for P3 = 6 ms (-Do-)



Average Turn Around Time = (Turn Around Time for all the processes) / No. of Processes
= (Turn Around Time for (P1 + P2 + P3))/3
= (12 + 10 + 6)/3 = 28/3
= 9.33 milliseconds

Average Turn Around Time (TAT) is the sum of average waiting time and average execution time.

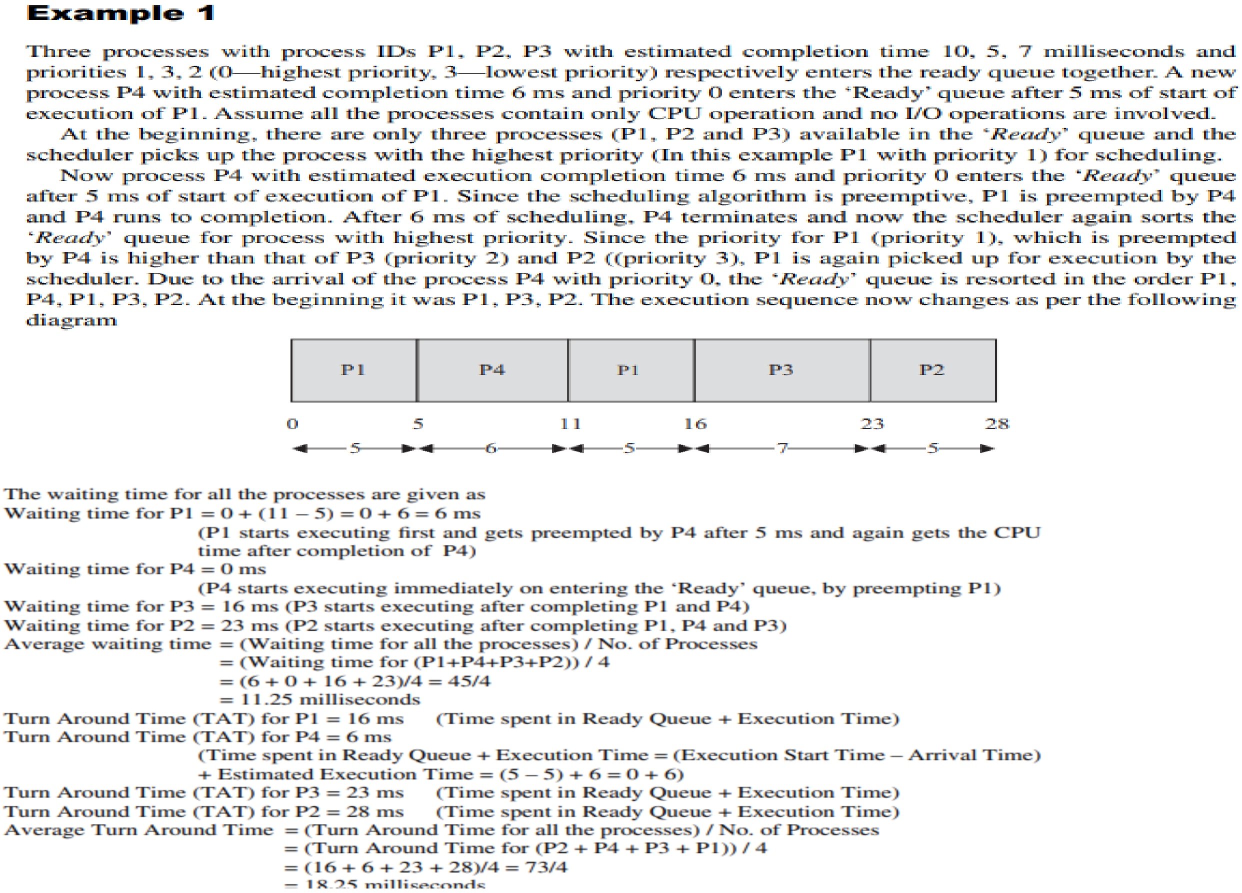
Average Execution time = (Execution time for all the process)/No. of processes
= (Execution time for (P1 + P2 + P3))/3
= (6 + 4 + 2)/3 = 12/3 = 4

Average Turn Around Time = Average Waiting time + Average Execution time
= 5.33 + 4
= 9.33 milliseconds

iii)Priority based scheduling

- 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.

- The priority of a task/process in preemptive scheduling is indicated in the same way as that of the mechanism adopted for nonpreemptive multitasking



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 0 enters the 'Ready' queue after 5 ms of start of execution of P1. Assume all the processes contain only CPU operation and no I/O operations are involved.

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 (In this example P1 with priority 1) 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), P1 is again picked up for execution by the scheduler. Due to the arrival of the process P4 with priority 0, the '*Ready*' queue is resorted in the order P1, P4, P1, P3, P2. At the beginning it was P1, P3, P2. The execution sequence now changes as per the following diagram

P1	P4	P1	P3	P2
0–5	5–11	11–16	16–23	23–28
5	6	5	7	5

The waiting time for all the processes are given as
Waiting time for P1 = 0 + (11 – 5) = 0 + 6 = 6 ms
(P1 starts executing first and gets preempted by P4 after 5 ms and again gets the CPU time after completion of P4)
Waiting time for P4 = 0 ms
(P4 starts executing immediately on entering the 'Ready' queue, by preempting P1)
Waiting time for P3 = 16 ms (P3 starts executing after completing P1 and P4)
Waiting time for P2 = 23 ms (P2 starts executing after completing P1, P4 and P3)
Average waiting time = (Waiting time for all the processes) / No. of Processes
= (Waiting time for (P1+P4+P3+P2)) / 4
= (6 + 0 + 16 + 23)/4 = 45/4
= 11.25 milliseconds
Turn Around Time (TAT) for P1 = 16 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P4 = 6 ms
(Time spent in Ready Queue + Execution Time = (Execution Start Time – Arrival Time) + Estimated Execution Time = (5 – 5) + 6 = 0 + 6)
Turn Around Time (TAT) for P3 = 23 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P2 = 28 ms (Time spent in Ready Queue + Execution Time)
Average Turn Around Time = (Turn Around Time for all the processes) / No. of Processes
= (Turn Around Time for (P2 + P4 + P3 + P1)) / 4
= (16 + 6 + 23 + 28)/4 = 73/4
= 18.25 milliseconds

Q.11.What are the different criteria for the selection of scheduling algorithms.

A good scheduling algorithm must havemaximum CPU utilization minimum turn around time maximum throughput least response time.

CPU Utilisation: Measure of h much percentage of CPU being utilized.(Always CPU utilization should be high)

Throughput: shows the number of processes executed per unit of time.

Turnaround time(TAT): amount of time taken by a process for completing its execution.

Waiting Time: Amount of time spent by a process in ready queue to get CPU time for execution.

Response Time: Time elapsed between submission of a process and its first response.

Q.12.Briefly describe task communication

OR

Identify the RPC based inter process communication.

- The mechanism through which processes/tasks communicate each other is known as Inter Process/Task Communication (IPC).
- Based on the degree of interaction, the processes running on an OS are classified as
- **Co-operating Processes**: In the co-operating interaction model one process requires the inputs from other processes to complete its execution. Co-

operating processes exchanges information and communicate through the following methods.

Co-operation through Sharing: The co-operating process exchange data through some shared resources.

Co-operation through Communication: No data is shared between the processes. But they communicate for synchronisation

- **Competing Processes**: The competing processes do not share anything among themselves but they share the system resources. The competing processes compete for the system resources such as fi le, display device, etc.
- Some IPC mechanisms are:

1.SHARED MEMORY-Pipes and memory mapped objects

2.MESSAGE PASSING-Message Queue,Mailbox,Signalling

3.REMOTE PROCEDURE CALLS AND SOCKET

1.SHARED MEMORY

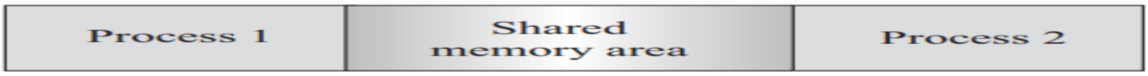


Fig. 10.16 Concept of Shared Memory

- Processes share some area of the memory to communicate among them.
- Information to be communicated by the process is written to the shared memory area.
- Other processes which require this information can read the same from the shared memory area.
- Different mechanisms for implementing shared memory are

Pipes

Memory mapped objects

Pipes

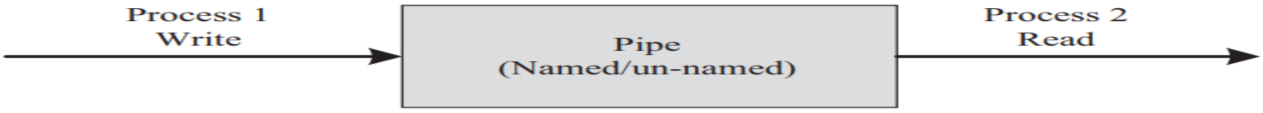


Fig. 10.17 Concept of Pipe for IPC

- Pipe is a section of the shared memory used by processes for communicating.
- Pipes follow the client-server architecture. A process which creates a pipe is known as a pipe server and a process which connects to a pipe is known as pipe client
- Pipes are kernel objects that provide unstructured data exchange and facilitate synchronization among tasks.
- It can be unidirectional, allowing information fl ow in one direction or bidirectional allowing bi-directional information fl ow.
- A unidirectional pipe allows the process connecting at one end of the pipe to write to the pipe and the process connected at the other end of the pipe to read the data, whereas a bi-directional pipe allows both reading and writing at one end.
- Two descriptors, one for each end of the pipe (one end for reading and one for writing), are returned when the pipe is created.
- Data is written via one descriptor and read via the other.
- Data is read from the pipe in FIFO order.
- A pipe provides a simple data flow facility so that the reader becomes blocked when the pipe is empty, and the writer becomes blocked when the pipe is full.

Anonymous Pipes: The anonymous pipes are unnamed, unidirectional pipes used for data transfer between two processes.

Named Pipes: Named pipe is a named, unidirectional or bi-directional pipe for data exchange between processes. Like anonymous pipes, the process which creates the named pipe is known as pipe server. A process which connects to the named pipe is known as pipe client. Named pipes can be used for communicating between processes running on the same machine or between processes running on different machines connected to a network.

Memory Mapped Objects

- Memory mapped object is a shared memory technique adopted by RTOS for allocating a shared block of memory which can be accessed by multiple process simultaneously
- A mapping object is created and physical storage for it is reserved and committed.
- A process can map the entire committed physical area or a block of it to its virtual address space.
- All read and write operation to this virtual address space by a process is directed to its committed physical area.
- Any process which wants to share data with other processes can map the physical memory area of the mapped object to its virtual memory space and use it for sharing the data.
- Windows Embedded Compact RTOS uses the memory mapped object based shared memory technique for Inter Process Communication
- The memory mapped object can be shared between the processes by either passing the handle of the object or by passing its name.
- If the handle of the memory mapped object created by a process is passed to another process for shared access, there is a possibility of closing the handle by the process which created the handle while it is in use by another process.
- If the name of the memory object is passed for shared access among processes, processes can use this name for creating a shared memory object which will open the shared memory object already existing with the given name.
- The OS will maintain a usage count for the named object and it is incremented each time when a process creates/opens a memory mapped object with existing name.
- This will prevent the destruction of a shared memory object by one process while it is being accessed by another process.
- Hence passing the name of the memory mapped object is strongly recommended for memory mapped object based IPC
- Reading and writing to a memory mapped area is same as any read write operation using pointers

2. MESSAGE PASSING

- Message passing is synchronous information exchange mechanism used for IPC
- Using shared memory lots of data can be shared whereas only limited amount of info/data is passed through message passing.
- Message passing is relatively fast and free from the synchronisation overheads
- Based on the message passing operation between the processes, message passing is classified

- into
    - Message Queue
    - Mailbox
    - Signalling

Message Queue

- A message queue is a buffer-like object FIFO queue through which tasks and ISRs send and receive messages to communicate and synchornize with data.
- A message queue is like a pipeline. It temporarily holds messages from a sender until the intended receiver is ready to read them.
- This temporary buffering decouples a sending and receiving task; that is, it frees the tasks from having to send and receive messages simultaneously.
- The elements holding the first and last messages are called the head and tail respectively.
- A thread which wants to communicate with another thread posts the message to the system message queue.
- The kernel picks up the message from the system message queue one at a time and examines the message for finding the destination thread and then posts the message to the message queue of the corresponding thread.
- For posting a message to a thread's message queue, the kernel fills a message structure MSG and copies it to the
- The messaging mechanism is classified into
    - Synchronous and
    - Asynchronous
- In asynchronous messaging, the message posting thread just posts the message to the queue and it will not wait for an acceptance (return) from the thread to which the message is posted
- In synchronous messaging, the thread which posts a message enters waiting state and waits for the message result from the thread to which the message is posted.
- The thread which invoked the send message becomes blocked and the scheduler will not pick it up for scheduling.

Mail box

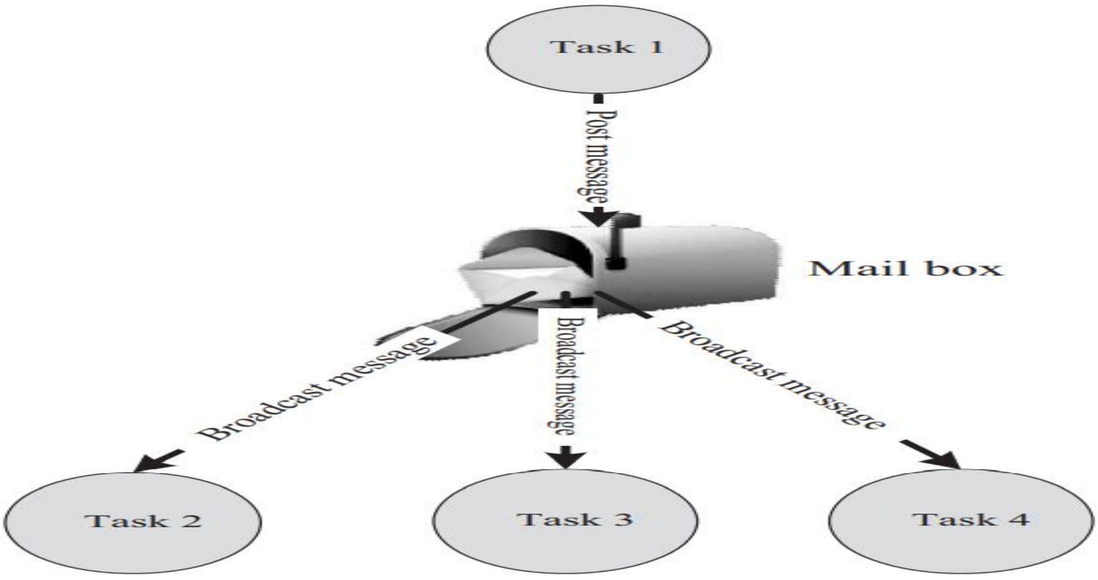


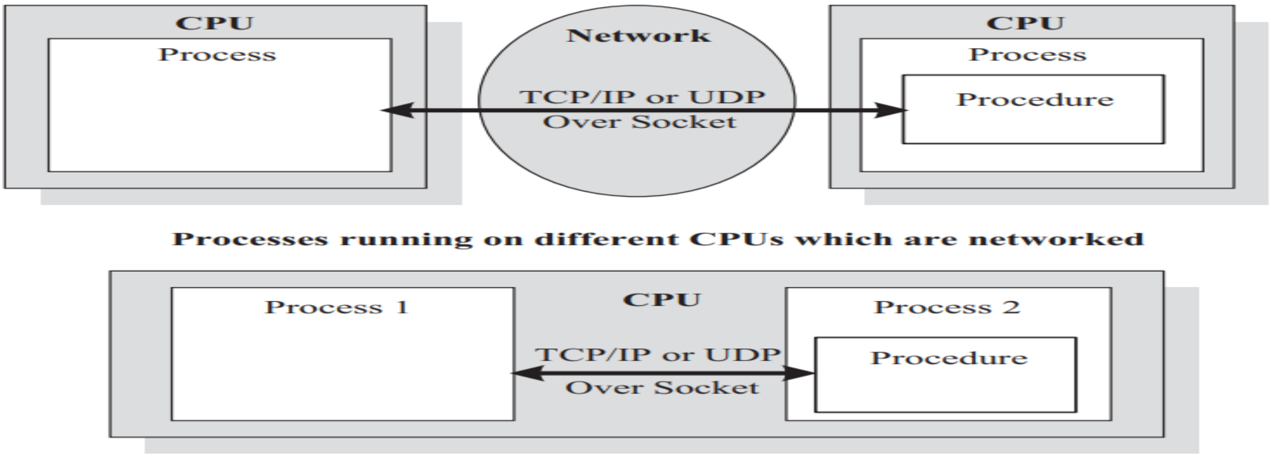
Fig. 10.21 Concept of Mailbox based indirect messaging for IPC

- Used in certain Real-Time Operating Systems for IPC.
- Used for one way messaging.
- The task/thread which wants to send a message to other tasks/threads creates a mailbox for posting the messages.
- The threads which are interested in receiving the messages posted to the mailbox by the mailbox creator thread can subscribe to the mailbox.
- The thread which creates the mailbox is known as 'mailbox server' and the threads which subscribe to the mailbox are known as 'mailbox clients'.
- The mailbox server posts messages to the mailbox and notifies it to the clients which are subscribed to the mailbox.
- The clients read the message from the mailbox on receiving the notification.
- The mailbox creation, subscription, message reading and writing are achieved through OS kernel provided API calls.
- The only difference is in the number of messages supported by them.
- Mailbox is used for exchanging a single message between two tasks .

Signalling

- Primitive way of communication between processes/threads.
- Signals are used for asynchronous notifications where one process/thread fires a signal, indicating the occurrence of a scenario which the other process(es)/thread(s) is waiting.
- Signals are not queued and they do not carry any data.
- The communication mechanisms used in RTX51 Tiny OS is an example for Signalling. The os_send_signal kernel call under RTX 51 sends a signal from one task to a specified task.

3.REMOTE PROCEDURE CALL AND SOCKET



Processes running on different CPUs which are networked

Processes running on same CPU

Fig. 10.22 Concept of Remote Procedure Call (RPC) for IPC

- Remote Procedure Call or is the Inter Process Communication (IPC) mechanism used by a process to call a procedure of another process running on the same CPU or on a different CPU which is interconnected in a network.
- RPC is mainly used for distributed applications like client-server applications. With RPC it is possible to communicate over a heterogeneous network
- The CPU/process containing the procedure which needs to be invoked remotely is known as server.

- The CPU/process which initiates an RPC request is known as client.
- Interface Definition Language (IDL) defines the interfaces for RPC.
- The RPC communication can be either Synchronous (Blocking) or Asynchronous (Non-blocking).
- In the Synchronous communication, the process which calls the remote procedure is blocked until it receives a response back from the other process.
- In asynchronous RPC calls, the calling process continues its execution while the remote process performs the execution of the procedure.
- The result from the remote procedure is returned back to the caller
- On security front, RPC employs authentication mechanisms to protect the systems against vulnerabilities.
- The client applications (processes) should authenticate themselves with the server for getting access.
- Eg:Authentication mechanisms like IDs, public key cryptography (like DES, 3DES), etc. are used by the client

- Sockets are used for RPC communication.
- Socket is a logical endpoint in a two-way communication link between two applications running on a network.
- A port number is associated with a socket so that the network layer of the communication channel can deliver the data to the designated application.
- Sockets are of different types, namely, Internet sockets (INET), UNIX sockets, etc.
- The INET socket works on internet communication protocol.
- TCP/IP, UDP, etc. are the communication protocols used by INET sockets.
- INET sockets are classified into:
  1. Stream sockets
  2. Datagram sockets
- Stream sockets are connection oriented and they use TCP to establish a reliable connection.
- Datagram sockets rely on UDP for establishing a connection.
- The client-server communication model uses a socket at the client side and a socket at the server side.
- A port number is assigned to both of these sockets.
- The client and server should be aware of the port number associated with the socket.
- In order to start the communication, the client needs to send a connection request to the server at the specified port number.
- The client should be aware of the name of the server along with its port number.
- The server always listens to the specified port number on the network.
- Upon receiving a connection request from the client, based on the success of authentication, the server grants the connection request and a communication channel is established between the client and server.
- The client uses the host name and port number of server for sending requests and server uses the client’s name and port number for sending responses.

Q.13. Explain shared memory

Ref Q.12.

Q.14.Explain message passing

Ref Q.12(message passing only

Q.15.Explain RPC (remote procedure call) and sockets

Ref q.12(RPC and socket only)

Q.16.Discuss the different functional and non-functional requirements that need to be addressed in the selection of a Real Time Operating System.

OR

How to chose an RTOS

- A lot of factors needs to be analysed carefully before making a decision on the selection of an RTOS.
- These factors can be either functional or non-functional.

FUNCTIONAL REQUIREMENTS

1.Processor Support- It is essential to ensure the processor support by the RTOS.

2.Memory Requirements-The OS requires ROM memory for holding the OS files and it is normally stored in a non-volatile memory like FLASH. OS also requires working memory RAM for loading the OS services. Since embedded systems are memory constrained, it is essential to evaluate the minimal ROM and RAM requirements for the OS under consideration.

3.Real time Capabilities-The task/process scheduling policies plays an important role in the 'Real-time' behaviour of an OS. Analyse the real-time capabilities of the OS under consideration and the standards met by the operating system for real-time capabilities.

4.Kernel and Interrupt Latency-The kernel of the OS may disable interrupts while executing certain services and it may lead to interrupt latency. For an embedded system whose response requirements are high, this latency should be minimal.

5.Inter Process Communication and Task Synchronisation-The implementation of Inter Process Communication and Synchronisation is OS kernel dependent. Certain kernels may provide a bunch of options whereas others provide very limited options. Certain kernels implement policies for avoiding priority inversion issues in resource sharing.

6.Modularisation Support- It is very useful if the OS supports modularisation where in which the developer can choose the essential modules and re-compile the OS image for functioning. Windows CE is an example for a highly modular operating system.

7.Support for Networking and Communication-The OS kernel may provide stack implementation and driver support for a bunch of communication interfaces and networking. Ensure that the OS under consideration provides support for all the interfaces required by the embedded product.

8.Development Language Support-Certain operating systems include the run time libraries required for running applications written in languages like Java and C#. A Java Virtual Machine (JVM) customised for the Operating System is essential for running java applications. Similarly the .NET Compact Framework (.NETCF) is required for running Microsoft® .NET applications on top of the Operating System.

NON-FUNCTIONAL REQUIREMENTS

1.Custom Developed or Off the Shelf-Depending on the OS requirement, it is possible to go for the complete development of an operating system suiting the embedded system needs or use an off the shelf, readily available operating system, which is either a commercial product or an Open Source product, which is in close match with the system requirements.

The decision on which to select is purely dependent on the development cost, licensing fees for the OS, development time and availability of skilled resources.

2.Cost-The total cost for developing or buying the OS and maintaining it in terms of commercial product and custom build needs to be evaluated before taking a decision on the selection of OS.

3.Development and Debugging Tools Availability-The availability of development and debugging tools is a critical decision making factor in the selection of an OS for embedded design. Certain Operating Systems may be superior in performance, but the availability of tools for supporting the development may be limited.Explore the different tools available for the OS under consideration.

4.Ease of Use-How easy it is to use a commercial RTOS is another important feature that needs to be considered in the RTOS selection.

5.After Sales-For a commercial embedded RTOS, after sales in the form of e-mail, on-call services, etc. for bug fixes, critical patch updates and support for production issues, etc. should be analysed thoroughly.

Q.17.State the need for task synchronisation in a multitasking environment

OR

Explain task synchronisation.

OR

What are the various task synchronization issues.

- The act of making processes aware of the access of shared resources by each process to avoid conflicts is known as 'Task/ Process Synchronisation'.
- Imagine a situation where two processes try to access a shared memory area where one process tries to write to a memory location when the other process is trying to read from this.

Various synchronisation issues are:

Racing/Race condition
Deadlock
Priority inheritance
Priority ceiling
Priority inversion

RACING/RACE CONDITION

- A situation in which multiple processes compete (race) 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.
- They occur when two computer program processes, or threads, attempt to access the same resource at the same time and cause problems in the system.
- Race conditions are considered a common issue for multithreaded applications.
- A race condition may occur if commands to read and write a large amount of data are received at almost the same instant, and the machine attempts to overwrite some or all of the old data while that old data is still being read.

DEADLOCK

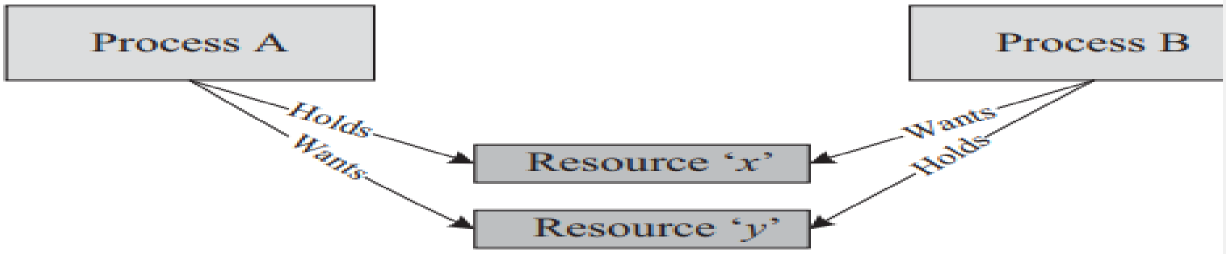


Fig. 10.25 Scenarios leading to deadlock

- Deadlock condition creates a situation where none of the processes are able to make any progress in their execution.
- '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.
- 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 resource x which is currently held by Process A.
- Both hold the respective resources and they compete each other to get the resource held by the respective processes.
- **None of the processes can move ahead.**
- Deadlock is a result of occurence of **all four conditions** listed below.
- The different conditions are:

  1.Mutual exclusion- The criteria that only one process can hold a resource at a time. Typical example is the accessing of display hardware in an embedded device.

  2.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.

  3.No resource preemption- 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.

  4.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.

PREVENT DEADLOCKS

Prevent by **negating one of the four conditions** favouring the deadlock situation.

Ensure that a process does not hold any other resources when it requests a resource.

This can be achieved by -

1. A process must request all its required resource and the resources should be allocated before the process begins its execution.
2. Grant resource allocation requests from processes only if the process does not hold a resource currently.

Ensure that resource preemption (resource releasing) is possible at OS level.

1. Release all the resources currently held by a process if a request made by the process for a new resource is not able to fulfi l immediately.
2. Add the resources which are preempted (released) to a resource list describing the resources which the process requires to complete its execution.
3. Reschedule the process for execution only when the process gets its old resources and the new resource which is requested by the process.

LIVELOCK

- Process in livelock condition changes its state with time.
- In deadlock a process enters in wait state for a resource and continues in that state forever without making any progress in the execution,
- In a livelock condition a process always does something but is unable to make any progress in the execution completion.

STARVATION

- 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 by product of preventive measures of deadlock, scheduling policies favouring high priority tasks and tasks with shortest execution time, etc.

PRIORITY INVERSION

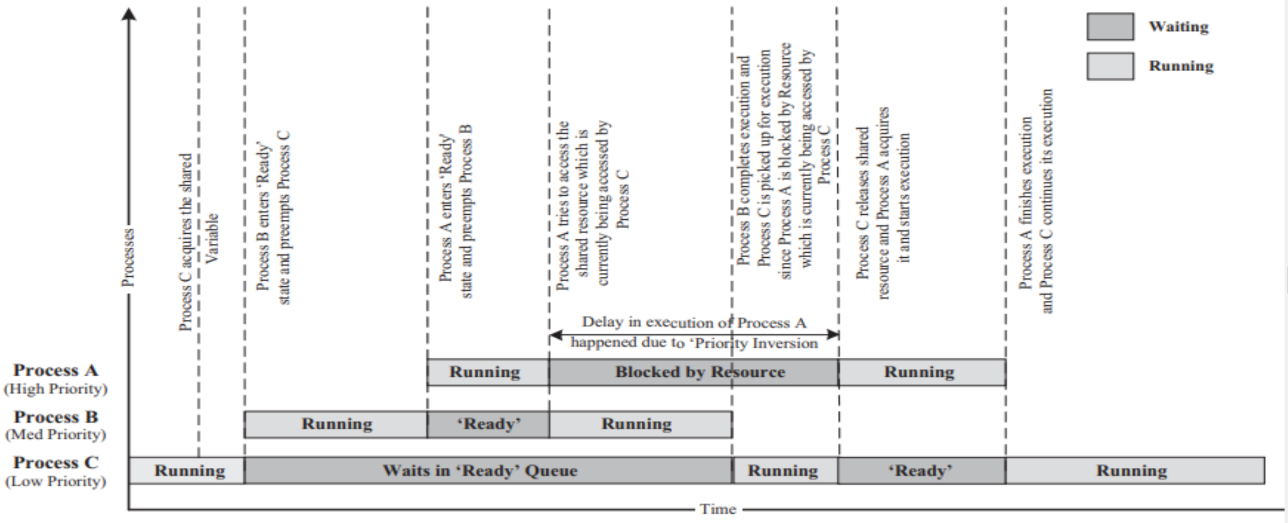
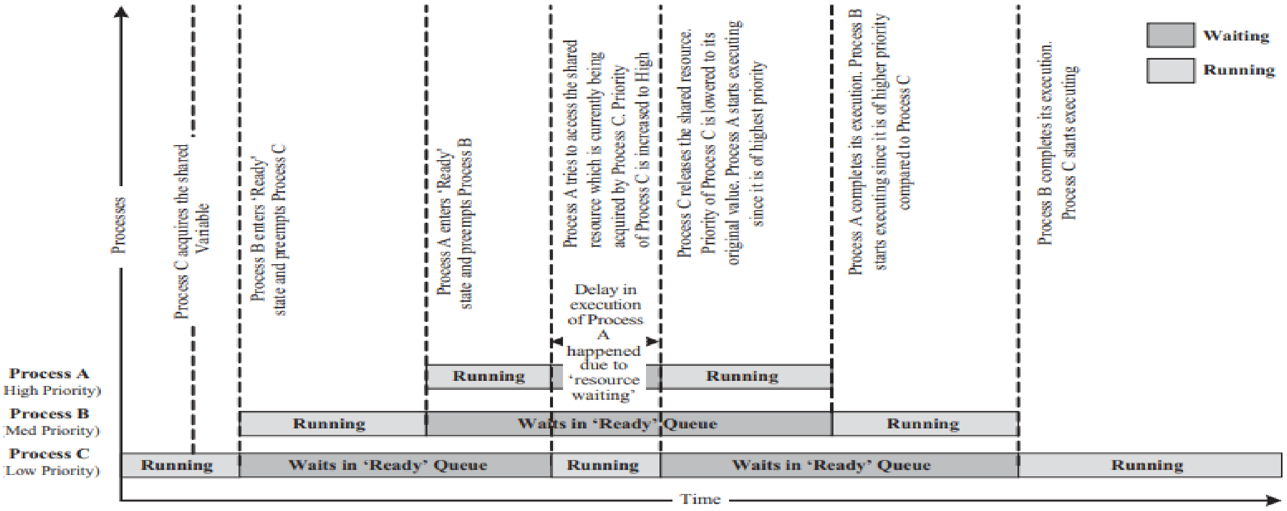

Waiting
Running
Processes
Process C acquires the shared Variable
Process B enters 'Ready' state and preempts Process C
Process A enters 'Ready' state and preempts Process B
Process A tries to access the shared resource which is currently being accessed by Process C
Process B completes execution and Process C is picked up for execution since Process A is blocked by Resource which is currently being accessed by Process C
Process C releases shared resource and Process A acquires it and starts execution
Process A finishes execution and Process C continues its execution
Delay in execution of Process A happened due to 'Priority Inversion'
Process A (High Priority)
Running
Blocked by Resource
Running
Process B (Med Priority)
Running
'Ready'
Running
Process C (Low Priority)
Running
Waits in 'Ready' Queue
Running
'Ready'
Running
Time

Fig. 10.30 Priority Inversion Problem

- 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.
- Let Process A, Process B and Process C be three processes with priorities High, Medium and Low.
- Process A and Process C share a variable 'X' and the access to this variable is synchronised through a mutual exclusion mechanism like Binary Semaphore S.
- Imagine a situation where Process C is ready and is picked up for execution by the scheduler and 'Process C' tries to access the shared variable 'X'.
- 'Process C' acquires the 'Semaphore S' to indicate the other processes that it is accessing the shared variable 'X'.
- Immediately after 'Process C' acquires the 'Semaphore S', 'Process B' enters the 'Ready' state.
- 'Process B' is of higher priority compared to 'Process C', 'Process C' is preempted and 'Process B' starts executing.
- Now imagine 'Process A' enters the 'Ready' state at this stage.

- Since ‘Process A’ is of higher priority than ‘Process B’, ‘Process B’ is preempted and ‘Process A’ is scheduled for execution.
- ‘Process A’ involves accessing of shared variable ‘X’ which is currently being accessed by ‘Process C’.
- Since ‘Process C’ acquired the semaphore for signalling the access of the shared variable ‘X’, ‘Process A’ will not be able to access it.
- Thus ‘Process A’ is put into blocked state (This condition is called **Pending on resource**).
- Now ‘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 highest priority process ‘Process 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’.
- Priority inversion ‘inverts’ the priority of a high priority task with that of a low priority task.
- The commonly adopted priority inversion workarounds are:
  Priority inheritance
  Priority ceiling

PRIORITY INHERITANCE



Processes
Process C acquires the shared Variable
Process B enters ‘Ready’ state and preempts Process C
Process A enters ‘Ready’ state and preempts Process B
Process A tries to access the shared resource which is currently being acquired by Process C. Priority of Process C is increased to High
Process C releases the shared resource. Priority of Process C is lowered to its original value. Process A starts executing since it is of highest priority
Process A completes its execution. Process B starts executing since it is of higher priority compared to Process C
Process B completes its execution. Process C starts executing
Waiting
Running
Delay in execution of Process A happened due to ‘resource waiting’
Process A
High Priority)
Process B
Med Priority)
Process C
Low Priority)
Running
Running
Running
Waits in ‘Ready’ Queue
Running
Running
Waits in ‘Ready’ Queue
Running
Waits in ‘Ready’ Queue
Running
Time

Fig. 10.31 Handling Priority Inversion Problem with Priority Inheritance

- A low-priority task that is currently accessing (by holding the lock) 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 .
- The priority of the low priority task which is temporarily boosted to high is brought to the original value when it releases the shared resource.
- Priority inheritance is only a work around and it **will not eliminate** the delay in waiting the high priority task to get the resource from the low priority task.

- The only thing is that it helps the low priority task to continue its execution and release the shared resource as soon as possible.
- The moment, at which the low priority task releases the shared resource, the high priority task kicks the low priority task out and grabs the CPU

PRIORITY CEILING

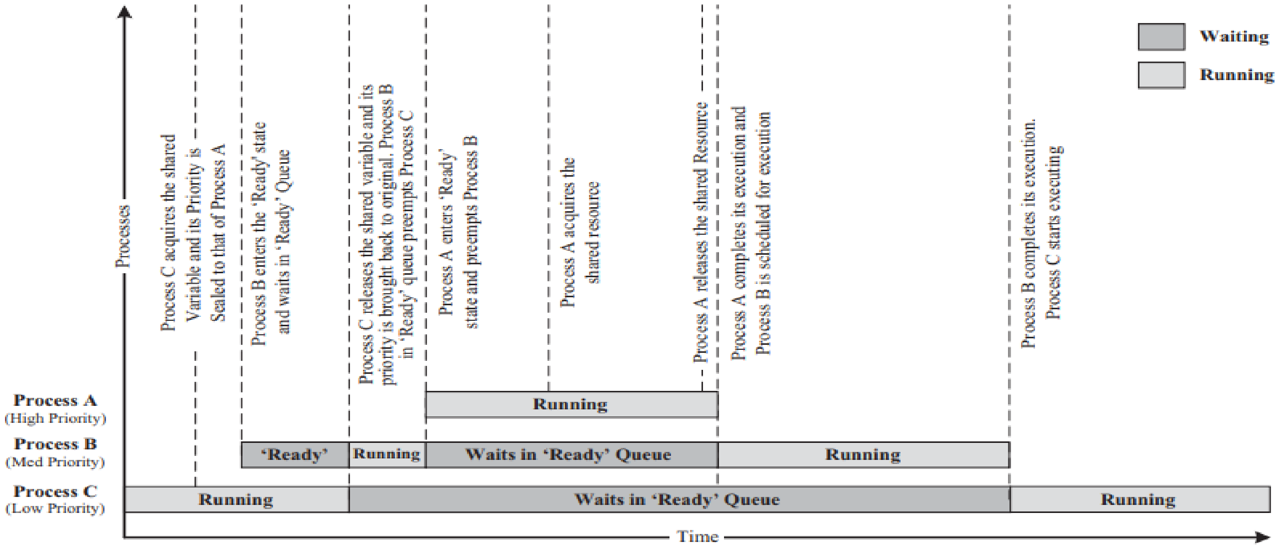

Waiting
Running
Processes
Process C acquires the shared Variable and its Priority is Sealed to that of Process A
Process B enters the 'Ready' state and waits in 'Ready' Queue
Process C releases the shared variable and its priority is brought back to original. Process B in 'Ready' queue preempts Process C
Process A enters 'Ready' state and preempts Process B
Process A acquires the shared resource
Process A releases the shared Resource
Process A completes its execution and Process B is scheduled for execution
Process B completes its execution. Process C starts executing
Process A (High Priority)
Process B (Med Priority)
Process C (Low Priority)
Running
'Ready'
Running
Waits in 'Ready' Queue
Running
Running
Waits in 'Ready' Queue
Running
Time

Fig. 10.32 Handling Priority Inversion Problem with Priority Ceiling

- Here 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 pre-emption 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.
- Since the priority of the task accessing a shared resource is boosted to the highest priority of the task among which the resource is shared, the concurrent access of shared resource is automatically handled..

Q.18. What are the solution for priority inversions

OR

What are the commonly adopted priority inversion techniques.

Ref PRIORITY INHERITANCE

Ref PRIORITY CEILING

Q.19. What is priority inversion

Ref Prev Question

Q.20. What are the different techniques of task synchronisation.

- Process/Task synchronisation is essential for

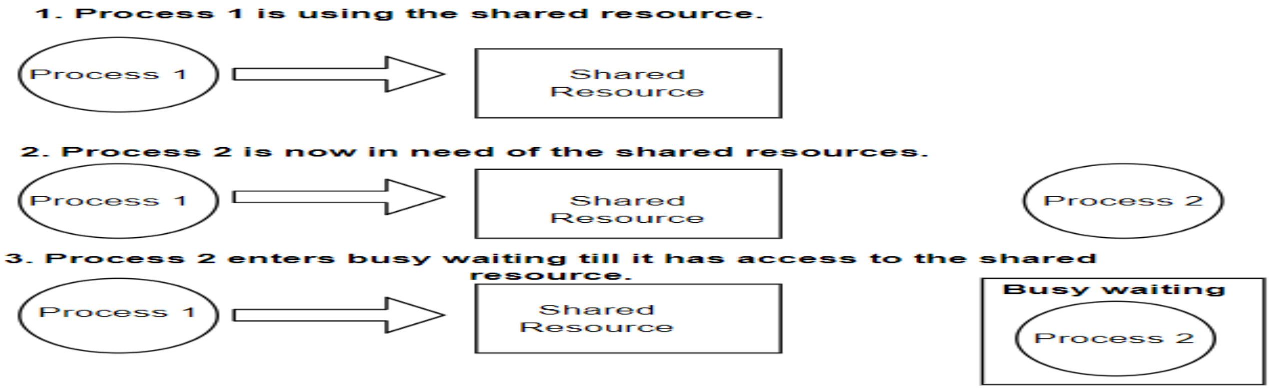
  1. Avoiding conflicts in resource access (racing, deadlock, starvation, livelock, etc.) in a multitasking environment.
  2. Ensuring proper sequence of operation across processes.
  3. Communicating between processes.

- The code memory area which holds the program instructions for accessing a shared resource is known as 'critical section'.
- The exclusive access to critical section of code is provided through mutual exclusion mechanism.
- 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.
- Mutual exclusions can be enforced in different ways.

  1.Mutual Exclusion through Busy Waiting/ Spin Lock
  2.Mutual Exclusion through Sleep & Wakeup
  3.Semaphore
  4.Binary Semaphore

1.**Mutual Exclusion through Busy Waiting/ Spin Lock**

- 'Busy waiting' is the simplest method for enforcing mutual exclusion.
- 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; else the lock is set to '0'.


1. Process 1 is using the shared resource.
Process 1
Shared Resource
2. Process 2 is now in need of the shared resources.
Process 1
Shared Resource
Process 2
3. Process 2 enters busy waiting till it has access to the shared resource.
Process 1
Shared Resource
Busy waiting
Process 2

- A process synchronization technique in which a process/task waits and constantly checks for a condition to be satisfied before proceeding with its execution.

- In busy waiting, a process executes instructions that test for the entry condition to be true, such as the availability of a lock or resource in the computer system.
- There are **two general approaches** to waiting in operating systems:

  Firstly, a process/task can continuously check for the condition to be satisfied while consuming the processor – busy waiting.

  Secondly, a process can wait without consuming the processor.
- In such a case, the process/task is alerted or awakened when the condition is satisfied.
- The latter is known as sleeping, blocked waiting, or sleep waiting.
- Busy looping is usually used to achieve mutual exclusion in operating systems.
- Mutual exclusion prevents processes from accessing a shared resource simultaneously.
- A process is granted exclusive control to resources in its critical section without interferences from other processes in mutual exclusion.
- A critical section is a section of a program code where concurrent access must be avoided.
- In some operating systems, busy waiting can be inefficient because the looping procedure is a waste of computer resources.
- In addition, the system is left idle while waiting. This is particularly wasteful if the task/process at hand is of low priority.
- In that case, resources that can be diverted to complete high-priority tasks are hogged by a low-priority task in busy waiting.
- A workaround solution for the inefficiency of busy waiting that is implemented in most operating systems is the use of a delay function.
- Also known as a sleep system call, a delay function places the process involved in busy waiting into an inactive state for a specified amount of time.

2.Mutual Exclusion through Sleep & Wakeup

- The 'Busy waiting' mutual exclusion enforcement mechanism used by processes makes the CPU always busy checking the lock to see whether they can proceed.
- This results in the wastage of CPU time and leads to high power consumption.
- 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.

3.Semaphore

- Semaphore is a system resource and the process which wants to access the shared 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 at a time.

- Semaphores provide two operations**: wait (P) and signal (V).** The wait operation decrements the value of the semaphore, and the signal operation increments the value of the semaphore. When the value of the semaphore is zero, any process that performs a wait operation will be blocked until another process performs a signal operation.
- Semaphores are used to implement critical sections, which are regions of code that must be executed by only one process at a time. By using semaphores, processes can coordinate access to shared resources, such as shared memory or I/O devices.

ADVANTAGES OF SEMAPHORES

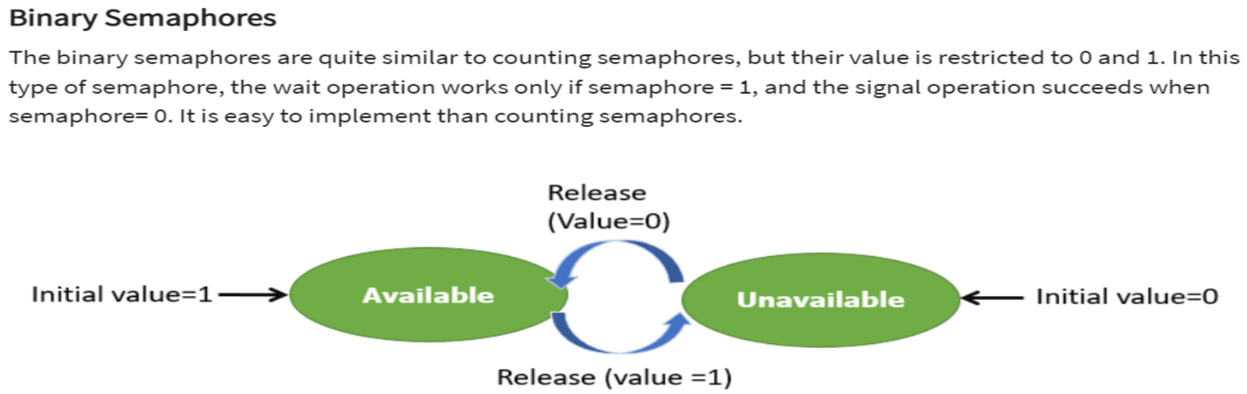
- Semaphores allow only one process into the critical section. They follow the mutual exclusion principle strictly
- There is no resource wastage because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.
- Semaphores are machine independent.

DISADVANTAGES OF SEMAPHORES

- Semaphores are complicated so the wait and signal operations must be implemented in the correct order to prevent deadlocks.
- Semaphores are impractical for last scale use as their use leads to loss of modularity.
- Semaphores may lead to a priority inversion where low priority processes may access the critical section first and high priority processes later.

TYPES OF SEMAPHORES

BINARY SEMAPHORE



Binary Semaphores

The binary semaphores are quite similar to counting semaphores, but their value is restricted to 0 and 1. In this type of semaphore, the wait operation works only if semaphore = 1, and the signal operation succeeds when semaphore= 0. It is easy to implement than counting semaphores.

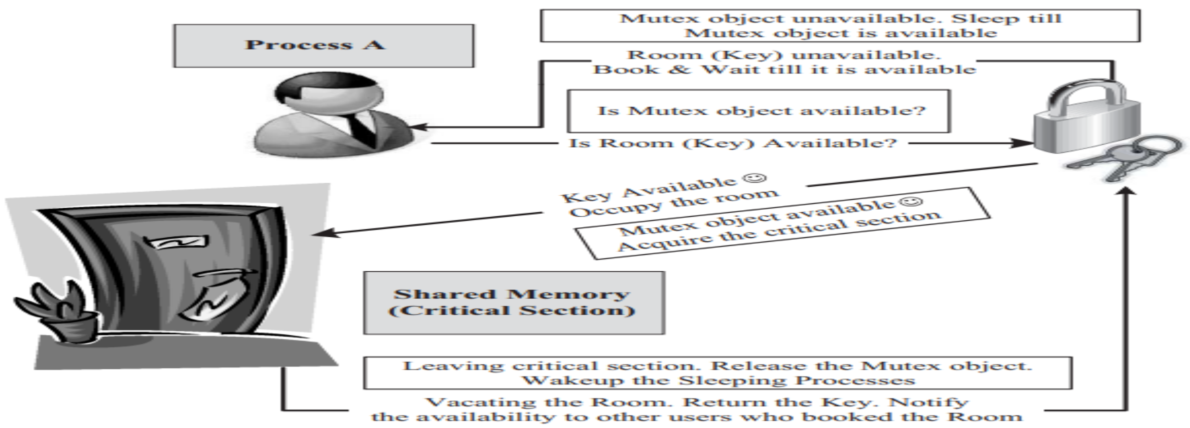

Process A
Mutex object unavailable. Sleep till Mutex object is available
Room (Key) unavailable. Book & Wait till it is available
Is Mutex object available?
Is Room (Key) Available?
Key Available ☺ Occupy the room
Mutex object available☺ Acquire the critical section
Shared Memory (Critical Section)
Leaving critical section. Release the Mutex object. Wakeup the Sleeping Processes
Vacating the Room. Return the Key. Notify the availability to other users who booked the Room

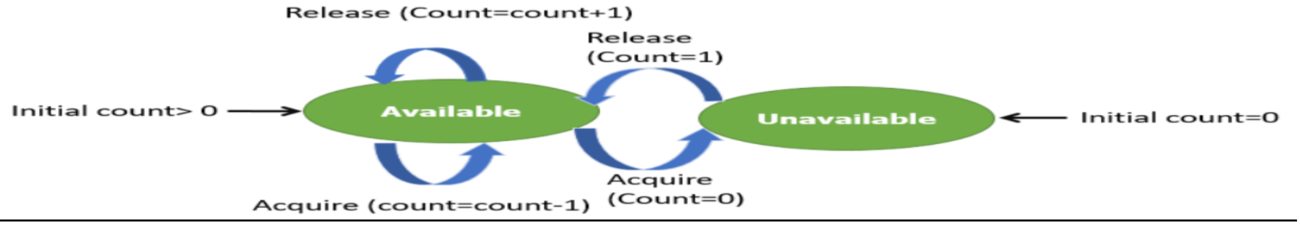
Fig. 10.35 The Concept of Binary Semaphore (Mutex)

- 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. The implementation of binary semaphore is OS kernel dependent.
- Any process/thread can create a 'mutex object' and other processes/threads of the system can use this 'mutex object' for synchronising the access to critical sections.
- Only one process/thread can own the 'mutex object' at a time.
- The state of a mutex object is set to signalled when it is not owned by any process/thread, and set to non-signalled when it is owned by any process/thread.
- A real world example for the mutex concept is the hotel accommodation system.
- The rooms in a hotel are shared for the public. Any user who pays and follows the norms of the hotel can avail the rooms for accommodation.
- A person wants to avail the hotel room facility can contact the hotel reception for checking the room availability .
- If room is available the receptionist will handover the room key to the user.
- If room is not available currently, the user can book the room to get notifications when a room is available. When a person gets a room he/she is granted the exclusive access. When a user vacates the room, he/she gives the keys back to the receptionist.
- The receptionist informs the users, who booked in advance, about the room's availability

COUNTING SEMAPHORE

Counting Semaphores

This type of Semaphore uses a count that helps task to be acquired or released numerous times. If the initial count = 0, the counting semaphore should be created in the unavailable state.



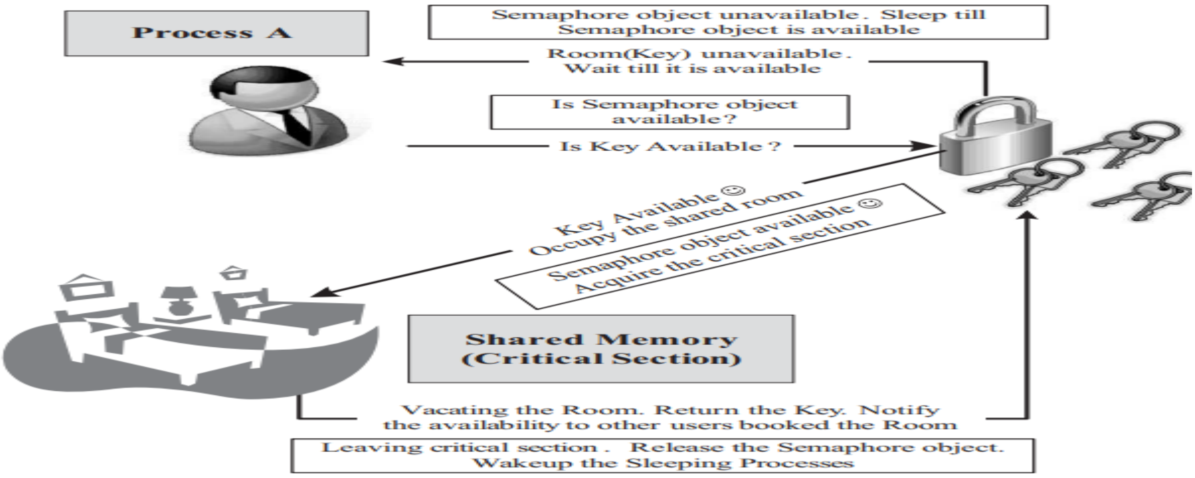

Process A
Semaphore object unavailable. Sleep till Semaphore object is available
Room(Key) unavailable. Wait till it is available
Is Semaphore object available?
Is Key Available ?
Key Available ☺ Occupy the shared room
Semaphore object available ☺ Acquire the critical section
Shared Memory (Critical Section)
Vacating the Room. Return the Key. Notify the availability to other users booked the Room
Leaving critical section. Release the Semaphore object. Wakeup the Sleeping Processes

Fig. 10.34 The Concept of Counting Semaphore

- ‘Counting Semaphore’ limits the access of resources by a fixed number of processes/threads.
- ‘Counting Semaphore’ maintains a count between zero and a maximum 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 ‘signalled’ when the count of the object is greater than zero
- A real world example for the counting semaphore concept is the dormitory system for accommodation .
- A dormitory contains a fixed number of beds (say 5) and at any point of time it can be shared by the maximum number of users supported by the dormitory.
- If a person wants to avail the dormitory facility, he/she can contact the dormitory caretaker for checking the availability. If beds are available in the dorm the caretaker will hand over the keys to the user.
- If beds are not available currently, the user can register his/her name to get notifications when a slot is available.
- Those who are availing the dormitory shares the dorm facilities like TV, telephone, toilet, etc.
- When a dorm user vacates, he/she gives the keys back to the caretaker. The caretaker informs the users, who booked in advance, about the dorm availability
- The count associated with a ‘Semaphore object’ is decremented by one when a process/thread acquires it and the count is incremented by one when a process/thread releases the ‘Semaphore object’.