

# MODULE V

## Real Time Operating System (RTOS) Based Embedded System Design

---

Dr. Rejeesh Rayaroth  
Asst. Professor  
CSE  
MITE

Module 5: Real Time Operating System(RTOS) Based Embedded System Design	
<b>Module 5: Real Time Operating System(RTOS) Based Embedded System Design</b>	<b>No. of Hrs: 9+4</b>
<p>RTOS: Concept, task, process and threads (Only POSIX Threads with an example program), Thread preemption, Multiprocessing and Multitasking, Task Scheduling, Task Communication: Shared memory, message passing, Remote Procedure call and socket, Task synchronization issues, Task synchronization Techniques.</p> <p><b>Laboratory Components:</b></p> <ol style="list-style-type: none"> <li>1. With the help of the Embedded controller (Arduino, Raspberry Pi) control a DC motor</li> <li>2. With the help of the Embedded controller (Arduino, Raspberry Pi) control a Stepper motor and rotate it in clockwise and anti-clockwise direction</li> </ol>	

## Operating System Basics

---

### ❑ **Operating System Functions and Management**

- ❖ Acts as a bridge between user applications/tasks and system resources.
- ❖ Provides a set of system functionalities and services.
- ❖ Manages system resources and makes them available to applications/tasks as needed.

### ❑ **A normal Computing System is A collection of different subsystems:**

- ❖ I/O subsystems
- ❖ Working memory
- ❖ Storage memory

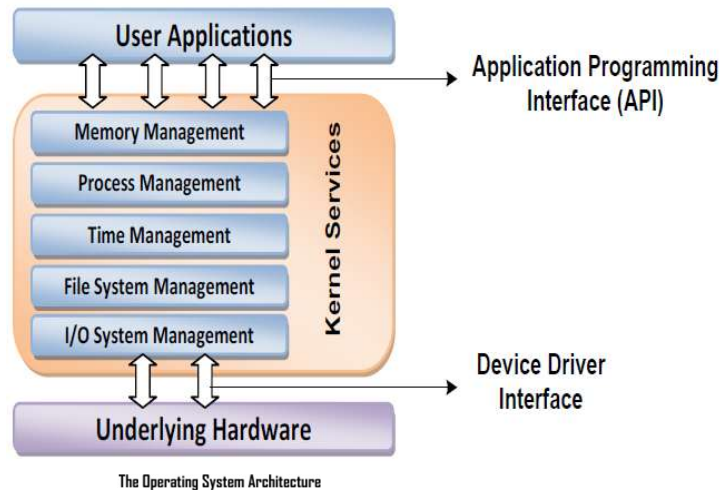
---

### ❑ **Primary Functions of an Operating System**

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

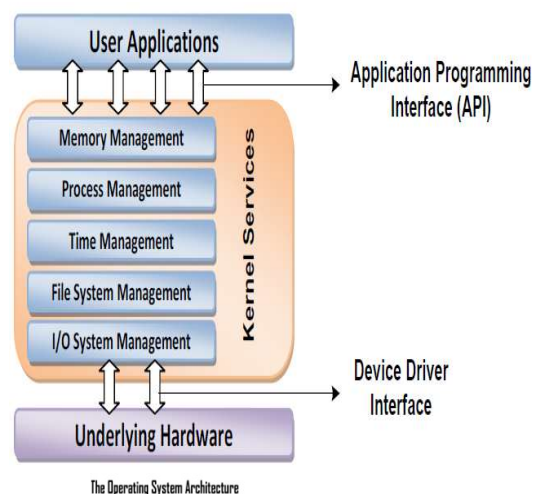
## Basic Components Of an Operating System

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



## The Kernel

- The kernel is the *core* of the **operating system** and is responsible for *managing the system resources* and the *communication among the hardware* and *other system services*
- Kernel acts as the *abstraction layer* between *system resources* and *user applications*.
- Kernel contains a set of system libraries and services.

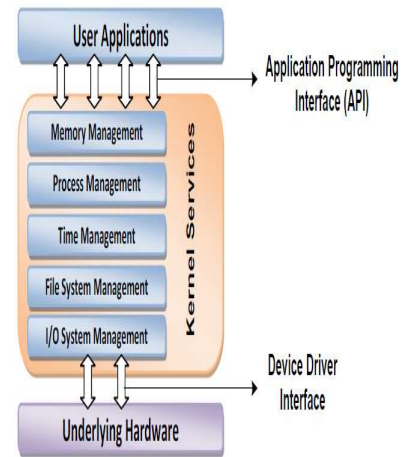


## Process Management:

❑ For a general-purpose OS, the kernel contains different services for handling the following

❑ **Process Management:** Process management deals with managing the processes/tasks. Process management include the following

- ❖ Setting up memory space for the process.
- ❖ Loading process code into the allocated memory.
- ❖ Allocating system resources as required.
- ❖ Scheduling and managing execution of the process.
- ❖ Setting up and managing the Process Control Block (PCB).
- ❖ Inter Process Communication and synchronization.
- ❖ Process termination/deletion.



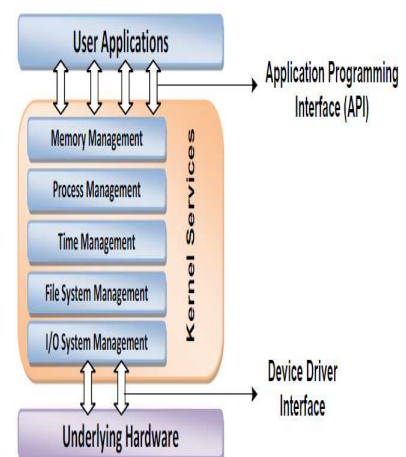
## Primary Memory Management

❑ **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.

❑ Memory Management Unit (MMU) of the kernel is Responsible of following functions

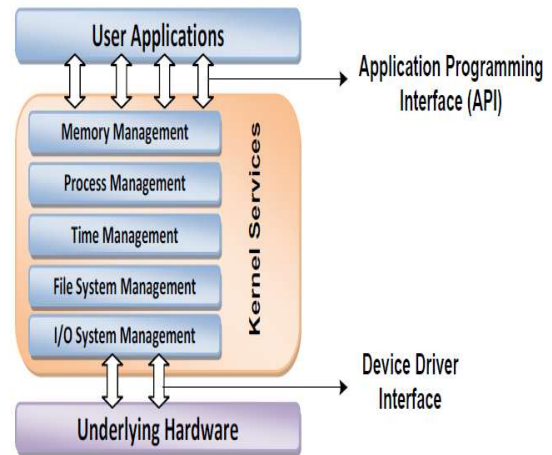
- ❖ Tracking memory usage – Identifies which part of the memory is currently used by each process.
- ❖ Dynamic memory allocation – Allocates and deallocates memory space as needed.



## File System Management

### ❑ 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.
- ❑ Each of these files differ in the kind of information they hold and the way in which the information is stored. The file operation is a useful service provided by the OS.



### ❑ File System Management in the Kernel is responsible for

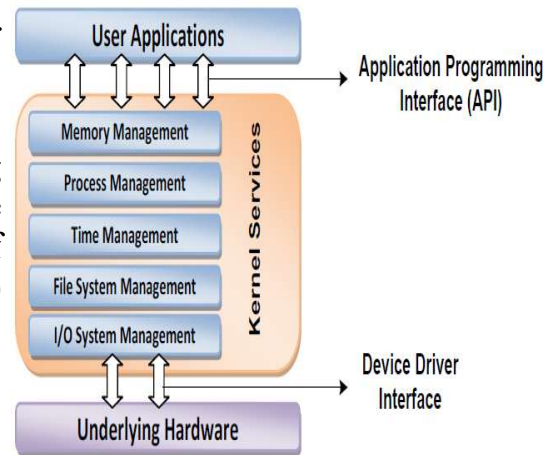
- ❖ File operations – Creation, deletion, and alteration of files.
- ❖ Directory operations – Creation, deletion, and alteration of directories.
- ❖ Storage management – Saving files in secondary storage (e.g., Hard disk).
- ❖ Automatic file space allocation – Based on available free space.
- ❖ Flexible naming convention – Allows ease of file identification.

### ❑ OS Dependency in File System Management

- ❖ Different OS kernels support specific file management operations.
- ❖ Example – Microsoft® DOS OS kernel has a different file system from UNIX Kernel.

## I/O System (Device) Management

- ❑ Kernel is responsible for routing the I/O requests coming from different user applications to the appropriate I/O devices of the system.
- ❑ In a well-structured OS, the direct accessing of I/O devices are not allowed and the access to them are provided through a set of Application Programming Interfaces (APIs) exposed by the kernel.



### ❑ I/O Device Management in the Kernel

- ❖ The kernel maintains a list of all I/O devices in the system.
- ❖ List may be predefined at *kernel build time* or updated dynamically when new devices are installed.
- ❖ **Example:** Windows NT kernel updates when a **plug 'n' play** USB device is attached.

### ❑ Device Manager Responsibilities

- ❖ Handles all I/O device operations (Name may vary across different OS kernels).
- ❖ Communicates with I/O devices using **low-level system calls** via **device drivers**.
- ❖ Device drivers are specific to a device or a class of devices.

### ❑ Device Manager Functions

- ❖ Loading and unloading of device drivers.
- ❖ Exchanging information and system-specific control signals between the system and the device.

## Secondary Storage Management

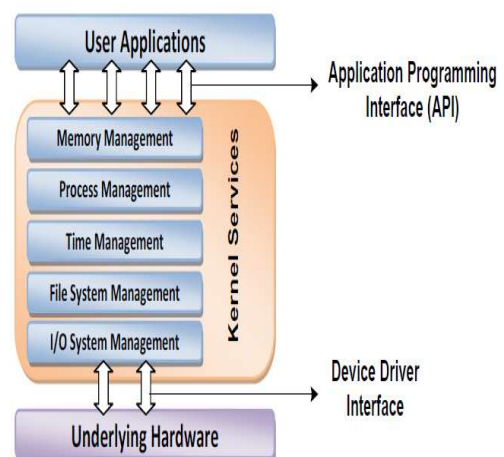
❑ **Secondary Storage Management:** deals with managing the secondary storage memory devices (if any) connected to the system.

❑ Secondary memory is used as backup medium for programs and data, as main memory is volatile.

❑ In most of the systems secondary storage is kept in disks (hard disks).

❑ The secondary storage management service of kernel deals with –

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



## Protection Systems

---

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

## Interrupt Handler:

---

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

## Other kernel services

---

- ❑ above discussed are the services provided by kernel to the operating system
- ❑ **Kernel Services in an Operating System**
  - ❖ The kernel offers important system services, but may include more or fewer components depending on the OS type.
  - ❖ Additional components/services may be provided beyond the core functionalities.
- ❑ **Examples of Add-On Kernel Components/Services**
  - ❖ Network communication & management.
  - ❖ User-interface graphics.
  - ❖ Timer services (delays, timeouts, etc.).
  - ❖ Error handling.
  - ❖ Database management.

- 
- ❑ **Kernel Interface and APIs**
    - ❖ The kernel hosts various applications/services.
    - ❖ Provides a standard Application Programming Interface (API) for user applications.
    - ❖ User applications use API calls to access kernel services

## Kernel Space and User Space:

---

### Kernel Space :

The program code corresponding to the kernel applications/ services are kept in a contiguous area of primary(working) memory and is protected from the unauthorized access by user programs/ applications.

The memory space at which the kernel code is located is known as "**Kernel Space**".

### User Space

All user applications are loaded to a specific area of primary memory and this memory area is referred as "**User Space**".

---

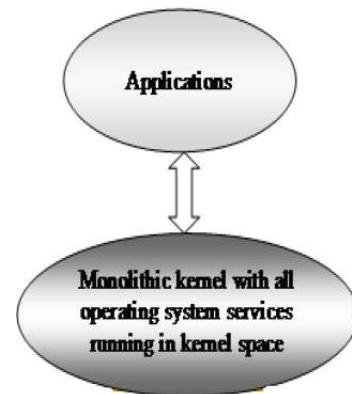
The partitioning of memory into kernel and user space is purely Operating System dependent.

Most of the operating systems keep the kernel application code in main memory and it is not swapped out into the secondary memory.

## Monolithic Kernel and Microkernel

---

- ❑ Kernel forms the heart of OS.
- ❑ Different approaches are adopted for building an operating system kernel.
- ❑ Based on the kernel design, kernels can be classified into
  - ❖ "Monolithic" and
  - ❖ "Micro".
- ❑ **Monolithic Kernel:** In monolithic kernel Architecture, all kernel services run in the kernel space.
- ❑ All kernel modules run within the same memory space under a single kernel thread.



## Drawback of monolithic kernel

---

- ❑ 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.

## Microkernel:

### ❑ **Microkernel:**

- ❑ The microkernel design incorporates only essential set of operating system services into the kernel.
- ❑ The rest of the operating systems services are implemented in program known as “Servers” which runs in user space.
- ❑ The memory management, timer systems and interrupt handlers are the essential services, which forms the part of the microkernel.

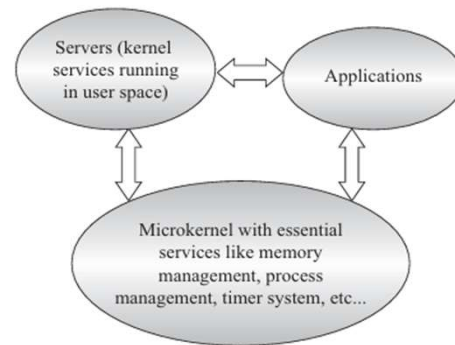


Fig. 10.3 The Microkernel model

## Benefits of MicroKernel

- ❑ The benefits of micro kernel based designs are –

### ❑ **Robustness:**

- ❖ If a problem is encountered in any of the services, which runs as a server can be reconfigured and restarted without the restarting the entire OS.
- ❖ Here chances of corruption of kernel services are ideally zero.

### ❑ **Configurability:**

- ❖ Any services, which runs as a server application can be changed without the need to restart the whole system.
- ❖ This makes the system dynamically configurable.

## Types of Operating Systems

---

- ❑ Depending on the type of kernel and kernel services, purpose and type of computing system,
- ❑ Operating Systems are classified into different types
  - ❖ General Purpose Operating Systems.
  - ❖ Real Time Operating Systems.

### General Purpose Operating Systems:

- ❑ The operating systems, which are deployed in general computing systems, are referred as GPOS. The GPOSs are often quite non-deterministic in behavior.
- ❑ Examples: Windows 10/ 8.x/ XP / MS-DOS

## Real Time Operating Systems

---

- ❑ *Real Time* implies *deterministic* in timing behavior.
- ❑ RTOS services consumes only *known and expected amounts of time* regardless the *number of services*.
- ❑ RTOS implements policies and rules concerning *time-critical allocation* of a system's resources.
- ❑ RTOS decides which applications should run in which order and how much time needs to be allocated for each application.
- ❑ **Examples:** Windows Embedded Compact, QNX, VxWorks MicroC /OS-II

## The Real-Time kernel

---

- ❑ **The Real-Time kernel:** The kernel of a Real-Time OS is referred as *Real-Time kernel*.
- ❑ The Real-Time kernel is highly specialized and it contains only the *minimal set of services* required for running user applications/ tasks.
- ❑ The basic functions of a Real-Time kernel are listed below:
  - ❖ **Task/ Process management**
  - ❖ **Task/ Process scheduling**
  - ❖ **Task/ Process synchronization**
  - ❖ **Error/ Exception handling**
  - ❖ **Memory management**
  - ❖ **Interrupt handling**
  - ❖ **Time management.**

## Task/ Process Management:

---

- ❑ **Task/ Process Management:** Deals with setting up the memory space for the tasks, loading the task's code into the memory space, allocating system resources and setting up a Task Control Block (TCB) for the task and task/process termination/deletion.
- ❑ A **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. (E.g. State = "Ready" for a task which is ready to execute)
  - ❖ **Task Type:** 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 :** Task priority(Ex:Task priority=1 for task with priority=1)
  - ❖ **Task Context Pointer:** 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)
  - ❖ **Other Parameters:** Other relevant task parameters.
- ❑ The parameters and implementation of the **TCB** is **kernel dependent**.
  - ❑ The **TCB** parameters vary **across different kernels** based on the **task management** implementation.

## ***Task/ Process Scheduling:***

---

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

## ***Task/ Process Synchronization:***

---

### **❑ *Task/ Process Synchronization:***

Deals with synchronizing the concurrent access of a resource, which is shared across multiple tasks and the communication between various tasks.

## ***Error/ Exception Handling:***

---

**❑ *Error/ Exception Handling:*** Deals with registering and handling the errors occurred/ exceptions rose during the execution of tasks.

**❑ *Errors/ Exceptions*** can happen at the kernel level services or at task level.

❖ **Deadlock is an example for kernel level exception**, whereas **timeout** is an example for a task level exception.

❖ **Deadlock** is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.

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

❖ The **OS kernel** gives the information about the error in the form of a **System call (API)**.

## ***Memory Management:***

---

- ❑ ***Memory Management:*** The memory management function of an RTOS kernel is slightly different compared to the General Purpose Operating Systems.
  - ❖ In general, the memory allocation time increases depending on the size of the block of memory need to be allocated and the state of the allocated memory block.
  - ❖ RTOS achieves predictable timing and deterministic behavior, by compromising the effectiveness of memory allocation.
  - ❖ RTOS generally uses “***block***” based memory allocation technique, instead of the usual ***dynamic memory allocation*** techniques used by the GPOS.
  - ❖ RTOS kernel uses blocks of fixed size of dynamic memory and the block is allocated for a task on a need basis.
  - ❖ The blocks are stored in a “***Free buffer Queue***”

- 
- ❖ Most of the RTOS kernels allow tasks to access any of the memory blocks without any memory protection to achieve predictable timing and avoid the timing overheads.
  - ❖ Some commercial RTOS kernels allow memory protection as optional and the kernel enters a “***fail-safe mode***” when an illegal memory access occurs.
  - ❑ The memory management function a block of fixed memory is always allocated for tasks on need basis and it is taken as a unit. Hence, there will not be any memory fragmentation issues.

## Interrupt Handling:

---

- ❑ **Interrupt Handling:** Deals with the handling of various interrupts. 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.
  - ❖ *Divide by zero, memory segmentation error* etc are examples of Synchronous interrupts.
- ❑ For “synchronous interrupts”, the *interrupt handler* runs in the same context of the *interrupting task*.

- 
- ❑ Interrupts which occurs at any point of execution of any task, and are not in sync with the currently executing task are *Asynchronous interrupts*.
    - ❖ *Timer overflow interrupts, serial data reception/ transmission interrupts* etc., are examples for asynchronous interrupts.
  - ❑ For asynchronous interrupts, the *interrupt handler* is usually written as *separate task*(depends on OS Kernel implementation) and it runs in a different context.
  - ❑ Hence, a *context switch* happens while handling the asynchronous interrupts.

## ***Time Management:***

---

- ❑ ***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 chip (hardware timer).
  - ❖ The hardware timer is programmed to interrupt the processor/controller at a fixed rate.
  - ❖ This timer interrupt is referred as "***Timer tick***". The "***Timer tick***" is taken as the timing reference by the kernel.
  - ❖ The "***Timer tick***" interval may vary depending on the hardware timer.
  - ❖ Usually, the "***Timer tick***" varies in the microseconds range.
  - ❖ The time parameters for tasks are expressed as the multiples of the "***Timer tick***".

- 
- ❑ The System time is updated based on the "***Timer tick***".
  - ❑ If the System time register is 32 bits wide and the "Timer tick" interval is 1 microsecond, the System time register will reset in,
    - ❖  $2^{32} * 10^{-6} / (24 * 60 * 60) = \sim 0.0497 \text{ Days} = 1.19 \text{ Hours}$
  - ❑ If the "Timer tick" interval is 1 millisecond, the System time register will reset in
    - ❖  $2^{32} * 10^{-3} / (24 * 60 * 60) = 49.7 \text{ Days} = \sim 50 \text{ Days}$
  - ❑ The "Timer tick" interrupt is handled by the "Timer Interrupt" handler of kernel.

- 
- ❑ The "*Timer tick*" interrupt can be utilized 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 the System time register if the timer tick count is greater than the **maximum range** available for System time register.
    - ❖ Update the timers implemented in kernel (Increment or decrement the timer registers for each timer depending on the count direction setting for each register. Increment registers with count direction setting = "*count up*" and decrement registers with count direction setting = "*count down*").
    - ❖ Activate the periodic tasks, which are in the idle state.

## *Hard Real-Time*

---

- ❑ ***Hard Real-Time***: A Real Time Operating Systems which strictly adheres to the timing constraints for a task is referred as hard real-time systems.
- ❑ A Hard Real Time system must meet the deadlines for a task without any slippage. Missing any deadline may produce catastrophic results for Hard Real Time Systems, including permanent data lose and irrecoverable damages to the system/users.
- ❑ Hard real-time systems emphasize on the principle. "*A late answer is a wrong answer*".
  - ❖ For example, Air bag control systems and Anti-lock Brake Systems (ABS) of vehicles are typical examples of Hard Real Time Systems.
  - ❖ Most of the Hard Real Time Systems are automatic.

## *Soft Real-Time:*

---

- ❑ **Soft Real-Time:** Real Time Operating Systems that does not guarantee meeting deadlines, but, offer the best effort to meet the deadline are referred as soft real-time systems.
- ❑ Missing deadlines for tasks are acceptable if the frequency of deadline missing is within the compliance limit of the Quality of Service (QoS).
  - ❖ Soft real-time system emphasizes on the principle “*A late answer is an acceptable answer, but it could have done bit faster*”.
  - ❖ Automatic Teller Machine (ATM) is a typical example of Soft Real Time System. If the ATM takes a few seconds more than the ideal operation time, nothing fatal happens.

## TASKS, PROCESS AND THREADS

---

- ❑ The term ‘*task*’ refers to something that needs to be done.
- ❑ In our day-to-day life, we are bound to the execution of a number of tasks. The task can be the one assigned by our **managers** or the one assigned by our **professors/teachers** or the one related to our personal or family needs.
- ❑ In addition, we will have an *order of priority* and *schedule/timeline* for executing these tasks.
- ❑ In the operating system context, a task is defined as the *program in execution* and the related information maintained by the operating system for the program.
- ❑ Task is also known as ‘**Job**’ in the operating system context.
- ❑ A program or part of it in execution is also called a ‘*Process*’.
- ❑ The terms ‘*Task*’, ‘*Job*’ and ‘*Process*’ refer to the same entity in the operating system context and most often they are used interchangeably

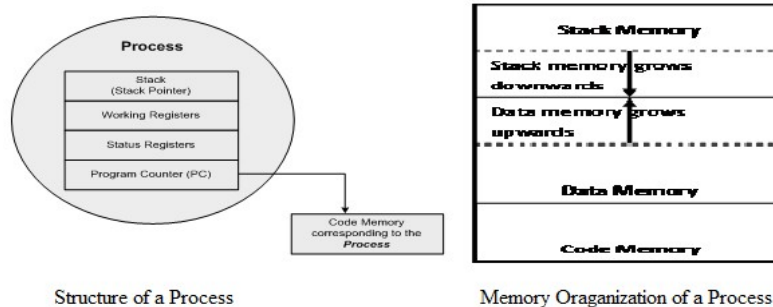
# PROCESS

## ❑ Process:

- ❑ “Process” is a program, or part of it, in execution. Process is also known as an instance of a program in execution.
- ❑ **Structure of a Processes:** The concept of “Process” leads to *concurrent execution of tasks* and thereby, *efficient utilization of the CPU and other system resources*.
- ❑ Concurrent execution is achieved through the *sharing of CPU among the processes*.

- ❑ A process mimics a processor in properties and holds a set of registers, process status, a Program Counter (PC) to point to the next executable instruction of the process, a stack for holding the local variables associated with the process and the code corresponding to the process.

This can be visualized as shown in the following Figure

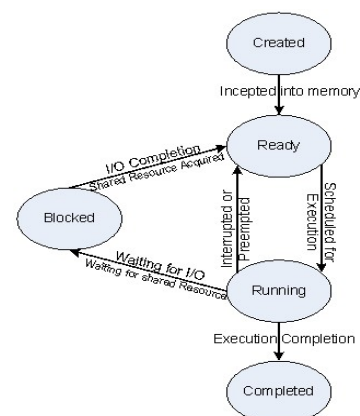


- ❑ A process which inherits all the properties of the CPU. When the process gets its turn, its registers and Program Counter register becomes mapped to the physical registers of the CPU.
- ❑ The memory occupied by the process is segregated into three regions namely; ***Stack memory, Data memory and Code memory***
  - ❖ The “***Stack***” memory holds all temporary data such as variables local to the process.
  - ❖ The “***Data***” memory holds all global data for the process.
  - ❖ The “***Code***” memory contains the program code (instructions) corresponding to the process.

## ***Process States & State Transition***

### ❑ ***Process States & State Transition***

- ❑ 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.

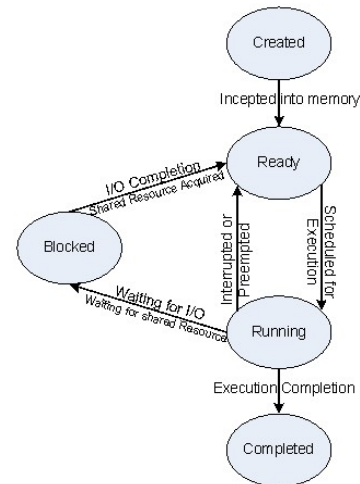


❑ The transition of a process from one state to another is called “**State transition**”.

❑ The process state and state transition representation are shown in the following fig.

❑ **Created State:** The state at which a process is being created is referred “**Created State**”.

❖ The OS recognizes a process in the “Created State” but no resources are allocated to the process



❑ **Ready State:**

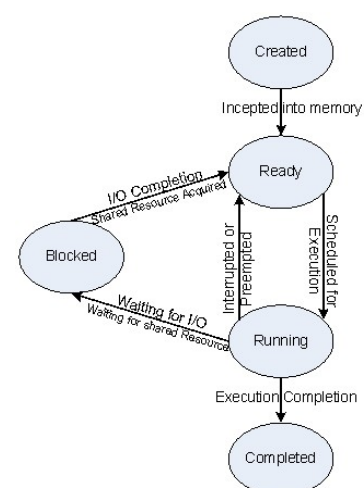
❑ The state, where a process is incepted into the memory and awaiting the processor time for execution, is known as “**Ready State**”.

❑ At this stage, the process is placed in the “**Ready list**” queue maintained by the OS.

❑ **Running State:**

❑ The state where in the source code instructions corresponding to the process is being executed is called “**Running State**”.

❑ Running state is the state at which the process execution happens.



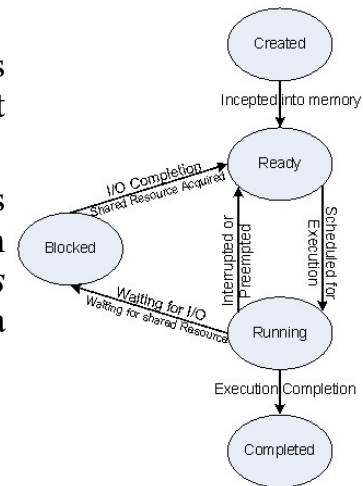
### ❑ **Blocked State/ Wait State:**

❑ Refers to a state where a running process is temporarily suspended from execution and does not have immediate access to resources.

❑ The blocked state might have invoked by various conditions like the process enters a wait state for an event to occur (***E.g. Waiting for user inputs such as keyboard input***) or waiting for getting access to a shared resource like semaphore, mutex etc.

### ❑ **Completed State:**

❑ A state where the process completes its execution



## ***THREADS***

❑ A **Thread** is the primitive that can execute code. A thread is a single sequential flow of control within a process.

❖ A thread is also known “***as lightweight process***”.

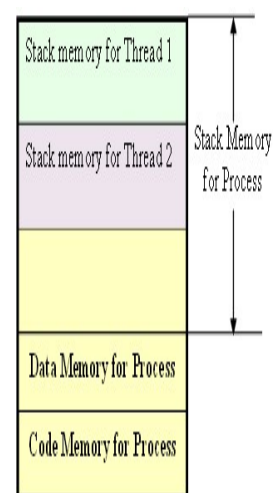
❑ A process can have many threads of execution.

❑ Different threads, which are part of a process, share the same address space;

❖ meaning they ***share the data memory, code memory and heap memory area.***

❑ Threads maintain their own thread status (***CPU register values***), ***Program Counter (PC)*** and ***stack***.

❑ The memory model for a process and its associated threads are given in the following figure.

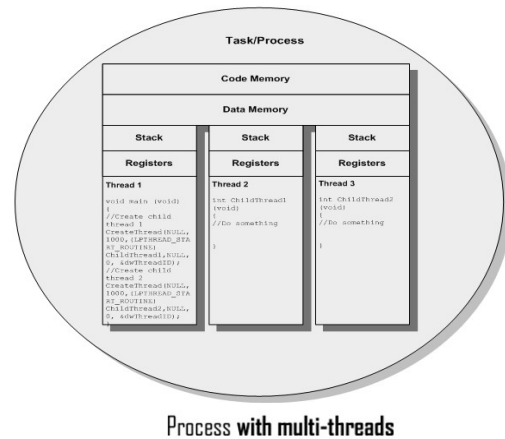


**Memory organisation of a Process and its associated Threads**

## Multithreading:

### ❑ *The Concept of Multithreading:*

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



## ADVANTAGES OF MULTITHREADING

- ❑ Use of multiple threads to execute a process brings the following advantage:
- ❑ **Better memory utilization:** Multiple threads of the same process **share** the **address space** for **data memory**. This also reduces the complexity of inter thread communication 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 utilized 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 utilization:** The CPU is engaged all time.

## ***Thread Standards***

---

- ❑ ***Thread Standards:*** deal with the different standards available for thread creation and management. These standards are utilized by the Operating Systems for thread creation and thread management. It is a set of thread class libraries.
- ❑ The commonly available thread class libraries are –
- ❑ ***POSIX Threads:*** POSIX stands for Portable Operating System Interface.
  - ❖ The ***POSIX.4*** standard deals with the Real Time extensions and ***POSIX.4a*** standard deals with thread extensions.
  - ❖ The ***POSIX*** standard library for thread creation and management is “***Pthreads***”.
  - ❖ “***Pthreads***” library defines the set of POSIX thread creation and management functions in “***C language***”.

## **Example 1**

---

- ❑ Write a multithreaded application to print “Hello I’m in main thread” from the main thread and “Hello I’m in new thread” 5 times each, using the `pthread_create()` and `pthread_join()` POSIX primitives.

```

//Assumes the application is running on an OS where POSIX library is
//available
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
//*****
//New thread function for printing "Hello I'm in new thread"
void *new_thread( void *thread_args )
{
    int i, j;
    for( j= 0; j < 5; j++ )
    {
        printf("Hello I'm in new thread\n" );
        //Wait for some time. Do nothing
        //The following line of code can be replaced with
        //OS supported delay function like sleep(), delay () etc...
        for( i= 0; i < 10000; i++ );
    }
    return NULL;
}
//*****
//Start of main thread
int main( void )
{
    int i, j;
    pthread_t tcb;
    //Create the new thread for executing new_thread function
    if (pthread_create( &tcb, NULL, new_thread, NULL ))
    {
        //New thread creation failed
        printf("Error in creating new thread\n" );
        return -1;
    }
    for( j= 0; j < 5; j++ )
    {
        printf("Hello I'm in main thread\n" );
        //Wait for some time. Do nothing
        //The following line of code can be replaced with
        //OS supported delay function like sleep(), delay etc...
        for( i= 0; i < 10000; i++ );
    }
    if (pthread_join(tcb, NULL ))
    {
        //Thread join failed
        printf("Error in Thread join\n" );
        return -1;
    }
    return 1;
}

```

### ❑ **Win32 Threads:**

- ❖ Win32 threads are the threads supported by various flavors of Windows Operating Systems.
- ❖ The Win32 Application Programming Interface (Win32 API) libraries provide the standard set of Win32 thread creation and management functions.
- ❖ Win32 threads are created with the API.

### ❑ **Java Threads:**

- ❖ Java threads are the threads supported by Java programming Language.
- ❖ The java thread class "**Thread**" is defined in the package "**java.lang**".
- ❖ This package needs to be imported for using the thread creation functions supported by the Java thread class.
- ❖ There are two ways of creating threads in Java: Either by extending the base "**Thread**" class or "**by implementing an interface**".

### ***User Level Thread & Kernel Level/ System Level Thread:***

---

- ❑ Threads falls into one of the following types:
  - ❑ ***User Level Thread:***
    - ❑ User level threads do not have ***kernel/ Operating System support*** and they exist only in the ***running process***.
      - ❖ A process may have ***multiple user level threads***; but the ***OS treats*** it as ***single thread*** and will not switch the execution among the different threads of it.
      - ❖ It is the responsibility of the ***process*** to ***schedule each thread*** as and when required.
  - ❑ ***Kernel Level/ System Level Thread:***
    - ❑ Kernel level threads are individual units of execution, which the OS treats as separate threads

### **Binding user level threads with kernel/system level threads**

---

- ❑ There are many ways for binding user level threads with kernel/system level threads; which are explained below:
  - ❖ ***Many-to-One Model:*** Many user level threads are mapped to a single kernel thread. Eg: Solaris Green threads and GNU Portable Threads
  - ❖ ***One-to-One Model:*** Each user level thread is bonded to a kernel/system level thread. Windows XP/NT/2000 and Linux threads
  - ❖ ***Many-to-Many Model:*** In this model many user level threads are allowed to be mapped to many kernel threads. Windows NT/2000 with ThreadFiberpackage

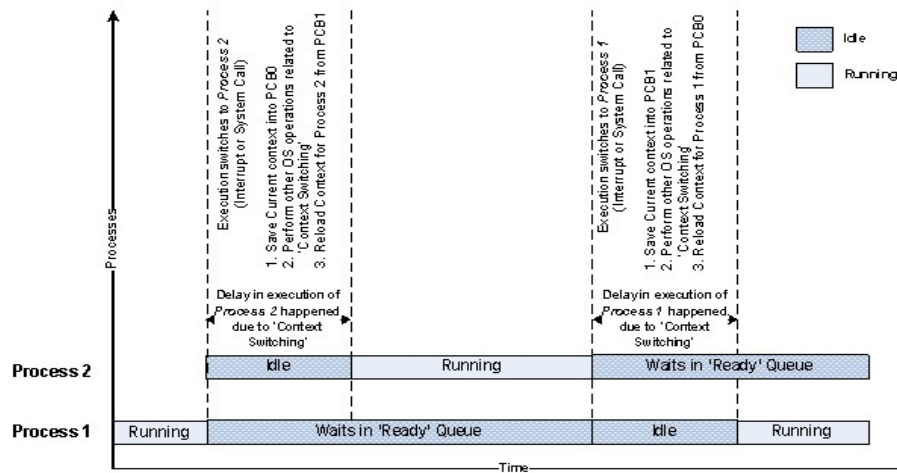
## Threads Versus Process

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

## MULTIPROCESSING AND MULTITASKING

- ❑ The ability to “*Execute multiple processes simultaneously*” is referred as **multiprocessing**.
  - ❖ Systems which are capable of performing multiprocessing are known as **multiprocessor systems**.
  - ❖ Multiprocessor systems possess multiple CPUs and can execute multiple processes simultaneously.
- ❑ The ability of the **Operating System** to have *multiple programs in memory*, which are ready for execution, is referred as **multiprogramming**.
- ❑ The *ability of operating system* to hold *multiple processes in memory* and *switch* the processor (CPU) from executing *one process to other* is called **multitasking**

- ❑ Multitasking involves “*Context switching*”(see the following Figure), “*Context saving*” and “*Context retrieval*”.



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

## ***Types of Multitasking:***

---

- ❑ ***Types of Multitasking:*** Depending on how the task/ process execution switching act is implemented multitasking can be classified into
  - ❖ **Co-operative Multitasking,**
  - ❖ **Pre-emptive Multitasking,**
  - ❖ **Non-Pre-emptive Multitasking**
- ❑ ***Co-operative Multitasking:*** In this method, any task/ process can avail the CPU as much time as it wants.
- ❑ ***Preemptive Multitasking:*** Preemptive multitasking ensures that every task/ process gets a chance to execute.
- ❑ ***Non-preemptive Multitasking:*** The process/ task, which is currently given the CPU time, is allowed to execute until it terminates (enters the “**Completed**” state) or enters the “**Blocked/ Wait**” state, waiting for an I/O.

## **TASK SCHEDULING**

---

- ❑ Determining which task/process is to be executed at a given point of time is known as task/process scheduling.
- ❑ Scheduling policies form the guidelines for determining which task is to be executed when.
- ❑ The kernel service/application which implements the scheduling algorithm, is known as 'Scheduler'.
- ❑ Process scheduling decision may take place when a process switches its state to:
  - 1) Ready state from Running state
  - 2) Blocked/Wait state from Running state
  - 3) Ready state from Blocked/Wait state
  - 4) Completed state

## Process Scheduling Scenarios

---

- ❑ **Scenario 1:** The process switches its state to the Ready state from the Running state
- ❖ **Preemptive Scheduling**
- ❖ **A process moves to the 'Ready' state from the 'Running' state when preempted.**
- ❖ **CPU allocation is interrupted for higher-priority tasks.**
- ❑ **Scenario 2:** The process switches its state to Blocked/Wait state from Running state
- ❖ **Preemptive or Non-Preemptive Scheduling**
- ❖ **The process relinquishes the CPU when it enters 'Blocked/Wait' or 'Completed' state.**
- ❖ **CPU switching occurs at this stage.**
- ❖ **Can follow either preemptive or non-preemptive scheduling.**

- 
- ❑ **Scenario 3:** The process switches its state to Ready state from Blocked/Wait state
  - ❖ **Priority-Based Preemptive Scheduling**
  - ❖ **A high-priority process in 'Blocked/Wait' state completes I/O.**
  - ❖ **The scheduler picks it for execution as soon as it moves to 'Ready' state.**
  - ❑ **Scenario 4:** The process switches its state to the Completed state
  - ❖ **Preemptive,**
  - ❖ **Non-Preemptive, or**
  - ❖ **Co-Operative Scheduling**

## Factors Influencing Scheduling Criteria/Algorithms

---

❑ 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.

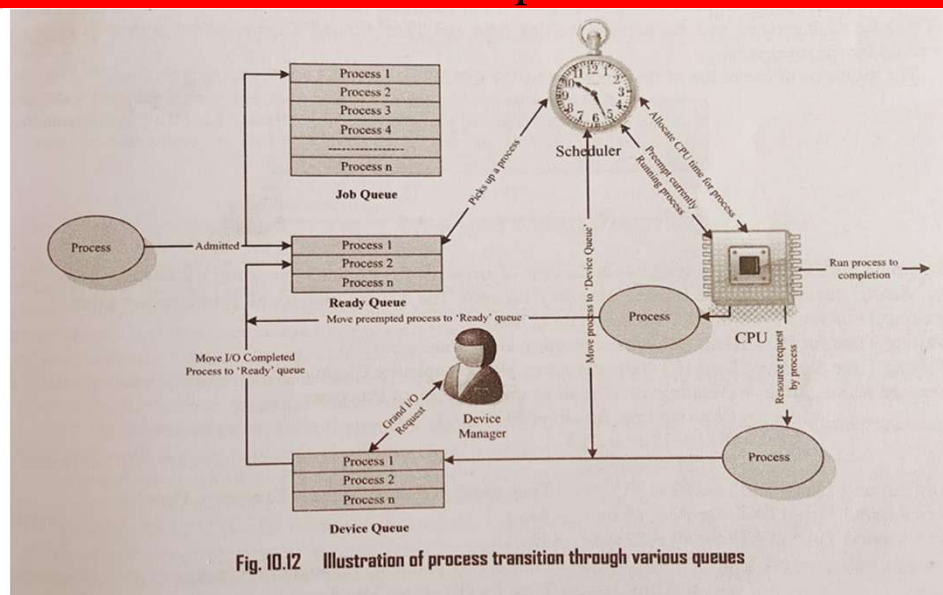
❑ **Response Time:**

- ❖ It is the time elapsed between the submission of a process and the first response. For a good scheduling algorithm, the response time should be as least as possible.

## QUEUES MAINTAINED BY OS IN CPU SCHEDULING

- ❑ The Operating System maintains various queues in connection with the CPU scheduling, and
  - ❖ a process passes through these queues during the course of its admittance to execution completion.
- ❑ The various queues maintained by OS in association with CPU scheduling are:
- ❑ **Job Queue:** Job queue contains all the processes in the system
- ❑ **Ready Queue:**
  - ❖ Contains all the processes, which are ready for execution and waiting for CPU to get their turn for execution.
  - ❖ The Ready queue is empty when there is no process ready for running.
- ❑ **Device Queue:**
  - ❖ Contains the set of processes, which are waiting for an I/O device.
  - ❖ A process migrates through all these queues during its journey from 'Admitted to 'Completed stage

## Illustration of the transition of a process through the various queues.



## Non-preemptive Scheduling

---

- ❑ Non-preemptive scheduling is employed in systems, which implement non-preemptive multitasking model.
- ❑ In this scheduling type, the currently executing task/process is allowed to run until it terminates or enters the 'Wait' state waiting for an I/O or system resource.
- ❑ The various types of non-preemptive scheduling adopted in task/process scheduling are listed below:
  1. First-Come-First-Served (FCFS)/ FIFO Scheduling
  2. Last-Come-First Served (LCFS)/LIFO Scheduling
  3. Shortest Job First (SJF) Scheduling
  4. Priority Based Scheduling:

## First-Come-First-Served (FCFS)/ FIFO Scheduling

---

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

## Example 1

---

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

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

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

❑ Assuming the CPU is readily available at the time of arrival of P1,

❑ P1 starts executing without any waiting in the 'Ready' queue. Hence the waiting time for P1 is zero.

❑ Thus, the waiting time for all processes are given as

---

❑ Waiting Time for P1 = 0

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

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

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

$$= (0 + 10 + 15) / 3 = 8.33 \text{ milliseconds.}$$

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

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

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

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

---

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

❑ **Average Execution Time=(Execution time for all processes)/No. of processes**  
 $= 10 + 5 + 7 / 3 = 7.33$  milliseconds.

Average Turn Around Time (TAT) =  $8.33 + 7.33 = \mathbf{15.66}$  milliseconds.

❑ **NOTE:**

❑ **Average Turn Around Time (TAT) = TAT of all processes / No. of Processes =**  
 $= (10+15+22)/3 = \mathbf{15.66}$  milliseconds.

---

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

❑ The Last-Come-First Served (LCFS) scheduling algorithm also allocates CPU time to the processes based on the order in which they are entered in the 'Ready' queue.

❑ The last entered process is serviced first.

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

## Example 1:

---

- ❑ Three processes with process ID's P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds respectively enters the ready queue together in the order P1, P2, P3. Now a new process P4 with estimated completion time 6ms enters the 'Ready' queue after 5ms of scheduling P1. Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time.
- ❑ **Solution:** Initially there is only P1 available in the Ready queue and the scheduling sequence will be P1, P3, P2.
- ❑ P4 enters the queue during the execution of P1 and becomes the last process entered the 'Ready' queue.
- ❑ Now the order of execution changes to P1, P4, P3, and P2 as given below:

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

- 
- ❑ The waiting time for all the processes is given as:
  - ❑ Waiting Time for P1 = 0ms (P1 starts executing first)
  - ❑ Waiting Time for P4 = 5ms (P4 starts executing after completing P1. But P4 arrived after 5ms of execution of P1. Hence it's waiting time = Execution start time - Arrival Time = 5)
  - ❑ Waiting Time for P3 = 16ms (P3 starts executing after completing P1 and P4)
  - ❑ Waiting Time for P2 = 23ms (P2 starts executing after completing P1, P4 and P3)
  - ❑ Average waiting time = (Waiting time for all processes) / No. of Processes =  $0+5+16+23 / 4 = \mathbf{11 \text{ milliseconds}}$ .

- 
- ❑ Turn Around Time (TAT) = Time spent in Ready Queue (Waiting time) + Execution Time
  - ❑ Turn Around Time (TAT) for P1 = 10ms (0 + 10)
  - ❑ Turn Around Time (TAT) for P4 = 11ms ((10-5) + 6)
  - ❑ Turn Around Time (TAT) for P3 = 23ms (16 + 7)
  - ❑ Turn Around Time (TAT) for P2 = 28ms (23 + 5)
  - ❑ Average Turn Around Time (TAT) = TAT of all processes / No. of Processes =  $(10+11+23+28) / 4 = \mathbf{18 \text{ milliseconds}}$ .

## Shortest Job First (SJF) Scheduling

---

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

## Example 1:

---

- ❑ Three processes with process IDs P1, P2, P3 with estimated completion time 12, 6, 8 milliseconds respectively enters the ready queue together. Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time (Assuming there is no I/O waiting for the processes) in SJF algorithm.
- ❑ **Solution:** The scheduler sorts the 'Ready' queue based on the shortest estimated completion time and schedules the process with the least estimated completion time first and the next least one as second, and so on.
- ❑ The order in which the processes are scheduled for execution is represented as

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

- 
- ❑ The waiting time for all processes are given as:
  - ❑ Waiting Time for P2 = 0ms (P2 starts executing first)
  - ❑ Waiting Time for P3 = 6ms (P3 starts executing after completing P2)
  - ❑ Waiting Time for P1 = 14ms (P1 starts executing after completing P2 and P3)
  - ❑ Average waiting time = (Waiting time for all processes) / No. of Processes =  

$$=(0+6+14) / 3 = \mathbf{6.66 \text{ milliseconds}}$$

- 
- ❑ Turn Around Time (TAT) = Time spent in Ready Queue (Waiting time) + Execution Time
  - ❑ Turn Around Time (TAT) for P2 = 6ms (0 + 6)
  - ❑ Turn Around Time (TAT) for P3 = 14ms (6 + 8)
  - ❑ Turn Around Time (TAT) for P1 = 24ms (14 + 10)
  - ❑ Average Turn Around Time (TAT) = TAT of all processes / No. of Processes = 
$$= (10+14+24) / 3 = \mathbf{16 \text{ milliseconds.}}$$

## Example 2:

---

- ❑ Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time for the above example if a new process P4 with estimated completion time 2ms enters the 'Ready' queue after 2ms of execution of P2. Assume all the processes contain only CPU operation and no I/O operations are involved.
- ❑ **Solution:** At the beginning, there are only three processes (P1, P2 and P3) available in the 'Ready' queue and the SJF scheduler picks up the process with the least execution completion time (P2) for scheduling.
- ❑ The execution sequence is P2, P3, P1.
- ❑ Now process P4 with estimated execution completion time 2ms enters the 'Ready' queue after 2ms of start of execution of P2. After 6ms of scheduling, P2 terminates and now the scheduler again sorts the 'Ready' queue for process with least execution completion time.
- ❑ Since the execution completion time for P4 (2ms) is less than that of P3 (8ms), which was supposed to be run after the completion of P2 as per the 'Ready' queue available at the beginning of execution scheduling, P4 is picked up for executing.
- ❑ Due to the arrival of the process P4 with execution time 2ms, the 'Ready' queue is re-sorted in the order P2, P4, P3, P1. At the beginning it was P2, P3, P1. The execution sequence now changes as per the following diagram

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

- ❑ The waiting time for all the processes are given as:
  - ❑ Waiting time for P2 = 0ms (P2 starts executing first)
  - ❑ Waiting time for P4 = 4ms (P4 starts executing after completing P2. But P4 arrived after 2ms of execution of P2.
  - ❑ Hence it's waiting time = Execution start time - Arrival Time = 4)
  - ❑ Waiting time for P3 = 8ms (P3 starts executing after completing P2 and P4)
  - ❑ Waiting time for P1 = 16ms (P1 starts executing after completing P2, P4 and P3)
- Average waiting time = (Waiting time for all processes) / No. of Processes**  
 $= (0+4+8+16) / 4 = 7 \text{ milliseconds}$

- ❑ Turn Around Time (TAT) = Time spent in Ready Queue (Waiting time) + Execution
- ❑ Time Turn Around Time (TAT) for P2 = 6ms (0 + 6)
- ❑ Turn Around Time (TAT) for P4 = 6ms ((6-2) + 2)
- ❑ Turn Around Time (TAT) for P3 = 16ms (8 + 8)
- ❑ Turn Around Time (TAT) for P1 = 28ms (16 + 12)
- ❑ Average Turn Around Time (TAT) = TAT of all processes / No. of Processes  
 $= (6+6+16+28) / 4 = 19 \text{ milliseconds.}$

## 4. Priority Based Scheduling:

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

### Example 1:

- ❑ Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds and priorities 0, 3, 2 (0-highest priority, 3-lowest priority) respectively enters the ready queue together. Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time (Assuming there is no I/O waiting for the processes) in priority based scheduling algorithm.
- ❑ **Solution:** The scheduler sorts the 'Ready' queue based on the priority and schedules the process with the highest priority (P1 with priority number 0) first and the next high priority process (P3 with priority number 2) as second, and so on.
- ❑ The order in which the processes are scheduled for execution is represented as

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

- 
- ❑ The waiting time for all the processes are given as:
  - ❑ Waiting time for P1 = 0ms (P1 starts executing first)
  - ❑ Waiting time for P3 = 10ms (P3 starts executing after completing P1)
  - ❑ Waiting time for P2 = 17ms (P2 starts executing after completing P1 and P3)
  - ❑ Average waiting time = (Waiting time for all processes) / No. of Processes  
=  $(0+10+17)/3 = \mathbf{9 \text{ milliseconds}}$

- 
- ❑ Turn Around Time (TAT) = Time spent in Ready Queue (Waiting time) + Execution Time
  - ❑ Turn Around Time (TAT) for P1 = 10ms (0+10)
  - ❑ Turn Around Time (TAT) for P3 = 17ms (10+7)
  - ❑ Turn Around Time (TAT) for P2 = 22ms (17+5)
  - ❑ Average Turn Around Time (TAT) = TAT of all processes / No. of Processes  
=  $(10+17+22)/3 = \mathbf{16.33 \text{ milliseconds}}$

## Example 2:

---

- ❑ Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time a Turn Around Time for the above example if a new process P4 with estimated completion time 6ms priority 1 enters the 'Ready' queue after 5ms of execution of P1. Assume all the processes contain only CP operation and no I/O operations are involved.
- ❑ **Solution:** At the beginning, there are only three processes (P1, P2 and P3) available in the 'Ready' queue and the scheduler picks up the process with the highest priority (P1) for scheduling.
- ❑ The execution sequence is P1, P3, P2.
- ❑ Now process P4 with estimate execution completion time 6ms and priority 1 enters the 'Ready' queue after 5ms of execution of P1.
- ❑ After 10ms of scheduling, 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 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 = 0ms (P1 starts executing first)

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

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

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

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

$$= (0+5+16+23) / 4 = \mathbf{11 \text{ milliseconds}}$$

---

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

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

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

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

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

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

$$= (10+11+23+28) / 4 = \mathbf{18 \text{ milliseconds}}$$

## Preemptive Scheduling

---

- ❑ In preemptive scheduling, every task in the 'Ready' queue gets a chance to execute.
- ❑ When and how often each process gets a chance to execute is dependent on the type of preemptive scheduling algorithm used for scheduling the processes.
- ❑ In this kind of scheduling, the scheduler can preempt (stop temporarily) the currently executing task/process and select another task from the 'Ready' queue for execution.
- ❑ A task which is preempted by the scheduler is moved to the 'Ready queue.
- ❑ The act of moving a 'Running process/task into the "Ready' queue by the scheduler, without the processes requesting for it is known as 'Preemption'.

## Types of preemptive scheduling

---

1. Preemptive SJF Scheduling/Shortest Remaining Time (SRT)
2. Round Robin (RR) Scheduling:
3. Priority Based Scheduling

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

---

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

### Example 1:

---

- ❑ Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds respectively enters the ready queue together. A new process P4 with estimated completion time 2ms enters the 'Ready' queue after 2ms. Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time in preemptive SJF/SRT based scheduling algorithm
- ❑ **Solution:** At the beginning, there are only three processes (P1, P2 and P3) available in the 'Ready' queue and the SRT scheduler picks up the process with the shortest remaining time for execution completion for scheduling.
- ❑ Now process P4 with estimated execution completion time 2ms enters the 'Ready' queue after 2ms of start of execution of P2.
- ❑ Since the SRT algorithm is preemptive, the remaining time for completion of process P2 is checked with the remaining time for completion of process P4. The remaining time for completion of P2 is 3ms which is greater than that of the remaining time for completion of the newly entered process P4 (2ms). Hence P2 is preempted and P4 is scheduled for execution.

- ❑ P4 continues its execution to finish since there is no new process entered in the 'Ready' queue during its execution.
- ❑ After 2ms of scheduling P4 terminates and now the scheduler again sorts the 'Ready' queue based on the remaining time for completion of the processes present in the 'Ready' queue. The execution sequence now changes as per the following diagram



- ❑ The waiting time for all the processes are given as:
- ❑ Waiting time for P2 =  $0\text{ms} + (4-2)\text{ms} = 2\text{ms}$  (P2 starts executing first and is interrupted by P4 and has to wait till the completion of P4 and has to wait till the completion of P4)
- ❑ Waiting time for P4 =  $0\text{ms}$  (P4 starts executing after preempting P2)
- ❑ Waiting time for P3 =  $7\text{ms}$  (P3 starts executing after completing P4 and P2)
- ❑ Waiting time for P1 =  $14\text{ms}$  (P1 starts executing after completing P2, P4 and P3)
- ❑ **Average waiting time = (Waiting time for all processes) / No. of Processes**  

$$= (0+2+7+14) / 4 = 5.75 \text{ milliseconds}$$

- ❑ Turn Around Time (TAT) = Time spent in Ready Queue (Waiting time) + Execution Time
- ❑ Turn Around Time (TAT) for P2 = 7ms (2+5)
- ❑ Turn Around Time (TAT) for P4 = 2ms (0+2)
- ❑ Turn Around Time (TAT) for P3 = 14ms (7+7)
- ❑ Turn Around Time (TAT) for P2 = 24ms (14+10)
- ❑ Average Turn Around Time (TAT) = TAT of all processes / No. of Processes  

$$= (7+2+14+24) / 4 = \mathbf{11.75 \text{ milliseconds}}$$

## 2. Round Robin (RR) Scheduling:

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

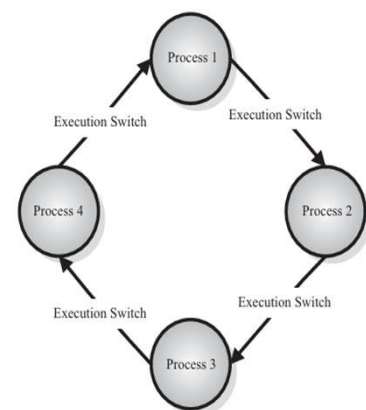
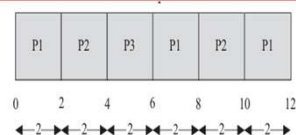


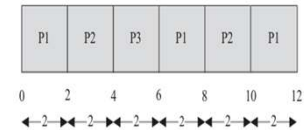
Fig. 10.13 Round Robin Scheduling

- 
- ❑ Example 1: Three processes with process IDs P1, P2, P3 with estimated completion time 6, 4, 2 milliseconds respectively enters the ready queue together in the order P1, P2, P3, Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time in RR algorithm with Time slice = 2 ms.
  - ❑ **Solution:** The scheduler sorts the 'Ready' queue and picks up the first process P1 from the 'Ready' queue and executes it for the time slice 2ms.
  - ❑ 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



- ❑ The waiting time for all the processes are given as:
- ❑ Waiting time for P1 =  $0 + (6 - 2) + (10 - 8) = 6$ 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 =  $(2 - 0) + (8 - 4) = 6$ ms (P2 starts executing after P1 executes for 1 time slice and waits for two time slices get the CPU time)
- ❑ Waiting time for P3 =  $(4 - 0) = 4$ 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 processes) / No. of Processes =  $(6+6+4) / 3 = 5.33$  milliseconds

- ❑ Turn Around Time (TAT) = Time spent in Ready Queue (Waiting time) + Execution Time
- ❑ Turn Around Time (TAT) for P1 = 12ms (6+6)
- ❑ Turn Around Time (TAT) for P2 = 10ms (6+4)
- ❑ Turn Around Time (TAT) for P3 = 6ms (4+2)
- ❑ Average Turn Around Time (TAT) = TAT of all processes / No. of Processes =  $(12+10+6) / 3 = 9.33$  milliseconds



### 3. Priority Based Scheduling

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

## Example 1:

- ❑ Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds and priorities 1, 3, 2 (0-highest priority, 3 lowest priority) respectively enters the ready queue together. A new process P4 with estimated completion time 6 ms and priority 0 enters the 'Ready' queue after 5 ms of start of execution of P1. Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time in preemptive priority based scheduling algorithm.
- ❑ **Solution:** At the beginning, there are only three processes (P1, P2 and P3) available in the 'Ready' queue and the scheduler picks up the process with the highest priority (P1) for scheduling.
- ❑ Now process P4 with estimated execution completion time 6 ms and priority 0 enters the 'Ready' queue after 5 ms of start of execution of P1.
- ❑ Since the scheduling algorithm is preemptive, P1 is preempted by P4 and P4 runs to completion. After 6 ms of scheduling, P4 terminates and now the scheduler again sorts the 'Ready' queue for process with highest priority. Since the priority for P1 (priority 1), which is preempted by P4 is higher than that of P3 (priority 2) and P2 ((priority 3), P1 is again picked up for execution by the scheduler. The execution sequence is as per the following:

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

- ❑ The waiting time for all the processes are given as:
- ❑ Waiting time for P1 = 0ms + (11-5) ms = 6ms (P1 starts executing first and gets preempted by P4 after 5ms and again gets CPU after completion of P4)
- ❑ Waiting time for P4 = 0ms (P4 starts executing immediately after preempting P1)
- ❑ Waiting time for P3 = 16ms (P3 starts executing after completing P1 and P4)
- ❑ Waiting time for P2 = 23ms (P2 starts executing after completing P1, P4 and P3)
- ❑ Average waiting time = (Waiting time for all processes) / No. of Processes  
= (6+0+16+23) / 4 = **11.25 milliseconds**

- 
- ❑ Turn Around Time (TAT) = Time spent in Ready Queue (Waiting time) + Execution
  - ❑ Time Turn Around Time (TAT) for P1 = 16ms (6+10)
  - ❑ Turn Around Time (TAT) for P4 = 6ms (0+6)
  - ❑ Turn Around Time (TAT) for P3 = 23ms (16+7)
  - ❑ Turn Around Time (TAT) for P2 = 28ms (23+5)
  - ❑ Average Turn Around Time (TAT) = TAT of all processes / No. of Processes =  $(16+6+23+28)/4 = 18.25$  milliseconds

## Task Communication System

---

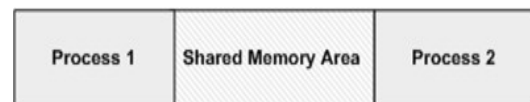
- ❑ Based on the degree of interaction, the processes/ tasks running on an OS are classified as –
- ❑ **Co-operating Process**
- ❑ **Competing Process**
  
- ❑ **Co-operating Processes:** In the co-operating interaction model, one process requires the inputs from other processes to complete its execution.
  
- ❑ **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 file, display device, etc.

## co-operating processes

- ❑ The co-operating processes exchanges information and communicate through the following methods:
  - ❑ **Co-operation through sharing:**
    - ❖ Exchange data through some shared resources.
  - ❑ **Co-operation through Communication:**
    - ❖ No data is shared between the processes.
    - ❖ But they communicate for execution synchronization.
- ❑ The mechanism through which tasks/ processes communicate each other is known as **Inter Process/Task Communication (IPC)**.
- ❑ IPC are as follows
  - ❖ **IPC – Shared Memory**
  - ❖ **IPC – Message Passing**
  - ❖ **IPC - Remote Procedure Calls and Sockets**

## IPC SHARED MEMORY

- ❑ **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.
  - ❖ Processes which requires the information can read the same from the shared memory area.
- ❑ Different mechanisms are adopted by different kernels for implementing this, a few among are as follows:
  - ❖ **1)Pipes**
  - ❖ **2)Memory Mapped Objects**



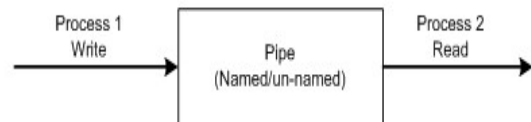
---

❑ **1) Pipes:**“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 **pipe server** and a process which connects to a pipe is known as **pipe client**.

❑ 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.




---

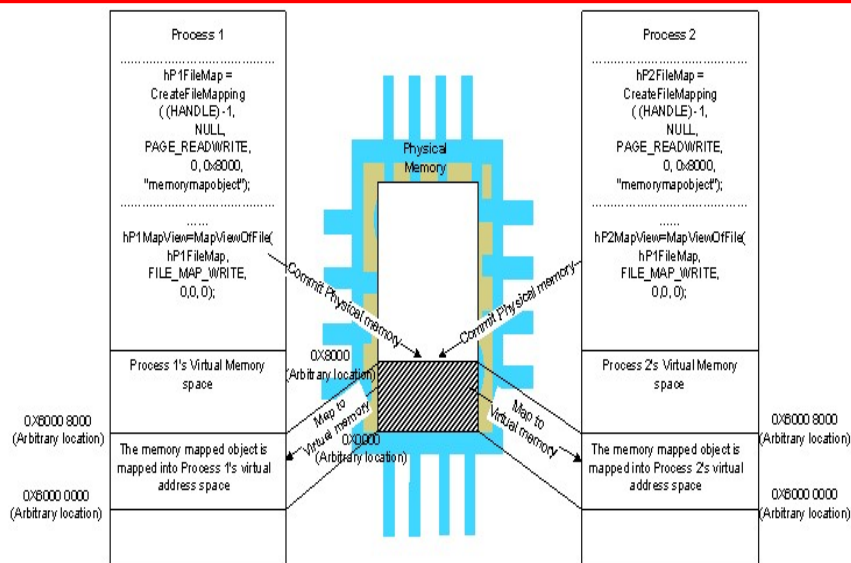
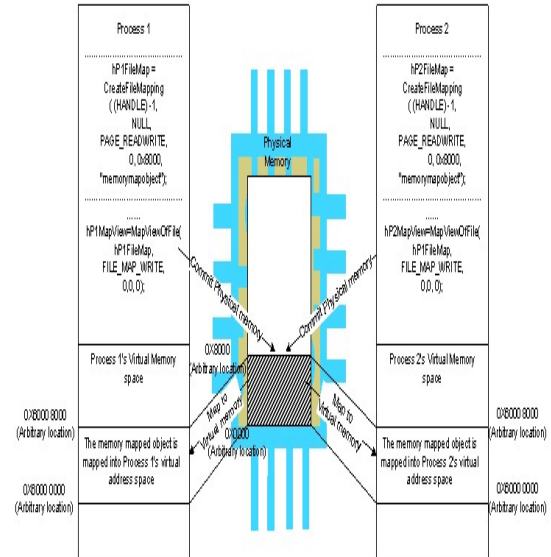
❑ Microsoft® Windows Desktop Operating Systems support two types of “Pipes” for Inter Process Communication. Namely;

❖ **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.

**2. Memory Mapped Objects:**

- ❑ Memory mapped object is a *shared memory technique* adopted by certain Real Time Operating Systems for *allocating a shared block of memory* which can be accessed by multiple process simultaneously.
- ❑ In this approach, 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.



Concept of memory mapped object

## ***IPC Mechanism - Message Passing***

---

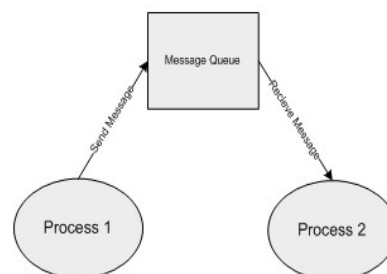
### ***IPC Mechanism - Message Passing***

- ❑ Message passing is a/ an synchronous/ asynchronous information exchange mechanism for Inter Process/ Thread Communication.
- ❑ Message passing is relatively fast and free from the synchronization overheads compared to shared memory.
- ❑ Based on the message passing operation between the processes, message passing is classified into three
  - ❖ 1) **Message Queue**
  - ❖ 2) **Mailbox**
  - ❖ 3) **Signaling**

---

### ❑ **1. Message Queues:**

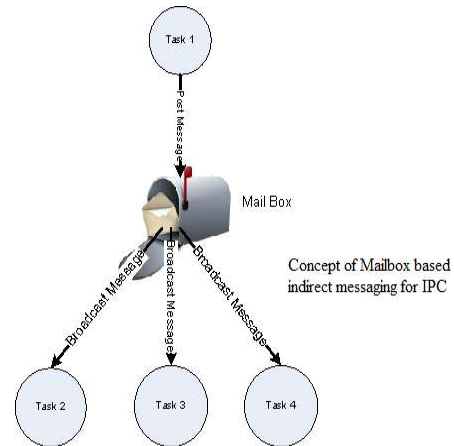
- ❑ Process which wants to talk to another process posts the message to a First-In-First-Out (FIFO) queue called “Message queue”, which stores the messages temporarily in a system defined memory object, to pass it to the desired process.



Concept of message queue based indirect messaging for IPC

## ❑ 2. Mailbox:

- ❑ Mailbox is a special implementation of message queue.
- ❑ Usually used for one way communication, only a single message is exchanged through mailbox whereas “message queue” can be used for exchanging multiple messages.
- ❑ One task/process creates the mailbox and other tasks/process can subscribe to this mailbox for getting message notification.

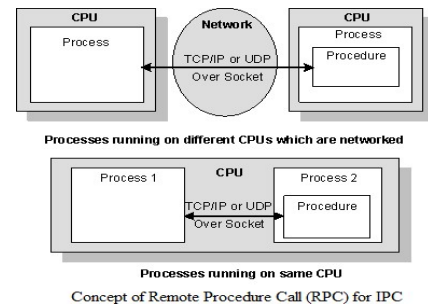


## ❑ 3. Signaling:

- ❑ Signals are used for an asynchronous notification mechanism.
- ❑ The signal mainly used for the execution synchronization of tasks process/tasks.
- ❑ Signals do not carry any data and are not queued.
- ❑ The implementation of signals is OS kernel dependent and VxWorks RTOS kernel implements “signals” for inter process communication.

## REMOTE PROCEDURE CALL (RPC)

- ❑ **Interface Definition Language (IDL)** defines the interfaces for RPC.(Remote Procedure Call)
- ❑ Remote Procedure Call or RPC is the inter process Communication (IPC) mechanism used by a process to call a procedure of a another process running on the same CPU or on a different CPU which is interconnected in a network
- ❑ **Microsoft Interface Definition Language(MIDL)** is the IDL implementation from Microsoft for all Microsoft Platforms.
- ❑ The RPC Communication can be either **Synchronous (Blocking)** or **Asynchronous (Non Blocking)**



## Sockets

- ❑ 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:
  - ❖ **Stream Sockets:** are connection oriented and they use TCP to establish a reliable connection.
  - ❖ **Datagram Sockets:** rely on UDP for establishing a connection.

## Task Synchronization

---

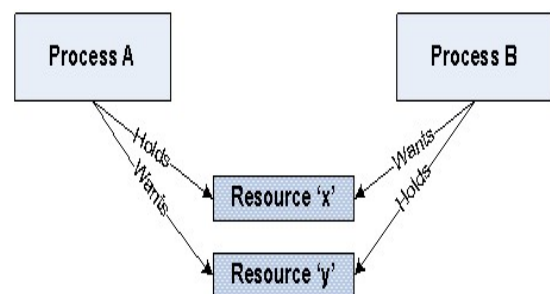
- ❑ There may be situations that; two processes try to access a shared memory area, where one process tries to write to the memory location when the other process is trying to read from the same memory location.
- ❑ This will lead to unexpected results.
- ❑ The solution is, make each process aware of access of a shared resource.
- ❑ The act of making the processes aware of the access of shared resources by each process to avoid conflicts is known as “**Task/ Process Synchronization**”.

- 
- ❑ **Task/ Process Synchronization is essential for –**
    - ❖ Avoiding conflicts in resource access (*racing, deadlock, etc.*) in multitasking environment.
    - ❖ Ensuring proper sequence of operation across processes.
    - ❖ Establish proper communication between processes.

## ***Task Communication/ Synchronization Issues:***

- ❑ ***Task Communication/ Synchronization Issues:***
- ❑ Various synchronization issues may arise in a multitasking environment; if processes are not synchronized properly in shared resource access, such as:
  - ❑ ***Racing: Race Condition***
  - ❑ A race condition is a situation that may occur inside a critical section. This happens when the result of multiple thread execution in critical section differs according to the order in which the threads execute.
  - ❑ Race conditions in critical sections can be avoided if the critical section is treated as an atomic instruction. Also, proper thread synchronization using locks or atomic variables can prevent race conditions.

- ❑ ***Deadlock:*** 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; hence, none of the processes are able to make any progress in their execution.
- ❑ 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.



## ***Conditions Favoring Deadlock.***

---

- ❑ **Mutual Exclusion:** The criteria that only one process can hold a resource at a time. Meaning processes should access shared resources with mutual exclusion.
- ❑ **Hold & Wait:** The condition in which a process holds a shared resource by acquiring the lock controlling the shared access and waiting for additional resources held by other processes.
- ❑ **No Resource 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.
- ❑ **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

## ***Handling Deadlock:***

---

- **Handling Deadlock:** The OS may adopt any of the following techniques to detect and prevent deadlock conditions.
- **Ignore Deadlocks:** Always assume that the system design is deadlock free.
  - This is acceptable for the reason the cost of removing a deadlock is large compared to the chance of happening a deadlock.
  - **UNIX** is an example for an OS following this principle.
  - A life critical system cannot pretend that it is deadlock free for any reason.

---

➤ **Detect and Recover:** This approach suggests the detection of a deadlock situation and recovery from it.

➤ This is similar to the deadlock condition that may arise at a traffic junction. When the vehicles from different directions compete to cross the junction, deadlock (traffic jam) condition is resulted. Once a deadlock (traffic jam) is happened at the junction, the only solution is to back up the vehicles from one direction and allow the vehicles from opposite direction to cross the junction. If the traffic is too high, lots of vehicles may have to be backed up to resolve the traffic jam. This technique is also known as “**back up cars**” technique.

---

❑ Operating Systems keep a resource graph in their memory. The resource graph is updated on each resource request and release

❑ A deadlock condition can be detected by analyzing the resource graph by graph analyzer algorithms.

❑ Once a deadlock condition is detected, the system can terminate a process or preempt the deadlocking cycle.

❑ **Avoid Deadlocks:** Deadlock is avoided by the careful resource allocation techniques by the Operating System. It is similar to the traffic light mechanism at junctions to avoid the traffic jams.

- 
- ❑ **Prevent Deadlocks:** Prevent the deadlock condition by negating one of the four conditions favoring the deadlock situation (*Mutual Exclusion, No Resource Preemption, Circular Wait*).
  - ❑ Ensure that a process does not hold any other resources when it requests a resource. This can be achieved by implementing the following set of rules/guidelines in allocating resources to processes.
    - ❖ A process must request all its required resource and the resources should be allocated before the process begins its execution.
    - ❖ 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 operating system level. This can be achieved by implementing the following set of rules/guidelines in resources allocation and releasing:
    - ❖ Release all the resources currently held by a process if a request made by the process for a new resource is not able to fulfill immediately.
    - ❖ Add the resources which are preempted(released) to a resource list describing the resources which the process requires to complete its execution.
    - ❖ Reschedule the process for execution only when the process gets its old resources and the new resource which is requested by the process.

## MUTUAL EXCLUSION

---

- ❑ The technique used for task synchronization in a multitasking environment is ***Mutual exclusion***.
- ❑ Mutual exclusion blocks a process. Based on the behavior of blocked process, mutual exclusion methods can be classified into two categories:
  - ❖ ***Mutual exclusion through busy waiting/ spin lock***
  - ❖ ***Mutual exclusion through sleep & wakeup.(Semaphore)***
- ❑ **Semaphore:** Semaphore is a sleep and wakeup based mutual exclusion implementation for shared resource access.
- ❑ Semaphore is a system resource; and a 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 in use by it.

## Livelock & Starvation in Process Management

---

- ❑ **Livelock Condition**
- ❑ Similar to deadlock, but the process keeps changing state instead of waiting indefinitely.
  - ❖ The process keeps taking action but makes no progress towards execution completion.
- ❑ **Real-world analogy:**
  - ❖ Two people in a narrow corridor trying to cross each other.
  - ❖ Both keep moving but never manage to pass.

---

### ❑ Starvation Condition

- ❖ Occurs in multitasking environments when a process does not get resources for a long time.
- ❖ As time progresses, the process starves for resources and execution slows down.

### ❑ Causes:

- ❖ Preventive measures for deadlocks.
- ❖ Scheduling policies that favor:
  - High-priority tasks
  - Shortest execution time tasks

---

## The Dining Philosophers' Problem

- ❑ The 'Dining philosophers' problem' is an interesting example for synchronisation issues in resource utilisation.
- ❑ The terms 'dining', 'philosophers', etc. may sound awkward in the operating system context, but it is the best way to explain technical things abstractly using non-technical terms.
- ❑

## ❑ Dining Philosophers Problem

❖ *A Classic Synchronization Issue in Resource Allocation*

### ❑ Problem Definition

- ❖ Five philosophers sit around a table, alternating between eating and brainstorming .
- ❖ Each philosopher needs 2 forks to eat.
- ❖ Only 5 forks available (one between every two philosophers).
- ❖ Must pick up left fork first, then right fork.

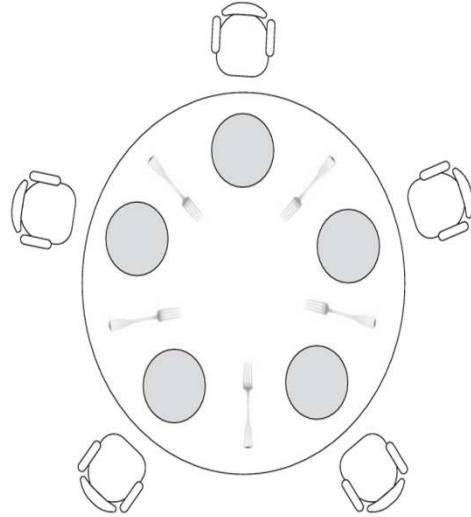


Fig. 10.27 Visualisation of the 'Dining Philosophers problem'

## ❑ Possible Scenarios & Issues

### ❑ Deadlock & Starvation

- ❖ All philosophers pick up left fork simultaneously.
- ❖ Each waits for the philosopher on their right to put down a fork.
- ❖ This forms a circular chain of waiting → **Deadlock**

### ❑ Race Condition

- ❖ One philosopher picks up the left fork.
- ❖ Another philosopher tries to pick up the same fork (which is their right fork).
- ❖ Leads to conflicting resource access → **Race Condition**

### ❑ Livelock & Starvation

- ❖ All philosophers pick up left fork but can't proceed.
- ❖ Each puts down the fork, waits, then tries again simultaneously.
- ❖ No one ever gets two forks → **Livelock** .

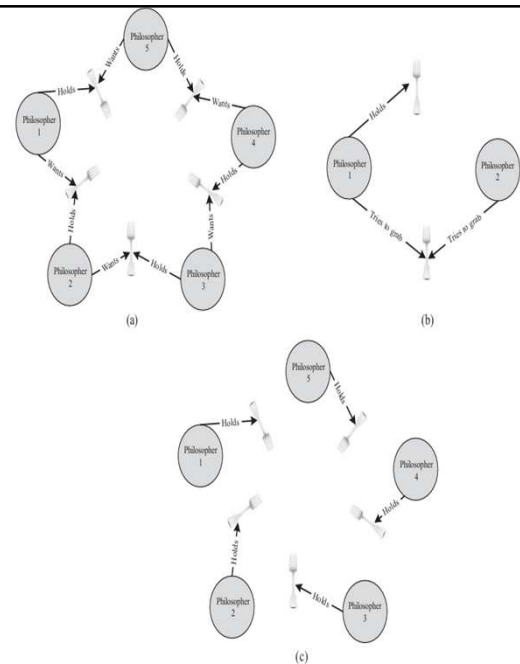


Fig. 10.28 The 'Real Problems' in the 'Dining Philosophers problem' (a) Starvation and Deadlock (b) Racing (c) Livelock and Starvation

---

❑ **Solutions to Avoid Deadlocks & Starvation**

- ❖ **Resource Allocation Techniques**
- ❖ **Semaphore/Mutex Mechanism**

---

❑ **Resource Allocation Techniques** :We need to find out alternative solutions to avoid the deadlock, livelock, racing and starvation condition that may arise due to the concurrent access of forks by philosophers.

❑ This situation can be handled in many ways by allocating the forks in different allocation techniques including

- ❖ Round Robin Allocation
- ❖ FIFO (First-In-First-Out)
- ❖ Priority-Based Scheduling

- 
- ❑ **Semaphore/Mutex Mechanism:** Another solution which gives maximum concurrency that can be thought of is each philosopher acquires a semaphore (mutex) before picking up any fork.
  - ❑ When a philosopher feels hungry he/she checks whether the philosopher sitting to the left and right of him is already using the fork, by checking the state of the associated semaphore.
  - ❑ If the forks are in use by the neighbouring philosophers, the philosopher waits till the forks are available. A philosopher when finished eating puts the forks down and informs the philosophers sitting to his/her left and right, who are hungry (waiting for the forks), by signalling the semaphores associated with the forks.

---

### ❑ **Key Takeaways**

- ❖ The Dining Philosophers Problem models real-world resource competition in operating systems.
- ❖ Key issues: Deadlock, Race Conditions, Livelock, and Starvation.
- ❖ Effective solutions involve smart resource allocation & synchronization mechanisms.

## Task Synchronization Techniques

---

- ❑ 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.
- ❑ The *display device of an embedded system* is a typical example of a shared resource which needs exclusive access by a process.
- ❑ The *Hard disk(secondary storage) of a system* is a typical example for sharing the resource among a limited number of multiple processes.

- 
- ❑ 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.

# SEMAPHORE

---

## ❑ What is a Semaphore?

- ❖ A sleep & wakeup-based mutual exclusion technique.
- ❖ Used for managing access to shared resources.
- ❖ A process acquires a semaphore to signal ownership of a resource.

---

## ❑ Types of Shared Resource Access

### 1. Exclusive Access

- ❖ Only one process can use the resource at a time.
- ❖ Example: Display device in an embedded system

### 2. Concurrent Access

- ❖ Multiple processes can access different parts simultaneously.
- ❖ Example: Hard disk (secondary storage)
- ❖ Various processes can read/write different sectors concurrently.

---

### ❑ **Why Use Semaphores?**

- ❖ Prevents race conditions in shared resource usage.
- ❖ Ensures synchronized access to prevent conflicts.
- ❖ Helps in efficient multitasking & process coordination.

---

### ❑ Based on the implementation, Semaphores can be classified into

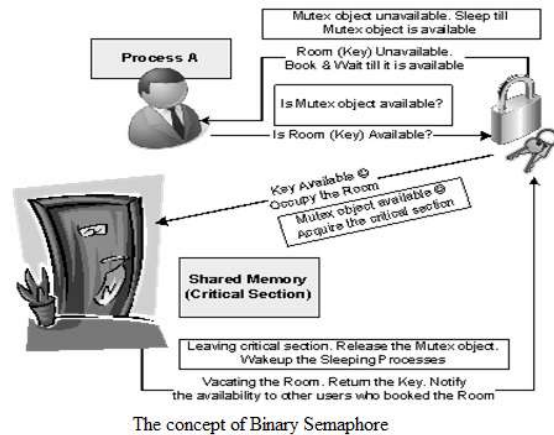
- ❖ Binary Semaphore and
- ❖ Counting Semaphore.

### ❑ **Binary Semaphore:**

- ❑ Implements 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 used by a process.
- ❑ “Only one process/thread” can own the binary semaphore at a time.

## Binary semaphore/ MUTEX

- The state of a “**Binary semaphore**” object is set to signaled when it is not owned by any process/ thread, and set to non-signaled when it is owned by any process/ thread.
- The implementation of binary semaphore is OS kernel dependent.
- Under certain OS kernel it is referred as *Mutex*



## COUNTING SEMAPHORE

- ❑ **Counting Semaphore:** Maintains a count between zero and a maximum value. It limits the usage of resource by a fixed number of processes/ threads.
- ❑ 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**”.
- ❑ The state of the counting semaphore object is set to “**signaled**” when the count of the object is greater than zero.
- ❑ The state of the “**Semaphore object**” is set to non-signaled when the semaphore is acquired by the maximum number of processes/ threads that the semaphore can support (i.e. when the count associated with the “**Semaphore object**” becomes zero).
- ❑ The creation and usage of “**counting semaphore object**” is OS kernel dependent.

