ATME College of Engineering

Embedded Systems-BEC601

Module-3

RTOS and IDE for Embedded System Design

Pradeep Kumar Y
Assistant Professor
Dept.of ECE

ATME College of Engineering

Topics

1. Operating System basics
2. Types of operating systems
3. Task, process and threads *(Only POSIX Threads with an example program)*
4. Thread preemption
5. Preemptive Task scheduling technique
6. Task Communication
7. Task synchronization issues – Racing and Deadlock

ATME College of Engineering

Topics

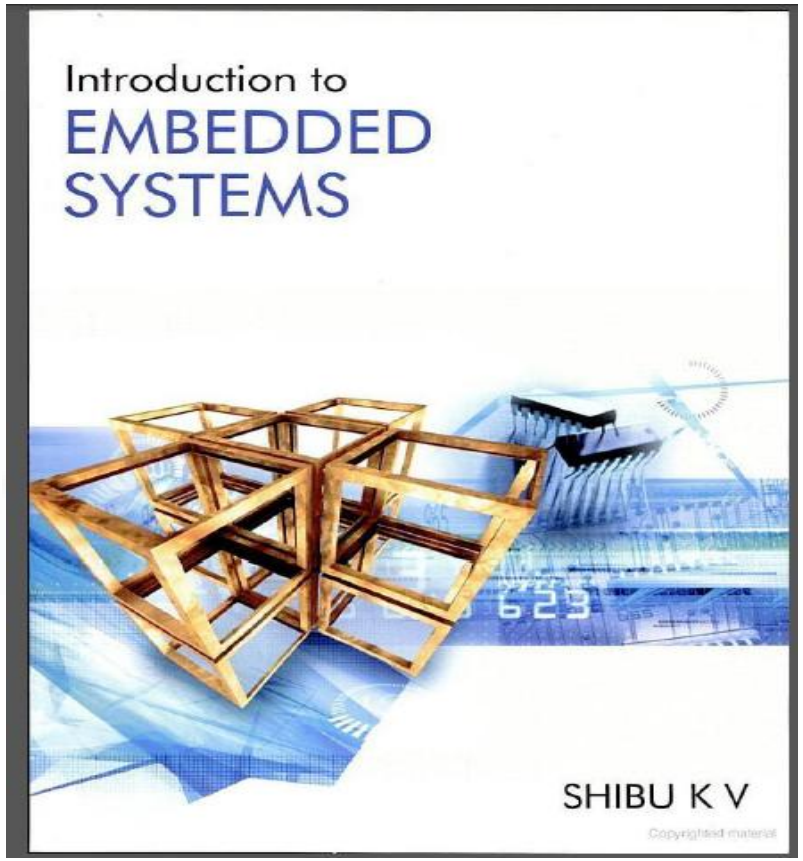
8. Concept of Binary and counting semaphores (Mutex example without any program)
9. How to choose an RTOS
10. Integration and testing of Embedded hardware and firmware
11. Embedded system Development Environment – Block diagram *(excluding Keil)*
12. Disassembler/decompiler, simulator, emulator and debugging techniques

ATME College of Engineering

Text Book

Shibu K V, "**Introduction to Embedded Systems**"
Tata McGraw Hill Education Private Limited, 2nd Edition

- Ch-10 (Sections 10.1, 10.2, 10.3, 10.5.2 , 10.7, 10.8.1.1, 10.8.1.2, 10.8.2.2, 10.10 only),
- Ch-12,
- Ch-13 (a block diagram before 13.1, 13.3, 13.4, 13.5, 13.6 only)



Introduction

Super Loop Based task Execution Model

- Response time for a task is not time-critical
- Electronic toy and video gaming

Application demand the time critical task response

- Flight control system,
- Airbag control and anti Locking brake system for vehicles
- Nuclear monitoring devices

Introduction

1. *Assign priority to task and execute high priority task whenever it is ready.*
2. *Dynamically change the priorities of tasks if required on a need basis.*
3. *Schedule the exception of task based on priorities.*
4. *Switching the execution of task when task waiting for external event.*

***Operating System (OS)** based firmware execution can address the these needs*

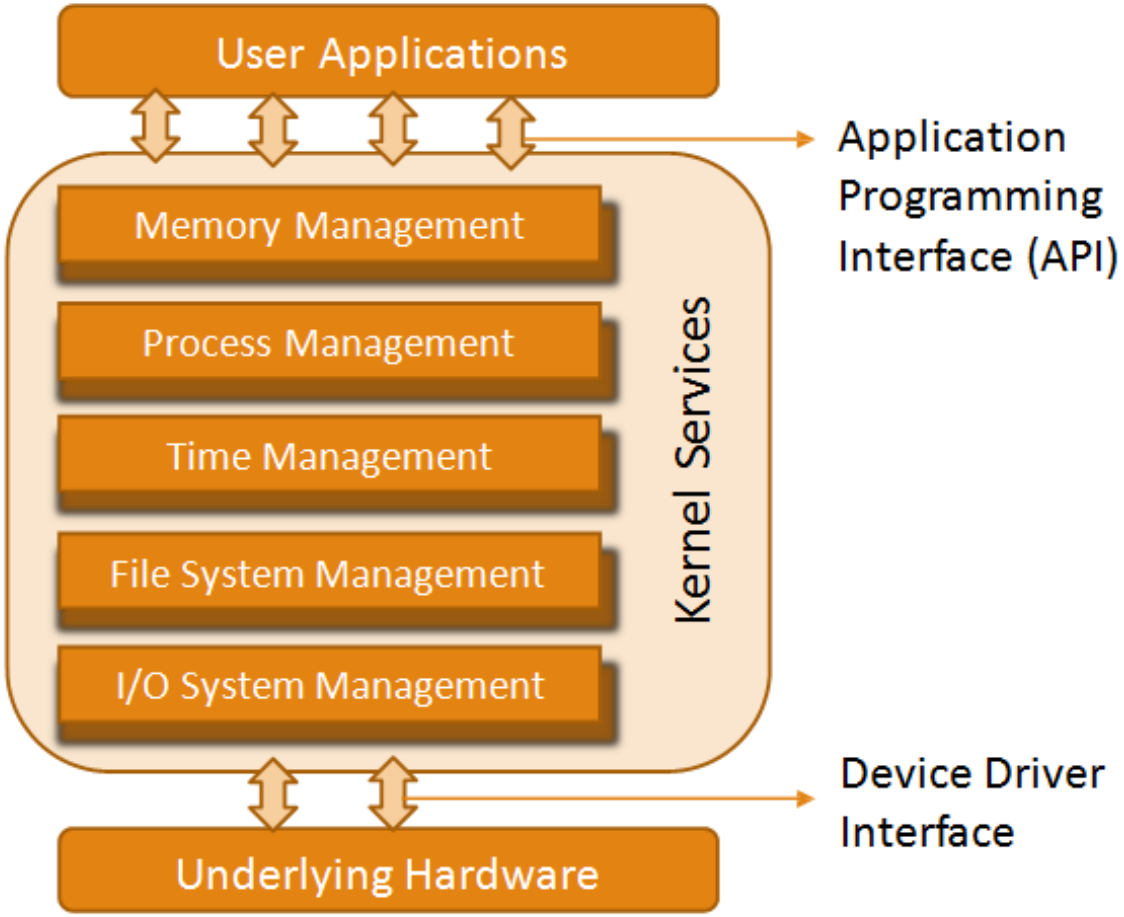
Operating System Basics

- The operating system acts as a bridge between the user applications/tasks and the underlying system resources through a set of system functionalities and services.
- The 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 are:
  - Make the system convenient to use
  - Organize and manage the system resources efficiently and correctly

Operating System Basics

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.



The Operating System Architecture

Operating System Basics- The Kernel

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

1. Process Management
2. Primary Memory Management
3. File System Management
4. I/O System (Device) Management
5. Secondary Storage Management
6. Protection Systems
7. Interrupt Handler

Operating System Basics- The Kernel

1. Process Management

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 synchronization
- Process termination/deletion, etc.

Operating System Basics- The Kernel

2. Primary Memory Management

- Primary memory refers to the volatile memory **(RAM)**

  Where processes are loaded and variables and shared data associated with each process are stored.

- The Memory Management Unit (MMU) of the kernel is responsible for:
  - Keeping track of which part of the memory area is currently used by which process.
  - Allocating and De-allocating memory space on a need basis *(Dynamic memory allocation).*

Operating System Basics- The Kernel

3. File System Management

- File is a collection of related information.
- A file could be a program (source code or executable), text files, image files, word documents, audio/video files, etc.

The file system management service of Kernel is responsible for:

- 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.

Operating System Basics- The Kernel

3. File System Management

- The various file system management operations are OS dependent.
- For example, the kernel of Microsoft DOS OS supports a specific set of file system management operations and they are not the same as the file system operations supported by UNIX Kernel.

Operating System Basics- The Kernel

4. I/O System (Device) Management

- Kernel is responsible for routing the I/O requests coming from different user applications to the appropriate I/O devices of the system.
- 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.
- The kernel maintains a list of all the I/O devices of the system.
- The service Device Manager of the kernel is responsible for handling all I/O device related operations.

Operating System Basics- The Kernel

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

1. Process Management
2. Primary Memory Management
3. File System Management
4. I/O System (Device) Management
5. Secondary Storage Management
6. Protection Systems
7. Interrupt Handler

Operating System Basics- The Kernel

4. I/O System (Device) Management

- The kernel talks to the I/O device through a set of low-level systems calls, which are implemented in a service called **device drivers**.
- Device Manager is responsible for
  - Loading and unloading of device drivers
  - Exchanging information and the system specific control signals to and from the device

Operating System Basics- The Kernel

5. Secondary Storage Management

- The secondary storage management deals with managing the secondary storage memory devices, if any, connected to the system.
- Secondary memory is used as backup medium for programs and data since the main memory is volatile.
- In most of the systems, the secondary storage is kept in disks (Hard Disk).
- The secondary storage management service of kernel deals with
  - Disk storage allocation
  - Disk scheduling (Time interval at which the disk is activated to backup data)
  - Free Disk space management

Operating System Basics- The Kernel

6. Protection Systems

- Most of the modern operating systems are designed in such a way to support multiple users with different levels of access permissions.

  E.g. 'Administrator', 'Standard', 'Restricted' permissions in Windows XP.

- Protection deals with implementing the security policies to restrict the access to both user and system resources by different applications or processes or users.
- In multiuser supported operating systems, one user may not be allowed to view or modify the whole or portions of another user's data or profile details.
- In addition, some application may not be granted with permission to make use of some of the system resources.
- This kind of protection is provided by the protection services running within the kernel.

Operating System Basics- The Kernel

6. Interrupt Handler

Kernel provides handler mechanism for all external/internal interrupts generated by the system

Operating System Basics- Kernel Space and User Space

- The applications/services are classified into two categories:
- **User applications**
- **Kernel applications**
- **Kernel Space** is the memory space at which the kernel code is located
  - Kernel applications/services are kept in this contiguous area of primary (working) memory.
  - It is protected from the unauthorized access by user programs/applications.
- **User Space** is the memory area where user applications are loaded and executed.

Operating System Basics- Kernel Space and User Space

- The partitioning of memory into kernel and user space is purely OS dependent.
- Some OS implement this kind of partitioning and protection whereas some OS do not segregate the kernel and user application code storage into two separate areas.
- In an operating system with virtual memory support, the user applications are loaded into its corresponding virtual memory space with demand paging technique.
- The entire code for the user application need not be loaded to the main (primary) memory at once.

Operating System Basics- Kernel Space and User Space

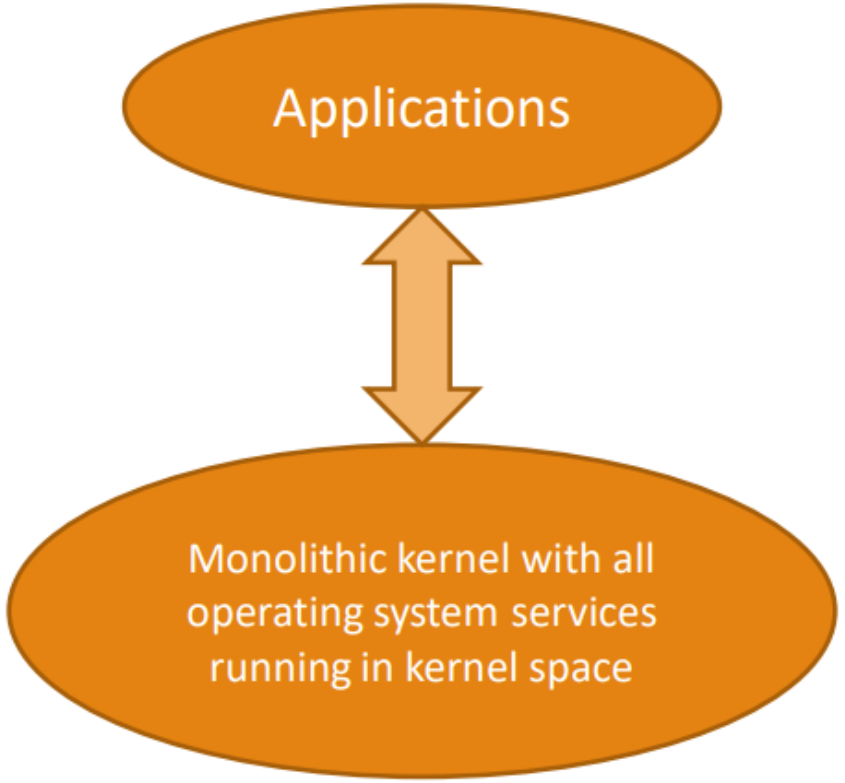
- The user application code is split into different pages and these pages are loaded into and out of the main memory area on a need basis.
- The act of loading the code into and out of the main memory is termed as 'Swapping'.
- Swapping happens between the main (primary) memory and secondary storage memory.

Operating System Basics- Monolithic Kernel and Microkernel

- The kernel forms the heart of an operating system.
- Different approaches are adopted for building an Operating System kernel.
- Based on the kernel design, kernels can be classified into:
  - Monolithic Kernel
  - Microkernel

Operating System Basics- Monolithic Kernel

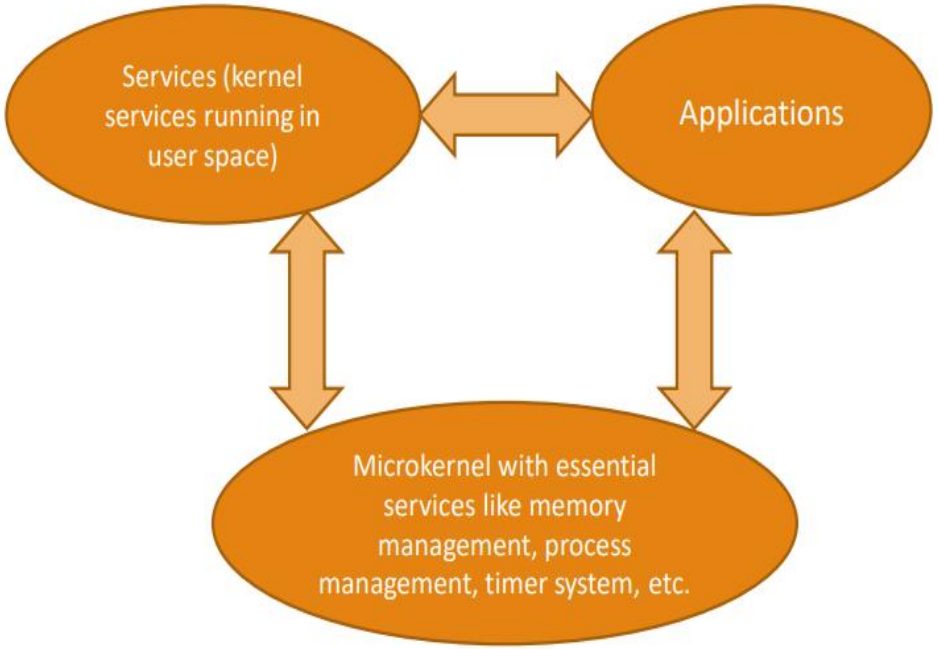
- All kernel services run in the kernel space.
- Here all kernel modules run within the same memory space under a single kernel thread.
- The tight internal integration of kernel modules in monolithic kernel architecture allows the effective utilization of the low-level features of the underlying system.
- 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.
- LINUX, SOLARIS, MS-DOS kernels are examples of monolithic kernel.



The Monolithic Kernel Model

Operating System Basics- 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 abstract to the kernel.
- Memory management, process management, timer systems and interrupt handlers are the essential services, which forms the part of the microkernel. \
- Mach, QNX, Minix 3 kernels are examples for microkernel.



The Microkernel Model

Operating System Basics- Microkernel

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 restarting the entire OS.
- Thus, this approach is highly useful for systems, which demands high 'availability'.
- Since the services which run as 'Servers' are running on a different memory space, the chances of corruption of kernel services are ideally zero.

❑ Configurability

- Any service which runs as 'Server' application can be changed without the need to restart the whole system.
- This makes the system dynamically configurable.

Types of Operating Systems

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

General Purpose Operating System (GPOS)

- The operating systems which are deployed in general computing systems are referred as General Purpose Operating Systems (GPOS).
- The kernel of such a GPOS is more generalized and it contains all kinds of services required for executing generic applications.
- General purpose operating systems are often quite non-deterministic in behavior.
- Their services can inject random delays into application software and may cause slow responsiveness of an application at unexpected times.
- GPOS are usually deployed in computing systems where deterministic behavior is not an important criterion.
- Personal Computer/Desktop system is a typical example for a system where GPOSs are deployed.
- Windows XP/MS-DOS etc. are examples for General Purpose Operating Systems.

Real-Time Operating System (RTOS)

- 'Real-Time' implies deterministic timing behavior.
- Deterministic timing behavior in RTOS context means the OS services consumes only known and expected amounts of time regardless the number of services.
- 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.

Real-Time Operating System (RTOS)

- Predictable performance is the hallmark of a well-designed RTOS.
- This is best achieved by the consistent application of policies and rules.
- Policies guide the design of an RTOS.
- Rules implement those policies and resolve policy conflicts.
- Windows CE, QNX, VxWorks, MicroC/OS-II, etc. are examples of Real-Time Operating Systems (RTOS).

The Real-Time Kernel

- The kernel of a Real-Time Operating System 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 the user applications/tasks.
- The basic functions of a Real-Time kernel are:

- ❑Task/Process Management
- ❑Task/Process Scheduling
- ❑Task/Process Synchronization
- ❑Error/Exception Handling
- ❑Memory Management
- ❑Interrupt Handling
- ❑Time Management

The Real-Time Kernel

❑ Task/Process Management

- 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
- Task/process termination/deletion
- A Task Control Block (TCB) is used for holding the information corresponding to a task

The Real-Time Kernel

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**: 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 (e.g. Task priority = 1 for task with priority = 1)
- **Task Context Pointer**: Pointer for context saving

The Real-Time Kernel

TCB usually contains the following set of information:

- **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 Real-Time Kernel

- The kernel of a Real-Time Operating System 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 the user applications/tasks.
- The basic functions of a Real-Time kernel are:

- ❑Task/Process Management
- ❑Task/Process Scheduling
- ❑Task/Process Synchronization
- ❑Error/Exception Handling
- ❑Memory Management
- ❑Interrupt Handling
- ❑Time Management

The Real-Time Kernel

❑ Task/Process Scheduling

- ➢ 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 optimum scheduling of tasks to provide a deterministic behavior.

The Real-Time Kernel

Task/Process Synchronization

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

The Real-Time Kernel

❑ Error/Exception Handling

- Deals with registering and handling the errors occurred/exceptions raised during the execution of tasks.
- Insufficient memory, timeouts, deadlocks, deadline missing, bus error, divide by zero, unknown instruction execution, etc. are examples of errors/exceptions.
- Errors/Exceptions can happen at the kernel level services or at task level. Deadlock is an example for kernel level exception, whereas timeout is an example for a task level exception.
- The OS kernel gives the information about the error in the form of a system call (API).
- Watchdog timer is a mechanism for handling the timeouts for tasks.

The Real-Time Kernel

❑ Memory Management

- 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'.
- To achieve predictable timing and avoid the timing overheads, most of the RTOS kernels allow tasks to access any of the memory blocks without any memory protection.
- RTOS kernels assume that the whole design is proven correct and protection is unnecessary.
- Some commercial RTOS kernels allow memory protection as optional.

The Real-Time Kernel

❑ Memory Management

- A few RTOS kernels implement Virtual Memory concept for memory allocation if the system supports secondary memory storage (like HDD and FLASH memory).
- In the 'block' based memory allocation, 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.
- The 'block' based memory allocation achieves deterministic behavior with the trade of limited choice of memory chunk size and suboptimal memory usage.

The Real-Time Kernel

❑ Interrupt Handling

- Deals with the handling of various types of interrupts.
- Interrupts provide Real-Time behavior to systems.
- 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.

The Real-Time Kernel

Synchronous interrupts:

- Occur in sync with the currently executing task.
- Usually the software interrupts fall under this 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.

The Real-Time Kernel

Asynchronous interrupts:

- Occur at any point of execution of any task, and are not in sync with the currently executing task.
- The interrupts generated by external devices
- For asynchronous interrupts, the interrupt handler is usually written as separate task and it runs in a different context.
- Context switch happens while handling the asynchronous interrupts.
- Priority levels can be assigned to the interrupts and each interrupt can be enabled or disabled individually.
- Most of the RTOS kernel implements 'Nested Interrupts' architecture.
- Interrupt nesting allows the pre-emption (interruption) of an Interrupt Service Routine (ISR), servicing an interrupt, by a high priority interrupt

The Real-Time Kernel

❑ 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’ and 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 Real-Time Kernel

❑ Time Management

- 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

$$2^{32} \times 10^{-6}\ seconds = \frac{2^{32} \times 10^{-6}}{24 \times 60 \times 60}\ Days = \sim 0.0497\ Days = 1.19\ Hours$$

- If the 'Timer tick' interval is 1 millisecond, the system time register will reset in

$$2^{32} \times 10^{-3}\ seconds = \frac{2^{32} \times 10^{-3}}{24 \times 60 \times 60}\ Days = 49.7\ Days = \sim 50\ Days$$

The Real-Time Kernel

- 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').

The Real-Time Kernel

- The 'Timer tick' interrupt can be utilized for implementing the following actions:
  - Activate the periodic tasks, which are in the idle state.
  - Invoke the scheduler and schedule the tasks again based on the scheduling algorithm.
  - Delete all the terminated tasks and their associated data structures (TCBs).
  - Load the context for the first task in the ready queue. Due to the re-scheduling, the ready task might be changed to a new one from the task, which was preempted by the 'Timer Interrupt' task.

Hard Real-Time

- Real-Time Operating Systems that strictly adhere to the timing constraints for a task are referred as 'Hard Real-Time' systems.
  - They 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 loss and irrecoverable damages to the system/users.
- Hard Real-Time systems emphasize the principle *'A late answer is a wrong answer'.*
- Air bag control systems and Anti-lock Brake Systems (ABS) of vehicles are typical examples for Hard Real-Time Systems.
  - Any delay in the deployment of the air bags makes the life of the passengers under threat.

Hard Real-Time

- Hard Real-Time Systems does not implement the virtual memory model for handling the memory.
  - This eliminates the delay in swapping in and out the code corresponding to the task to and from the primary memory.
- Most of the Hard Real-Time Systems are automatic and does not contain a Human in the Loop (HITL).
  - The presence of human in the loop for tasks introduces unexpected delays in the task execution.

Soft Real-Time

- Real-Time Operating Systems that do 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 Real-time system if the frequency of deadline missing is within the compliance limit of the Quality of Service (QoS).
- A Soft Real-Time system emphasizes the principle 'A late answer is an acceptable answer, but it could have done bit faster'.

Soft Real-Time

- 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.

Soft Real-Time

- 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.

Tasks, Process and Threads

- The term 'task' refers to something that needs to be done.
- In the operating system context, a task is defined as the program in execution and the related information maintained by the operating system for the program.
- Task is also known as 'Job' in the operating system context.
- A program or part of it in execution is also called a 'Process'.
- The terms 'Task', 'Job' and 'Process' refer to the same entity in the operating system context and most often they are used interchangeably.

Process

- A 'Process' is a program, or part of it, in execution.
- Process is also known as an instance of a program in execution.
- Multiple instances of the same program can execute simultaneously.
- A process requires various system resources:
  - ***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.

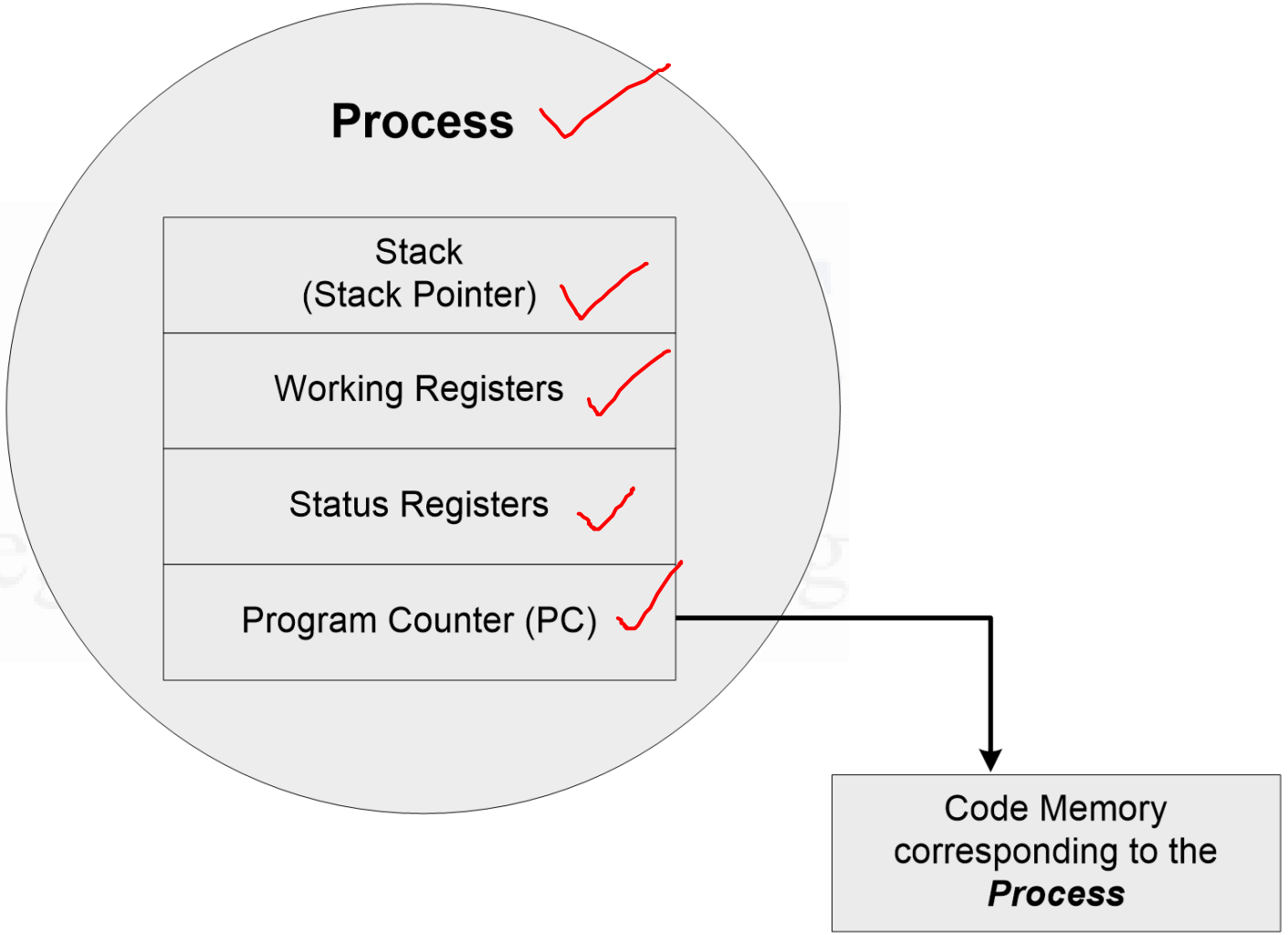
Process

- The concept of 'Process' leads to concurrent execution.
- 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,
  - Program Counter (PC)
  - Point to the next executable instruction of the process
  - Stack for holding the local variables

Process

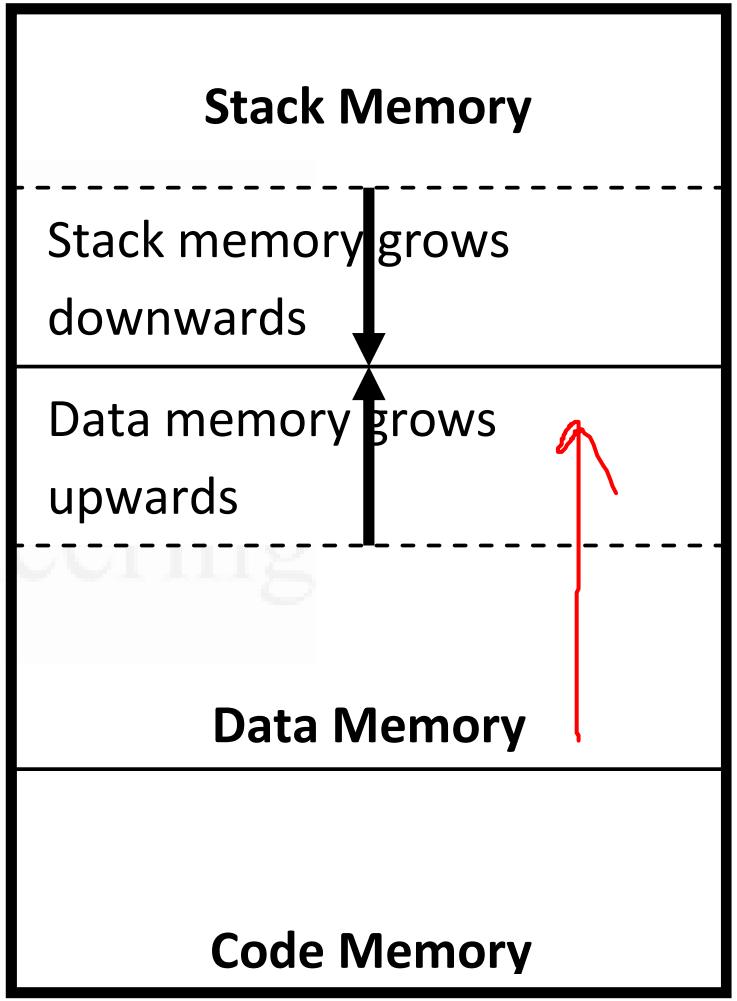
- A process, which inherits all the properties of the CPU, can be considered as a virtual processor, awaiting its turn to have its properties switched into the physical processor

- When the process gets its turn, its registers and Program counter register becomes mapped to the physical registers of the CPU



Process

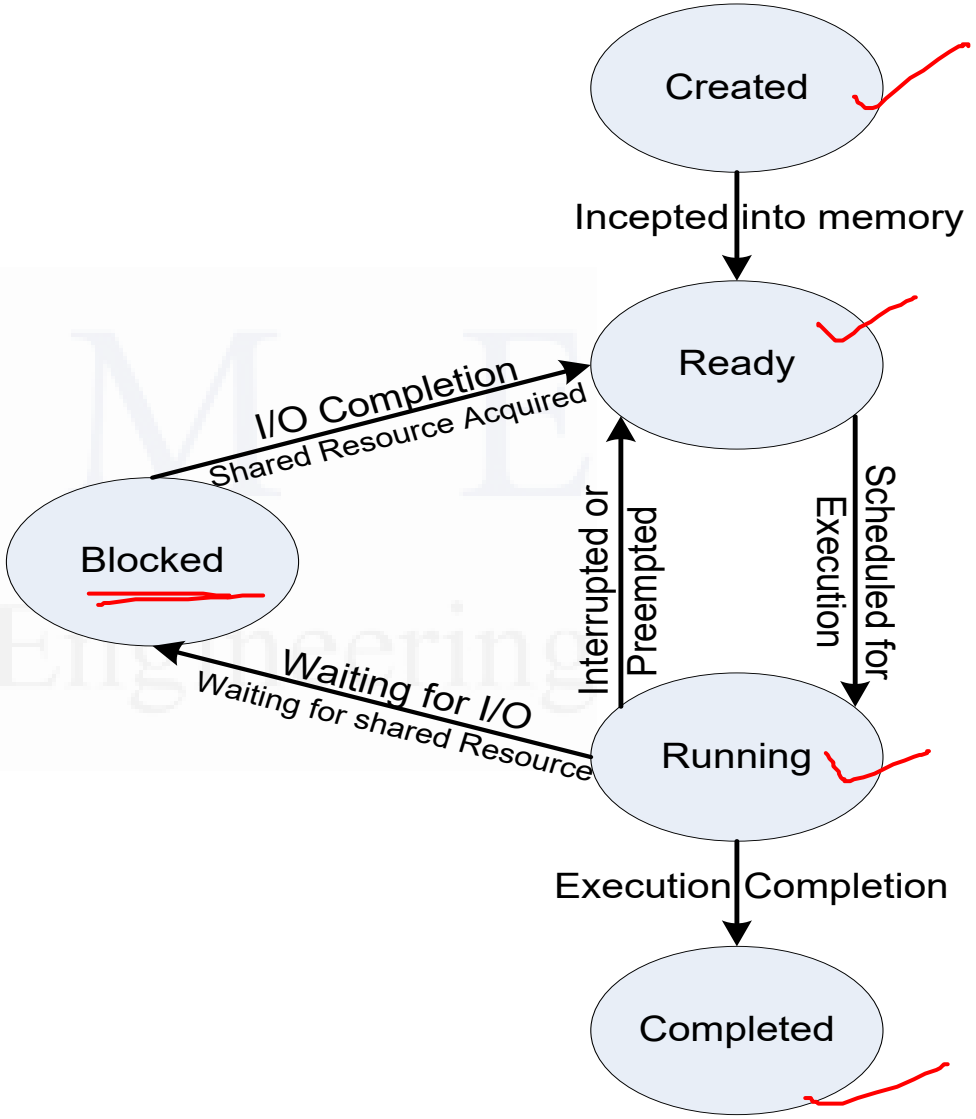
- The memory occupied by the *process* is segregated into three regions namely; ***Stack memory, Data memory and Code memory***
- The 'Stack' memory holds all temporary data such as variables local to the process
- Data memory holds all global data for the process
- The code memory contains the program code (instructions) corresponding to the process
- On loading a process into the main memory, a specific area of memory is allocated for the process
- The stack memory usually starts at the highest memory address from the memory area allocated for the process (Depending on the OS kernel implementation)



Process

Process States & State Transition

- The creation of a process to its termination is not a single step operation
- The process traverses through a series of states during its transition from the newly created state to the terminated state
- The cycle through which a process changes its state from '*newly created*' to '*execution completed*' is known as '*Process Life Cycle*'.



Process

Process States & State Transition

- **Created State:** The state at which a process is being created is referred as 'Created State'. The Operating System 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.

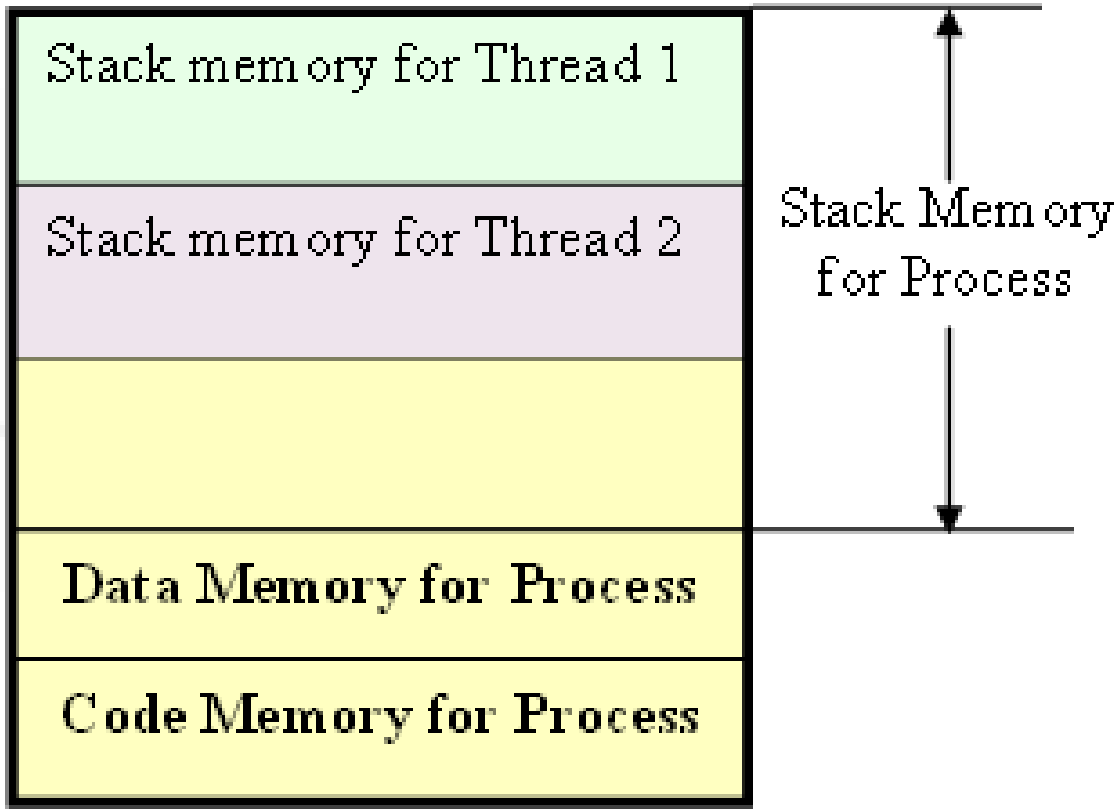
Process

Process States & State Transition

- **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
- 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

Threads

- 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

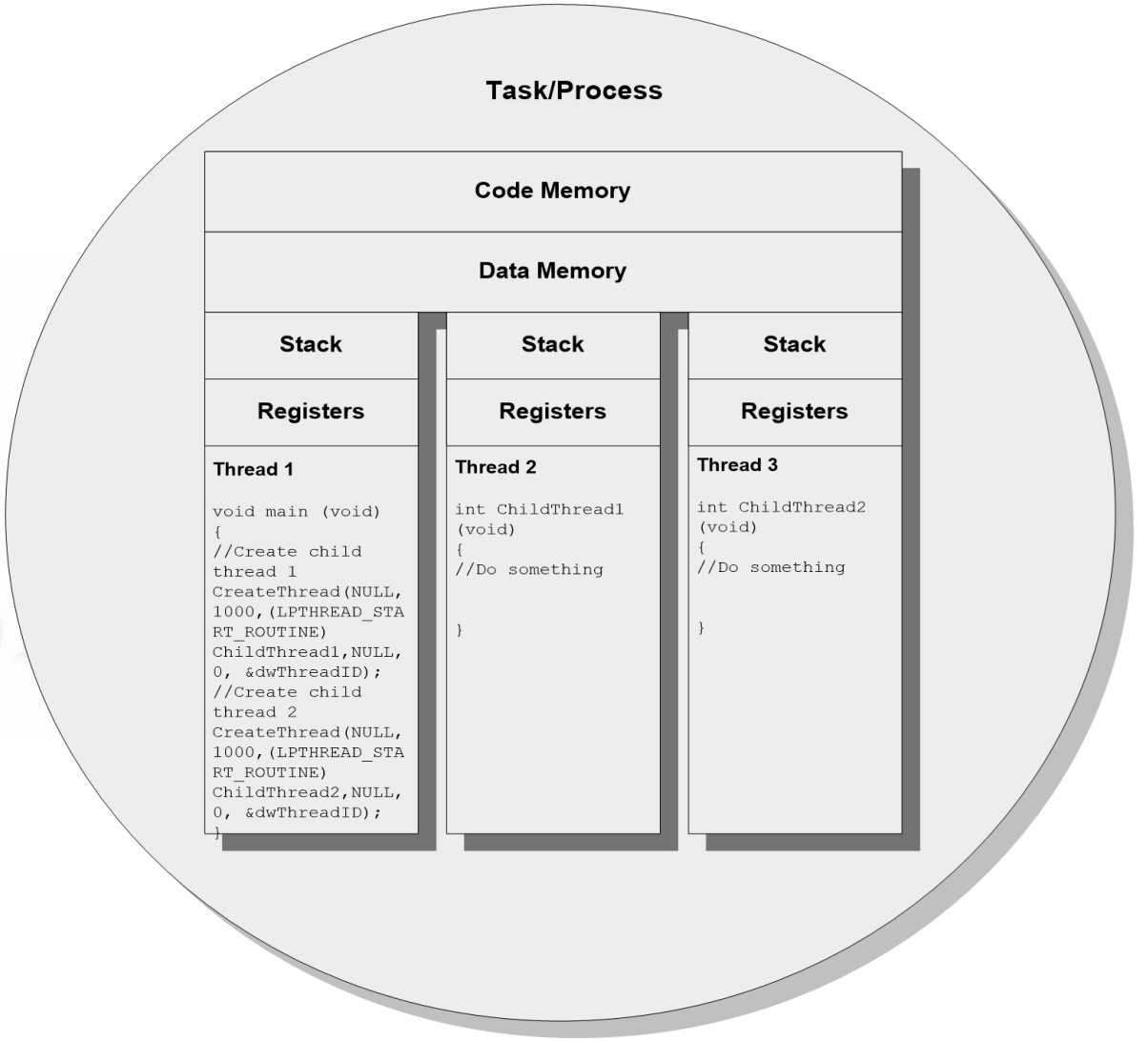


Threads

The Concept 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	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 overhead and is comparatively slower.
If a thread expires, its stack is reclaimed by the process.	If a process dies, the resources allocated to it are reclaimed by the OS and all the associated threads of the process also dies.

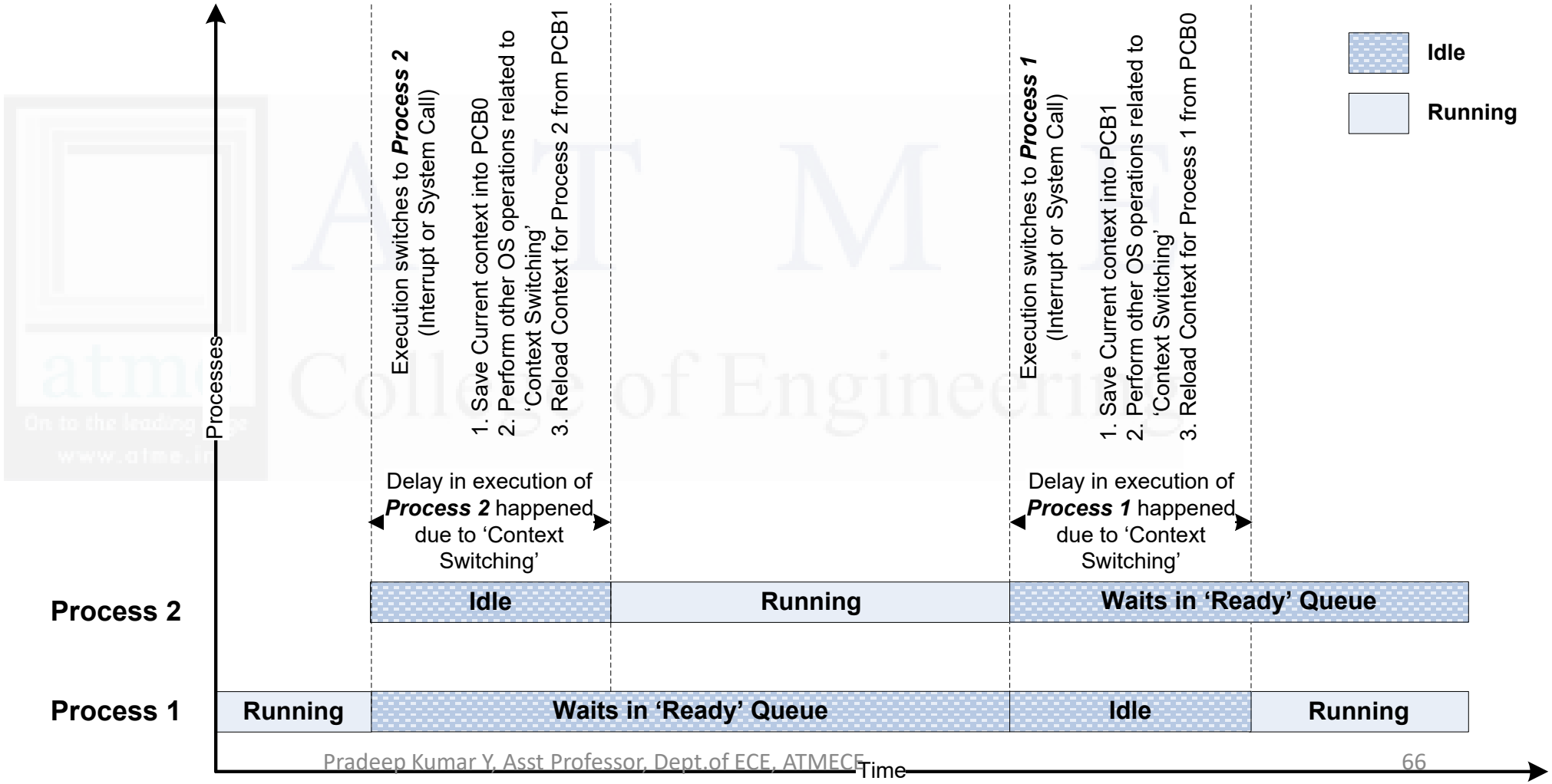
Multiprocessing & Multitasking

- 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

Multiprocessing & Multitasking

- The ability of the Operating System to have multiple programs in memory, which are ready for execution, is referred as *multiprogramming*
- *Multitasking* refers to the ability of an operating system to hold multiple processes in memory and switch the processor (CPU) from executing one process to another process
- *Multitasking* involves '*Context switching*', '*Context saving*' and '*Context retrieval*'
- *Context switching* refers to the switching of execution context from task to other
- When a task/process switching happens, the current context of execution should be saved to (*Context saving*) retrieve it at a later point of time when the CPU executes the process, which is interrupted currently due to execution switching
- During context switching, the context of the task to be executed is retrieved from the saved context list. This is known as *Context retrieval*

Multitasking – Context Switching



Types of Multitasking

Depending on how the task/process execution switching act is implemented, multitasking can is classified into

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

Types of Multitasking

1. Co-operative Multitasking:

- Co-operative multitasking is the most primitive form of multitasking in which a task/process gets a chance to execute only when the currently executing task/process voluntarily relinquishes the CPU.
- In this method, any task/process can avail the CPU as much time as it wants.
- Since this type of implementation involves the mercy of the tasks each other for getting the CPU time for execution, it is known as co-operative multitasking. If the currently executing task is non-cooperative, the other tasks may have to wait for a long time to get the CPU

Types of Multitasking

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

Types of Multitasking

3. 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.
- The co-operative and non-preemptive multitasking differs in their behavior when they are in the 'Blocked/Wait' state.
- In co-operative multitasking, the currently executing process/task need not relinquish the CPU when it enters the 'Blocked/Wait' sate, waiting for an I/O, or a shared resource access or an event to occur whereas in non-preemptive multitasking the currently executing task relinquishes the CPU when it waits for an I/O.

Task Scheduling

- ✓ In a multitasking system, 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 task scheduling policy can be *pre-emptive*, *non-preemptive* or *co-operative*

Task Scheduling

Depending on the scheduling policy the process scheduling decision may take place when a process switches its state to

- ✓ '*Ready*' state from '*Running*' state
- ✓ '*Blocked/Wait*' state from '*Running*' state
- ✓ '*Ready*' state from '*Blocked/Wait*' state
- ✓ '*Completed*' state

Task Scheduling - Scheduler Selection

The selection of a scheduling criteria/algorithm should consider

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

Task Scheduling - Scheduler Selection

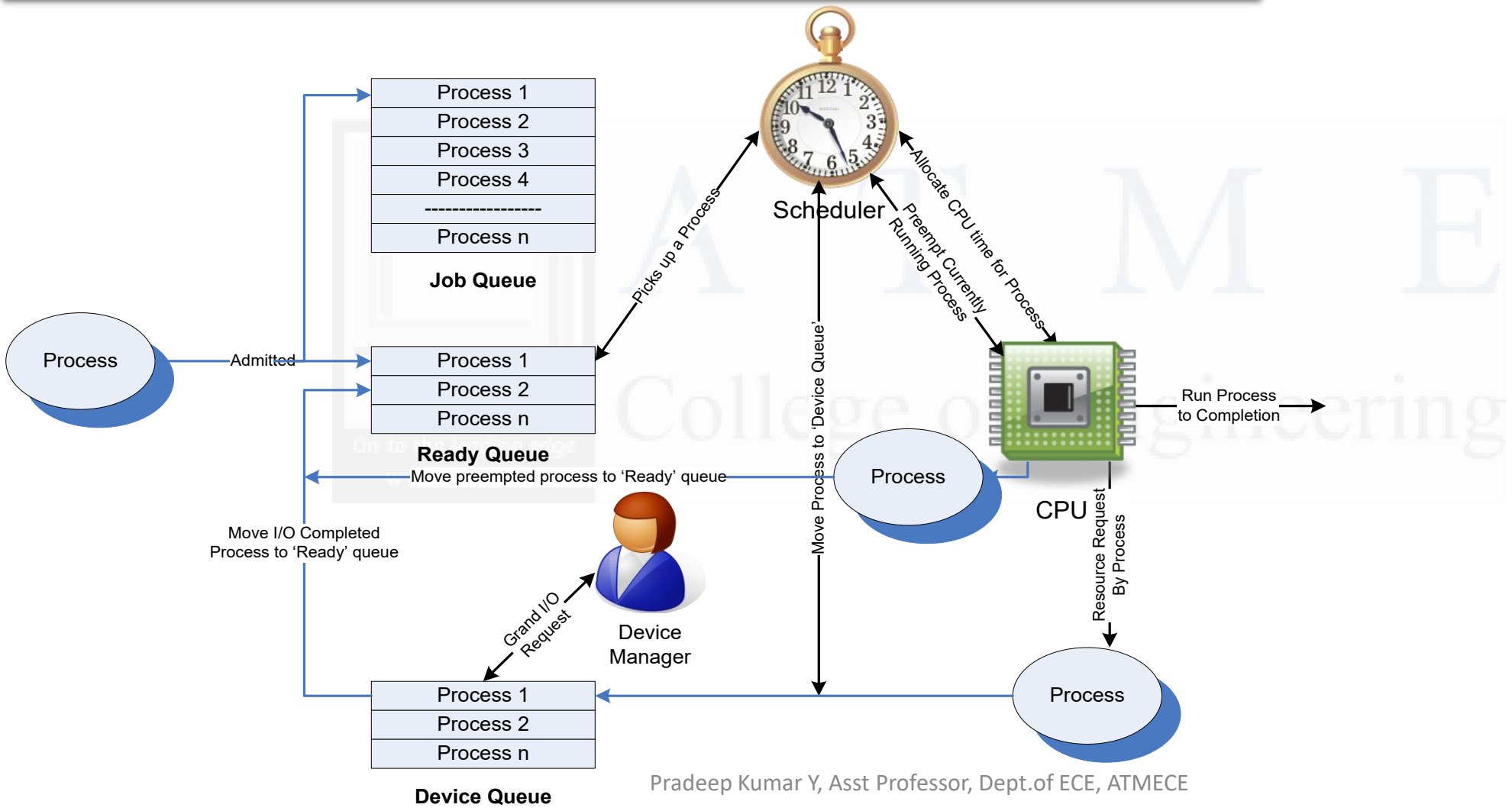
- **Turnaround Time:** It is the amount of time taken by a process for completing its execution. It includes the time spent by the process for waiting for the main memory, time spent in the ready queue, time spent on completing the I/O operations, and the time spent in execution. The turnaround time should be a minimum 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.

Task Scheduling - Queues

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

Task Scheduling - Task transition through various Queues



Non-preemptive scheduling

First Come First Served (FCFS)/ First In First Out (FIFO) Scheduling

- ✓ Allocates CPU time to the processes based on the order in which they enters 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

Drawbacks:

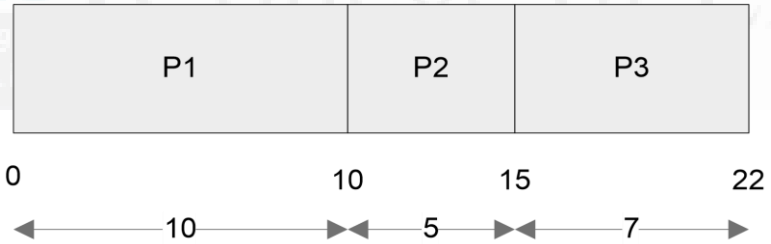
- ✓ Favors monopoly of process. A process, which does not contain any I/O operation, continues its execution until it finishes its task
- ✓ In general, FCFS favors CPU bound processes and I/O bound processes may have to wait until the completion of CPU bound process, if the currently executing process is a CPU bound process. This leads to poor device utilization.
- ✓ The average waiting time is not minimal for FCFS scheduling algorithm

Non-preemptive scheduling

First Come First Served (FCFS)/ First In First Out (FIFO) Scheduling

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

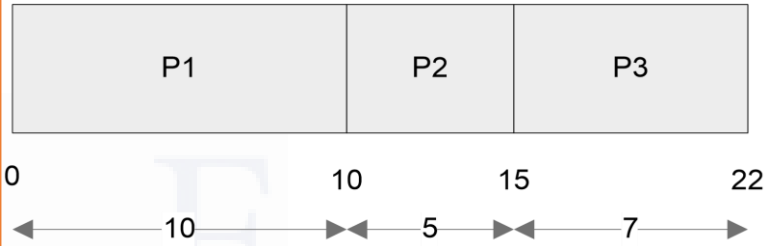


Assuming the CPU is readily available at the time of arrival of P1, P1 starts executing without any waiting in the '*Ready*' queue. Hence the waiting time for P1 is zero.

Non-preemptive scheduling

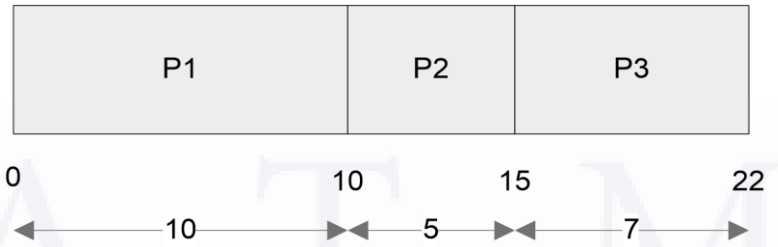
Waiting Time for P1 = 0 ms (P1 starts executing first)
Waiting Time for P2 = 10 ms (P2 starts executing after completing P1)
Waiting Time for P3 = 15 ms (P3 starts executing after completing P1 and P2)

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

Non-preemptive scheduling



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 milliseconds

Average Turn Around Time = Average Waiting time + Average execution time

= 8.33 + 7.33

= **15.66 milliseconds**

Non-preemptive scheduling

Last Come First Served (LCFS)/ Last In First Out (LIFO) Scheduling

- ✓ 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

Drawbacks:

- ✓ Favors monopoly of process. A process, which does not contain any I/O operation, continues its execution until it finishes its task
- ✓ In general, LCFS favors CPU bound processes and I/O bound processes may have to wait until the completion of CPU bound process, if the currently executing process is a CPU bound process. This leads to poor device utilization.
- ✓ The average waiting time is not minimal for LCFS scheduling algorithm

Non-preemptive scheduling

Last Come First Served (LCFS)/ Last In First Out (LIFO) Scheduling

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 up it and P2, P3 entered '*Ready*' queue after that).

Now a new process P4 with estimated completion time 6ms enters the 'Ready' queue after 5ms of scheduling P1.

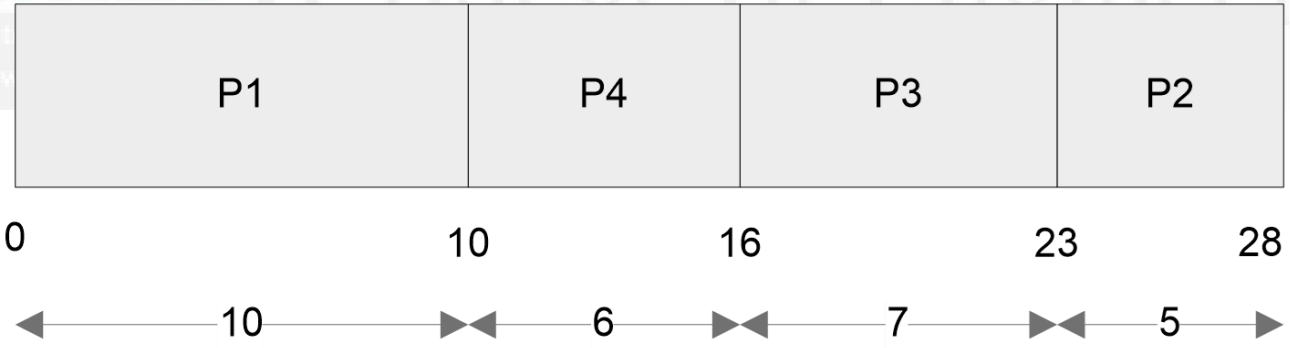
Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time (Assuming there is no I/O waiting for the processes).

Assume all the processes contain only CPU operation and no I/O operations are involved

Non-preemptive scheduling

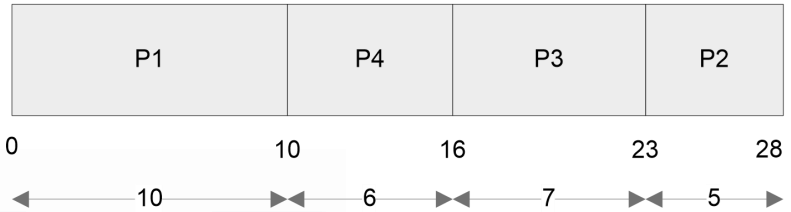
Last Come First Served (LCFS)/ Last In First Out (LIFO) Scheduling

- 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



Non-preemptive scheduling

Last Come First Served (LCFS)/ Last In First Out (LIFO) Scheduling



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 5ms 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

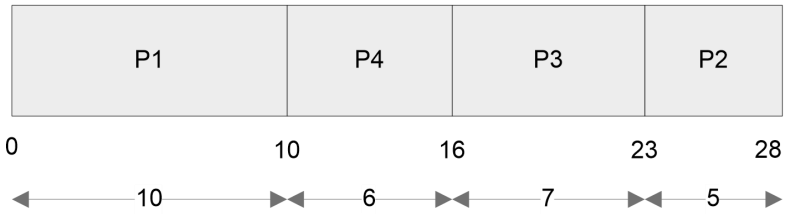
= (Waiting time for (P1+P4+P3+P2)) / 4

= (0 + 5 + 16 + 23)/4 = 44/4

= ***11 milliseconds***

Non-preemptive scheduling

Last Come First Served (LCFS)/ Last In First Out (LIFO) Scheduling



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***

Non-preemptive scheduling

Shortest Job First (SJF) Scheduling

- ✓ Allocates CPU time to the processes based on the execution completion time for tasks
- ✓ The average waiting time for a given set of processes is minimal in SJF scheduling
- ✓ Optimal compared to other non-preemptive scheduling like FCFS

Drawbacks:

- ✓ 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 starts its execution
- ✓ May lead to the 'Starvation' of processes with high estimated completion time
- ✓ Difficult to know in advance the next shortest process in the '*Ready*' queue for scheduling since new processes with different estimated execution time keep entering the '*Ready*' queue at any point of time.

Non-preemptive scheduling

Shortest Job First (SJF) Scheduling

Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7milliseconds 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 up it and P2, P3 entered 'Ready' queue after that).

Now a new process P4 with estimated completion time 6ms enters the 'Ready' queue after 5ms of scheduling P1.

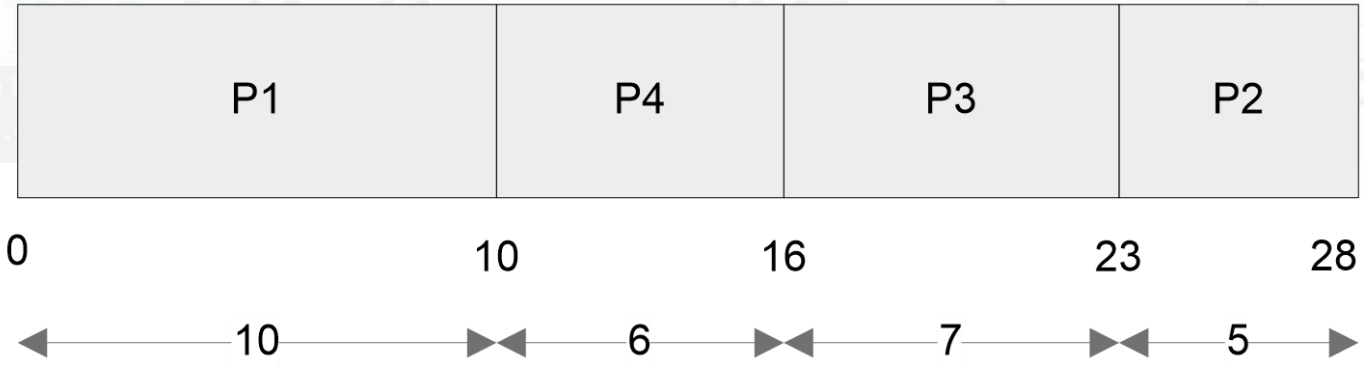
Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time (Assuming there is no I/O waiting for the processes).

Assume all the processes contain only CPU operation and no I/O operations are involved.

Non-preemptive scheduling

Shortest Job First (SJF) Scheduling

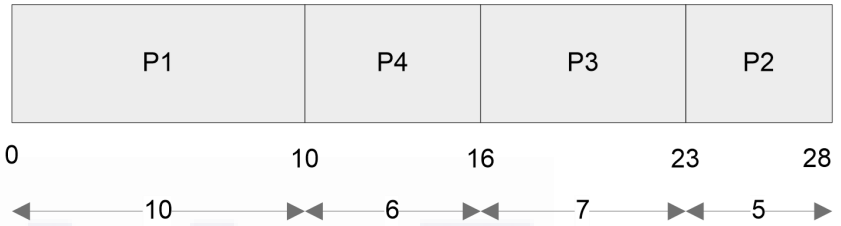
- Initially there is only P1 available in the Ready queue and the scheduling sequence will be P1, P3, P2.
- P4 enters the queue during the execution of P1 and becomes the last process entered the 'Ready' queue.
- Now the order of execution changes to P1, P4, P3, and P2 as



Non-preemptive scheduling

Shortest Job First (SJF) Scheduling

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 5ms 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)

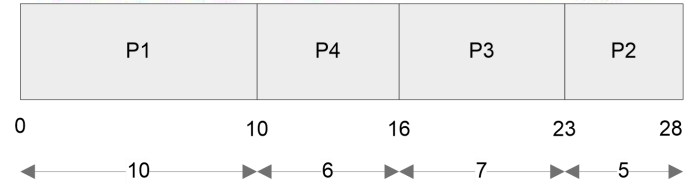
Average waiting time = (Waiting time for all processes) / No. of Processes

= (Waiting time for (P1+P4+P3+P2)) / 4

= (0 + 5 + 16 + 23)/4 = 44/4

= 11 milliseconds

Non-preemptive scheduling



Shortest Job First (SJF) Scheduling

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

Non-preemptive scheduling

Priority based Scheduling

- ✓ A priority, which is unique or same is associated with each task
- ✓ The priority of a task is expressed in different ways, like a priority number, the time required to complete the execution etc.
- ✓ In number based priority assignment the priority is a number ranging from 0 to the maximum priority supported by the OS. The maximum level of priority is OS dependent.
- ✓ Windows CE supports 256 levels of priority (0 to 255 priority numbers, with 0 being the highest priority)
- ✓ The priority is assigned to the task on creating it. It can also be changed dynamically (If the Operating System supports this feature)
- ✓ The non-preemptive priority based scheduler sorts the 'Ready' queue based on the priority and picks the process with the highest level of priority for execution

Non-preemptive scheduling

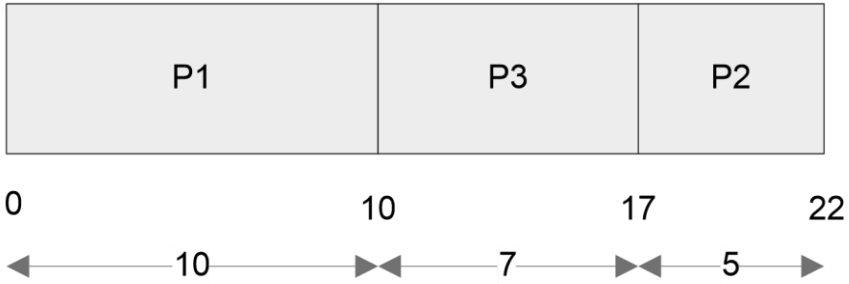
Priority based Scheduling

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



Non-preemptive scheduling

Priority based Scheduling

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

Non-preemptive scheduling

Priority based Scheduling

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

Non-preemptive scheduling

Priority based Scheduling

Drawbacks:

- Similar to SJF scheduling algorithm, non-preemptive priority based algorithm also possess the drawback of '*Starvation*' where a process whose priority is low may not get a chance to execute if more and more processes with higher priorities enter the '*Ready*' queue before the process with lower priority starts its execution.
- '*Starvation*' can be effectively tackled in priority based non-preemptive scheduling by dynamically raising the priority of the low priority task/process which is under starvation (waiting in the ready queue for a longer time for getting the CPU time)
- The technique of gradually raising the priority of processes which are waiting in the 'Ready' queue as time progresses, for preventing '*Starvation*', is known as '*Aging*'.

Preemptive scheduling

- ✓ Employed in systems, which implements preemptive multitasking model
- ✓ Every task in the '*Ready*' queue gets a chance to execute. When and how often each process gets a chance to execute (gets the CPU time) is dependent on the type of preemptive scheduling algorithm used for scheduling the processes
- ✓ 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

Preemptive scheduling

- ✓ 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*'
- ✓ Time-based preemption and priority-based preemption are the two important approaches adopted in preemptive scheduling

Preemptive scheduling

Preemptive SJF Scheduling/ Shortest Remaining Time (SRT)

- ✓ The ***non preemptive SJF*** scheduling algorithm sorts the 'Ready' queue only after the current process completes execution or enters wait state, whereas 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 execution time of 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
- ✓ Always compares the execution completion time (ie the remaining execution 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 scheduling

Preemptive SJF Scheduling

Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds respectively enters the ready queue together.

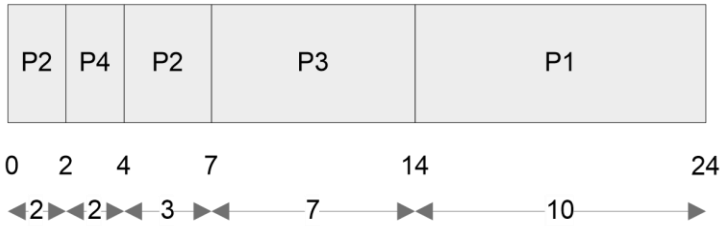
A new process P4 with estimated completion time 2ms enters the 'Ready' queue after 2ms.

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 5ms) for scheduling.

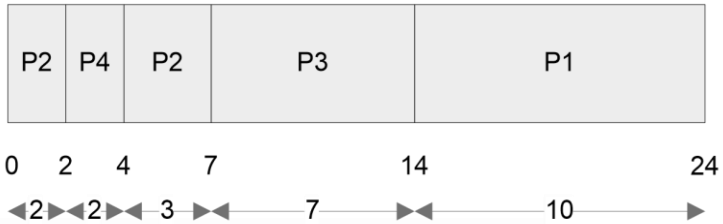
Now process P4 with estimated execution completion time 2ms enters the '*Ready*' queue after 2ms of start of execution of P2.

The processes are re-scheduled for execution in the following order



Preemptive scheduling

Preemptive SJF Scheduling



The waiting time for all the processes are given as

Waiting Time for P2 = 0 ms + (4 -2) ms = 2ms (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 (2ms) is less than that of the Remaining time for execution completion of P2 (Here it is 3ms))

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

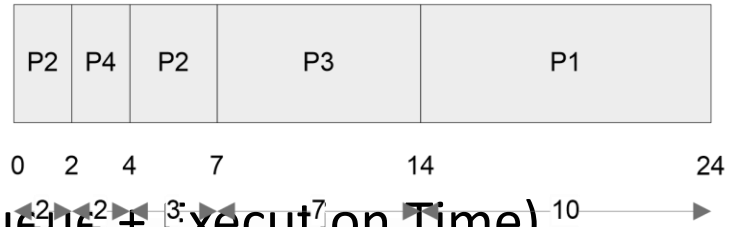
Waiting Time for P1 = 14 ms (P1 starts executing after completing P4, P2 and P3)

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***

Preemptive scheduling



Preemptive SJF Scheduling

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

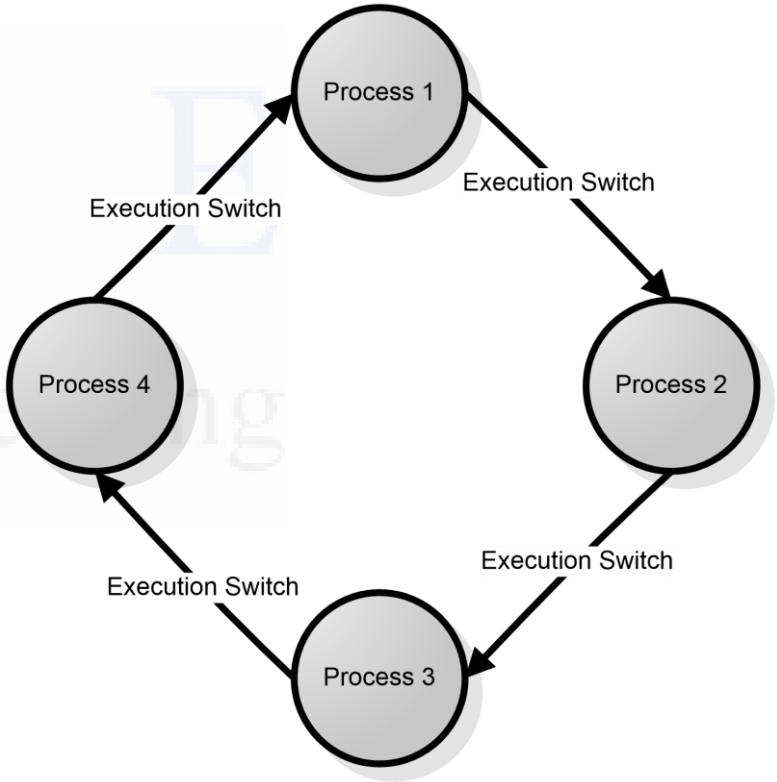
= (7+2+14+24)/4 = 47/4

= 11.75 milliseconds

Preemptive scheduling

Round Robin (RR) 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. It is executed for a pre-defined time
- ✓ 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
- ✓ Round Robin scheduling is similar to the FCFS scheduling and the only difference is that a time slice based preemption is added to switch the execution between the processes in the 'Ready' queue



Preemptive scheduling

Round Robin (RR) Scheduling

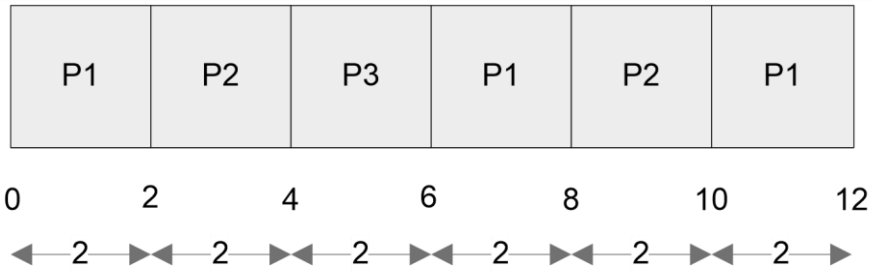
Three processes with process IDs P1, P2, P3 with estimated completion time 6, 4, 2 milliseconds respectively, enters the ready queue together in the order P1, P2, P3.

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= 2ms.

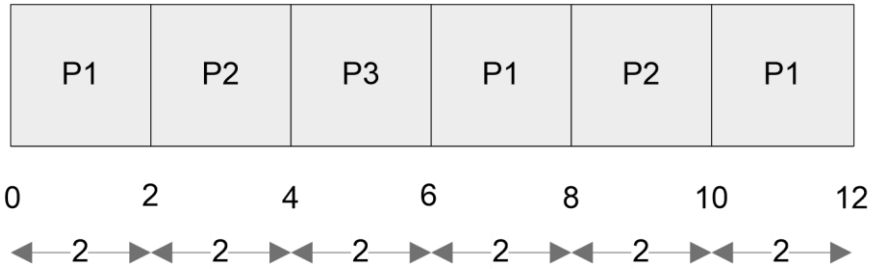
Preemptive scheduling

Round Robin (RR) Scheduling

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 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



Preemptive scheduling



Round Robin (RR) Scheduling

The waiting time for all the processes are given as

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

Waiting Time for P2 = (2-0) + (8-4) = 2+4 = 6ms (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) = 4ms (P3 starts executing after completing the first time slices for P1 and P2 and completes its execution in a single time slice.)

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

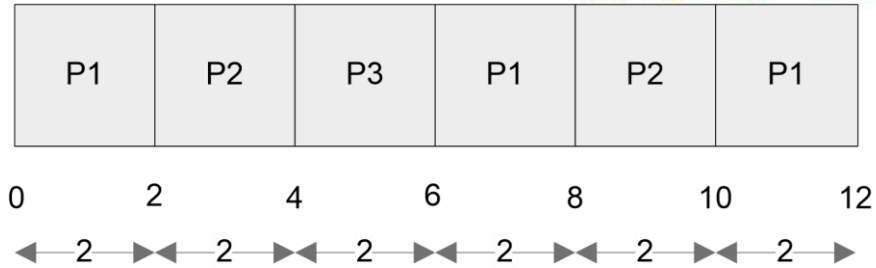
= (Waiting time for (P1+P2+P3)) / 3

= (6+6+4)/3 = 16/3

= 5.33 milliseconds

Preemptive scheduling

Round Robin (RR) Scheduling



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***

Preemptive scheduling

Priority based Scheduling

- ✓ Same as that of the ***non-preemptive priority*** based scheduling except for the switching of execution between tasks
- ✓ In ***preemptive priority*** based 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 releases the CPU
- ✓ The priority of a task/process in preemptive priority based scheduling is indicated in the same way as that of the mechanisms adopted for non-preemptive multitasking

Preemptive scheduling

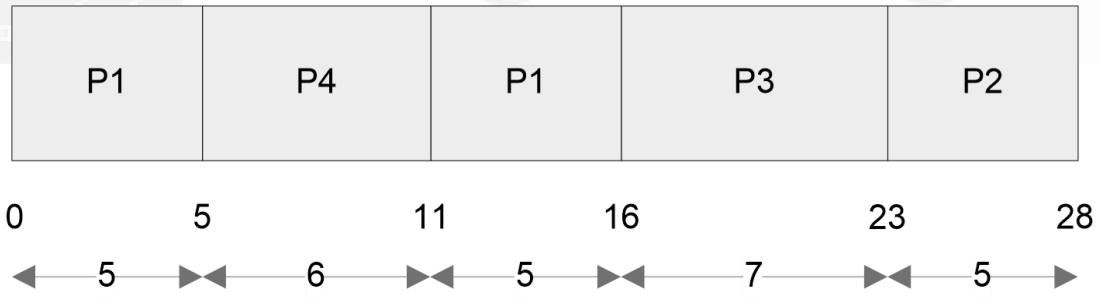
Priority based Scheduling

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 6ms and priority 0 enters the 'Ready' queue after 5ms of start of execution of P1. Assume all the processes contain only CPU operation and no I/O operations are involved.

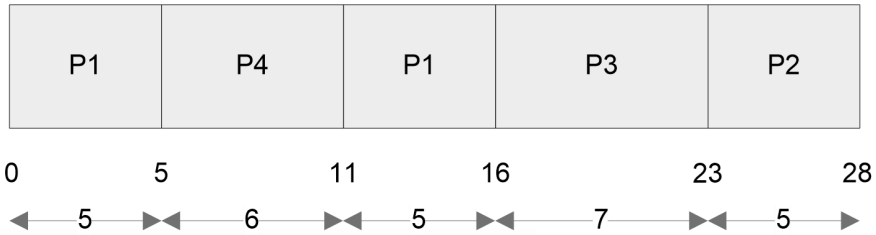
Preemptive scheduling

Priority based Scheduling

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 6ms and priority 0 enters the '*Ready*' queue after 5ms of start of execution of P1. The processes are re-scheduled for execution in the following order



Preemptive scheduling



Priority based Scheduling

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 5ms 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)

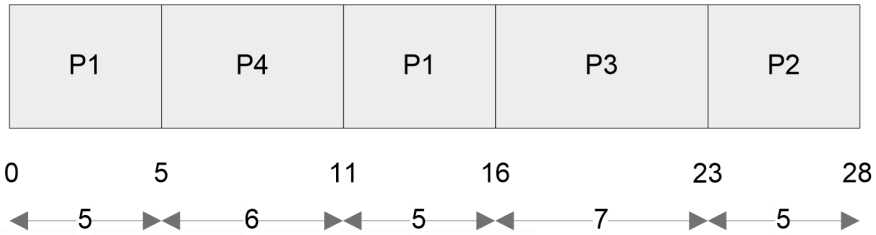
Waiting Time for P3 = 16 ms (P3 starts executing after completing P1 and P4)

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

Preemptive scheduling

Priority based Scheduling



Turn Around Time (TAT) for P1 = 16 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P4 = 6ms (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

Task Communication

In a multitasking system, multiple tasks/processes run concurrently (in pseudo parallelism) and each process may or may not interact between. Based on the degree of interaction, the processes /tasks running on an OS are classified as

1. **Co-operating Processes:** In the co-operating interaction model one process requires the inputs from other processes to complete its execution.
2. **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

The co-operating processes exchanges information and communicate through

- **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.

Inter Process (Task) Communication (IPC)

- ✓ IPC refers to the mechanism through which tasks/processes communicate each other
- ✓ IPC is essential for task /process execution co-ordination and synchronization
- ✓ Implementation of IPC mechanism is OS kernel dependent
- ✓ Some important IPC mechanisms adopted by OS kernels are:
  - ✓ Shared Memory
    - ✓ Global Variables
    - ✓ Pipes (Named & Un-named)
    - ✓ Memory mapped Objects
  - ✓ Message Passing
    - ✓ Message Queues
    - ✓ Mailbox
    - ✓ Mail slot
    - ✓ Signals
  - ✓ Remote Procedure Calls (RPC)

Inter Process (Task) Communication (IPC)

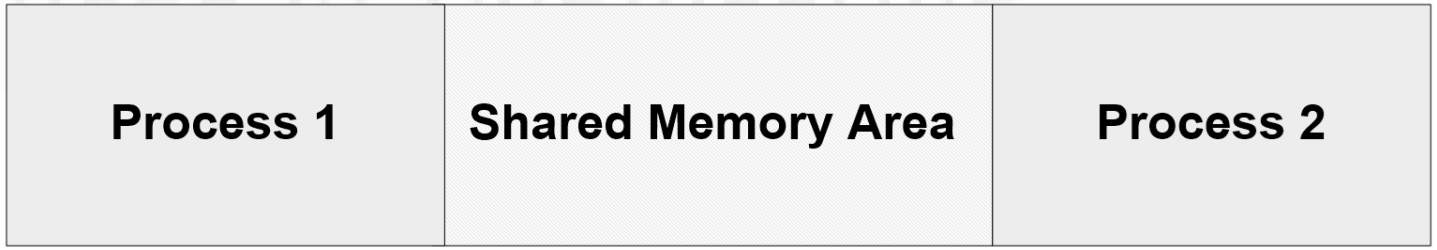
IPC – Shared Memory

✓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 require this information can read the same from the shared memory area

✓Same as the real world concept where a 'Notice Board' is used by the college to publish the information for students



Concept of Shared Memory

Inter Process (Task) Communication (IPC)

IPC – Shared Memory: Pipes

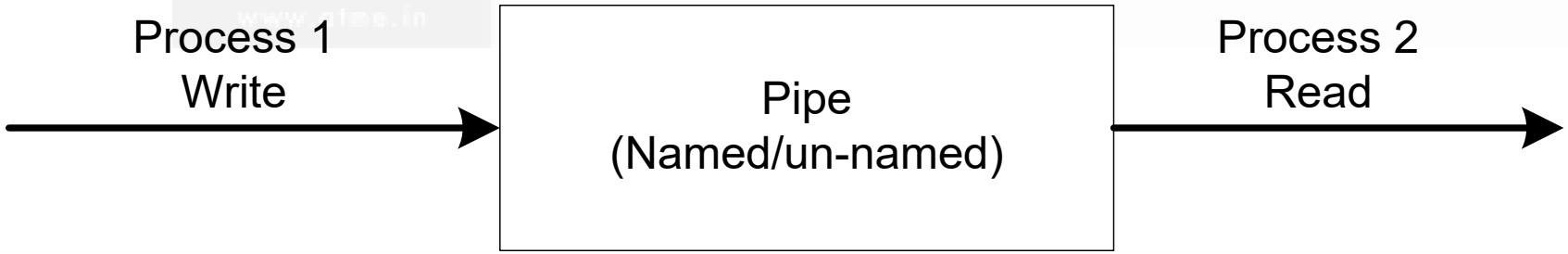
- 'Pipe' is a section of the shared memory used by processes for communicating.
- Pipes follow the client-server architecture. A process which creates a pipe is known as pipe server and a process which connects to a pipe is known as pipe client.
- A pipe can be considered as a conduit for information flow and has two conceptual ends. It can be unidirectional, allowing information flow in one direction or bidirectional allowing bi-directional information flow.
- 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

Inter Process (Task) Communication (IPC)

IPC – Shared Memory: Pipes

The implementation of 'Pipes' is OS dependent. 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.

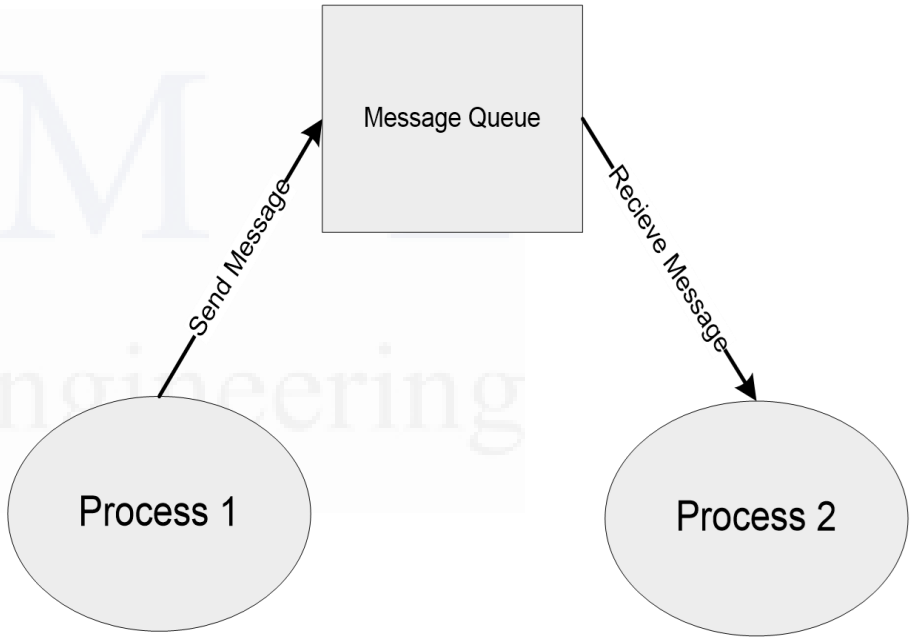


Concept of Shared Memory

Inter Process (Task) Communication (IPC)

IPC – Message Passing: 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
- ✓ Messages are sent and received through *send* (Name of the process to which the message is to be sent, message) and *receive* (Name of the process from which the message is to be received, message) methods
- ✓ The messages are exchanged through a message queue
- ✓ The implementation of the message queue, *send* and *receive* methods are OS kernel dependent.



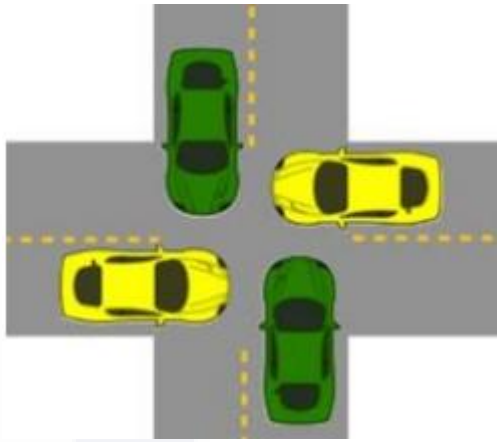
Task/Process Synchronization

- ✓Multiple processes may try to access and modify shared resources in a multitasking environment. This may lead to conflicts and inconsistent results
- ✓Processes should be made aware of the access of a shared resource by each process and should not be allowed to access a shared resource when it is currently being accessed by another processes
- ✓The act of making processes aware of the access of shared resources by each process to avoid conflicts is known as '***Task/Process Synchronization***'
- ✓Task Synchronization is essential for avoiding conflicts in shared resource access and ensuring a specified sequence for task execution
- ✓Various synchronization issues may arise in a multitasking environment if processes are not synchronized properly in shared resource access

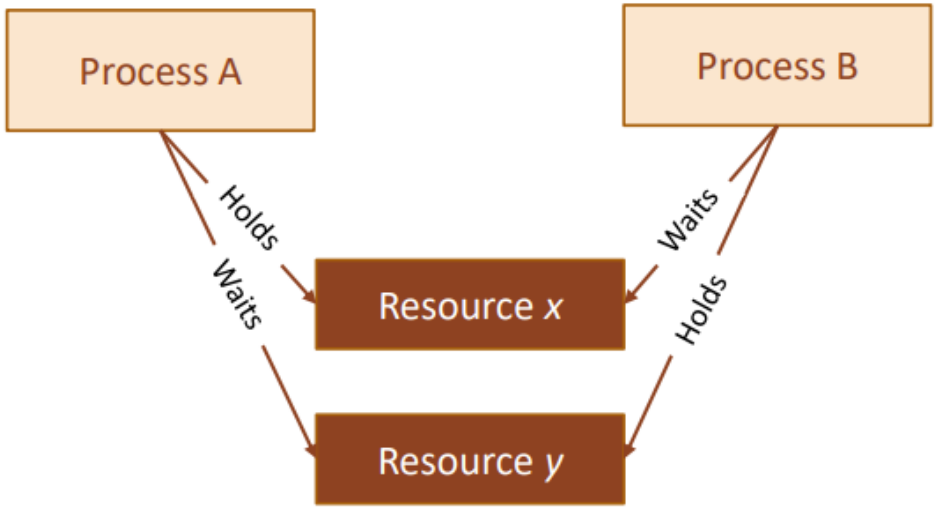
Task/Process Synchronization

Task Synchronization Issues – 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



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



Task/Process Synchronization

Task Synchronization Issues – Livelock

- The *Livelock* condition is similar to the deadlock condition except that a process in livelock condition changes its state with time. While 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 do something but unable to make any progress in the execution completion. The livelock condition is better explained with the real world example, two people attempting to cross each other in a narrow corridor.
- Both of the persons move towards each side of the corridor to allow the opposite person to cross. Since the corridor is narrow, none of them are able to cross each other. Here both of the persons perform some action but still they are unable to achieve their target- Cross each other

Task/Process Synchronization

Task Synchronization Issues – Starvation

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

How to Choose an RTOS

The decision of choosing an RTOS for an embedded design is very crucial.

A lot of factors needs to be analyzed carefully before making a decision on the selection of an RTOS.

The requirements that needs to be analyzed in the selection of an RTOS for an embedded design fall under two categories:

- Functional requirements
- Non-functional requirements

How to Choose an RTOS

Functional Requirements

Processor Support

- ✓ It is not necessary that all RTOS's support all kinds of processor architecture.
- ✓ It is essential to ensure the processor support by the RTOS.

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.

How to Choose an RTOS

Functional Requirements

Real-time Capabilities

- ✓ It is not mandatory that the operating system for all embedded systems need to be Real-time and all embedded Operating systems are 'Real-time' in behavior.
- ✓ The task/process scheduling policies plays an important role in the 'Real-time' behavior of an OS.
- ✓ Analyze the real-time capabilities of the OS under consideration and the standards met by the operating system for real-time capabilities.

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

How to Choose an RTOS

Functional Requirements

Inter Process Communication and Task Synchronization

- 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.

Modularization Support

Most of the operating systems provide a bunch of features. • At times it may not be necessary for an embedded product for its functioning. • It is very useful if the OS supports modularization where in 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.

How to Choose an RTOS

Functional Requirements

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.

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) customized 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. • The OS may include these components as built-in component, if not, check the availability of the same from a third party vendor for the OS under consideration.

How to Choose an RTOS

Non-Functional Requirements

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.

Sometimes it may be possible to build the required features by customizing an Open source OS. 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.

How to Choose an RTOS

Non-Functional Requirements

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.

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.

How to Choose an RTOS

Non-Functional Requirements

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.

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.

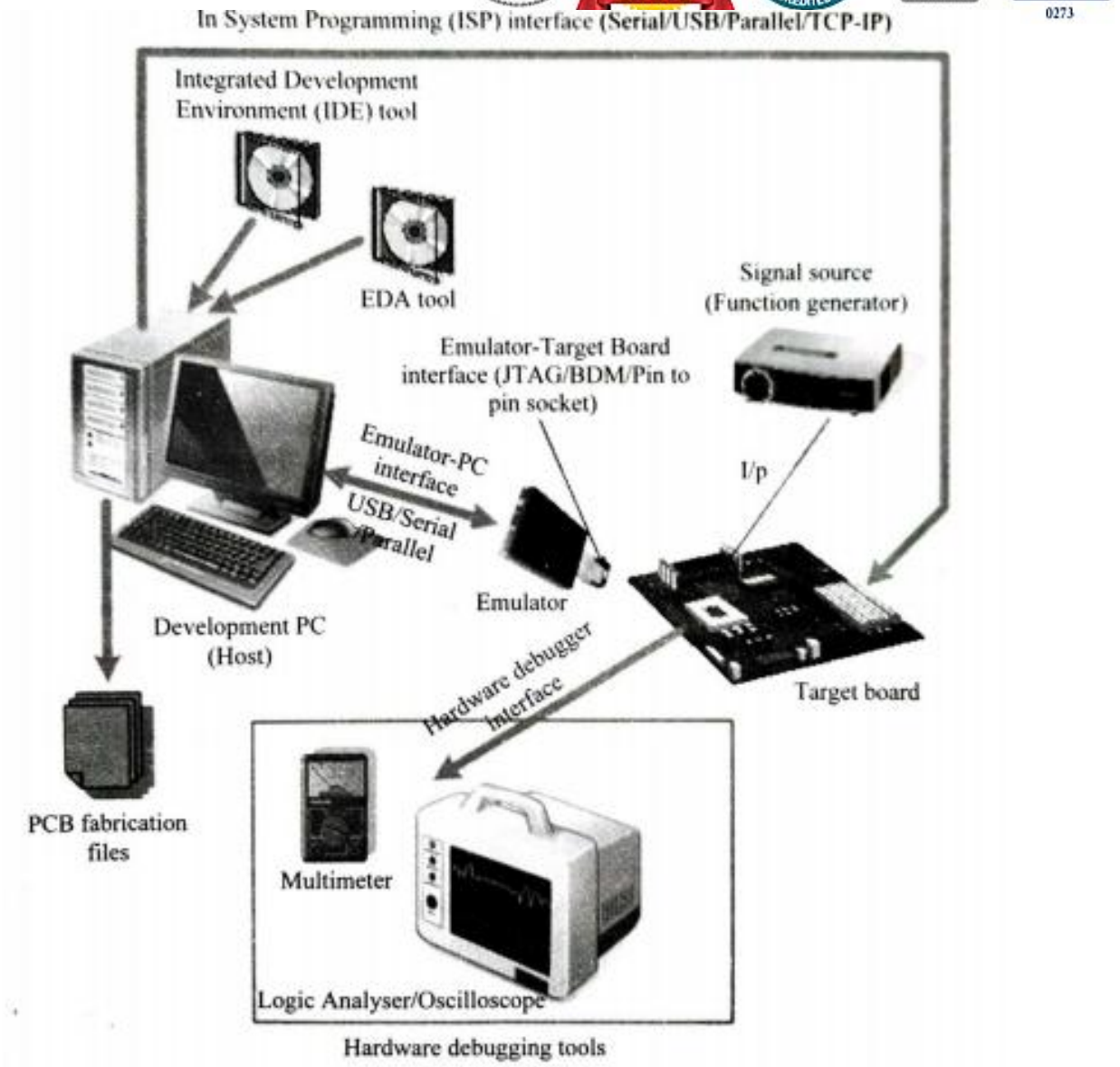
Embedded System Development Environment

The embedded system development environment consists of:

- A Development Computer (PC) or Host, which acts as the heart of the development environment
- Integrated Development Environment (IDE) Tool for embedded firmware development and debugging
- Electronic Design Automation (EDA) Tool for Embedded Hardware design
- An emulator hardware for debugging the target board
- Signal sources (like Function generator) for simulating the inputs to the target board,
- Target hardware debugging tools (Digital CRO, Multimeter, Logic Analyser, etc.)
- The target hardware.

ATME
College of Engineering

Embedded System Development Environment



Integrated Development Environment (IDE)

In embedded system development context, Integrated Development Environment (IDE) stands for an integrated environment for developing and debugging the target processor specific embedded firmware.

IDE is a software package which bundles a

- Text Editor (Source Code Editor)
- Cross-compiler (for cross platform development and compiler for same platform development)
- Linker
- Debugger

Integrated Development Environment (IDE)

- IDEs used in embedded firmware development are slightly different from the generic IDEs used for high level language based development for desktop applications.
- In embedded applications, the IDE is either supplied by the target processor/controller manufacturer or by third party vendors or as Open Source.
- Keil µVision from Keil software is an example for a third party IDE, which is used for developing embedded firmware for 8051 family microcontrollers and also ARM microcontrollers.
- MPLAB is an IDE tool supplied by microchip for developing embedded firmware using their PIC family of microcontrollers.
- CodeWarrior by Metrowerks is an example of IDE for ARM family of processors

Integrated Development Environment (IDE)

Disassembler/Decompiler

- Disassembler is a utility program which converts machine codes into target processor specific Assembly codes/instructions.
- The process of converting machine codes into Assembly code is known as 'Disassembling'.
- In operation, disassembling is complementary to assembling/cross-assembling.
- Decompiler is the utility program for translating machine codes into corresponding high level language instructions.
- Decompiler performs the reverse operation of compiler/cross-compiler.
- The disassemblers/decompilers for different family of processors/controllers are different.

Integrated Development Environment (IDE)

Simulators

- Simulator is a software tool used for simulating the various conditions for checking the functionality of the application firmware.
- The Integrated Development Environment (IDE) itself will be providing simulator support and they help in debugging the firmware for checking its required functionality.
- Simulators simulate the target hardware and the firmware execution can be inspected using simulators.

Integrated Development Environment (IDE)

Simulators

The features of simulator based debugging are:

- Purely software based
- Doesn't require a real target system
- Very primitive (Lack of featured I/O support. Everything is a simulated one)
- Lack of Real-time behavior