# 10 Real-Time Operating System (RTOS) based Embedded System Design

## LEARNING OBJECTIVES

- ✓ *Learn the basics of an operating system and the need for an operating system*
- ✓ *Learn the internals of Real-Time Operating System and the fundamentals of RTOS based embedded firmware design*
- ✓ *Learn the basic kernel services of an operating system*
- ✓ *Learn about the classification of operating systems*
- ✓ *Learn about the different real-time kernels and the features that make a kernel Real-Time*
- ✓ *Learn about tasks, processes and threads in the operating system context*
- ✓ *Learn about the structure of a process, the different states of a process, process life cycle and process management*
- ✓ *Learn the concept of multithreading, thread standards and thread scheduling*
- ✓ *Understand the difference between multiprocessing and multitasking,*
- ✓ *Learn about the different types of multitasking (Co-operative, Preemptive and Non-preemptive)*
- ✓ *Learn about the FCFS/FIFO, LCFS/LIFO, SJF and priority based task/process scheduling*
- ✓ *Learn about the shortest remaining time (SRT), Round Robin and priority based preemptive task/process scheduling*
- ✓ *Learn about the different Inter Process Communication (IPC) mechanisms used by tasks/process to communicate and co-operate each other in a multitasking environment*
- ✓ *Learn the different types of shared memory techniques (Pipes, memory mapped object, etc.) for IPC*
- ✓ *Learn the different types of message passing techniques (Message queue, mailbox, signals, etc.) for IPC*
- ✓ *Learn the RPC based Inter Process Communication*
- ✓ *Learn the need for task synchronisation in a multitasking environment*
- ✓ *Learn the different issues related to the accessing of a shared resource by multiple processes concurrently*
- ✓ *Learn about 'Racing', 'Starvation', 'Livelock', 'Deadlock', 'Dining Philosopher's Problem', 'Producer-Consumer/Bounded Buffer Problem', 'Readers-Writers Problem' and 'Priority Inversion'*
- ✓ *Learn about the 'Priority Inheritance' and 'Priority Ceiling' based Priority avoidance mechanisms*
- ✓ *Learn the need for task synchronisation and the different mechanisms for task synchronisation in a multitasking environment*
- ✓ *Learn about mutual exclusion and the different policies for mutual exclusion implementation*
- ✓ *Learn about semaphores, different types of semaphores, mutex, critical section objects and events for task synchronisation*
- ✓ *Learn about device drivers, their role in an operating system based embedded system design, the structure of a device driver, and interrupt handling inside device drivers*
- ✓ *Understand the different functional and non-functional requirements that need to be addressed in the selection of a Real-Time Operating System*

In the previous chapter, we discussed about the *Super loop* based task execution model for firmware execution. The super loop executes the tasks sequentially in the order in which the tasks are listed within the loop. Here every task is repeated at regular intervals and the task execution is non-real time. As the number of task increases, the time intervals at which a task gets serviced also increases. If some of the tasks involve waiting for external events or I/O device usage, the task execution time also gets pushed off in accordance with the 'wait' time consumed by the task. The priority in which a task is to be executed is fixed and is determined by the task placement within the loop, in a super loop based execution. This type of firmware execution is suited for embedded devices where response time for a task is not time critical. Typical examples are electronic toys and video gaming devices. Here any response delay is acceptable and it will not create any operational issues or potential hazards. Whereas certain applications demand time critical response to tasks/events and any delay in the response may become catastrophic. Flight Control systems, Air bag control and Anti Locking Brake (ABS) systems for vehicles, Nuclear monitoring devices, etc. are typical examples of applications/devices demanding time critical task response.

How the increasing need for time critical response for tasks/events is addressed in embedded applications? Well the answer is

1. Assign priority to tasks and execute the high priority task when the task is ready to execute.
2. Dynamically change the priorities of tasks if required on a need basis.
3. Schedule the execution of tasks based on the priorities.
4. Switch the execution of task when a task is waiting for an external event or a system resource including I/O device operation.

The introduction of operating system based firmware execution in embedded devices can address these needs to a greater extent.



## 10.1 OPERATING SYSTEM BASICS

The operating system acts as a bridge between the user applications/tasks and the underlying system resources through a set of system functionalities and services. The OS manages the system resources and makes them available to the user applications/tasks on a need basis. A normal computing system is a collection of different I/O subsystems, working, and storage memory. The primary functions of an operating system is

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

Figure 10.1 gives an insight into the basic components of an operating system and their interfaces with rest of the world.

### 10.1.1 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. 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 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

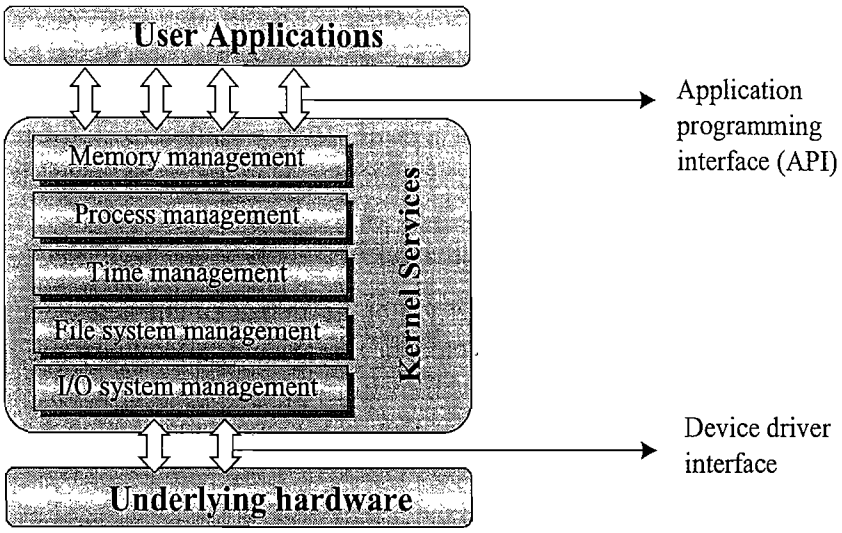


Fig. 10.1 **The Operating System Architecture**

up and managing the Process Control Block (PCB), Inter Process Communication and synchronisation, process termination/deletion, etc. We will look into the description of process and process management in a later section of this chapter.

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

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

***File System Management*** File is a collection of related information. A file could be a program (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. The file system management service of Kernel is responsible for

- The creation, deletion and alteration of files
- Creation, deletion and alteration of directories
- Saving of files in the secondary storage memory (e.g. Hard disk storage)
- Providing automatic allocation of file space based on the amount of free space available
- Providing a flexible naming convention for the files

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.

***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. This list may be available in advance, at the time of building the kernel. Some kernels, dynamically updates the list of available devices as and when a new device is installed

(e.g. Windows XP kernel keeps the list updated when a new plug '*n*' play USB device is attached to the system). The service 'Device Manager' (Name may vary across different OS kernels) of the kernel is responsible for handling all I/O device related operations. The kernel talks to the I/O device through a set of low-level systems calls, which are implemented in a service, called device drivers. The device drivers are specific to a device or a class of devices. The Device Manager is responsible for

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

***Secondary Storage Management*** The secondary storage management deals with managing the secondary storage memory devices, 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

***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. Windows XP with user permissions like 'Administrator', 'Standard', 'Restricted', etc.). 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/ 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.

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

These are some of the important services offered by the kernel of an operating system. It does not mean that a kernel contains no more than components/services explained above. Depending on the type of the operating system, a kernel may contain lesser number of components/services or more number of components/services. In addition to the components/services listed above, many operating systems offer a number of add-on system components/services to the kernel. Network communication, network management, user-interface graphics, timer services (delays, timeouts, etc.), error handler, database management, etc. are examples for such components/services. Kernel exposes the interface to the various kernel applications/services, hosted by kernel, to the user applications through a set of standard Application Programming Interfaces (APIs). User applications can avail these API calls to access the various kernel application/services.

*10.1.1.1 Kernel Space and User Space* As we discussed in the earlier section, the applications/ services are classified into two categories, namely: user applications and kernel applications. The program code corresponding to the kernel applications/services are kept in a contiguous area (OS dependent) of primary (working) memory and is protected from the un-authorised access by user programs/applications. The memory space at which the kernel code is located is known as '*Kernel Space*'. Similarly, all user applications are loaded to a specific area of primary memory and this memory area is referred as '*User Space*'. User space is the memory area where user applications are loaded and executed. The partitioning of memory into kernel and user space is purely Operating System dependent.

Some OS implements 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; Meaning, the entire code for the user application need not be loaded to the main (primary) memory at once; instead the user application code is split into different pages and these pages are loaded into and out of the main memory area on a need basis. The act of loading the code into and out of the main memory is termed as '*Swapping*'. Swapping happens between the main (primary) memory and secondary storage memory. Each process run in its own virtual memory space and are not allowed accessing the memory space corresponding to another processes, unless explicitly requested by the process. Each process will have certain privilege levels on accessing the memory of other processes and based on the privilege settings, processes can request kernel to map another process's memory to its own or share through some other mechanism. Most of the operating systems keep the kernel application code in main memory and it is not swapped out into the secondary memory.

*10.1.1.2 Monolithic Kernel and Microkernel* As we know, 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*' and '*Micro*'.

***Monolithic Kernel*** In monolithic kernel architecture, all kernel services run in the kernel space. Here all kernel modules run within the same memory space under a single kernel thread. The tight internal integration of kernel modules in monolithic kernel architecture allows the effective utilisation of the low-level features of the underlying system. The major drawback of monolithic kernel is that any error or failure in any one of the kernel modules leads to the crashing of the entire kernel application. LINUX, SOLARIS, MS-DOS kernels are examples of monolithic kernel. The architecture representation of a monolithic kernel is given in Fig. 10.2.



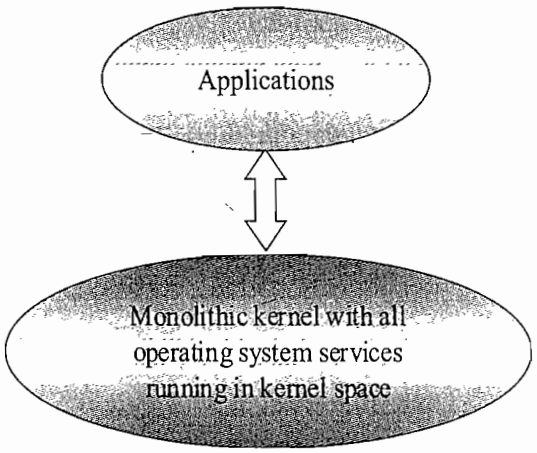


Fig. 10.2 **The Monolithic Kernel Model**

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

are the essential services, which forms the part of the microkernel. Mach, QNX, Minix 3 kernels are examples for microkernel. The architecture representation of a microkernel is shown in Fig. 10.3.

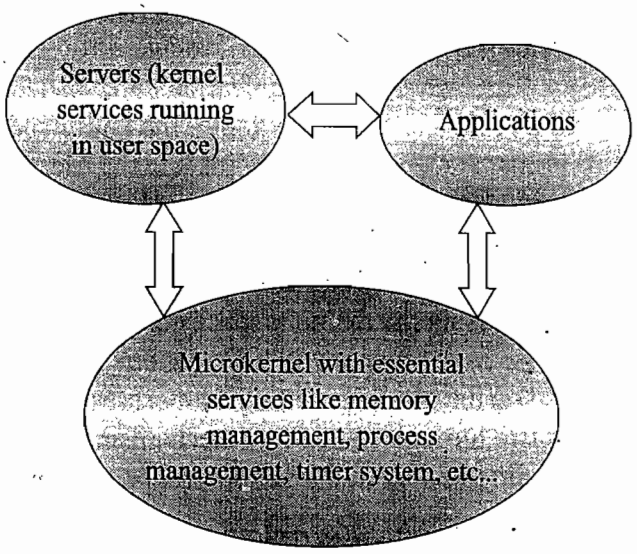


Fig. 10.3 **The Microkernel model**

Microkernel based design approach offers the following benefits

- **Robustness:** If a problem is encountered in any of the services, which runs as '*Server*' application, the same can be reconfigured and re-started without the need for re-starting the entire OS. Thus, this approach is highly useful for systems, which demands high '*availability*'. Refer Chapter 3 to get an understanding of '*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 services, which run as '*Server*' application can be changed without the need to restart the whole system. This makes the system dynamically configurable.

## 10.2 TYPES OF OPERATING SYSTEMS

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

### 10.2.1 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 an OS is more generalised and it contains all kinds of services required for executing generic applications. General-purpose operating systems are often quite non-deterministic in behaviour. 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 behaviour 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.

### 10.2.2 Real-Time Operating System (RTOS)

There is no universal definition available for the term '*Real-Time*' when it is used in conjunction with operating systems. What '*Real-Time*' means in Operating System context is still a debatable topic and there are many definitions available. In a broad sense, '*Real-Time*' implies deterministic timing behaviour. Deterministic timing behaviour 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. 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).

*10.2.2.1 The Real-Time Kernel* The kernel of a Real-Time Operating System is referred as Real. Time kernel. In complement to the conventional OS kernel, the Real-Time kernel is highly specialised 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 listed below:

- Task/Process management
- Task/Process scheduling
- Task/Process synchronisation
- Error/Exception handling
- Memory management
- Interrupt handling
- Time 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, 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 (e.g. 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 management service utilises the TCB of a task in the following way

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

***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 optimal scheduling of tasks to provide a deterministic behaviour. We will discuss the various types of scheduling in a later section of this chapter.

***Task/Process Synchronisation*** Deals with synchronising the concurrent access of a resource, which is shared across multiple tasks and the communication between various tasks. We will discuss the various synchronisation techniques and inter task /process communication in a later section of this chapter.

***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). *GetLastError()* API provided by Windows CE RTOS is an example for such a system call. Watchdog timer is a mechanism for handling the timeouts for tasks. Certain tasks may involve the waiting of external events from devices. These tasks will wait infinitely when the external device is not responding and the task will generate a hang-up behaviour. In order to avoid these types of scenarios, a proper timeout mechanism should be implemented. A watchdog is normally used in such situations. The watchdog will be loaded with the maximum expected wait time for the event and if the event is not triggered within this wait time, the same is informed to the task and the task is timed out. If the event happens before the timeout, the watchdog is resetted.

***Memory Management*** Compared to the General Purpose Operating Systems, the memory management function of an RTOS kernel is slightly different. In general, the memory allocation time increases depending on the size of the block of memory needs to be allocated and the state of the allocated memory block (initialised memory block consumes more allocation time than un-initialised memory block). Since predictable timing and deterministic behaviour are the primary focus of an RTOS, RTOS achieves this by compromising the effectiveness of memory allocation. 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 and the kernel enters a *fail-safe* mode when an illegal memory access occurs.

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 memory allocation can be implemented as constant functions and thereby it consumes fixed amount of time for memory allocation. This leaves the deterministic behaviour of the RTOS kernel untouched. The '*block*' memory concept avoids the garbage collection overhead also. (We will explore this technique under the MicroC/OS-II kernel in a

† *Virtual Memory* is an imaginary memory supported by certain operating systems. Virtual memory expands the address space available to a task beyond the actual physical memory (RAM) supported by the system. Virtual memory is implemented with the help of a Memory Management Unit (MMU) and 'memory paging'. The program memory for a task can be viewed as different pages and the page corresponding to a piece of code that needs to be executed is loaded into the main physical memory (RAM). When a memory page is no longer required, it is moved out to secondary storage memory and another page which contains the code snippet to be executed is loaded into the main memory. This memory movement technique is known as demand paging. The MMU handles the demand paging and converts the virtual address of a location in a page to corresponding physical address in the RAM.

latter chapter).The '*block*' based memory allocation achieves deterministic behaviour with the trade-of limited choice of memory chunk size and suboptimal memory usage.

***Interrupt Handling*** Deals with the handling of various types of interrupts. Interrupts provide Real-Time behaviour 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*. 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. Asynchronous interrupts are interrupts, which occurs at any point of execution of any task, and are not in sync with the currently executing task. The interrupts generated by external devices (by asserting the interrupt line of the processor/controller to which the interrupt line of the device is connected) connected to the processor/controller, timer overflow interrupts, serial data reception/ transmission interrupts, etc. are examples for asynchronous interrupts. 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. Priority levels can be assigned to the interrupts and each interrupts 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.

***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

$$2^{32} * 10^{-6} / (24 * 60 * 60) = 49700 \text{ Days} = \sim 0.0497 \text{ Days} = 1.19 \text{ Hours}$$

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

$$2^{32} * 10^{-3} / (24 * 60 * 60) = 497 \text{ Days} = 49.7 \text{ Days} = \sim 50 \text{ Days}$$

The '*Timer tick*' interrupt is handled by the 'Timer Interrupt' handler of kernel. The '*Timer tick*' interrupt can be utilised for implementing the following actions.

- Save the current context (Context of the currently executing task).
- Increment the System time register by one. Generate timing error and reset 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.
- 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.

Apart from these basic functions, some RTOS provide other functionalities also (Examples are file management and network functions). Some RTOS kernel provides options for selecting the required kernel functions at the time of building a kernel. The user can pick the required functions from the set of available functions and compile the same to generate the kernel binary. Windows CE is a typical example for such an RTOS. While building the target, the user can select the required components for the kernel.

*10.2.2.2 Hard Real-Time* Real-Time Operating Systems that strictly adhere 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 emphasise the principle '*A late answer is a wrong answer*'. A system can have several such tasks and the key to their correct operation lies in scheduling them so that they meet their time constraints. Air bag control systems and Anti-lock Brake Systems (ABS) of vehicles are typical examples for Hard Real-Time Systems. The Air bag control system should be into action and deploy the air bags when the vehicle meets a severe accident. Ideally speaking, the time for triggering the air bag deployment task, when an accident is sensed by the Air bag control system, should be zero and the air bags should be deployed exactly within the time frame, which is predefined for the air bag deployment task. Any delay in the deployment of the air bags makes the life of the passengers under threat. When the air bag deployment task is triggered, the currently executing task must be pre-empted, the air bag deployment task should be brought into execution, and the necessary I/O systems should be made readily available for the air bag deployment task. To meet the strict deadline, the time between the air bag deployment event triggering and start of the air bag deployment task execution should be minimum, ideally zero. As a rule of thumb, 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. In general, the presence of *Human in the loop* (*HITL*) for tasks introduces unexpected delays in the task execution. Most of the Hard Real-Time Systems are automatic and does not contain a 'human in the loop'.

*10.2.2.3 Soft Real-Time* Real-Time Operating System that does not guarantee meeting deadlines, but offer the best effort to meet the deadline are referred as '*Soft Real-Time*' systems. Missing deadlines for tasks are acceptable for a Soft 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 emphasises the principle '*A late answer is an acceptable answer, but it could have done bit faster*'. Soft Real-Time systems most often have a '*human in the loop (HITL)*'. Automatic Teller Machine (ATM) is a typical example for Soft-Real-Time System. 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.

## 10.3 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

3. Register an Interrupt Service Routine (ISR) with an Interrupt Request (IRQ). ISR is the handler for an Interrupt. In order to service an interrupt, an ISR should be associated with an IRQ. Registering an ISR with an IRQ takes care of it.

With these the interrupt configuration is complete. If an interrupt occurs, depending on its priority, it is serviced and the corresponding ISR is invoked. The processing part of an interrupt is handled in an ISR. The whole interrupt processing can be done by the ISR itself or by invoking an Interrupt Service Thread (IST). The IST performs interrupt processing on behalf of the ISR. To make the ISR compact and short, it is always advised to use an IST for interrupt processing. The intention of an interrupt is to send or receive command or data to and from the hardware device and make the received data available to user programs for application specific processing. Since interrupt processing happens at kernel level, user applications may not have direct access to the drivers to pass and receive data. Hence it is the responsibility of the Interrupt processing routine or thread to inform the user applications that an interrupt is occurred and data is available for further processing. The client interfacing part of the device driver takes care of this. The client interfacing implementation makes use of the Inter Process communication mechanisms supported by the embedded OS for communicating and synchronising with user applications and drivers. For example, to inform a user application that an interrupt is occurred and the data received from the device is placed in a shared buffer, the client interfacing code can signal (or set) an event. The user application creates the event, registers it and waits for the driver to signal it. The driver can share the received data through shared memory techniques. IOCTLs, shared buffers, etc. can be used for data sharing. The story line is incomplete without performing an interrupt done (Interrupt processing completed) functionality in the driver. Whenever an interrupt is asserted, while vectoring to its corresponding ISR, all interrupts of equal and low priorities are disabled. They are re-enable only on executing the interrupt done function (Same as the Return from Interrupt RETI instruction execution for *8051*) by the driver. The interrupt done function can be invoked at the end of corresponding ISR or IST.



We will discuss more about device driver development in a dedicated book coming under this book-series.

## 10.10 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 analysed carefully before making a decision on the selection of an RTOS. These factors can be either functional or non-functional. The following section gives a brief introduction to the important functional and non-functional requirements that needs to be analysed in the selection of an RTOS for an embedded design.

### 10.10.1 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.

***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 behaviour. The task/process

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

***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.

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

***Modularisation 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 modularisation where in which the developer can choose the essential modules and re-compile the OS image for functioning. Windows CE is an example for a highly modular operating system.

***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) customised for the Operating System is essential for running java applications. Similarly the .NET Compact Framework (.NETCF) is required for running Microsoft® .NET applications on top of the Operating System. 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.

## 10.10.2 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 customising 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.

***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.

***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.

## Summary

- ✓ The *Operating System* is responsible for making the system convenient to use, organise and manage system resources efficiently and properly.
- ✓ Process/Task management, Primary memory management, File system management, I/O system (Device) management, Secondary Storage Management, protection implementation, Time management, Interrupt handling, etc. are the important services handled by the OS kernel.
- ✓ The core of the operating system is known as *kernel*. Depending on the implementation of the different kernel services, the kernel is classified as *Monolithic* and *Micro*. *User Space* is the memory area in which user applications are confined to run, whereas *kernel space* is the memory area reserved for kernel applications.
- ✓ Operating systems with a generalised kernel are known as *General Purpose Operating Systems* (*GPOS*), whereas operating systems with a specialised kernel with deterministic timing behaviour are known as *Real-Time Operating Systems* (*RTOS*).
- ✓ In the operating system context a task/process is a program, or part of it, in execution. The process holds a set of registers, process status, a Program Counter (PC) to point to the next executable instruction of the process, a stack for holding the local variables associated with the process and the code corresponding to the process.
- ✓ The different states through which a process traverses through during its journey from the newly created state to finished state is known as *Process Life Cycle*
- ✓ Process management deals with the creation of a process, setting up the memory space for the process, loading the process's code into the memory space, allocating system resources, setting up a Process Control Block (PCB) for the process and process termination/deletion.
- ✓ A thread is the primitive that can execute code. It is a single sequential flow of control within a process. A process may contain multiple threads. The act of concurrent execution of multiple threads under an operating system is known as *multithreading*.
- ✓ Thread standards are the different standards available for thread creation and management. POSIX, Win32, Java, etc. are the commonly used thread creation and management libraries.
- ✓ The ability of a system to execute multiple processes simultaneously is known as *multiprocessing*, whereas the ability of an operating system to hold multiple processes in memory and switch the processor (CPU) from executing one process to another process is known as *multitasking*. Multitasking involves *Context Switching*, *Context Saving* and *Context Retrieval*.
- ✓ *Co-operative* multitasking, *Preemptive* multitasking and *Non-preemptive* multitasking are the three important types of multitasking which exist in the Operating system context.
- ✓ CPU utilisation, Throughput, Turn Around Time (TAT), Waiting Time and Response Time are the important criterions that need to be considered for the selection of a scheduling algorithm for task scheduling.
- ✓ Job queue, Ready queue and Device queue are the important queues maintained by an operating system in association with CPU scheduling.
- ✓ First Come First Served (FCFS)/First in First Out (FIFO), Last Come First Served (LCFS)/Last in First Out (LIFO), Shortest Job First (SJF), priority based scheduling, etc. are examples for Non-preemptive scheduling, whereas Preemptive SJF Scheduling/Shortest Remaining Time (SRT), Round Robin (RR) scheduling and priority based scheduling are examples for preemptive scheduling.
- ✓ Processes in a multitasking system falls into either *Co-operating* or *Competing*. The co-operating processes share data for communicating among the processes through *Inter Process Communication* (IPC), whereas competing

processes do not share anything among themselves but they share the system resources like display devices, keyboard, etc.

- ✓ *Shared memory*, *message passing* and *Remote Procedure Calls* (RPC) are the important IPC mechanisms through which the co-operating processes communicate in an operating system environment. The implementation of the IPC mechanism is OS kernel dependent.
- ✓ Racing, deadlock, livelock, starvation, producer-consumer problem, Readers-Writers problem and priority inversion are some of the problems involved in shared resource access in task communication through sharing.
- ✓ The 'Dining Philosophers' 'Problem' is a real-life representation of the deadlock, starvation, livelock and 'Racing' issues in shared resource access in operating system context.
- ✓ *Priority inversion* is the condition in which a medium priority task gets the CPU for execution, when a high priority task needs to wait for a low priority task to release a resource which is shared between the high priority task and the low priority task.
- ✓ *Priority inheritance* and *Priority ceiling* are the two mechanisms for avoiding *Priority Inversion* in a multitasking environment.
- ✓ The act of preventing the access of a shared resource by a task/process when it is currently being held by another task/process is known as *mutual exclusion*. Mutual exclusion can be implemented through either *busy waiting* (*spin lock*) or *sleep and wakeup* technique.
- ✓ *Test and Set*, *Flags*, etc. are examples of *Busy waiting* based *mutual exclusion* implementation, whereas *Semaphores*, *mutex*, *Critical Section Objects* and *events* are examples for *Sleep and Wakeup* based mutual exclusion.
- ✓ *Binary semaphore* implements exclusive shared resource access, whereas *counting semaphore* limits the concurrent access to a shared resource, and *mutual exclusion semaphore* prevents priority inversion in shared resource access.
- ✓ *Device driver* is a piece of software that acts as a bridge between the operating system and the hardware. Device drivers are responsible for initiating and managing the communication with the hardware peripherals.
- ✓ Various functional and non-functional requirements need to be evaluated before the selection of an RTOS for an embedded design.

## Keywords

| | | |
|---|---|---|
| **Operating System** | : | A piece of software designed to manage and allocate system resources and execute other pieces of the software |
| **Kernel** | : | The core of the operating system which is responsible for managing the system resources and the communication among the hardware and other system services |
| **Kernel space** | : | The primary memory area where the kernel applications are confined to run |
| **User space** | : | The primary memory area where the user applications are confined to run |
| **Monolithic kernel** | : | A kernel with all kernel services run in the kernel space under a single kernel thread |
| **Microkernel** | : | A kernel which incorporates only the essential services within the kernel space and the rest is installed as loadable modules called *servers* |
| **Real-Time Operating System (RTOS)** | : | Operating system with a specialised kernel with a deterministic timing behaviour |
| **Scheduler** | : | OS kernel service which deals with the scheduling of processes/tasks |
| **Hard Real-Time** | : | Real-time operating systems that strictly adhere to the timing constraints for a task |
| **Soft Real-Time** | : | Real-time operating systems that does not guarantee meeting deadlines, but, offer the best effort to meet the deadline |
| **Task/Job/Process** | : | In the operating system context a task/process is a program, or part of it, in execution |

| | | |
|---|---|---|
| **Process Life Cycle** | : | The different *states* through which a process traverses through during its journey from the newly created state to completed state |
| **Thread** | : | The primitive that can execute code. It is a single sequential flow of control within a process |
| **Multiprocessing systems** | : | Systems which contain multiple CPUs and are capable of executing multiple processes simultaneously |
| **Multitasking** | : | The ability of an operating system to hold multiple processes in memory and switch the processor (CPU) from executing one process to another process |
| **Context switching** | : | The act of switching CPU among the processes and changing the current execution context |
| **Co-operative multitasking** | : | Multitasking model in which a task/process gets a chance when the currently executing task relinquishes the CPU voluntarily |
| **Preemptive multitasking** | : | Multitasking model in which a currently running task/process is preempted to execute another task/process |
| **Non-preemptive multitasking** | : | Multitasking model in which a task gets a chance to execute when the currently executing task relinquishes the CPU or when it enters a wait state |
| **First Come First Served (FCFS)/First in First Out (FIFO)** | : | Scheduling policy which sorts the *Ready Queue* with FCFS model and schedules the first arrived process from the *Ready queue* for execution |
| **Last Come First Served (LCFS)/Last in.First Out (LIFO)** | : | Scheduling policy which sorts the *Ready Queue* with LCFS model and schedules the last arrived process from the *Ready queue* for execution |
| **Shortest Job First (SJF)** | : | Scheduling policy which sorts the *Ready queue* with the order of the shortest execution time for process and schedules the process with least estimated execution completion time from the *Ready queue* for execution |
| **Priority based Scheduling** | : | Scheduling policy which sorts the *Ready queue* based on priority and schedules the process with highest priority from the *Ready queue* for execution |
| **Shortest Remaining Time (SRT)** | : | Preemptive scheduling policy which sorts the *Ready queue* with the order of the shortest remaining time for execution completion for process and schedules the process with the least remaining time for estimated execution completion from the *Ready queue* for execution |
| **Round Robin** | : | Preemptive scheduling policy in which the currently running process is preempted based on time slice |
| **Co-operating processes** | : | Processes which share data for communicating among them |
| **Inter Process/Task Communication (IPC)** | : | Mechanism for communicating between co-operating processes of a system |
| **Shared memory** | : | A memory sharing mechanism used for inter process communication |
| **Message passing** | : | IPC mechanism based on exchanging of messages between processes through a message queue or mailbox |
| **Message queue** | : | A queue for holding messages for exchanging between processes of a multitasking system |
| **Mailbox** | : | A special implementation of message queue under certain OS kernel, which supports only a single message |
| **Signal** | : | A form of asynchronous message notification |
| **Remote Procedure Call or RPC** | : | The IPC mechanism used by a process to invoke a procedure of another process running on the same CPU or on a different CPU which is interconnected in a network |
| **Racing** | : | The situation in which multiple processes compete (race) each other to access and manipulate shared data concurrently |

| | | |
|---|---|---|
| **Deadlock** | : | A situation where none of the processes are able to make any progress in their execution. Deadlock is the condition in which a process is waiting for a resource held by another process which is waiting for a resource held by the first process |
| **Livelock** | : | A condition where a process always does something but is unable to make any progress in the execution completion |
| **Starvation** | : | The condition in which a process does not get the CPU or system resources required to continue its execution for a long time |
| **Dining Philosophers' Problem** | : | A real-life representation of the *deadlock*, *starvation*, *livelock* and *racing* issues in shared resource access in operating system context |
| **Producer-Consumer problem** | : | A common data sharing problem where two processes concurrently access a shared buffer with fixed size |
| **Readers-Writers problem** | : | A data sharing problem characterised by multiple processes trying to read and write shared data concurrently |
| **Priority inversion** | : | The condition in which a medium priority task gets the CPU for execution, when a high priority task needs to wait for a low priority task to release a resource which is shared between the high priority task and the low priority task |
| **Priority inheritance** | : | A mechanism by which the priority of a low-priority task which is currently holding a resource requested by a high priority task, is raised to that of the high priority task to avoid priority inversion |
| **Priority Ceiling** | : | The mechanism in which a priority is associated with a shared resource (The priority of the highest priority task which uses the shared resource) and the priority of the task is temporarily boosted to the priority of the shared resource when the resource is being held by the task, for avoiding priority inversion |
| **Task/Process synchronisation** | : | The act of synchronising the access of shared resources by multiple processes and enforcing proper sequence of operation among multiple processes of a multitasking system |
| **Mutual Exclusion** | : | The act of preventing the access of a shared resource by a task/process when it is being held by another task/process |
| **Semaphore** | : | A system resource for implementing mutual exclusion in shared resource access or for restricting the access to the shared resource |
| **Mutex** | : | The *binary semaphore* implementation for exclusive resource access under certain OS kernel |
| **Device driver** | : | A piece of software that acts as a bridge between the operating system and the hardware |

## Objective Questions

### Operating System Basics

1. Which of the following is true about a kernel?
   (a) The kernel is the core of the operating system
   (b) It is responsible for managing the system resources and the communication among the hardware and other system services
   (c) It acts as the abstraction layer between system resources and user applications.
   (d) It contains a set of system libraries and services
   (e) All of these
2. The user application and kernel interface is provided through
   (a) System calls (b) Shared memory (d) None of these

3. The process management service of the kernel is responsible for
   (a) Setting up the memory space for the process
   (b) Allocating system resources
   (c) Scheduling and managing the execution of the process
   (d) Setting up and managing the Process Control Block (PCB), inter-process communication and synchronisation
   (e) All of these
4. The Memory Management Unit (MMU) of the kernel is responsible for
   (a) Keeping track of which part of the memory area is currently used by which process
   (b) Allocating and de-allocating memory space on a need basis (Dynamic memory allocation)
   (c) Handling all virtual memory operations in a kernel with virtual memory support
   (d) All of these
5. The memory area which holds the program code corresponding to the core OS applications/services is known as
   (a) User space (b) Kernel space (c) Shared memory (d) All of these
6. Which of the following is true about *Privilege separation*?
   (a) The user applications/processes runs at user space and kernel applications run at kernel space
   (b) Each user application/process runs on its own virtual memory space
   (c) A process is not allowed to access the memory space of another process directly
   (d) All of these
7. Which of the following is true about monolithic kernel?
   (a) All kernel services run in the kernel space under a single kernel thread.
   (b) The tight internal integration of kernel modules in monolithic kernel architecture allows the effective utilisation of the low-level features of the underlying system
   (c) Error prone. Any error or failure in any one of the kernel modules may lead to the crashing of the entire kernel
   (d) All of these
8. Which of the following is true about microkernel?
   (a) 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.
   (b) Highly modular and OS neutral
   (c) Less Error prone. Any 'Server' where error occurs can be restarted without restarting the entire kernel
   (d) All of these

**Real-Time Operating System (RTOS)**

1. Which of the following is true for Real-Time Operating Systems (RTOSes)?
   (a) Possess specialised kernel (b) Deterministic in behaviour
   (c) Predictable performance (d) All of these
2. Which of the following is (are) example(s) for RTOS?
   (a) Windows CE (b) Windows XP (c) Windows 2000 (d) QNX
   (e) (a) and (d)
3. Interrupts which occur in sync with the currently executing task are known as
   (a) Asynchronous interrupts (b) Synchronous interrupts
   (c) External interrupts (d) None of these
4. Which of the following is an example of a synchronous interrupt?
   (a) TRAP (b) External interrupt (c) Divide by zero (d) Timer interrupt
5. Which of the following is true about 'Timer tick' for RTOS?
   (a) The high resolution hardware timer interrupt is referred as 'Timer tick'
   (b) The 'Timer tick' is taken as the timing reference by the kernel
   (c) The time parameters for tasks are expressed as the multiples of the 'Timer tick'
   (d) All of these

6. Which of the following is true about hard real-time systems?
   (a) Strictly adhere to the timing constraints for a task
   (b) Missing any deadline may produce catastrophic results
   (c) Most of the hard real-time systems are automatic and may not contain a human in the loop
   (d) May not implement virtual memory based memory management
   (e) All of these.
7. Which of the following is true about soft real-time systems?
   (a) Does not guarantee meeting deadlines, but offer the best effort to meet the deadline are referred
   (b) Missing deadlines for tasks are acceptable
   (c) Most of the soft real-time systems contain a human in the loop
   (d) All of these

**Tasks, Process and Threads**

1. Which of the following is true about *Process* in the operating system context?
   (a) A '*Process*' is a program, or part of it, in execution
   (b) It can be an instance of a program in execution
   (c) A process requires various system resources like CPU for executing the process, memory for storing the code corresponding to the process and associated variables, I/O devices for information exchange, etc.
   (d) A process is sequential in execution
   (e) All of these
2. A process has
   (a) Stack memory (b) Program memory (c) Working Registers (d) Data memory
   (e) All of these
3. The 'Stack' memory of a process holds all temporary data such as variables local to the process. State 'True' or 'False'
   (a) True (b) False
4. The data memory of a process holds
   (a) Local variables (b) Global variables (c) Program instructions (d) None of these
5. A process has its own memory space, when residing at the main memory. State 'True' or 'False'
   (a) True (b) False
6. A process when loaded to the memory is allocated a virtual memory space in the range 0x08000 to 0x08FF8. What is the content of the Stack pointer of the process when it is created?
   (a) 0x07FFF (b) 0x08000 (c) 0x08FF7 (d) 0x08FF8
7. What is the content of the program counter for the above example when the process is loaded for the first time?
   (a) 0x07FFF (b) 0x08000 (c) 0x08FF7 (d) 0x08FF8
8. The state where a process is incepted into the memory and awaiting the processor time for execution, is known as
   (a) Created state (b) Blocked state (c) Ready state (d) Waiting state
   (e) Completed state
9. The CPU allocation for a process may change when it changes its state from _____?
   (a) 'Running' to 'Ready' (b) 'Ready' to 'Running'
   (c) 'Running' to 'Blocked' (d) 'Running' to 'Completed'
   (e) All of these
10. Which of the following is true about threads?
    (a) A thread is the primitive that can execute code
    (b) A thread is a single sequential flow of control within a process
    (c) '*Thread*' is also known as lightweight process
    (d) All of these (e) None of these
11. A process can have many threads of execution. State 'True' or 'False'
    (a) True (b) False

12. Different threads, which are part of a process, share the same address space. State 'True' or 'False'
    (a) True (b) False
13. Multiple threads of the same process share _____?
    (a) Data memory (b) Code memory
    (c) Stack memory (d) All of these
    (e) only (a) and (b)
14. Which of the following is true about multithreading?
    (a) Better memory utilisation
    (b) Better CPU utilisation
    (c) Reduced complexity in inter-thread communication
    (d) Faster process execution (e) All of these
15. Which of the following is a thread creation and management library?
    (a) POSIX (b) Win32
    (c) Java Thread Library (d) All of these
16. Which of the following is the POSIX standard library for thread creation and management
    (a) Pthreads (b) Threads (c) Jthreads (d) None of these
17. What happens when a thread object's *wait()* method is invoked in Java?
    (a) Causes the thread object to wait
    (b) The thread will remain in the 'Wait' state until another thread invokes the *notify()* or *notifyAll()* method of the thread object which is waiting
    (c) Both of these
    (d) None of these
18. Which of the following is true about user level threads?
    (a) Even if a process contains multiple user level threads, the OS treats it as a single thread
    (b) The user level threads are executed non-preemptively in relation to each other
    (c) User level threads follow co-operative execution model
    (d) All of these
19. Which of the following are the valid thread binding models for user level to kernel level thread binding?
    (a) One to Many (b) Many to One (c) One to One (d) Many to Many
    (e) All of these (f) only (b), (c) and (d)
20. If a thread expires, the stack memory allocated to it is reclaimed by the process to which the thread belongs. State 'True' or 'False'
    (a) True (b) False

**Multiprocessing and Multitasking**

1. Multitasking and multiprocessing refers to the same entity in the operating system context. State 'True' or 'False'
    (a) True (b) False
2. Multiprocessor systems contain
    (a) Single CPU (b) Multiple CPUs (c) No CPU
3. The ability of the operating system to have multiple programs in memory, which are ready for execution, is referred as
    (a) Multitasking (b) Multiprocessing (c) Multiprogramming
4. In a multiprocessing system
    (a) Only a single process can run at a time (b) Multiple processes can run simultaneously
    (c) Multiple processes run in pseudo parallelism
5. In a multitasking system
    (a) Only a single process can run at a time (b) Multiple processes can run simultaneously
    (c) Multiple processes run in pseudo parallelism (d) Only (a) and (c)
6. Multitasking involves
    (a) CPU execution switching of processes (b) CPU halting (c) No CPU operation

7. Multitasking involves
   (a) Context switching (b) Context saving (c) Context retrieval (d) All of these
   (e) None of these
8. What are the different types of multitasking present in operating systems?
   (a) Co-operative (b) Preemptive (c) Non-preemptive (d) All of these
9. In Co-operative multitasking, a process/task gets the CPU time when
   (a) The currently executing task terminates its execution
   (b) The currently executing task enters 'Wait' state
   (c) The currently executing task relinquishes the CPU before terminating
   (d) Never get a chance to execute (e) Either (a) or (c)
10. In Preemptive multitasking
   (a) Each process gets an equal chance for execution
   (b) The execution of a process is preempted based on the scheduling policy
   (c) Both of these (d) None of these
11. In Non-preemptive multitasking, a process/task gets the CPU time when
   (a) The currently executing task terminates its execution
   (b) The currently executing task enters 'Wait' state
   (c) The currently executing task relinquishes the CPU before terminating
   (d) All of these
   (e) None of these
12. MSDOS Operating System supports
   (a) Single user process with single thread
   (b) Single user process with multiple threads
   (c) Multiple user process with single thread per process
   (d) Multiple user process with multiple threads per process

**Task Scheduling**

1. Who determines which task/process is to be executed at a given point of time?
   (a) Process manager (b) Context manager (c) Scheduler (d) None of these
2. Task scheduling is an essential part of multitasking.
   (a) True (b) False
3. The process scheduling decision may take place when a process switches its state from
   (a) 'Running' to 'Ready' (b) 'Running' to 'Blocked'
   (c) 'Blocked' to 'Ready' (d) 'Running' to 'Completed'
   (e) All of these
   (f) Any one among (a) to (d) depending on the type of multitasking supported by OS
4. A process switched its state from 'Running' to 'Ready' due to scheduling act. What is the type of multitasking supported by the OS?
   (a) Co-operative (b) Preemptive (c) Non-preemptive (d) None of these
5. A process switched its state from 'Running' to 'Wait' due to scheduling act. What is the type of multitasking supported by the OS?
   (a) Co-operative (b) Preemptive (c) Non-preemptive (d) (b) or (c)
6. Which one of the following criteria plays an important role in the selection of a scheduling algorithm?
   (a) CPU utilisation (b) Throughput (c) Turnaround time (d) Waiting time
   (e) Response time (f) All of these
7. For a good scheduling algorithm, the CPU utilisation is
   (a) High (b) Medium (c) Non-defined

8. Under the process scheduling context, 'Throughput' is
   (a) The number of processes executed per unit of time
   (b) The time taken by a process to complete its execution
   (c) None of these
9. Under the process scheduling context, 'Turnaround Time' for a process is
   (a) The time taken to complete its execution
   (b) The time spent in the 'Ready' queue
   (c) The time spent on waiting on I/O
   (d) None of these
10. Turnaround Time (TAT) for a process includes
   (a) The time spent for waiting for the main memory
   (b) The time spent in the ready queue
   (c) The time spent on completing the I/O operations
   (d) The time spent in execution
   (e) All of these
11. For a good scheduling algorithm, the Turn Around Time (TAT) for a process should be
   (a) Minimum (b) Maximum (c) Average (d) Varying
12. Under the process scheduling context, 'Waiting time' for a process is
   (a) The time spent in the 'Ready queue'
   (b) The time spent on I/O operation (time spent in wait state)
   (c) Sum of (a) and (b)
   (d) None of these
13. For a good scheduling algorithm, the waiting time for a process should be
   (a) Minimum (b) Maximum (c) Average (d) Varying
14. Under the process scheduling context, 'Response time' for a process is
   (a) The time spent in 'Ready queue'
   (b) The time between the submission of a process and the first response
   (c) The time spent on I/O operation (time spent in wait state)
   (d) None of these
15. For a good scheduling algorithm, the response time for a process should be
   (a) Maximum (b) Average (c) Least (d) Varying
16. What are the different queues associated with process scheduling?
   (a) Ready Queue (b) Process Queue (c) Job Queue (d) Device Queue
   (e) All of the Above (f) (a), (c) and (d)
17. The 'Ready Queue' contains
   (a) All the processes present in the system
   (b) All the processes which are 'Ready' for execution
   (c) The currently running processes
   (d) Processes which are waiting for I/O
18. Which among the following scheduling is (are) Non-preemptive scheduling
   (a) First In First Out (FIFO/FCFS)
   (b) Last In First Out (LIFO/LCFS)
   (c) Shortest Job First (SJF)
   (d) All of these
   (e) None of these
19. Which of the following is true about FCFS scheduling
   (a) Favours CPU bound processes
   (b) The device utilisation is poor
   (c) Both of these
   (d) None of these
20. The average waiting time for a given set of process is _____ in SJF scheduling compared to FIFO scheduling
   (a) Minimal (b) Maximum (c) Average
21. Which among the following scheduling is (are) preemptive scheduling
   (a) Shortest Remaining Time First (SRT)
   (b) Preemptive Priority based
   (c) Round Robin (RR)
   (d) All of these
   (e) None of these
22. The Shortest Job First (SJF) algorithm is a priority based scheduling. State 'True' or 'False'
   (a) True (b) False

23. Which among the following is true about preemptive scheduling
    (a) A process is moved to the 'Ready' state from 'Running' state (preempted) without getting an explicit request from the process
    (b) A process is moved to the 'Ready' state from 'Running' state (preempted) on receiving an explicit request from the process
    (c) A process is moved to the 'Wait' state from the 'Running' state without getting an explicit request from the process
    (d) None of these
24. Which of the following scheduling technique(s) possess the drawback of 'Starvation'
    (a) Round Robin (b) Priority based preemptive
    (c) Shortest Job First (SJF) (d) (b) and (c)
    (e) None of these
25. Starvation describes the condition in which
    (a) A process is ready to execute and is waiting in the 'Ready' queue for a long time and is unable to get the CPU time due to various reasons
    (b) A process is waiting for a shared resource for a long time, and is unable to get it for various reasons.
    (c) Both of the above
    (d) None of these
26. Which of the scheduling policy offers equal opportunity for execution for all processes?
    (a) Priority based scheduling (b) Round Robin (RR) scheduling
    (c) Shortest Job First (SJF) (d) All of these
    (e) None of these
27. Round Robin (RR) scheduling commonly uses which one of the following policies for sorting the 'Ready' queue?
    (a) Priority (b) FCFS (FIFO) (c) LIFO (d) SRT
    (e) SJF
28. Which among the following is used for avoiding 'Starvation' of processes in priority based scheduling?
    (a) Priority Inversion (b) Aging (c) Priority Ceiling (d) All of these
29. Which of the following is true about 'Aging'
    (a) Changes the priority of a process at run time
    (b) Raises the priority of a process temporarily
    (c) It is a technique used for avoiding 'Starvation' of processes
    (d) All of these
    (e) None of these
30. Which is the most commonly used scheduling policy in Real-Time Operating Systems?
    (a) Round Robin (RR) (b) Priority based preemptive
    (c) Priority based non-preemptive
    (d) Shortest Job First (SJF)
31. In the process scheduling context, the IDLE TASK is executed for
    (a) To handle system interrupts
    (b) To keep the CPU always engaged or to keep the CPU in idle mode depending on the system design
    (c) To keep track of the resource usage by a process
    (d) All of these

**Task Communication and Synchronisation**

1. Processes use IPC mechanisms for
    (a) Communicating between process (b) Synchronising the access of shared resource
    (c) Both of these (d) None of these
2. Which of the following techniques is used by operating systems for inter process communication?
    (a) Shared memory (b) Messaging (c) Signalling (d) All of these

3. Under Windows Operating system, the input and output buffer memory for a named pipe is allocated in
   (a) Non-paged system memory (b) Paged system memory
   (c) Virtual memory (d) None of the above
4. Which among the following techniques is used for sharing data between processes?
   (a) Semaphores (b) Shared memory (c) Messages d) (b) and (c)
5. Which among the following is a shared memory technique for IPC?
   (a) Pipes (b) Memory mapped Object
   (c) Message blocks (d) Events (e) (a) and (b)
6. Which of the following is best advised for sharing a memory mapped object between processes under windows kernel?
   (a) Passing the handle of the shared memory object (b) Passing the name of the memory mapped object
   (c) None of these
7. Why is message passing relatively fast compared to shared memory based IPC?
   (a) Message passing is relatively free from synchronisation overheads
   (b) Message passing does not involve any OS intervention
   (c) All of these
   (d) None of these
8. In asynchronous messaging, the message posting thread just posts the message to the queue and will not wait for an acceptance (return) from the thread to which the message is posted. State 'True' or 'False'
   (a) True (b) False
9. Which of the following is a blocking message passing call in Windows?
   (a) PostMessage (b) PostThreadMessage (c) SendMessage (d) All of these
   (e) None of these
10. Under Windows operating system, the message is passed through ____ for Inter Process Communication (IPC) between processes?
   (a) Message structure (b) Memory mapped object
   (c) Semaphore (d) All of these
11. Which of the following is true about 'Signals' for Inter Process Communication?
   (a) Signals are used for asynchronous notifications (b) Signals are not queued
   (c) Signals do not carry any data (d) All of these
12. Which of the following is true about *Racing* **or** *Race condition*?
   (a) It is the condition in which multiple processes compete (race) each other to access and manipulate shared data concurrently
   (b) In a race condition the final value of the shared data depends on the process which acted on the data finally
   (c) Racing will not occur if the shared data access is atomic
   (d) All of these
13. Which of the following is true about *deadlock*?
   (a) 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
   (b) Is the situation in which none of the competing process will be able to access the resources held by other processes since they are locked by the respective processes
   (c) Is a result of chain of circular wait
   (d) All of these
14. What are the conditions favouring deadlock in multitasking?
   (a) Mutual Exclusion (b) Hold and Wait (c) No kernel resource preemption at kernel level
   (d) Chain of circular waits (e) All of these
15. Livelock describes the situation where
   (a) A process waits on a resource is not blocked on it and it makes frequent attempts to acquire the resource. But unable to acquire it since it is held by other process

(b) A process waiting in the 'Ready' queue is unable to get the CPU time for execution
(c) Both of these
(d) None of these

16. *Priority inversion* is
(a) The condition in which a high priority task needs to wait for a low priority task to release a resource which is shared between the high priority task and the low priority task
(b) The act of increasing the priority of a process dynamically
(c) The act of decreasing the priority of a process dynamically
(d) All of these

17. Which of the following is true about Priority inheritance?
(a) A low priority task which currently holds a shared resource requested by a high priority task temporarily inherits the priority of the high priority task
(b) The priority of the low priority task which is temporarily boosted to high is brought to the original value when it releases the shared resource
(c) All of these
(d) None of these

18. Which of the following is true about Priority Ceiling based Priority inversion handling?
(a) A priority is associated with each shared resource
(b) The priority associated to each resource is the priority of the highest priority task which uses this shared resource
(c) Whenever a task accesses a shared resource, the scheduler elevates the priority of the task to that of the ceiling priority of the resource
(d) The priority of the task is brought back to the original level once the task completes the accessing of the shared resource
(e) All of these



19. Process/Task synchronisation is essential for?
(a) Avoiding conflicts in resource access in multitasking environment
(b) Ensuring proper sequence of operation across processes.
(c) Communicating between processes
(d) All of these
(e) None of these

20. Which of the following is true about *Critical Section*?
(a) It is the code memory area which holds the program instructions (piece of code) for accessing a shared resource
(b) The access to the critical section should be exclusive
(c) All of these
(d) None of these

21. Which of the following is true about mutual exclusion?
(a) Mutual exclusion enforces mutually exclusive access of resources by processes
(b) Mutual exclusion may lead to deadlock
(c) Both of these
(d) None of these

22. Which of the following is an example of mutual exclusion enforcing policy?
(a) Busy Waiting (Spin lock)
(b) Sleep & Wake up
(c) Both of these
(d) None of these

23. Which of the following is true about lock based synchronisation mechanism?
(a) It is CPU intensive
(b) Locks are useful in handling situations where the processes is likely to be blocked for a shorter period of time on waiting the lock

(c) If the lock is being held for a long time by a process and if it is preempted by the OS, the other threads waiting for this lock may have to spin a longer time for getting
(d) All of these
(e) None of these

24. Which of the following synchronisation techniques follow the '*Sleep & Wakeup*' mechanism for mutual exclusion?
(a) Mutex (b) Semaphore (c) Critical Section (d) Spin lock
(e) (a), (b) and (c)

25. Which of the following is true about *mutex objects* for IPC synchronisation under Windows OS?
(a) Only one process/thread can own the '*mutex object*' at a time
(b) The state of a mutex object is set to non-signalled when it is not owned by any process/thread, and set to signalled when it is owned by any process/thread
(c) The state of a mutex object is set to signalled when it is not owned by any process/thread, and set to non-signalled when it is owned by any process/thread
(d) Both (a) & (b) (e) Both (a) & (c)

26. Which of the following is (are) the *wait functions* provided by windows for synchronisation purpose?
(a) WaitForSingleObject (b) WaitForMultipleObjects
(c) Sleep (d) Both (a) and (b)

27. Which of the following is true about *Critical Section object*?
(a) It can only be used by the threads of a single process (Intra process)
(b) The 'Critical Section' must be initialised before the threads of a process can use it
(c) Accessing Critical Section blocks the execution of the caller thread if the critical section is already in use by other threads
(d) Threads which are blocked by the Critical Section access call, waiting on a critical section, are added to a wait queue and are woken when the Critical Section is available to the requested thread
(e) All of these

28. Which of the following is a non-blocking Critical Section accessing call under windows?
(a) *EnterCriticalSection* (b) *TryEnterCriticalSection*
(c) Both of these (d) None of these

29. The Critical Section object makes the piece of code residing inside it ____?
(a) Non-reentrant (b) Re-entrant (c) Thread safe (d) Both (a) and (c)

30. Which of the following synchronisation techniques is exclusively used for synchronising the access of shared resources by the threads of a process (Intra Process Synchronisation) under Windows kernel?
(a) Mutex object (b) Critical Section object (c) Interlocked functions
(d) Both (c) and (d)

## Review Questions

**Operating System Basics**

1. What is an Operating System? Where is it used and what are its primary functions?
2. What is kernel? What are the different functions handled by a general purpose kernel?
3. What is kernel space and user space? How is kernel space and user space interfaced?
4. What is monolithic and microkernel? Which one is widely used in real-time operating systems?
5. What is the difference between a General Purpose kernel and a Real-Time kernel? Give an example for both.

**Real-Time Operating System (RTOS)**

1. Explain the basic functions of a real-time kernel?
2. What is task control block (TCB)? Explain the structure of TCB

3. Explain the difference between the memory management of general purpose kernel and real-time kernel.
4. What is virtual memory? What are the advantages and disadvantages of virtual memory?
5. Explain how 'accurate time management' is achieved in real-time kernel
6. What is the difference between 'Hard' and 'Soft' real-time systems? Give an example for 'Hard' and 'Soft' Real-Time kernels

**Tasks, Process and Threads**

1. Explain *Task* in the operating system context
2. What is *Process* in the operating system context?
3. Explain the memory architecture of a process
4. What is *Process Life Cycle*?
5. Explain the various activities involved in the creation of process and threads
6. What is *Process Control Block* (*PCB*)? Explain the structure of *PCB*
7. Explain *Process Management* in the Operating System Context
8. What is *Thread* in the operating system context?
9. Explain how *Threads* and *Processes* are related? What are common to *Process* and *Threads*?
10. Explain the memory model of a 'thread'.
11. Explain the concept of 'multithreading'. What are the advantages of multithreading?
12. Explain how multithreading can improve the performance of an application with an illustrative example
13. Why is thread creation faster than process creation?
14. Explain the commonly used thread standards for thread creation and management by different operating systems
15. Explain *Thread context switch* and the various activities performed in thread context switching for user level and kernel level threads
16. What all information is held by the thread control data structure of a user/kernel thread?
17. What are the differences between user level and kernel level threads?
18. What are the advantages and disadvantages of using user level threads?
19. Explain the different thread binding models for user and kernel level threads
20. Compare threads and processes in detail

**Multiprocessing and Multitasking**

1. Explain multiprocessing, multitasking and multiprogramming
2. Explain context switching, context saving and context retrieval
3. What all activities are involved in context switching?
4. Explain the different multitasking models in the operating system context

**Task Scheduling**

1. What is *task scheduling* in the operating system context?
2. Explain the various factors to be considered for the selection of a scheduling criteria
3. Explain the different *queues* associated with process scheduling
4. Explain the different types of *non-preemptive* scheduling algorithms. State the merits and de-merits of each
5. Explain the different types of *preemptive* scheduling algorithms. State the merits and de-merits of each
6. Explain *Round Robin (RR)* process scheduling with interrupts
7. Explain *starvation* in the process scheduling context. Explain how starvation can be effectively tackled?
8. What is *IDLEPROCESS*? What is the significance of IDLEPROCESS in the process scheduling context?
9. Three processes with process IDs P1, P2, P3 with estimated completion time 5, 10, 7 milliseconds respectively enters the ready queue together in the order P1, P2, P3. Process P4 with estimated execution completion time 2 milliseconds enters the ready queue after 5 milliseconds. 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 the FIFO scheduling
10. Three processes with process IDs P1, P2, P3 with estimated completion time 12, 10, 2 milliseconds respectively enters the ready queue together in the order P2, P3, P1. Process P4 with estimated execution completion time

4 milliseconds enters the Ready queue after 8 milliseconds. 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 the FIFO scheduling

11. Three processes with process IDs P1, P2, P3 with estimated completion time 8, 4, 7 milliseconds respectively enters the ready queue together in the order P3, P1, P2. P1 contains an I/O waiting time of 2 milliseconds when it completes 4 milliseconds of its execution. P2 and P3 do not contain any I/O waiting. Calculate the waiting time and Turn Around Time (TAT) for each process and the average waiting time and Turn Around Time in the LIFO scheduling. All the estimated execution completion time is excluding I/O wait time
12. Three processes with process IDs P1, P2, P3 with estimated completion time 12, 10, 2 milliseconds respectively enters the ready queue together in the order P2, P3, P1. Process P4 with estimated execution completion time 4 milliseconds enters the Ready queue after 8 milliseconds. 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 the LIFO scheduling
13. Three processes with process IDs P1, P2, P3 with estimated completion time 6, 8, 2 milliseconds respectively enters the ready queue together. Process P4 with estimated execution completion time 4 milliseconds enters the Ready queue after 1 millisecond. 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 the non-preemptive SJF scheduling
14. Three processes with process IDs P1, P2, P3 with estimated completion time 4, 6, 5 milliseconds and priorities 1, 0, 3 (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 non-preemptive priority based scheduling algorithm
15. Three processes with process IDs P1, P2, P3 with estimated completion time 4, 6, 5 milliseconds and priorities 1, 0, 3 (0—highest priority, 3 lowest priority) respectively enters the ready queue together. Process P4 with estimated execution completion time 6 milliseconds and priority 2 enters the 'Ready' queue after 5 milliseconds. 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 non-preemptive priority based scheduling algorithm
16. Three processes with process IDs P1, P2, P3 with estimated completion time 8, 4, 7 milliseconds respectively enters the ready queue together. P1 contains an I/O waiting time of 2 milliseconds when it completes 4 milliseconds of its execution. P2 and P3 do not contain any I/O waiting. Calculate the waiting time and Turn Around Time (TAT) for each process and the average waiting time and Turn Around Time in the SRT scheduling. All the estimated execution completion time is excluding I/O waiting time
17. Three processes with process IDs P1, P2, P3 with estimated completion time 12, 10, 6 milliseconds respectively enters the ready queue together. Process P4 with estimated execution completion time 2 milliseconds enters the Ready queue after 3 milliseconds. 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 the SRT scheduling
18. Three processes with process IDs P1, P2, P3 with estimated completion time 10, 14, 20 milliseconds respectively, enters the ready queue together in the order P3, P2, 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) in RR algorithm with Time slice = 2 ms
19. Three processes with process IDs P1, P2, P3 with estimated completion time 12, 10, 12 milliseconds respectively enters the ready queue together in the order P2, P3, P1. Process P4 with estimated execution completion time 4 milliseconds enters the Ready queue after 8 milliseconds. 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 = 4 ms
20. Three processes with process IDs P1, P2, P3 with estimated completion time 4, 6, 5 milliseconds and priorities 1, 0, 3 (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 preemptive priority based scheduling algorithm

21. Three processes with process IDs P1, P2, P3 with estimated completion time 6, 2, 4 milliseconds respectively, enters the ready queue together in the order P1, P3, P2. Process P4 with estimated execution time 4 milliseconds entered the 'Ready' queue 3 milliseconds later the 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 (Assuming there is no I/O waiting for the processes) in RR algorithm with Time slice = 2 ms

**Task Communication and Synchronisation**

1. Explain the various process interaction models in detail.
2. What is Inter Process Communication (IPC)? Give an overview of different IPC mechanisms adopted by various operating systems.
3. Explain how multiple processes in a system co-operate.
4. Explain how multiple threads of a process co-operate.
5. Explain the shared memory based IPC.
6. Explain the concept of memory mapped objects for IPC.
7. Explain the handle sharing and name sharing based memory mapped object technique for IPC under Windows Operating System.
8. Explain the message passing technique for IPC. What are the merits and de-merits of message based IPC?
9. Explain the synchronous and asynchronous messaging mechanisms for IPC under Windows kernel.
10. Explain *Race condition* in detail, in relation to the shared resource access.
11. What is *deadlock*? What are the different conditions favouring deadlock?
12. Explain by *Coffman conditions*?
13. Explain the different methods of handling deadlocks.
14. Explain *livelock* in the resource sharing context.
15. Explain *starvation* in the resource sharing context.
16. Explain the *Dining Philosophers* problem in the process synchronisation context.
17. Explain the *Produces-consumer* problem in the inter process communication context.
18. Explain *bounded-buffer* problem in the interprocess communication context.
19. Explain *buffer overrun* and *buffer under-run*.
20. What is *priority inversion*? What are the different techniques adopted for handling priority inversion?
21. What are the merits and de-merits of *priority ceiling*?
22. Explain the different task communication synchronisation issues encountered in Interprocess Communication
23. What is *task (process) synchronisation*? What is the role of process synchronisation in IPC?
24. What is *mutual exclusion* in the process synchronisation context? Explain the different mechanisms for mutual exclusion
25. What are the merits and de-merits of *busy-waiting* (*spinlock*) based mutual exclusion?
26. Explain the *Test and Set Lock (TSL)* based mutual exclusion technique. Explain how *TSL* is implemented in Intel family of processors
27. Explain the *interlocked functions* for lock based mutual exclusion under Windows OS
28. Explain the advantages and limitations of *interlocked function* based synchronisation under Windows
29. Explain the *sleep & wakeup* mechanism for mutual exclusion
30. What are the merits and de-merits of *sleep & wakeup* mechanism based mutual exclusion?
31. What is *mutex*?
32. Explain the *mutex* based process synchronisation under Windows OS
33. What is *semaphore*? Explain the different types of semaphores. Where is it used?
34. What is binary *semaphore*? Where is it used?
35. What is the difference between *mutex* and *semaphore*?
36. What is the difference between *semaphore* and *binary semaphore*?
37. What is the difference between *mutex* and *binary semaphore*?

38. Explain the *semaphore* based process synchronisation under Windows OS
39. Explain the *critical section problem*?
40. What is *critical section*? What are the different techniques for controlling access to *critical section*?
41. Explain the *critical section object* for process synchronisation. Why is critical section object based synchronisation fast?
42. Explain the *critical section object* based process synchronisation under Windows OS.
43. Explain the *Event* based synchronisation mechanism for IPC.
44. Explain the *Event object* based synchronisation mechanism for IPC under Windows OS.
45. What is a *device driver*? Explain its role in the OS context.
46. Explain the architecture of device drivers.
47. Explain the different functional and non-functional requirements that needs to be evaluated in the selection of an RTOS.

## Lab Assignments

1. Write a multithreaded Win32 console application satisfying:
   (a) The main thread of the application creates a child thread with name "child_thread" and passes the pointer of a buffer holding the data "Data passed from Main thread".
   (b) The main thread sleeps for 10 seconds after creating the child thread and then quits.
   (c) The child thread retrieves the message from the memory location pointed by the buffer pointer and prints the retrieved data to the console and sleeps for 100 milliseconds and then quits.
   (d) Use appropriate error handling mechanisms wherever possible.
2. Write a multithreaded Win32 console application for creating '*n*' number of child threads (*n* is configurable). Each thread prints the message "I'm in thread thread no" ('thread no' is the number passed to the thread when it is created. It varies from 0 to $n-1$) and sleeps for 50 milliseconds and then quits. The main thread, after creating the child threads, wait for the completion of the execution of child threads and quits when all the child threads are completed their execution.
3. Write a multithreaded application using 'PThreads' for creating '*n*' number of child threads (*n* is configurable). The application should receive '*n*' as command line parameter. Each thread prints the message "I'm in thread thread no" ('thread no' is the number passed to the thread when it is created. It varies from 0 to $n-1$) and sleeps for 1 second and then quits. The main thread, after creating the child threads, wait for the completion of the execution of child threads and quits when all the child threads are completed their execution. Compile and execute the application in Linux.
4. Write a multithreaded application in Win32 satisfying the following:
   (a) Two child threads are created with normal priority
   (b) Thread 1 retrieves and prints its priority and sleeps for 500 milliseconds and then quits
   (c) Thread 2 prints the priority of thread 1 and raises its priority to above normal and retrieves the new priority of thread 1, prints it and then quits
   (d) The main thread waits for the completion of both the child threads and then terminates.
5. Write a Win32 console application illustrating the usage of anonymous pipes for data sharing between a parent and child thread of a process. The application should satisfy the following conditions:
   (a) The main thread of the process creates an anonymous pipe with size 512KB and assigns the handle of the pipe to a global handle
   (b) The main thread creates an event object "synchronise" with state non-signalled and a child thread with name 'child_thread'.
   (c) The main thread waits for the signalling of the event object "synchronise" and reads data from the anonymous pipe when the event is signalled and prints the data read from the pipe on the console window.

(d) The main thread waits for the execution completion of the child thread and quits when the child thread completes its execution.

(e) The child thread writes the data "Hi from child thread" to the anonymous pipe and sets the event object "synchronise" and sleeps for 500 milliseconds and then quits.

Compile and execute the application using Visual Studio under Windows XP/NT OS.

6. Write a Win32 console application (Process 1) illustrating the creation of a memory mapped object of size 512KB with name "mysharedobject". Create an event object with name "synchronise" with state non-signalled. Read the memory mapped object when the event is signalled and display the contents on the console window. Create a second console application (Process 2) for opening the memory mapped object with name "mysharedobject" and event object with name "synchronise". Write the message "Message from Process 2" to the memory mapped object and set the event object "synchronise". Use appropriate error handling mechanisms wherever possible. Compile both the applications using Visual Studio and execute them in the order Process 1 followed by Process 2 under Windows XP/NT OS.
7. Write a multithreaded Win32 console application where:
   (a) The main thread creates a child thread with default stack size and name 'Child_Thread'.
   (b) The main thread sends user defined messages and the message 'WM_QUIT' randomly to the child thread.
   (c) The child thread processes the message posted by the main thread and quits when it receives the 'WM_QUIT' message.
   (d) The main thread checks the termination of the child thread and quits when the child thread completes its execution.
   (e) The main thread continues sending random messages to the child thread till the WM_QUIT message is sent to child thread.
   (f) The messaging mechanism between the main thread and child thread is synchronous.

   Compile the application using Visual Studio and execute it under Windows XP/NT OS.
8. Write a Win32 console application illustrating the usage of anonymous pipes for data sharing between a parent and child processes using handle inheritance mechanism. Compile and execute the application using Visual Studio under Windows XP/NT OS.
9. Write a Win32 console application for creating an anonymous pipe with 512 bytes of size and pass the 'Read handle' of the pipe to a second process (another Win32 console application) using a memory mapped object. The first process writes a message "Hi from Pipe Server". The second process reads the data written by the pipe server to the pipe and displays it on the console window. Use event object for indicating the availability of data on the pipe and mutex objects for synchronising the access to the pipe.
10. Write a multithreaded Win32 Process addressing:
    (a) The main thread of the process creates an unnamed memory mapped object with size 1K and shares the handle of the memory mapped object with other threads of the process
    (b) The main thread writes the message "Hi from main thread" and informs the availability of data to the child thread by signalling an event object
    (c) The main thread waits for the execution completion of the child thread after writing the message to the memory mapped area and quits when the child thread completes its execution
    (d) The child thread reads the data from the memory mapped area and prints it on the console window when the event object is signalled by the main thread
    (e) The read write access to the memory mapped area is synchronised using a mutex object
11. Write a multithreaded application using Java thread library satisfying:
    (a) The first thread prints "Hello I'm going to the wait queue" and enters wait state by invoking the wait method.
    (b) The second thread sleeps for 500 milliseconds and then prints "Hello I'm going to invoke first thread" and invokes the first thread.
    (c) The first thread prints "Hello I'm invoked by the second thread" when invoked by the second thread.