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

LEARNING OBJECTIVES

LO 1 Understand the basics of an operating system and the need for an operating system
- Learn the basic kernel services of an operating system

LO 2 Classify the types of operating systems
- Learn the internals of Real-Time Operating System and the fundamentals of RTOS based embedded firmware design
- Learn about the different real-time kernels and the features that make a kernel Real-Time

LO 3 Discuss 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

LO 4 Understand the difference between multiprocessing and multitasking
- Learn about the different types of multitasking (Co-operative, Preemptive and Non-preemptive

LO 5 Describe 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

LO 6 Explain the different Inter Process Communication (IPC) mechanisms used by tasks/process to communicate and co-operate each other in a multitasking environment

LO 7 Identify the RPC based Inter Process Communication
- 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

LO 8 State 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

LO 9 Analyse device drivers, their role in an operating system based embedded system design, the structure of a device driver, and interrupt handling inside device drivers

LO 10 Discuss 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-lock Brake System (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

LO 1 Understand the basics of an operating system and the need for an operating system

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.

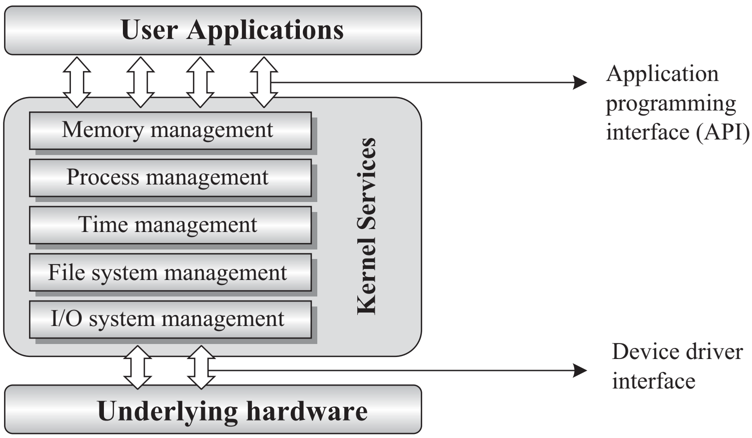


Fig. 10.1 The Operating System Architecture

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 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 NT 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 10 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.

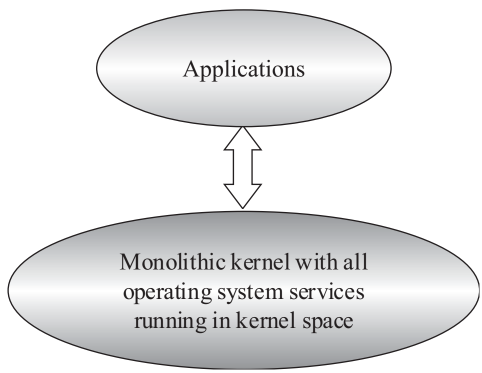


Fig. 10.2 The Monolithic Kernel Model

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.

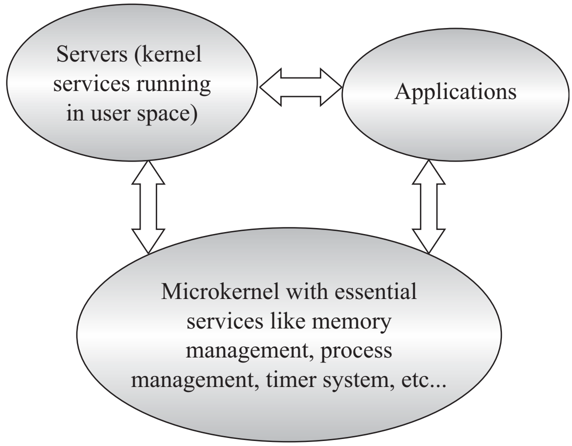


Fig. 10.3 The Microkernel model

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

LO 2 Classify the 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 10/8.x/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 Embedded Compact, 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/Embedded Compact 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 latter chapter).The '*block*' based memory

* *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.

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

LO 3 Discuss tasks, processes and threads in the operating system context

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

10.3.1 Process

A '*Process*' is a program, or part of it, in execution. Process is also known as an instance of a program in execution. Multiple instances of the same program can execute simultaneously. A process requires various system resources like CPU for executing the process, memory for storing the code corresponding to the process and associated variables, I/O devices for information exchange, etc. A process is sequential in execution.

10.3.1.1 The Structure of a Process

The concept of '*Process*' leads to concurrent execution (pseudo parallelism) of tasks and thereby the efficient utilisation of the CPU and other system resources. Concurrent execution is achieved through the sharing of CPU among the processes. A process mimics a processor in properties and holds a set of registers, process status, a Program Counter (PC) to point to the next executable instruction of the process, a stack for holding the local variables associated with the process and the code corresponding to the process. This can be visualised as shown in Fig. 10.4.

A process which inherits all the properties of the CPU can be considered as a virtual processor, awaiting its turn to have its properties switched into the physical processor. When the process gets its turn, its registers and the program counter register becomes mapped to the physical registers of the CPU. From a memory perspective, the memory occupied by the *process* is segregated into three regions, namely, Stack memory, Data memory and Code memory (Fig. 10.5).

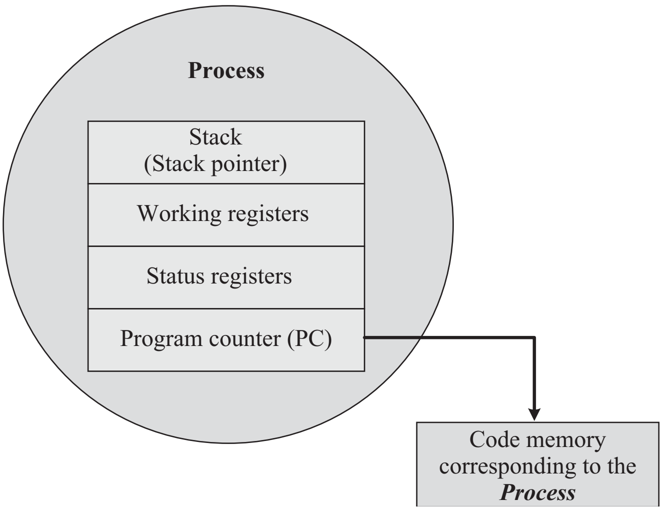


Fig. 10.4 Structure of a Process

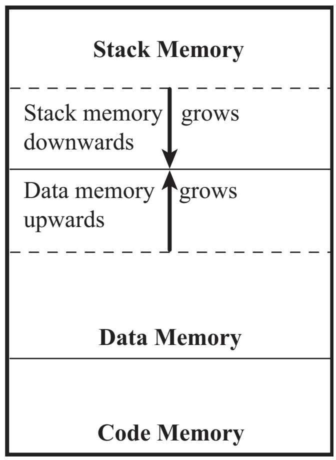


Fig. 10.5 Memory organisation of a Process

The 'Stack' memory holds all temporary data such as variables local to the process. Data memory holds all global data for the process. The code memory contains the program code (instructions) corresponding to the process. On loading a process into the main memory, a specific area of memory is allocated for the process. The stack memory usually starts (OS Kernel implementation dependent) at the highest memory address from the memory area allocated for the process. Say for example, the memory map of the memory area allocated for the process is 2048 to 2100, the stack memory starts at address 2100 and grows downwards to accommodate the variables local to the process.

10.3.1.2 Process States and State Transition

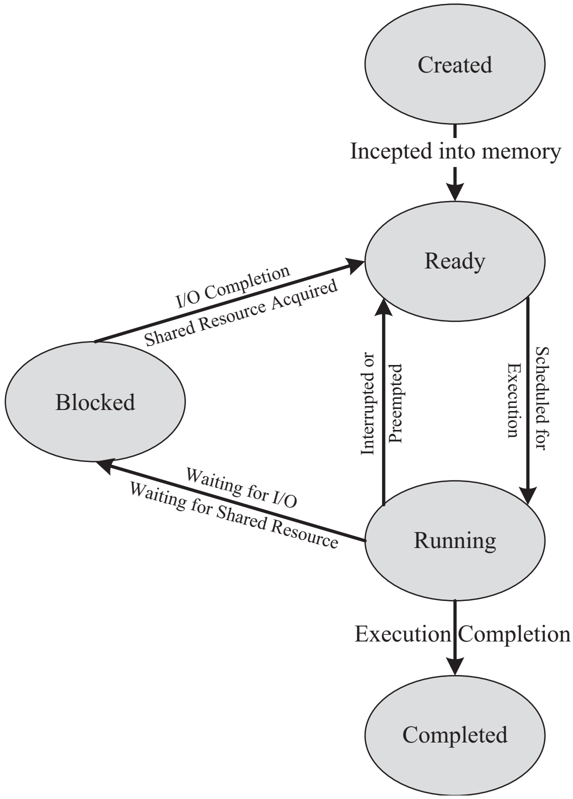


Fig. 10.6 Process states and state transition representation

The creation of a process to its termination is not a single step operation. The process traverses through a series of states during its transition from the newly created state to the terminated state. The cycle through which a process changes its state from '*newly created*' to '*execution completed*' is known as '*Process Life Cycle*'. The various states through which a process traverses through during a Process Life Cycle indicates the current status of the process with respect to time and also provides information on what it is allowed to do next. Figure 10.6 represents the various states associated with a process.

The state at which a process is being created is referred as 'Created State'. The Operating System recognises a process in the '*Created State*' but no resources are allocated to the process. The state, where a process is incepted into the memory and awaiting the processor time for execution, is known as '*Ready State*'. At this stage, the process is placed in the '*Ready list*' queue maintained by the OS. The state where in the source code instructions corresponding to the process is being executed is called '*Running State*'. Running state is the state at which the process execution happens. '*Blocked State/Wait State*' refers to a state where a running process is temporarily suspended from execution and does not have immediate access to resources. The blocked state might be invoked by various conditions like: the process enters a wait state for an event to occur (e.g. Waiting for user inputs such as keyboard input) or waiting for getting access to a shared resource (will be discussed at a later section of this chapter). A state where the process completes its execution is known as '*Completed State*'. The transition of a process from one state to another is known as '*State transition*'. When a process changes its state from Ready to running or from running to blocked or terminated or from blocked to running, the CPU allocation for the process may also change.

It should be noted that the state representation for a process/task mentioned here is a generic representation. The states associated with a task may be known with a different name or there may be more or less number of states than the one explained here under different OS kernel. For example, under VxWorks' kernel, the tasks may be in either one or a specific combination of the states READY, PEND, DELAY and SUSPEND. The PEND state represents a state where the task/process is blocked on waiting for I/O or system resource. The DELAY state represents a state in which the task/process is sleeping and the SUSPEND state represents a state where a task/process is temporarily suspended from execution and not available for execution. Under MicroC/OS-II kernel, the tasks may be in one of the states, DORMANT, READY, RUNNING, WAITING or INTERRUPTED. The DORMANT state represents the 'Created' state and WAITING state represents the state in which a process waits for shared resource or I/O access. We will discuss about the states and state transition for tasks under VxWorks and uC/OS-II kernel in a later chapter.

10.3.1.3 Process Management

Process management deals with the creation of a process, setting up the memory space for the process, loading the process's code into the memory space, allocating system resources, setting up a Process Control Block (PCB) for the process and process termination/deletion. For more details on Process Management, refer to the section 'Task/Process management' given under the topic 'The Real-Time Kernel' of this chapter.

10.3.2 Threads

A *thread* is the primitive that can execute code. A *thread* is a single sequential flow of control within a process. '*Thread*' is also known as lightweight process. A process can have many threads of execution. Different threads, which are part of a process, share the same address space; meaning they share the data memory, code memory and heap memory area. Threads maintain their own thread status (CPU register values), Program Counter (PC) and stack. The memory model for a process and its associated threads are given in Fig. 10.7.

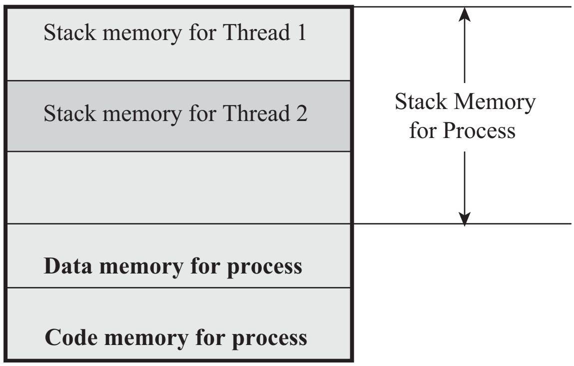


Fig. 10.7 Memory organisation of a Process and its associated Threads

10.3.2.1 The Concept of Multithreading

A process/task in embedded application may be a complex or lengthy one and it may contain various suboperations like getting input from I/O devices connected to the processor, performing some internal calculations/operations, updating some I/O devices etc. If all the subfunctions of a task are executed in sequence, the CPU utilisation may not be efficient. For example, if the process is waiting for a user input, the CPU enters the wait state for the event, and the process execution also enters a wait state. Instead of this single sequential execution of the whole process, if the task/process is split into different threads carrying out the different subfunctionalities of the process, the CPU can be effectively utilised and when the thread corresponding to the I/O operation enters the wait state, another threads which do not require the I/O event for their operation can be switched into execution. This leads to more speedy execution of the process and the efficient utilisation of the processor time and resources. The multithreaded architecture of a process can be better visualised with the thread-process diagram shown in Fig. 10.8.

If the process is split into multiple threads, which executes a portion of the process, there will be a main thread and rest of the threads will be created within the main thread. Use of multiple threads to execute a process brings the following advantage.

- Better memory utilisation. Multiple threads of the same process share the address space for data memory. This also reduces the complexity of inter thread communication since variables can be shared across the threads.
- Since the process is split into different threads, when one thread enters a wait state, the CPU can be utilised by other threads of the process that do not require the event, which the other thread is waiting, for processing. This speeds up the execution of the process.
- Efficient CPU utilisation. The CPU is engaged all time.

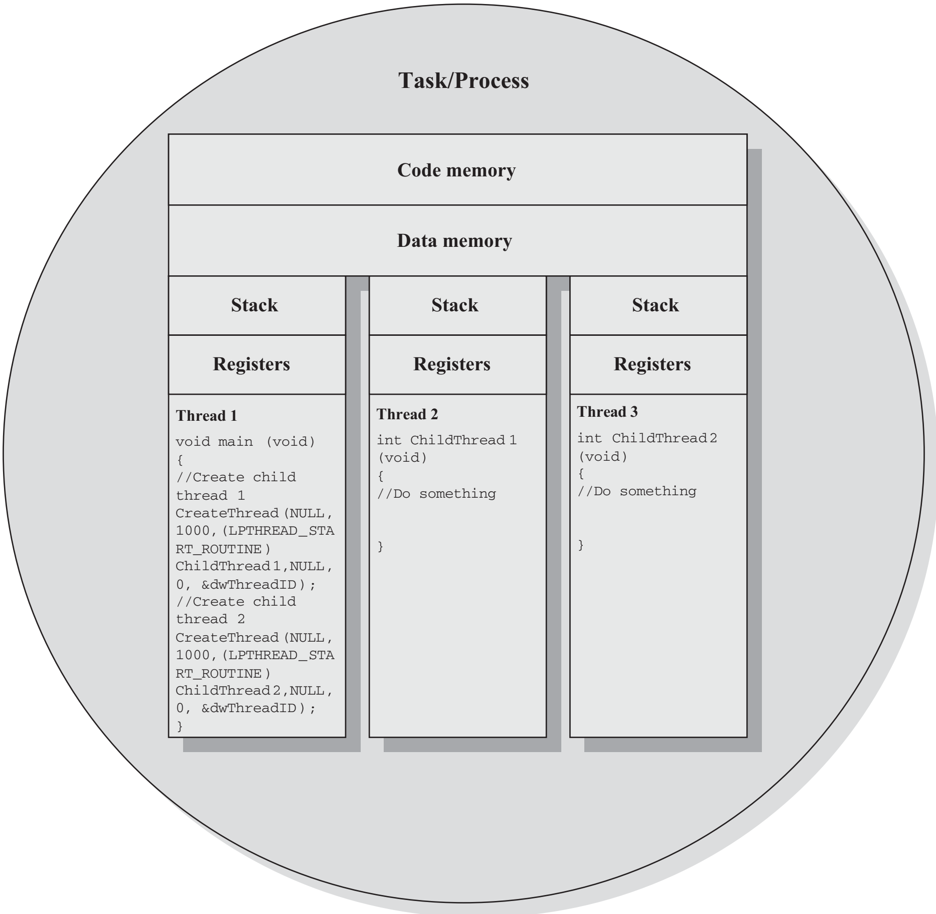


Fig. 10.8 Process **with multi-threads**

10.3.2.2 Thread Standards

Thread standards deal with the different standards available for thread creation and management. These standards are utilised by the operating systems for thread creation and thread management. It is a set of thread class libraries. The commonly available thread class libraries are explained below.

POSIX Threads POSIX stands for Portable Operating System Interface. The *POSIX.4* standard deals with the Real-Time extensions and *POSIX.4a* standard deals with thread extensions. The POSIX standard library for thread creation and management is '*Pthreads*'. '*Pthreads*' library defines the set of POSIX thread creation and management functions in 'C' language.

The primitive

```
int pthread_create(pthread_t *new_thread_ID, const pthread_attr_t
*attribute, void * (*start_function)(void *), void *arguments);
```

creates a new thread for running the function *start_ function*. Here *pthread_t* is the handle to the newly created thread and *pthread_attr_t* is the data type for holding the thread attributes. '*start_function*' is the function the thread is going to execute and *arguments* is the arguments for '*start_function*' (It is a void * in the above example). On successful creation of a *Pthread*, *pthread_create()* associates the Thread Control Block (TCB) corresponding to the newly created thread to the variable of type *pthread_t* (*new_thread_ID* in our example).

The primitive

```
int pthread_join(pthread_t new_thread,void * *thread_status);
```

blocks the current thread and waits until the completion of the thread pointed by it (In this example *new_thread*)

All the POSIX 'thread calls' returns an integer. A return value of zero indicates the success of the call. It is always good to check the return value of each call.

Example 1

Write a multithreaded application to print "Hello I'm in main thread" from the main thread and "Hello I'm in new thread" 5 times each, using the *pthread_create()* and *pthread_join()* POSIX primitives.

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

```
        return -1;
     }
   for( j= 0; j < 5; j++ )
    {
     printf("Hello I'm in main thread\n" );
     //Wait for some time. Do nothing
     //The following line of code can be replaced with
     //OS supported delay function like sleep(), delay etc…
     for( i= 0; i < 10000; i++ );
    }
   if (pthread_join(tcb, NULL ))
     {
        //Thread join failed
        printf("Error in Thread join\n" );
        return -1;
     }
   return 1;
}
```

You can compile this application using the *gcc* compiler. Examine the output to figure out the thread execution switching. The lines printed will give an idea of the order in which the thread execution is switched between. The *pthread_join* call forces the main thread to wait until the completion of the thread *tcb*, if the main thread finishes the execution first.

The termination of a thread can happen in different ways. The thread can terminate either by completing its execution (natural termination) or by a forced termination. In a natural termination, the thread completes its execution and returns back to the main thread through a simple *return* or by executing the *pthread_exit()* call. Forced termination can be achieved by the call *pthread_cancel()* or through the termination of the main thread with *exit* or *exec* functions. *pthread_cancel()* call is used by a thread to terminate another thread.

pthread_exit() call is used by a thread to explicitly exit after it completes its work and is no longer required to exist. If the main thread finishes before the threads it has created, and exits with *pthread_exit()*, the other threads continue to execute. If the main thread uses *exit* call to exit the thread, all threads created by the main thread is terminated forcefully. Exiting a thread with the call *pthread_exit()* will not perform a cleanup. It will not close any files opened by the thread and files will remain in the open status even after the thread terminates. Calling *pthread_join* at the end of the main thread is the best way to achieve synchronisation and proper cleanup. The main thread, after finishing its task waits for the completion of other threads, which were joined to it using the *pthread_join* call. With a *pthread_join* call, the main thread waits other threads, which were joined to it, and finally merges to the single main thread. If a new thread spawned by the main thread is still not joined to the main thread, it will be counted against the system's maximum thread limit. Improper cleanup will lead to the failure of new thread creation.

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

```
HANDLE    CreateThread(LPSECURITY_ATTRIBUTES    lpThreadAttributes,DWORD
dwStackSize,  LPTHREAD_START_ROUTINE  lpStartAddress,  LPVOID  lpParameter,
DWORD dwCreationFlags, LPDWORD lpThreadId );
```

The parameter *lpThreadAttributes* defines the security attributes for the thread and *dwStackSize* defines the stack size for the thread. These two parameters are not supported by the Windows CE/Embedded Compact Real-Time Operating Systems and it should be kept as NULL and 0 respectively in a *CreateThread* API Call. The other parameters are

lpStartAddress: Pointer to the function which is to be executed by the thread.

lpParameter: Parameter specifying an application-defined value that is passed to the thread routine.

dwCreationFlags: Defines the state of the thread when it is created. Usually it is kept as 0 or CREATE_SUSPENDED implying the thread is created and kept at the suspended state.

lpThreadId: Pointer to a DWORD that receives the identifier for the thread.

On successful creation of the thread, *CreateThread* returns the handle to the thread and the thread identifier.

The API *GetCurrentThread(void)* returns the handle of the current thread and *GetCurrentThreadId(void)* returns its ID. *GetThreadPriority (HANDLE hThread)* API returns an integer value representing the current priority of the thread whose handle is passed as *hThread*. Threads are always created with normal priority (THREAD_PRIORITY_NORMAL. Refer MSDN documentation for the different thread priorities and their meaning). *SetThreadPriority (HANDLE hThread, int nPriority)* API is used for setting the priority of a thread. The first parameter to this function represents the thread handle and the second one the thread priority.

For Win32 threads, the normal thread termination happens when an exception occurs in the thread, or when the thread's execution is completed or when the primary thread or the process to which the thread is associated is terminated. A thread can exit itself by calling the *ExitThread (DWORD dwExitCode)* API. The parameter *dwExitCode* sets the exit code for thread termination. Calling *ExitThread* API frees all the resources utilised by the thread. The exit code of a thread can be checked by other threads by calling the *GetExitCodeThread (HANDLE hThread, LPDWORD lpExitCode)*. *TerminateThread (HANDLE hThread, DWORD dwExitCode)* API is used for terminating a thread from another thread. The handle *hThread* indicates which thread is to be terminated and *dwExitCode* sets the exit code for the thread. This API will not execute the thread termination and clean up code and may not free the resources occupied by the thread. *TerminateThread* is a potentially dangerous call and it should not be used in normal conditions as a mechanism for terminating a thread. Use this call only as a final choice. When a thread is terminated through *TerminateThread* method, the system releases the thread's initial stack and the thread will not get a chance to execute any user-mode code. Also any dynamic link libraries (dlls) attached to the thread are not notified that the thread is terminating. *TerminateThread* can lead to potential issues like: Non-releasing of the critical section object, any, owned by the thread, non-releasing of heap lock, if the thread is allocating memory from the heap, inconsistencies of the kernel32 state for the thread's process if the thread was executing certain kernel32 call when it is terminated, issues in shared dll functions, if the thread was manipulating the global state of a shared dll when it is terminated etc. *SuspendThread(HANDLE hThread)* API can be used for suspending a thread from execution provided the *handle hThread* possesses *THREAD_SUSPEND_RESUME* access right. If the *SuspendThread* API call succeeds, the thread stops executing and increments its internal suspend count. The thread becomes suspended if its suspend count is greater than zero. The *SuspendThread* function is primarily designed for use by debuggers. One must be cautious in using this API for the reason it may cause *deadlock* condition if the thread is suspended at a stage where it acquired a mutex or shared resource and another thread tries to access the same. The *ResumeThread(HANDLE hThread)* API is used for resuming a suspended thread. The *ResumeThread* API checks the suspend count of the specified thread. A suspend count of zero indicates that the specified thread is not currently in the suspended mode. If the count is not zero, the count is decremented by one and if the resulting count value is zero, the thread is resumed. The API *Sleep (DWORD* *dwMilliseconds)* can be used for suspending a thread for the duration specified in milliseconds by the *Sleep* function. The *Sleep* call is initiated by the thread.

Example 2

Write a multithreaded application using Win32 APIs to set up a counter in the main thread and secondary thread to count from 0 to 10 and print the counts from both the threads. Put a delay of 500 ms in between the successive printing in both the threads.

```
#include "stdafx.h"
#include "windows.h"
#include "stdio.h"
//********************************************************************
//Child thread
//********************************************************************
void ChildThread(void)
{
 char i;
 for (i = 0; i <= 10; ++i)
 {
  printf("Executing Child Thread : Counter = %d\n", i);
  Sleep(500);
 }
}
//********************************************************************
//Primary thread
//********************************************************************
int main(int argc, char* argv[])
{
 HANDLE hThread;
 DWORD dwThreadID;
 char i;
 hThread = CreateThread(NULL, 1000, (LPTHREAD_START_ROUTINE)ChildThread,
                        NULL, 0, &dwThreadID);
 if (hThread == NULL)
 {
  printf("Thread Creation Failed\nError No : %d\n", GetLastError());
  return 1;
 }
 for (i = 0; i <= 10; ++i)
 {
  printf("Executing Main Thread : Counter = %d\n", i);
  Sleep(500);
 }
 return 0;
}
```

To execute this program, create a new Win32 Console Application with Microsoft Visual Studio using Visual C++ and add the above piece of code to it and compile. The output obtained on running this application on a machine with Windows 10 operating system is given in Fig. 10.9.

Fig. 10.9 Output of the Win32 Multithreaded application

If you examine the output, you can see the switching between main and child threads. The output need not be the same always. The output is purely dependent on the scheduling policies implemented by the windows operating system for thread scheduling. You may get the same output or a different output each time you run the application.

Java Threads Java threads are the threads supported by Java programming Language. The java thread class '*Thread*' is defined in the package '*java.lang*'. This package needs to be imported for using the thread creation functions supported by the Java thread class. There are two ways of creating threads in Java: Either by extending the base '*Thread*' class or by implementing an interface. Extending the thread class allows inheriting the methods and variables of the parent class (Thread class) only whereas interface allows a way to achieve the requirements for a set of classes. The following piece of code illustrates the implementation of Java threads with extending the thread base class '*Thread*'.

```
import java.lang.*;
public class MyThread extends Thread
{
	public void run()
	{
		System.out.println("Hello from MyThread!");
	}
	public static void main(String args[])
	{
		(new MyThread()).start();
	}
}
```

The above piece of code creates a new class *MyThread* by extending the base class *Thread*. It also overrides the *run()* method inherited from the base class with its own *run()* method. The *run()* method of *MyThread* implements all the task for the *MyThread* thread. The method *start()* moves the thread to a pool of threads waiting for their turn to be picked up for execution by the scheduler. The thread is said to be in the 'Ready' state at this stage. The scheduler picks the threads for execution from the pool based on the thread priorities.

```
E.g.  MyThread.start();
```

The output of the above piece of code when executed on Windows 10 platform is given in Fig. 10.10.

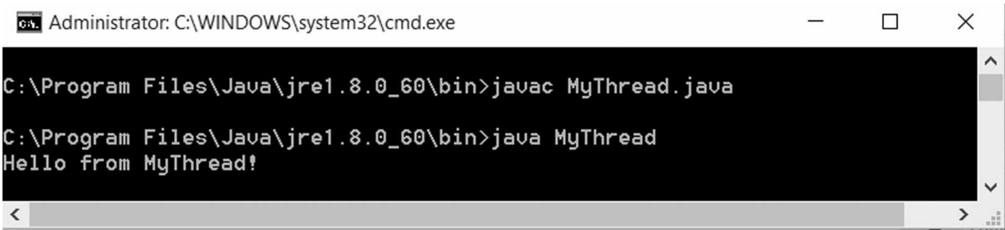


Fig. 10.10 Output of the Java Multithreaded application

Invoking the static method *yield()* voluntarily give up the execution of the thread and the thread is moved to the pool of threads waiting to get their turn for execution, i.e. the thread enters the 'Ready' state.

```
E.g.  MyThread.yield();
```

The static method *sleep()* forces the thread to sleep for the duration mentioned by the sleep call, i.e. the thread enters the 'Suspend' mode. Once the sleep period is expired, the thread is moved to the pool of threads waiting to get their turn for execution, i.e. the thread enters the 'Ready' state. The method *sleep()* only guarantees that the thread will sleep for the minimum period mentioned by the argument to the call. It will not guarantee anything on the resume of the thread after the sleep period. It is dependent on the scheduler.

```
E.g.  MyThread.sleep(100); Sleep for 100 milliseconds.
```

Calling a thread Object's *wait()* method causes the thread object to wait. The thread will remain in the 'Wait' state until another thread invokes the *notify()* or *notifyAll()* method of the thread object which is waiting. The thread enters the 'Blocked' state when waiting for input from I/O devices or waiting for object lock in case of accessing shared resources. The thread is moved to the 'Ready' state on receiving the I/O input or on acquiring the object lock. The thread enters the 'Finished/Dead' state on completion of the task assigned to it or when the *stop()* method is explicitly invoked. The thread may also enter this state if it is terminated by an unrecoverable error condition.

For more information on Java threads, visit Sun Micro System's tutorial on Threads, available at http://java.sun.com/tutorial/applet/overview/threads.html

Summary So far we discussed about the various thread classes available for creation and management of threads in a multithreaded system in a General Purpose Operating System's perspective. From an RTOS perspective, POSIX threads and Win32 threads are the most commonly used thread class libraries for thread creation and management. Many non-standard, proprietary thread classes are also used by some proprietary RTOS. Portable threads (*Pth*), a very portable POSIX/ANSI-C based library from GNU, may be the "next generation" threads library. *Pth* provides non-preemptive priority based scheduling for multiple threads inside event driven applications. Visit http://www.gnu.org/software/pth/ for more details on GNU Portable threads.

10.3.2.3 Thread Pre-emption

Thread pre-emption is the act of pre-empting the currently running thread (stopping the currently running thread temporarily). Thread pre-emption ability is solely dependent on the Operating System. Thread pre-emption is performed for sharing the CPU time among all the threads. The execution switching among threads are known as '*Thread context switching*'. Thread context switching is dependent on the Operating system's scheduler and the type of the thread. When we say 'Thread', it falls into any one of the following types.

User Level Thread User level threads do not have kernel/Operating System support and they exist solely in the running process. Even if a process contains multiple user level threads, the OS treats it as single thread and will not switch the execution among the different threads of it. It is the responsibility of the process to schedule each thread as and when required. In summary, user level threads of a process are non-preemptive at thread level from OS perspective.

Kernel/System Level Thread Kernel level threads are individual units of execution, which the OS treats as separate threads. The OS interrupts the execution of the currently running kernel thread and switches the execution to another kernel thread based on the scheduling policies implemented by the OS. In summary kernel level threads are pre-emptive.

For user level threads, the execution switching (thread context switching) happens only when the currently executing user level thread is voluntarily blocked. Hence, no OS intervention and system calls are involved in the context switching of user level threads. This makes context switching of user level threads very fast. On the other hand, kernel level threads involve lots of kernel overhead and involve system calls for context switching. However, kernel threads maintain a clear layer of abstraction and allow threads to use system calls independently. There are many ways for binding user level threads with system/kernel level threads. The following section gives an overview of various thread binding models.

Many-to-One Model Here many user level threads are mapped to a single kernel thread. In this model, the kernel treats all user level threads as single thread and the execution switching among the user level threads happens when a currently executing user level thread voluntarily blocks itself or relinquishes the CPU. Solaris Green threads and GNU Portable Threads are examples for this. The '*PThread*' example given under the POSIX thread library section is an illustrative example for application with Many-to-One thread model.

One-to-One Model In One-to-One model, each user level thread is bonded to a kernel/system level thread. Windows NT and Linux threads are examples for One-to-One thread models. The modified '*PThread*' example given under the '*Thread Pre-emption*' section is an illustrative example for application with One-to-One thread model.

Many-to-Many Model In this model many user level threads are allowed to be mapped to many kernel threads. Windows NT/2000 with *ThreadFibre* package is an example for this.

10.3.2.4 Thread v/s Process

I hope, by now you got a reasonably good knowledge of *process* and *threads*. Now let us summarise the properties of *process* and *threads*.

Thread	Process
Thread is a single unit of execution and is part of process.	Process is a program in execution and contains one or more threads.
A thread does not have its own data memory and heap memory. It shares the data memory and heap memory with other threads of the same process.	Process has its own code memory, data memory and stack memory.

A thread cannot live independently; it lives within the process.	A process contains at least one thread.
There can be multiple threads in a process. The first thread (main thread) calls the main function and occupies the start of the stack memory of the process.	Threads within a process share the code, data and heap memory. Each thread holds separate memory area for stack (shares the total stack memory of the process).
Threads are very inexpensive to create	Processes are very expensive to create. Involves many OS overhead.
Context switching is inexpensive and fast	Context switching is complex and involves lot of OS overhead and is comparatively slower.
If a thread expires, its stack is reclaimed by the process.	If a process dies, the resources allocated to it are reclaimed by the OS and all the associated threads of the process also dies.

10.4 MULTIPROCESSING AND MULTITASKING

LO 4 Understand the difference between multiprocessing and multitasking

The terms *multiprocessing* and *multitasking* are a little confusing and sounds alike. In the operating system context *multiprocessing* describes the ability to execute multiple processes simultaneously. Systems which are capable of performing multiprocessing, are known as *multiprocessor* systems. *Multiprocessor* systems possess multiple CPUs and can execute multiple processes simultaneously.

The ability of the operating system to have multiple programs in memory, which are ready for execution, is referred as *multiprogramming*. In a uniprocessor system, it is not possible to execute multiple processes simultaneously. However, it is possible for a uniprocessor system to achieve some degree of pseudo parallelism in the execution of multiple processes by switching the execution among different processes. The ability of an operating system to hold multiple processes in memory and switch the processor (CPU) from executing one process to another process is known as *multitasking*. Multitasking creates the illusion of multiple tasks executing in parallel. Multitasking involves the switching of CPU from executing one task to another. In an earlier section '*The Structure of a Process*' of this chapter, we learned that a Process is identical to the physical processor in the sense it has own register set which mirrors the CPU registers, stack and Program Counter (PC). Hence, a '*process*' is considered as a '*Virtual processor*', awaiting its turn to have its properties switched into the physical processor. In a multitasking environment, when task/process switching happens, the virtual processor (task/process) gets its properties converted into that of the physical processor. The switching of the virtual processor to physical processor is controlled by the scheduler of the OS kernel. Whenever a CPU switching happens, the current context of execution should be saved to retrieve it at a later point of time when the CPU executes the process, which is interrupted currently due to execution switching. The context saving and retrieval is essential for resuming a process exactly from the point where it was interrupted due to CPU switching. The act of switching CPU among the processes or changing the current execution context is known as '*Context switching*'. The act of saving the current context which contains the context details (Register details, memory details, system resource usage details, execution details, etc.) for the currently running process at the time of CPU switching is known as '*Context saving*'. The process of retrieving the saved context details for a process, which is going to be executed due to CPU switching, is known as '*Context retrieval*'. Multitasking involves '*Context switching*' (Fig. 10.11), '*Context saving*' and '*Context retrieval*'.

Toss Juggling The skilful object manipulation game is a classic real world example for the multitasking illusion. The juggler uses a number of objects (balls, rings, etc.) and throws them up and catches them. At any point of time, he throws only one ball and catches only one per hand. However, the speed at which he is switching the balls for throwing and catching creates the illusion, he is throwing and catching multiple balls or using more than two hands ☺ simultaneously, to the spectators.

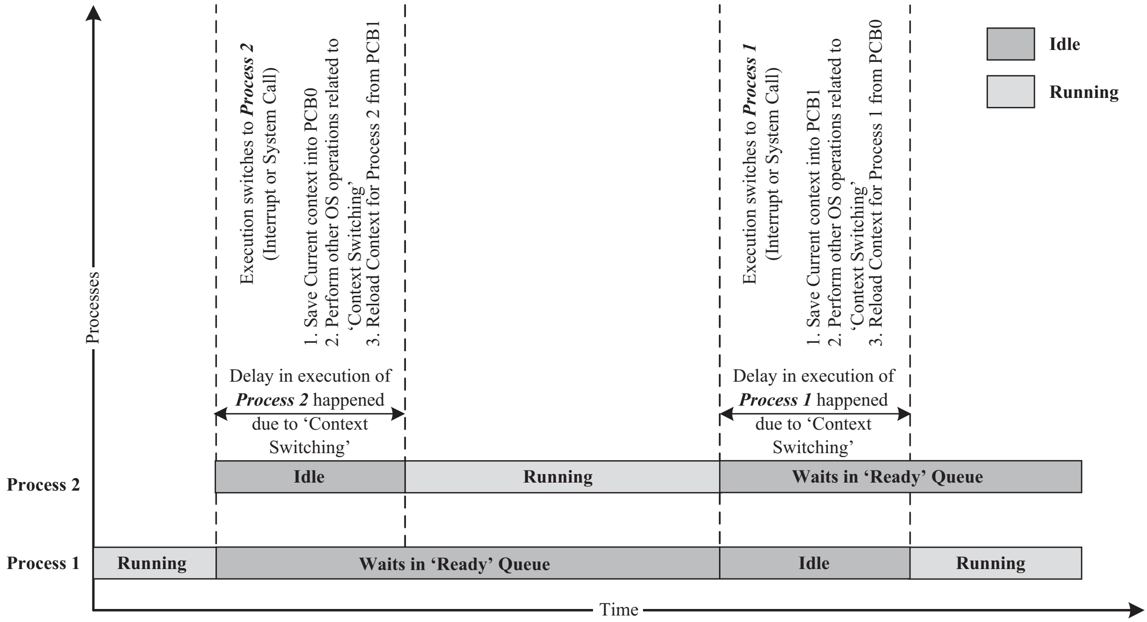


Fig. 10.11 Context switching

10.4.1 Types of Multitasking

As we discussed earlier, multitasking involves the switching of execution among multiple tasks. Depending on how the switching act is implemented, multitasking can be classified into different types. The following section describes the various types of multitasking existing in the Operating System's context.

10.4.1.1 Co-operative Multitasking

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

10.4.1.2 Preemptive Multitasking

Preemptive multitasking ensures that every task/process gets a chance to execute. When and how much time a process gets is dependent on the implementation of the preemptive scheduling. As the name indicates, in preemptive multitasking, the currently running task/process is preempted to give a chance to other tasks/process to execute. The preemption of task may be based on time slots or task/process priority.

10.4.1.3 Non-preemptive Multitasking

In non-preemptive multitasking, the process/task, which is currently given the CPU time, is allowed to execute until it terminates (enters the '*Completed*' state) or enters the '*Blocked/Wait*' state, waiting for an I/O or system resource. The co-operative and non-preemptive multitasking differs in their behaviour when they are in the '*Blocked/Wait*' state. In co-operative multitasking, the currently executing process/task need not relinquish the CPU when it enters the '*Blocked/Wait*' state, waiting for an I/O, or a shared resource access or an event to occur whereas in non-preemptive multitasking the currently executing task relinquishes the CPU when it waits for an I/O or system resource or an event to occur.

10.5 TASK SCHEDULING

LO 5 Describe the FCFS/FIFO, LCFS/LIFO, SJF and priority based task/process scheduling

As we already discussed, multitasking involves the execution switching among the different tasks. There should be some mechanism in place to share the CPU among the different tasks and to decide which process/task is to be executed at a given point of time. Determining which task/process is to be executed at a given point of time is known as task/process scheduling. Task scheduling forms the basis of multitasking. Scheduling policies forms the guidelines for determining which task is to be executed when. The scheduling policies are implemented in an algorithm and it is run by the kernel as a service. The kernel service/application, which implements the scheduling algorithm, is known as '*Scheduler*'. The process scheduling decision may take place when a process switches its state to

1. '*Ready*' state from '*Running*' state
2. '*Blocked/Wait*' state from '*Running*' state
3. '*Ready*' state from '*Blocked/Wait*' state
4. '*Completed*' state

A process switches to '*Ready*' state from the '*Running*' state when it is preempted. Hence, the type of scheduling in scenario 1 is pre-emptive. When a high priority process in the '*Blocked/Wait*' state completes its I/O and switches to the '*Ready*' state, the scheduler picks it for execution if the scheduling policy used is priority based preemptive. This is indicated by scenario 3. In preemptive/non-preemptive multitasking, the process relinquishes the CPU when it enters the '*Blocked/Wait*' state or the '*Completed*' state and switching of the CPU happens at this stage. Scheduling under scenario 2 can be either preemptive or non-preemptive. Scheduling under scenario 4 can be preemptive, non-preemptive or co-operative.

The selection of a scheduling criterion/algorithm should consider the following factors:

CPU Utilisation: The scheduling algorithm should always make the CPU utilisation high. CPU utilisation is a direct measure of how much percentage of the CPU is being utilised.

Throughput: This gives an indication of the number of processes executed per unit of time. The throughput for a good scheduler should always be higher.

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

Waiting Time: It is the amount of time spent by a process in the '*Ready*' queue waiting to get the CPU time for execution. The waiting time should be minimal for a good scheduling algorithm.

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

To summarise, a good scheduling algorithm has high CPU utilisation, minimum Turn Around Time (TAT), maximum throughput and least response time.

The Operating System maintains various queues† in connection with the CPU scheduling, and a process passes through these queues during the course of its admittance to execution completion.

The various queues maintained by OS in association with CPU scheduling are:

Job Queue: Job queue contains all the processes in the system

Ready Queue: Contains all the processes, which are ready for execution and waiting for CPU to get their turn for execution. The Ready queue is empty when there is no process ready for running.

Device Queue: Contains the set of processes, which are waiting for an I/O device.

A process migrates through all these queues during its journey from '*Admitted*' to '*Completed*' stage. The following diagrammatic representation (Fig. 10.12) illustrates the transition of a process through the various queues.

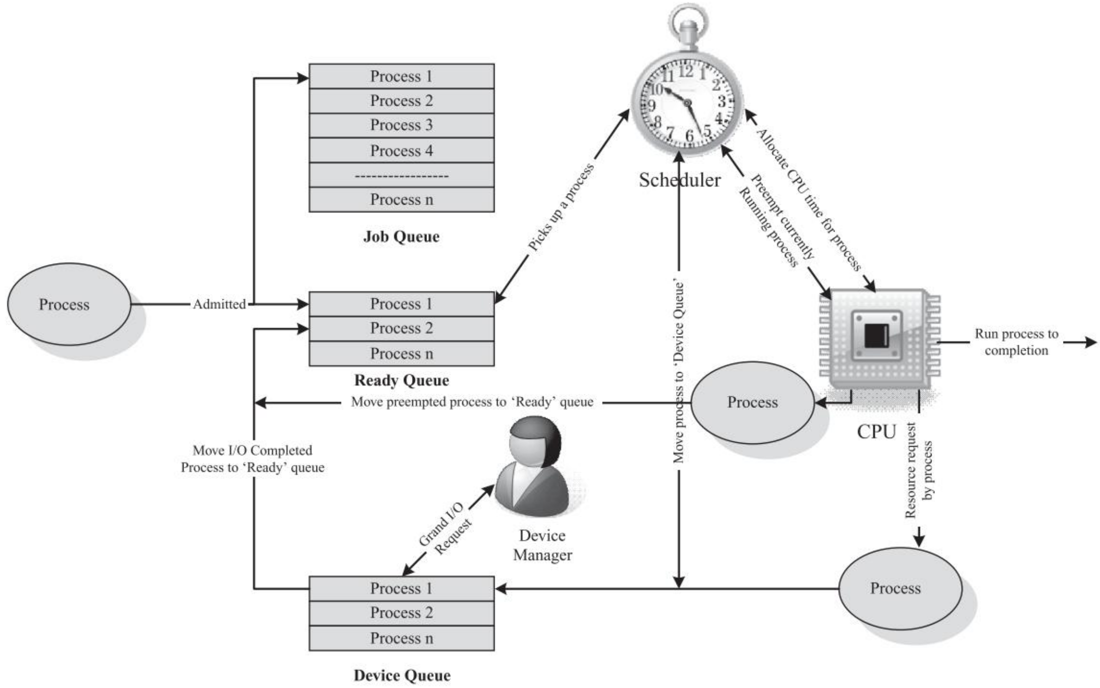


Fig. 10.12 Illustration of process transition through various queues

Based on the scheduling algorithm used, the scheduling can be classified into the following categories.

10.5.1 Non-preemptive Scheduling

Non-preemptive scheduling is employed in systems, which implement non-preemptive multitasking model. In this scheduling type, the currently executing task/process is allowed to run until it terminates or enters the '*Wait*' state waiting for an I/O or system resource. The various types of non-preemptive scheduling adopted in task/process scheduling are listed below.

† Queue is a special kind of arrangement of a collection of objects. In the operating system context queue is considered as a buffer.

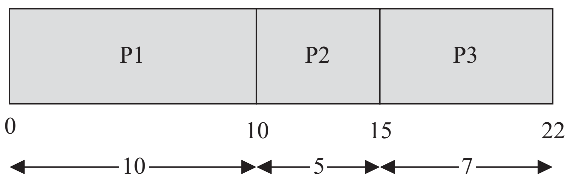
10.5.1.1 First-Come-First-Served (FCFS)/ FIFO Scheduling

As the name indicates, the First-Come-First-Served (FCFS) scheduling algorithm allocates CPU time to the processes based on the order in which they enter the '*Ready*' queue. The first entered process is serviced first. It is same as any real world application where queue systems are used; e.g. Ticketing reservation system where people need to stand in a queue and the first person standing in the queue is serviced first. FCFS scheduling is also known as First In First Out (FIFO) where the process which is put first into the '*Ready*' queue is serviced first.

Example 1

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

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



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

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

Average waiting time = (Waiting time for all processes) / No. of Processes
= (Waiting time for (P1+P2+P3)) / 3
= (0+10+15)/3 = 25/3
= 8.33 milliseconds

Turn Around Time (TAT) for P1 = 10 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P2 = 15 ms (-Do-)
Turn Around Time (TAT) for P3 = 22 ms (-Do-)

Average Turn Around Time = (Turn Around Time for all processes) / No. of Processes
= (Turn Around Time for (P1+P2+P3)) / 3
= (10+15+22)/3 = 47/3
= 15.66 milliseconds

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

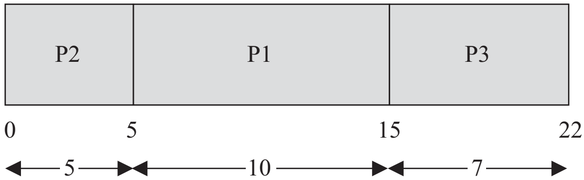
Average Execution Time = (Execution time for all processes)/No. of processes
= (Execution time for (P1+P2+P3))/3
= (10+5+7)/3 = 22/3
= 7.33

Average Turn Around Time = Average waiting time + Average execution time
= 8.33 + 7.33
= 15.66 milliseconds

Example 2

Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time (Assuming there is no I/O waiting for the processes) for the above example if the process enters the 'Ready' queue together in the order P2, P1, P3.

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



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

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

The Average waiting time and Turn Around Time (TAT) depends on the order in which the processes enter the '*Ready*' queue, regardless there estimated completion time.

From the above two examples it is clear that the Average waiting time and Turn Around Time improve if the process with shortest execution completion time is scheduled first.

The major drawback of FCFS algorithm is that it favours monopoly of process. A process, which does not contain any I/O operation, continues its execution until it finishes its task. If the process contains any I/O operation, the CPU is relinquished by the process. In general, FCFS favours CPU bound processes and I/O bound processes may have to wait until the completion of CPU bound process, if the currently executing process is a CPU bound process. This leads to poor device utilisation. The average waiting time is not minimal for FCFS scheduling algorithm.

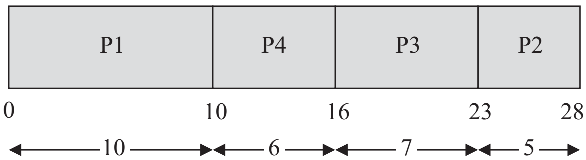
10.5.1.2 Last-Come-First Served (LCFS)/LIFO Scheduling

The Last-Come-First Served (LCFS) scheduling algorithm also allocates CPU time to the processes based on the order in which they are entered in the '*Ready*' queue. The last entered process is serviced first. LCFS scheduling is also known as Last In First Out (LIFO) where the process, which is put last into the '*Ready*' queue, is serviced first.

Example 1

Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds respectively enters the ready queue together in the order P1, P2, P3 (Assume only P1 is present in the '*Ready*' queue when the scheduler picks it up and P2, P3 entered '*Ready*' queue after that). Now a new process P4 with estimated completion time 6 ms enters the 'Ready' queue after 5 ms of scheduling P1. Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time (Assuming there is no I/O waiting for the processes). Assume all the processes contain only CPU operation and no I/O operations are involved.

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



The waiting time for all the processes is given as

Waiting Time for P1 = 0 ms (P1 starts executing first)

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

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

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

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

Turn Around Time (TAT) for P1 = 10 ms (Time spent in Ready Queue + Execution Time)

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

Turn Around Time (TAT) for P3 = 23 ms (Time spent in Ready Queue + Execution Time)

Turn Around Time (TAT) for P2 = 28 ms (Time spent in Ready Queue + Execution Time)

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

LCFS scheduling is not optimal and it also possesses the same drawback as that of FCFS algorithm.

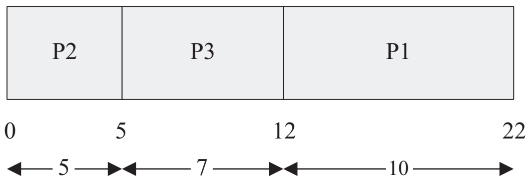
10.5.1.3 Shortest Job First (SJF) Scheduling

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

Example 1

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

The scheduler sorts the '*Ready*' queue based on the shortest estimated completion time and schedules the process with the least estimated completion time first and the next least one as second, and so on. The order in which the processes are scheduled for execution is represented as



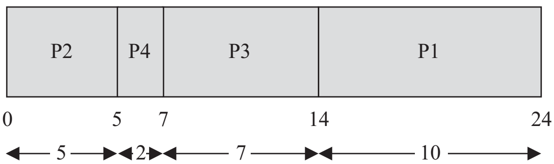
The estimated execution time of P2 is the least (5 ms) followed by P3 (7 ms) and P1 (10 ms).
The waiting time for all processes are given as
Waiting Time for P2 = 0 ms (P2 starts executing first)
Waiting Time for P3 = 5 ms (P3 starts executing after completing P2)
Waiting Time for P1 = 12 ms (P1 starts executing after completing P2 and P3)
Average waiting time = (Waiting time for all processes) / No. of Processes
= (Waiting time for (P2+P3+P1)) / 3
= (0+5+12)/3 = 17/3
= 5.66 milliseconds
Turn Around Time (TAT) for P2 = 5 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P3 = 12 ms (-Do-)
Turn Around Time (TAT) for P1 = 22 ms (-Do-)
Average Turn Around Time = (Turn Around Time for all processes) / No. of Processes
= (Turn Around Time for (P2+P3+P1)) / 3
= (5+12+22)/3 = 39/3
= 13 milliseconds
Average Turn Around Time (TAT) is the sum of average waiting time and average execution time.
The average Execution time = (Execution time for all processes)/No. of processes
= (Execution time for (P1+P2+P3))/3
= (10+5+7)/3 = 22/3 = 7.33
Average Turn Around Time = Average Waiting time + Average Execution time
= 5.66 + 7.33
= 13 milliseconds
From this example, it is clear that the average waiting time and turn around time is much improved with the SJF scheduling for the same processes when compared to the FCFS algorithm.

Example 2

Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time for the above example if a new process P4 with estimated completion time 2 ms enters the 'Ready' queue after 2 ms of execution of P2. Assume all the processes contain only CPU operation and no I/O operations are involved.

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

Now process P4 with estimated execution completion time 2 ms enters the '*Ready*' queue after 2 ms of start of execution of P2. Since the SJF algorithm is non-preemptive and process P2 does not contain any I/O operations, P2 continues its execution. After 5 ms of scheduling, P2 terminates and now the scheduler again sorts the '*Ready*' queue for process with least execution completion time. Since the execution completion time for P4 (2 ms) is less than that of P3 (7 ms), which was supposed to be run after the completion of P2 as per the '*Ready*' queue available at the beginning of execution scheduling, P4 is picked up for executing. Due to the arrival of the process P4 with execution time 2 ms, the '*Ready*' queue is re-sorted in the order P2, P4, P3, P1. At the beginning it was P2, P3, P1. The execution sequence now changes as per the following diagram



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

The average waiting time for a given set of process is minimal in SJF scheduling and so it is optimal compared to other non-preemptive scheduling like FCFS. The major drawback of SJF algorithm is that a process whose estimated execution completion time is high may not get a chance to execute if more and more processes with least estimated execution time enters the '*Ready*' queue before the process with longest estimated execution time started its execution (In non-preemptive SJF). This condition is known as '*Starvation*'. Another drawback of SJF is that it is difficult to know in advance the next shortest process in the '*Ready*' queue for scheduling since new processes with different estimated execution time keep entering the '*Ready*' queue at any point of time.

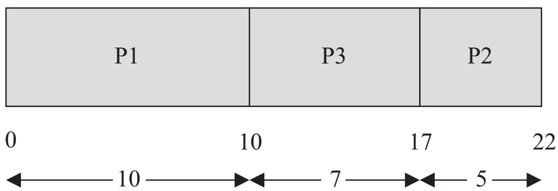
10.5.1.4 Priority Based Scheduling

The Turn Around Time (TAT) and waiting time for processes in non-preemptive scheduling varies with the type of scheduling algorithm. Priority based non-preemptive scheduling algorithm ensures that a process with high priority is serviced at the earliest compared to other low priority processes in the 'Ready' queue. The priority of a task/process can be indicated through various mechanisms. The Shortest Job First (SJF) algorithm can be viewed as a priority based scheduling where each task is prioritised in the order of the time required to complete the task. The lower the time required for completing a process the higher is its priority in SJF algorithm. Another way of priority assigning is associating a priority to the task/process at the time of creation of the task/process. The priority is a number ranging from 0 to the maximum priority supported by the OS. The maximum level of priority is OS dependent. For Example, Windows CE supports 256 levels of priority (0 to 255 priority numbers). While creating the process/task, the priority can be assigned to it. The priority number associated with a task/process is the direct indication of its priority. The priority variation from high to low is represented by numbers from 0 to the maximum priority or by numbers from maximum priority to 0. For Windows CE operating system a priority number 0 indicates the highest priority and 255 indicates the lowest priority. This convention need not be universal and it depends on the kernel level implementation of the priority structure. The non-preemptive priority based scheduler sorts the 'Ready' queue based on priority and picks the process with the highest level of priority for execution.

Example 1

Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds and priorities 0, 3, 2 (0—highest priority, 3—lowest priority) respectively enters the ready queue together. Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time (Assuming there is no I/O waiting for the processes) in priority based scheduling algorithm.

The scheduler sorts the '*Ready*' queue based on the priority and schedules the process with the highest priority (P1 with priority number 0) first and the next high priority process (P3 with priority number 2) as second, and so on. The order in which the processes are scheduled for execution is represented as



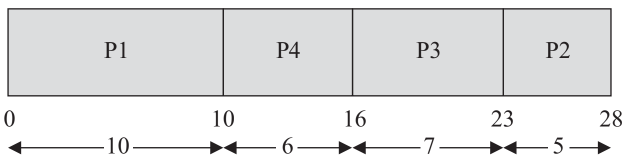
The waiting time for all the processes are given as
Waiting time for P1 = 0 ms (P1 starts executing first)
Waiting time for P3 = 10 ms (P3 starts executing after completing P1)
Waiting time for P2 = 17 ms (P2 starts executing after completing P1 and P3)
Average waiting time = (Waiting time for all processes)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

= (Turn Around Time for (P1+P3+P2)) / 3
= (10+17+22)/3 = 49/3
= 16.33 milliseconds

Example 2

Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time for the above example if a new process P4 with estimated completion time 6 ms and priority 1 enters the 'Ready' queue after 5 ms of execution of P1. Assume all the processes contain only CPU operation and no I/O operations are involved.

At the beginning, there are only three processes (P1, P2 and P3) available in the '*Ready*' queue and the scheduler picks up the process with the highest priority (In this example P1 with priority 0) for scheduling. The execution sequence diagram for this is same as that of Example 1. Now process P4 with estimated execution completion time 6 ms and priority 1 enters the '*Ready*' queue after 5 ms of execution of P1. Since the scheduling algorithm is non-preemptive and process P1 does not contain any I/O operations, P1 continues its execution. After 10 ms of scheduling, P1 terminates and now the scheduler again sorts the '*Ready*' queue for process with highest priority. Since the priority for P4 (priority 1) is higher than that of P3 (priority 2), which was supposed to be run after the completion of P1 as per the '*Ready*' queue available at the beginning of execution scheduling, P4 is picked up for executing. Due to the arrival of the process P4 with priority 1, the '*Ready*' queue is resorted in the order P1, P4, P3, P2. At the beginning it was P1, P3, P2. The execution sequence now changes as per the following diagram



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

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

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

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

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

10.5.2 Preemptive Scheduling

Preemptive scheduling is employed in systems, which implements preemptive multitasking model. In preemptive scheduling, every task in the '*Ready*' queue gets a chance to execute. When and how often each process gets a chance to execute (gets the CPU time) is dependent on the type of preemptive scheduling algorithm used for scheduling the processes. In this kind of scheduling, the scheduler can preempt (stop temporarily) the currently executing task/process and select another task from the '*Ready*' queue for execution. When to pre-empt a task and which task is to be picked up from the '*Ready*' queue for execution after preempting the current task is purely dependent on the scheduling algorithm. A task which is preempted by the scheduler is moved to the '*Ready*' queue. The act of moving a '*Running*' process/task into the '*Ready*' queue by the scheduler, without the processes requesting for it is known as '*Preemption*'. Preemptive scheduling can be implemented in different approaches. The two important approaches adopted in preemptive scheduling are time-based preemption and priority-based preemption. The various types of preemptive scheduling adopted in task/process scheduling are explained below.

10.5.2.1 Preemptive SJF Scheduling/Shortest Remaining Time (SRT)

The non-preemptive SJF scheduling algorithm sorts the 'Ready' queue only after completing the execution of the current process or when the process enters 'Wait' state, whereas the preemptive SJF scheduling algorithm sorts the 'Ready' queue when a new process enters the 'Ready' queue and checks whether the execution time of the new process is shorter than the remaining of the total estimated time for the currently executing process. If the execution time of the new process is less, the currently executing process is preempted and the new process is scheduled for execution. Thus preemptive SJF scheduling always compares the execution completion time (It is same as the remaining time for the new process) of a new process entered the 'Ready' queue with the remaining time for completion of the currently executing process and schedules the process with shortest remaining time for execution. Preemptive SJF scheduling is also known as Shortest Remaining Time (SRT) scheduling.

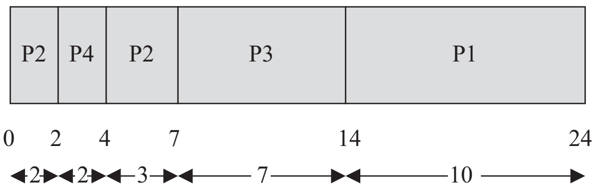
Now let us solve Example 2 given under the Non-preemptive SJF scheduling for preemptive SJF scheduling. The problem statement and solution is explained in the following example.

Example 1

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

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

Now process P4 with estimated execution completion time 2 ms enters the '*Ready*' queue after 2 ms of start of execution of P2. Since the SRT algorithm is preemptive, the remaining time for completion of process P2 is checked with the remaining time for completion of process P4. The remaining time for completion of P2 is 3 ms which is greater than that of the remaining time for completion of the newly entered process P4 (2 ms). Hence P2 is preempted and P4 is scheduled for execution. P4 continues its execution to finish since there is no new process entered in the 'Ready' queue during its execution. After 2 ms of scheduling P4 terminates and now the scheduler again sorts the '*Ready*' queue based on the remaining time for completion of the processes present in the 'Ready' queue. Since the remaining time for P2 (3 ms), which is preempted by P4 is less than that of the remaining time for other processes in the 'Ready' queue, P2 is scheduled for execution. Due to the arrival of the process P4 with execution time 2 ms, the '*Ready*' queue is re-sorted in the order P2, P4, P2, P3, P1. At the beginning it was P2, P3, P1. The execution sequence now changes as per the following diagram



The waiting time for all the processes are given as

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

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

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

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

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

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

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

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

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

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

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

Average Waiting Time in non-preemptive SJF scheduling = 6 ms
Average Waiting Time in preemptive SJF scheduling = 5.75 ms

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

This reveals that the Average waiting Time and Turn Around Time (TAT) improves significantly with preemptive SJF scheduling.

10.5.2.2 Round Robin (RR) Scheduling

The term *Round Robin* is very popular among the sports and games activities. You might have heard about 'Round Robin' league or 'Knock out' league associated with any football or cricket tournament. In the 'Round Robin' league each team in a group gets an equal chance to play against the rest of the teams in the same group whereas in the 'Knock out' league the losing team in a match moves out of the tournament ☺.

In the process scheduling context also, '*Round Robin*' brings the same message "Equal chance to all". In Round Robin scheduling, each process in the 'Ready' queue is executed for a pre-defined time slot. The execution starts with picking up the first process in the 'Ready' queue (see Fig. 10.13). It is executed for a pre-defined time and when the pre-defined time elapses or the process completes (before the pre-defined time slice), the next process in the 'Ready' queue is selected for execution. This is repeated for all the processes in the 'Ready' queue. Once each process in the 'Ready' queue is executed for the pre-defined time period, the scheduler comes back and picks the first process in the 'Ready' queue again for execution. The sequence is repeated. This reveals that the Round Robin scheduling is similar to the FCFS scheduling and the only difference is that a time slice based preemption is added to switch the execution between the processes in the 'Ready' queue. The 'Ready' queue can be considered as a circular queue in which the scheduler picks up the first process for execution and moves to the next till the end of the queue and then comes back to the beginning of the queue to pick up the first process.

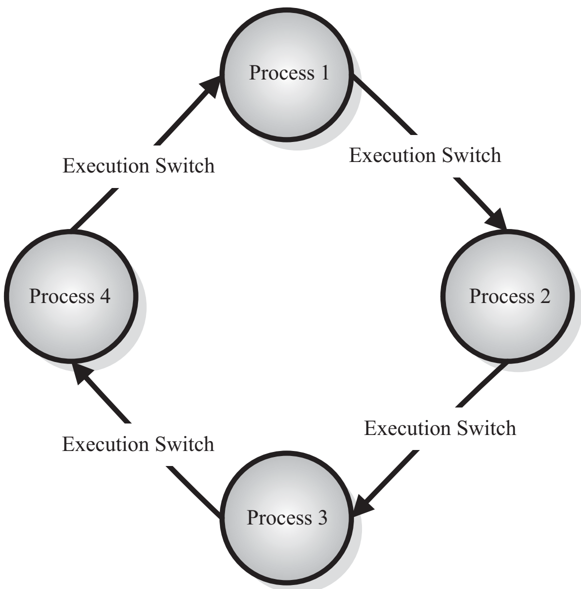


Fig. 10.13 Round Robin Scheduling

The time slice is provided by the *timer tick* feature of the time management unit of the OS kernel (Refer the Time management section under the subtopic '*The Real-Time kernel*' for more details on Timer tick). Time slice is kernel dependent and it varies in the order of a few microseconds to milliseconds. Certain OS kernels may allow the time slice as user configurable. Round Robin scheduling ensures that every process gets a fixed amount of CPU time for execution. When the process gets its fixed time for execution is determined by the

FCFS policy (That is, a process entering the Ready queue first gets its fixed execution time first and so on…). If a process terminates before the elapse of the time slice, the process releases the CPU voluntarily and the next process in the queue is scheduled for execution by the scheduler. The implementation of RR scheduling is kernel dependent. The following code snippet illustrates the RR scheduling implementation for RTX51 Tiny OS, an 8bit OS for *8051* microcontroller from Keil Software (www.keil.com), an ARM® Company.

```
#include <rtx51tny.h>    /* Definitions for RTX51 Tiny */
int counter0;
int counter1;

job0 () _task_ 0 {
  os_create_task (1);    /* Mark task 1 as "ready"       */

  while (1)  {           /* Endless loop                 */
    counter0++;          /* Increment counter 0          */
  }
}
job1 () _task_ 1 {
  while (1)  {           /* Endless loop                 */
    counter1++;          /* Increment counter 1          */
  }
}
```

RTX51 defines the tasks as simple C functions with void return type and void argument list. The attribute *_task_* is used for declaring a function as task. The general form of declaring a task is

```
     void func (void) _task_ task_id
```

where *func* is the name of the task and *task_id* is the ID of the task. RTX51 supports up to 16 tasks and so *task_id* varies from 0 to 15. All tasks should be implemented as endless loops.

The two tasks in this program are counter loops. RTX51 Tiny starts executing task 0 which is the function named job0. This function creates another task called job1. After job0 executes for its time slice, RTX51 Tiny switches to job1. After job1 executes for its time slice, RTX51 Tiny switches back to job0. This process is repeated forever.

Now let's check how the RTX51 Tiny RR Scheduling can be implemented in an embedded device (A smart card reader) which addresses the following requirements.

- Check the presence of a card
- Process the data received from the card
- Update the Display
- Check the serial port for command/data
- Process the data received from serial port

These four requirements can be considered as four tasks. Implement them as four RTX51 tasks as explained below.

```
void check_card_task (void) _task_ 1
{
/* This task checks for the presence of a card */
/* Implement the necessary functionality here */
}
```

```
void process_card_task (void) _task_ 2
{
/* This task processes the data received from the card */
/* Implement the necessary functionality here */
}
void check_serial_io_task (void) _task_ 3
{
/* This task checks for serial I/O */
/* Implement the necessary functionality here */
}
void process_serial_data_task (void) _task_ 4
{
/* This task processes the data received from the serial port */
/* Implement the necessary functionality here */
}
```

Now the tasks are created. Next step is scheduling the tasks. The following code snippet illustrates the scheduling of tasks.

```
void startup_task (void) _task_ 0
{
os_create_task (1);    /* Create check_card_task Task */
os_create_task (2);    /* Create process_card_task Task */
os_create_task (3);    /* Create serial_io_task Task */

os_create_task (4);    /* Create serial_data_task Task */
os_delete_task (0);    /* Delete the Startup Task */
}
```

The *os_create_task* (task_ID) RTX51 Tiny kernel call puts the task with task ID *task_ID* in the 'Ready' state. All the ready tasks begin their execution at the next available opportunity. RTX51 Tiny does not have a *main ()* function to begin the code execution; instead it starts with executing task 0. Task 0 is used for creating other tasks. Once all the tasks are created, task 0 is stopped and removed from the task list with the *os_delete_task* kernel call. The RR scheduler selects each task based on the time slice and continues the execution. If we observe the tasks we can see that there is no point in executing the task *process_card_task* (Task 2) without detecting a card and executing the task *process_serial_data_task* (Task 4) without receiving some data in the serial port. In summary task 2 needs to be executed only when task 1 reports the presence of a card and task 4 needs to be executed only when task 3 reports the arrival of data at serial port. So these tasks (tasks 2 and 4) need to be put in the 'Ready' state only on satisfying these conditions. Till then these tasks can be put in the 'Wait' state so that the RR scheduler will not pick them for scheduling and the RR scheduling is effectively utilised among the other tasks. This can be achieved by implementing the wait and notify mechanism in the related tasks. Task 2 can be coded in a way that it waits for the card present event and task 1 signals the event 'card detected'. In a similar fashion Task 4 can be coded in such a way that it waits for the serial data received event and task 3 signals the reception of serial data on receiving serial data from serial port. The following code snippet explains the same.

```
void check_card_task (void) _task_ 1
{
/* This task checks for the presence of a card */
```

```
/* Implement the necessary functionality here */
while (1)
{
//Function for checking the presence of card and card reading
//……………………………
if (card is present)
//Signal card detected to task 2
os_send_signal (2)
}
}

void process_card_task (void) _task_ 2
{
/* This task processes the data received from the card */
/* Implement the necessary functionality here */
while (1)
{
//Function for checking the signaling of card present event
os_wait1(K_SIG);
//Process card data
}
}

void check_serial_io_task (void) _task_ 3
{
/* This task checks for serial I/O */
/* Implement the necessary functionality here */
while (1)
{
//Function for checking the reception of serial data
//……………………………
if (data is received)
//Signal serial data reception to task 4
os_send_signal (4)
}
}

void process_serial_data_task (void) _task_ 4
{
/* This task processes the data received from the serial port */
/* Implement the necessary functionality here */
while (1)
{
//Function for checking the signaling of serial data received event
os_wait1(K_SIG);
//Process card data
}
}
```

The *os_send_signal* (*Task ID*) kernel call sends a signal to task *Task ID*. If the specified task is already waiting for a signal, this function call readies the task for execution but does not start it. The *os_wait1* (*event*) kernel call halts the current task and waits for an event to occur. The *event* argument specifies the event to wait for and may have only the value K_SIG which waits for a signal. RTX51 uses the Timer 0 of *8051* for time slice generation. The time slice can be configured by the user by changing the time slice related parameters in the RTX51 Tiny OS configuration file **CONF_TNY.A51** file which is located in the **\Keil_v5\C51\RtxTiny2\SourceCode** folder. Configuration options in **CONF_TNY.A51** allow users to:

- Specify the Timer Tick Interrupt Register Bank.
- Specify the Timer Tick Interval (in *8051* machine cycles).
- Specify user code to execute in the Timer Tick Interrupt.
- Specify the Round-Robin Timeout.
- Enable or disable Round-Robin Task Switching.
- Specify that your application includes long duration interrupts.
- Specify whether or not code banking is used.
- Define the top of the RTX51 Tiny stack.
- Specify the minimum stack space required.
- Specify code to execute in the event of a stack error.
- Define idle task operations.

The RTX51 kernel provides a set of task management functions for managing the tasks. At any point of time each RTX51 task is exactly in any one of the following state.

Task State	State Description
RUNNING	The task that is currently running is in the RUNNING State. Only one task at a time may be in this state. The *os_running_task_id* kernel call returns the task number (ID) of the currently executing task.
READY	Tasks which are ready to run are in the READY State. Once the Running task has completed processing, RTX51 Tiny selects and starts the next Ready task. A task may be made ready immediately (even if the task is waiting for a timeout or signal) by setting its ready flag using the *os_set_ready or isr_set_ready* kernel functions.
WAITING	Tasks which are waiting for an event are in the WAITING State. Once the event occurs, the task is switched to the READY State. The *os_wait function* is used for placing a task in the WAITING State.
DELETED	Tasks which have not been started or tasks which have been deleted are in the DELETED State. The *os_delete_task* routine places a task that has been started (with *os_create_task*) into the DELETED State.
TIME-OUT	Tasks which were interrupted by a Round-Robin Time-Out are in the TIME-OUT State. This state is equivalent to the READY State for Round-Robin programs.

Refer the documentation available with RTX51 Tiny OS for more information on the various RTX51 task management kernel functions and their usage.

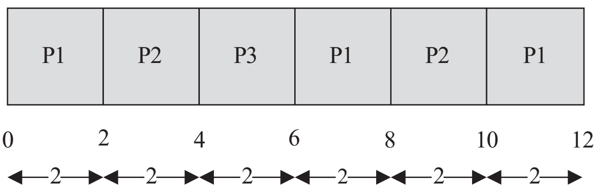
RR scheduling with interrupts is a good choice for the design of comparatively less complex *Real-Time Embedded Systems*. In this approach, the tasks which require less *Real-Time* attention can be scheduled with Round Robin scheduling and the tasks which require *Real-Time* attention can be scheduled through Interrupt Service Routines. RTX51 Tiny supports Interrupts with RR scheduling. For RTX51 the time slice for RR scheduling is provided by the Timer interrupt and if the interrupt is of high priority than that of the timer interrupt and if its service time (ISR) is longer than the timer tick interval, the RTX51 timer interrupt may be interrupted by the ISR and it may be reentered by a subsequent RX51 Tiny timer interrupt. Hence proper care must be taken to limit the ISR time within the timer tick interval or to protect the timer tick interrupt code from reentrancy. Otherwise unexpected results may occur. The limitations of RR with interrupt generic approach are the limited number of interrupts supported by embedded processors and the interrupt latency happening due to the context switching overhead.

RR can also be used as technique for resolving the priority in scheduling among the tasks with same level of priority. We will discuss about how RR scheduling can be used for resolving the priority among equal tasks under the VxWorks kernel in a later chapter.

Example 1

Three processes with process IDs P1, P2, P3 with estimated completion time 6, 4, 2 milliseconds respectively, enters the ready queue together in the order P1, P2, P3. Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time (Assuming there is no I/O waiting for the processes) in RR algorithm with Time slice = 2 ms.

The scheduler sorts the '*Ready*' queue based on the FCFS policy and picks up the first process P1 from the 'Ready' queue and executes it for the time slice 2 ms. When the time slice is expired, P1 is preempted and P2 is scheduled for execution. The Time slice expires after 2ms of execution of P2. Now P2 is preempted and P3 is picked up for execution. P3 completes its execution within the time slice and the scheduler picks P1 again for execution for the next time slice. This procedure is repeated till all the processes are serviced. The order in which the processes are scheduled for execution is represented as



The waiting time for all the processes are given as

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

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

Waiting time for P3 = (4 – 0) = 4 ms
(P3 starts executing after completing the first time slices for P1 and P2 and completes its execution in a single time slice)

Average waiting time = (Waiting time for all the processes) / No. of Processes
= (Waiting time for (P1 + P2 + P3)) / 3
= (6 + 6 + 4)/3 = 16/3
= 5.33 milliseconds

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

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

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

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

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

RR scheduling involves lot of overhead in maintaining the time slice information for every process which is currently being executed.

10.5.2.3 Priority Based Scheduling

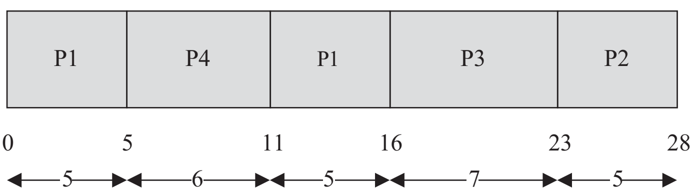
Priority based preemptive scheduling algorithm is same as that of the non-preemptive priority based scheduling except for the switching of execution between tasks. In preemptive scheduling, any high priority process entering the 'Ready' queue is immediately scheduled for execution whereas in the non-preemptive scheduling any high priority process entering the 'Ready' queue is scheduled only after the currently executing process completes its execution or only when it voluntarily relinquishes the CPU. The priority of a task/process in preemptive scheduling is indicated in the same way as that of the mechanism adopted for non-preemptive multitasking. Refer the non-preemptive priority based scheduling discussed in an earlier section of this chapter for more details.

Example 1

Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds and priorities 1, 3, 2 (0—highest priority, 3—lowest priority) respectively enters the ready queue together. A new process P4 with estimated completion time 6 ms and priority 0 enters the 'Ready' queue after 5 ms of start of execution of P1. Assume all the processes contain only CPU operation and no I/O operations are involved.

At the beginning, there are only three processes (P1, P2 and P3) available in the '*Ready*' queue and the scheduler picks up the process with the highest priority (In this example P1 with priority 1) for scheduling.

Now process P4 with estimated execution completion time 6 ms and priority 0 enters the '*Ready*' queue after 5 ms of start of execution of P1. Since the scheduling algorithm is preemptive, P1 is preempted by P4 and P4 runs to completion. After 6 ms of scheduling, P4 terminates and now the scheduler again sorts the '*Ready*' queue for process with highest priority. Since the priority for P1 (priority 1), which is preempted by P4 is higher than that of P3 (priority 2) and P2 ((priority 3), P1 is again picked up for execution by the scheduler. Due to the arrival of the process P4 with priority 0, the '*Ready*' queue is resorted in the order P1, P4, P1, P3, P2. At the beginning it was P1, P3, P2. The execution sequence now changes as per the following diagram



The waiting time for all the processes are given as

Waiting time for P1 = 0 + (11 – 5) = 0 + 6 = 6 ms

(P1 starts executing first and gets preempted by P4 after 5 ms and again gets the CPU time after completion of P4)

Waiting time for P4 = 0 ms

(P4 starts executing immediately on entering the 'Ready' queue, by preempting P1)

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

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

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

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

= (6 + 0 + 16 + 23)/4 = 45/4

= 11.25 milliseconds

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

Turn Around Time (TAT) for P4 = 6 ms

(Time spent in Ready Queue + Execution Time = (Execution Start Time – Arrival Time) + Estimated Execution Time = (5 – 5) + 6 = 0 + 6)

Turn Around Time (TAT) for P3 = 23 ms (Time spent in Ready Queue + Execution Time)

Turn Around Time (TAT) for P2 = 28 ms (Time spent in Ready Queue + Execution Time)

Average Turn Around Time = (Turn Around Time for all the processes) / No. of Processes

= (Turn Around Time for (P2 + P4 + P3 + P1)) / 4

= (16 + 6 + 23 + 28)/4 = 73/4

= 18.25 milliseconds

Priority based preemptive scheduling gives Real-Time attention to high priority tasks. Thus priority based preemptive scheduling is adopted in systems which demands 'Real-Time' behaviour. Most of the RTOSs make use of the preemptive priority based scheduling algorithm for process scheduling. Preemptive priority based scheduling also possesses the same drawback of non-preemptive priority based scheduling– '*Starvation*'. This can be eliminated by the '*Aging*' technique. Refer the section Non-preemptive priority based scheduling for more details on '*Starvation*' and '*Aging*'.

10.6 THREADS, PROCESSES AND SCHEDULING: PUTTING THEM ALTOGETHER

LO 6 Explain the different Inter Process Communication (IPC) mechanisms used by tasks/process to communicate and co-operate each other in a multitasking environment

So far we discussed about threads, processes and process/thread scheduling. Now let us have a look at how these entities are addressed in a real world implementation. Let's examine the following pieces of code.

```
//*******************************************************************
//Process 1
//*******************************************************************
#include "stdafx.h"
#include <windows.h>
#include <stdio.h>
//*******************************************************************
//Thread for executing Task
//*******************************************************************
void Task(void) {
while (1)
```

```
{
  //Perform some task
  //Task execution time is 7.5 units of execution
  //Sleep for 17.5 units of execution
  Sleep(17.5); //Parameter given is not in milliseconds
               //Repeat task
 }
}
//*******************************************************************
//Main Thread.
//*******************************************************************
void main(void) {
 DWORD id;
 HANDLE hThread;
 //Create thread with normal priority
 //*******************************************************************
 hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Task,
                       (LPVOID)0, 0, &id);
 if (NULL == hThread)
 {//Thread Creation failed. Exit process
  printf("Creating thread failed : Error Code =   %d", GetLastError());
  return;
 }
 WaitForSingleObject(hThread, INFINITE);
 return;
}
//*******************************************************************
//Process 2
//*******************************************************************
#include "stdafx.h"
#include <windows.h>
#include <stdio.h>
//*******************************************************************
//Thread for executing Task
//*******************************************************************
void Task(void) {
 while (1)
 {
  //Perform some task
  ///Task execution time is 10 units of execution
  //Sleep for 5 units of execution
  Sleep(5); //Parameter given is not in milliseconds
            //Repeat task
 }
}
//*******************************************************************
//Main Thread.
```

```
//****************************************************************
void main(void) {
 DWORD id;
 HANDLE hThread;
 //Create thread with above normal priority
 //****************************************************************
 hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Task,
                       (LPVOID)0, CREATE_SUSPENDED, &id);
 if (NULL == hThread)
 {//Thread Creation failed. Exit process
  printf("Creating thread failed : Error Code = %d", GetLastError());
  return;
 }
 SetThreadPriority(hThread, THREAD_PRIORITY_ABOVE_NORMAL);
 ResumeThread(hThread);
 WaitForSingleObject(hThread, INFINITE);
 return;
}
```

The first piece of code represents a process (Process 1) with priority normal and it performs a task which requires 7.5 units of execution time. After performing this task, the process sleeps for 17.5 units of execution time and this is repeated forever. The second piece of code represents a process (Process 2) with priority above normal and it performs a task which requires 10 units of execution time. After performing this task, the process sleeps for 5 units of execution time and this is repeated forever. Process 2 is of higher priority compared to process 1, since its priority is above 'Normal'.

Now let us examine what happens if these processes are executed on a Real-Time kernel with pre-emptive priority based scheduling policy. Imagine Process 1 and Process 2 are ready for execution. Both of them enters the 'Ready' queue and the scheduler picks up Process 2 for execution since it is of higher priority (Assuming there is no other process running/ready for execution, when both the processes are 'Ready' for execution) compared to Process 1. Process 2 starts executing and runs until it executes the Sleep instruction (i.e. after10 units of execution time). When the Sleep instruction is executed, Process 2 enters the wait state. Since Process 1 is waiting for its turn in the 'Ready' queue, the scheduler picks up it for execution, resulting in a context switch. The Process Control Block (PCB) of Process 2 is updated with the values of the Program Counter (PC), stack pointer, etc. at the time of context switch. The estimated task execution time for Process 1 is 7.5 units of execution time and the sleeping time for Process 2 is 5 units of execution. After 5 units of execution time, Process 2 enters the 'Ready' state and moves to the 'Ready' queue. Since it is of higher priority compared to the running process, the running process (Process 1) is pre-empted and Process 2 is scheduled for execution. Process 1 is moved to the 'Ready' queue, resulting in context switching. The Process Control Block of Process 1 is updated with the current values of the Program Counter (PC), Stack pointer, etc. when the context switch is happened. The Program Counter (PC), Stack pointer, etc. for Process 2 is loaded with the values stored in the Process Control Block (PCB) of Process 2 and Process 2 continues its execution form where it was stopped earlier. Process 2 executes the Sleep instruction after 10 units of execution time and enters the wait state. At this point Process 1 is waiting in the 'Ready' queue and it requires 2.5 units of execution time for completing the task associated with it (The total time for completing the task is 7.5 units of time, out of this it has already completed 5 units of execution when Process 2 was in the wait state). The scheduler schedules Process 1 for execution. The Program Counter (PC), Stack pointer, etc. for Process 1 is loaded with the values stored in the Process Control Block (PCB) of Process 1 and Process 1 continues its execution form where it was stopped earlier. After 2.5 units of execution time, Process 1 executes the Sleep instruction and enters the wait state. Process 2 is already in the wait state and the scheduler finds no other process for scheduling. In order to keep the CPU always busy, the scheduler runs a dummy process (task) called '*IDLE PROCESS (TASK)*'. The '*IDLE PROCESS (TASK)*' executes some dummy task and keeps the CPU engaged. The execution diagram depicted in Fig. 10.14 explains the sequence of operations.

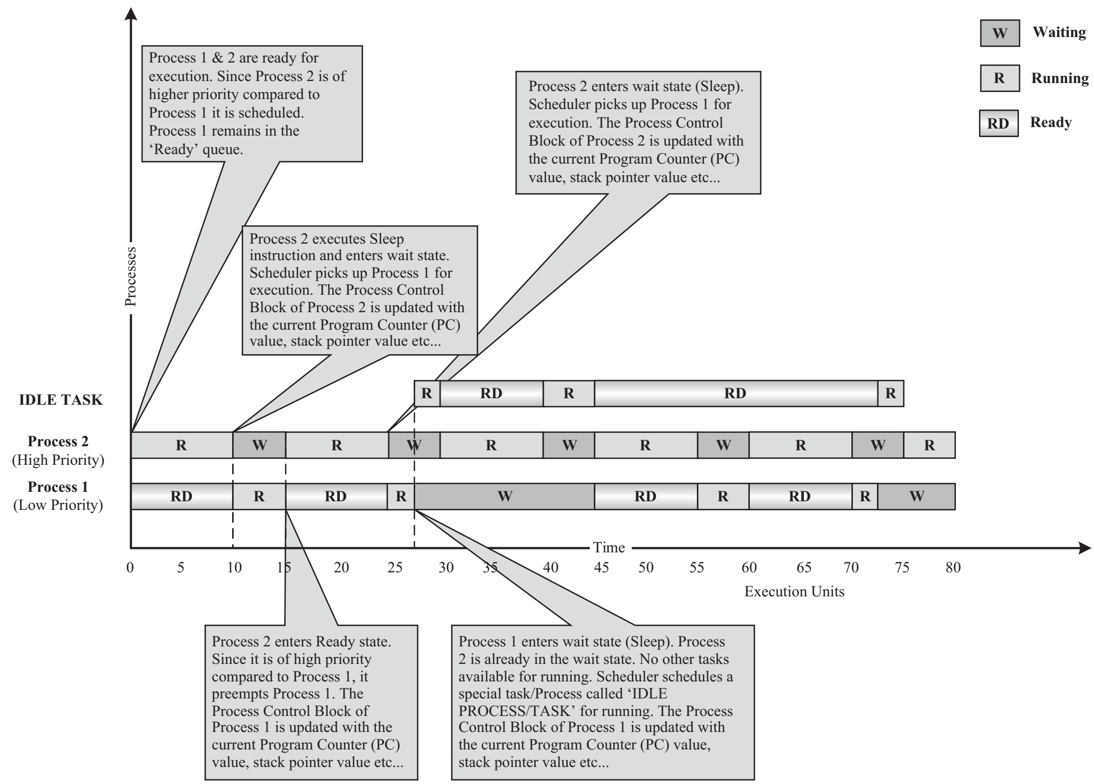


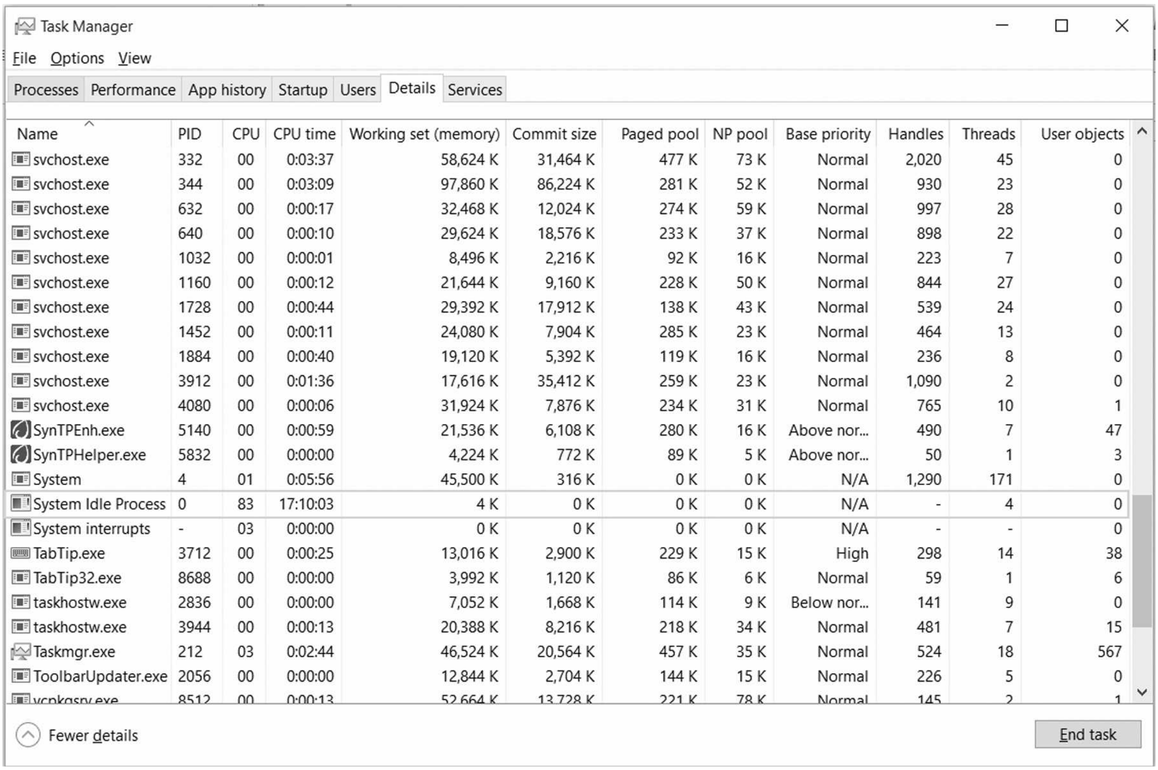
Fig. 10.14 Process scheduling and context switch

The implementation of the '*IDLE PROCESS (TASK)*' is dependent on the kernel and a typical implementation for a desktop OS may look like. It is simply an endless loop.

```
void Idle_Process (void)
{
//Simply wait.
//Do nothing...
while(1);
}
```

The Real-Time kernels deployed in embedded systems, where operating power is a big constraint (like systems which are battery powered); the '*IDLE TASK*' is used for putting the CPU into *IDLE* mode for saving the power. A typical example is the RTX51 Tiny Real-Time kernel, where the '*IDLE TASK*' sets the 8051 CPU to *IDLE* mode, a power saving mode. In the '*IDLE*' mode, the program execution is halted and all peripherals and the interrupt system continues its operation. Once the CPU is put into the '*IDLE*' mode, it comes out of this mode when an Interrupt occurs or when the RTX51 Tiny Timer Tick Interrupt (The timer interrupt used for task scheduling in Round robin scheduling) occurs. It should be noted that the '*IDLE PROCESS (TASK)*' execution is not pre-emptive priority scheduling specific, it is applicable to all types of scheduling policies which demand 100% CPU utilisation/CPU power saving.

Back to the desktop OS environment, let's analyse the process, threads and scheduling in the Windows desktop environment. Windows provides a utility called task manager for monitoring the different process running on the system and the resources used by each process. A snapshot of the process details returned by the task manager for Windows 10 kernel is shown in Fig. 10.15. It should be noted that this snap- shot is purely machine dependent and it varies with the number of processes running on the machine. 'Name' represents the name of the process. 'PID' represents the Process Identification Number (Process ID). As mentioned in the 'Threads and Process' section, when a process is created an ID is associated to it. CPU usage gives the % of CPU utilised by the process during an interval. 'CPU Time' gives the total processor time, in seconds, used by a process since it started. 'Working set (memory)' represents the amount of memory in the private working set plus the amount of memory the process is using that can be shared by other processes. 'Commit Size' represents the amount of virtual memory that's reserved for use by a process. 'Paged Pool' represents the amount of pageable kernel memory allocated by the kernel or drivers on behalf of a process. Pageable memory is memory that can be written to another storage medium, such as the hard disk. 'NP Pool' is the amount of non-pageable kernel memory allocated by the kernel or drivers on behalf of a process.



Name	PID	CPU	CPU time	Working set (memory)	Commit size	Paged pool	NP pool	Base priority	Handles	Threads	User objects
svchost.exe	332	00	0:03:37	58,624 K	31,464 K	477 K	73 K	Normal	2,020	45	0
svchost.exe	344	00	0:03:09	97,860 K	86,224 K	281 K	52 K	Normal	930	23	0
svchost.exe	632	00	0:00:17	32,468 K	12,024 K	274 K	59 K	Normal	997	28	0
svchost.exe	640	00	0:00:10	29,624 K	18,576 K	233 K	37 K	Normal	898	22	0
svchost.exe	1032	00	0:00:01	8,496 K	2,216 K	92 K	16 K	Normal	223	7	0
svchost.exe	1160	00	0:00:12	21,644 K	9,160 K	228 K	50 K	Normal	844	27	0
svchost.exe	1728	00	0:00:44	29,392 K	17,912 K	138 K	43 K	Normal	539	24	0
svchost.exe	1452	00	0:00:11	24,080 K	7,904 K	285 K	23 K	Normal	464	13	0
svchost.exe	1884	00	0:00:40	19,120 K	5,392 K	119 K	16 K	Normal	236	8	0
svchost.exe	3912	00	0:01:36	17,616 K	35,412 K	259 K	23 K	Normal	1,090	2	0
svchost.exe	4080	00	0:00:06	31,924 K	7,876 K	234 K	31 K	Normal	765	10	1
SynTPEnh.exe	5140	00	0:00:59	21,536 K	6,108 K	280 K	16 K	Above nor...	490	7	47
SynTPHelper.exe	5832	00	0:00:00	4,224 K	772 K	89 K	5 K	Above nor...	50	1	3
System	4	01	0:05:56	45,500 K	316 K	0 K	0 K	N/A	1,290	171	0
System Idle Process	0	83	17:10:03	4 K	0 K	0 K	0 K	N/A	-	4	0
System interrupts	-	03	0:00:00	0 K	0 K	0 K	0 K	N/A	-	-	0
TabTip.exe	3712	00	0:00:25	13,016 K	2,900 K	229 K	15 K	High	298	14	38
TabTip32.exe	8688	00	0:00:00	3,992 K	1,120 K	86 K	6 K	Normal	59	1	6
taskhostw.exe	2836	00	0:00:00	7,052 K	1,668 K	114 K	9 K	Below nor...	141	9	0
taskhostw.exe	3944	00	0:00:13	20,388 K	8,216 K	218 K	34 K	Normal	481	7	15
Taskmgr.exe	212	03	0:02:44	46,524 K	20,564 K	457 K	35 K	Normal	524	18	567
ToolbarUpdater.exe	2056	00	0:00:00	12,844 K	2,704 K	144 K	15 K	Normal	226	5	0
vcpkgsrv.exe	8512	00	0:00:13	52,664 K	13,728 K	221 K	78 K	Normal	145	2	1

Fig. 10.15 Windows 10 Task Manager for monitoring process and resource usage

The non-paged memory cannot be swapped to the secondary storage disk. 'Base Priority' represents the priority of the process (A precedence ranking that determines the order in which the threads of a process are scheduled.). As mentioned in an earlier section, a process may contain multiple threads. The 'Threads' section gives the number of threads running in a process. 'Handles' reflects the number of object handles owned by the process. This value is the reflection of the object handles present in the process's object table. 'User Objects' reflects the number of objects active in the user mode for a process (A USER object is an object from Window Manager, which includes windows, menus, cursors, icons, hooks, accelerators, monitors, keyboard layouts, and other internal objects). Use 'Ctrl' + 'Alt' + 'Del' key for accessing the task manager and select the 'Details' tab. Right click the mouse on the row title header displaying the parameters and choose 'Select columns' option to select the different monitoring parameters for a process.

10.7 TASK COMMUNICATION

LO 7 Identify the RPC based Inter Process Communication

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

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

Competing Processes: The competing processes do not share anything among themselves but they share the system resources. The competing processes compete for the system resources such as file, display device, etc.

Co-operating processes exchanges information and communicate through the following methods.

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

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

The mechanism through which processes/tasks communicate each other is known as Inter Process/Task Communication (IPC). Inter Process Communication is essential for process co-ordination. The various types of Inter Process Communication (IPC) mechanisms adopted by process are kernel (Operating System) dependent. Some of the important IPC mechanisms adopted by various kernels are explained below.

10.7.1 Shared Memory

Processes share some area of the memory to communicate among them (Fig. 10.16). Information to be communicated by the process is written to the shared memory area. Other processes which require this information can read the same from the shared memory area. It is same as the real world example where 'Notice Board' is used by corporate to publish the public information among the employees (The only exception is; only corporate have the right to modify the information published on the Notice board and employees are given 'Read' only access, meaning it is only a one way channel).

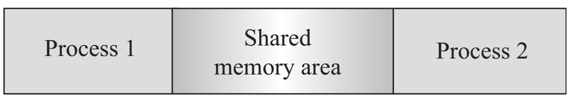


Fig. 10.16 Concept of Shared Memory

The implementation of shared memory concept is kernel dependent. Different mechanisms are adopted by different kernels for implementing this. A few among them are:

10.7.1.1 Pipes

'Pipe' is a section of the shared memory used by processes for communicating. Pipes follow the client-server‡ architecture. A process which creates a pipe is known as a pipe server and a process which connects to a pipe is known as pipe client. A pipe can be considered as a conduit for information flow and has two conceptual ends. It can be unidirectional, allowing information flow in one direction or bidirectional allowing bi-directional information flow. A unidirectional pipe allows the process connecting at one end of the pipe to write to the pipe and the process connected at the other end of the pipe to read the data, whereas a bi-directional pipe allows both reading and writing at one end. The unidirectional pipe can be visualised as

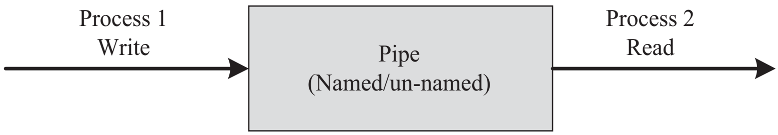


Fig. 10.17 Concept of Pipe for IPC

The implementation of 'Pipes' is also OS dependent. Microsoft® Windows Desktop Operating Systems support two types of 'Pipes' for Inter Process Communication. They are:

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

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

Please refer to the Online Learning Centre for details on the Pipe implementation under Windows Operating Systems.

Under VxWorks kernel, *pipe* is a special implementation of message queues. We will discuss the same in a latter chapter.

10.7.1.2 Memory Mapped Objects

Memory mapped object is a shared memory technique adopted by certain Real-Time Operating Systems for allocating a shared block of memory which can be accessed by multiple process simultaneously (of course certain synchronisation techniques should be applied to prevent inconsistent results). In this approach a mapping object is created and physical storage for it is reserved and committed. A process can map the entire committed physical area or a block of it to its virtual address space. All read and write operation to this virtual address space by a process is directed to its committed physical area. Any process which wants to share data with other processes can map the physical memory area of the mapped object to its virtual memory space and use it for sharing the data.

Windows Embedded Compact RTOS uses the memory mapped object based shared memory technique for Inter Process Communication (Fig. 10.18). The *CreateFileMapping (HANDLE hFile, LPSECURITY_ATTRIBUTES lpFileMappingAttributes, DWORD flProtect, DWORD dwMaximumSizeHigh,*

‡Client Server is a software architecture containing a client application and a server application. The application which sends request is known as client and the application which receives the request process it and sends a response back to the client is known as server. A server is capable of receiving request from multiple clients.

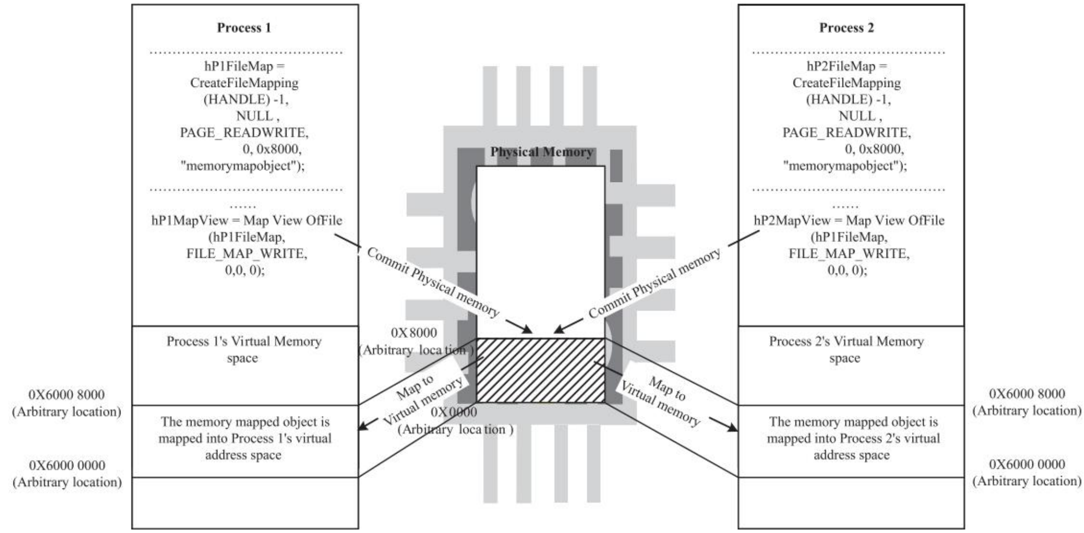


Fig. 10.18 Concept of memory mapped object

DWORD dwMaximumSizeLow, LPCTSTR lpName) system call is used for sharing the memory. This API call is used for creating a mapping from a file. In order to create the mapping from the system paging memory, the handle parameter should be passed as *INVALID_HANDLE_VALUE (–1)*. The *lpFileMappingAttributes* parameter represents the security attributes and it must be *NULL.* The *flProtect* parameter represents the read write access for the shared memory area. A value of *PAGE_READONLY* makes the shared memory read only whereas the value *PAGE_READWRITE* gives read-write access to the shared memory. The parameter *dwMaximumSizeHigh* specifies the higher order 32 bits of the maximum size of the memory mapped object and *dwMaximumSizeLow* specifies the lower order 32 bits of the maximum size of the memory mapped object. The parameter *lpName* points to a null terminated string specifying the name of the memory mapped object. The memory mapped object is created as unnamed object if the parameter *lpName* is *NULL*. If *lpName* specifies the name of an existing memory mapped object, the function returns the handle of the existing memory mapped object to the caller process. The memory mapped object can be shared between the processes by either passing the handle of the object or by passing its name. If the handle of the memory mapped object created by a process is passed to another process for shared access, there is a possibility of closing the handle by the process which created the handle while it is in use by another process. This will throw OS level exceptions. If the name of the memory object is passed for shared access among processes, processes can use this name for creating a shared memory object which will open the shared memory object already existing with the given name. The OS will maintain a usage count for the named object and it is incremented each time when a process creates/opens a memory mapped object with existing name. This will prevent the destruction of a shared memory object by one process while it is being accessed by another process. Hence passing the name of the memory mapped object is strongly recommended for memory mapped object based inter process communication. The *MapViewOfFile (HANDLE hFileMappingObject DWORD dwDesiredAccess, DWORD dwFileOffsetHigh, DWORD dwFileOffsetLow, DWORD dwNumberOfBytesToMap)* system call maps a view of the memory mapped object to the address space of the calling process. The parameter *hFileMappingObject* specifies the handle to an existing memory mapped object. The *dwDesiredAccess* parameter represents the read write access for the mapped view area. A value of *FILE_MAP_WRITE* makes the view access read-write, provided the memory mapped object *hFileMappingObject* is created with read-write access, whereas the value *FILE_MAP_READ* gives read only access to the shared memory, provided the memory mapped object *hFileMappingObject* is created with read-write/read only access. The parameter *dwFileOffsetHigh* specifies the higher order 32 bits and *dwFileOffsetLow* specifies the lower order 32 bits of the memory offset where mapping is to begin from the memory mapped object. A value of '0' for both of these maps the view from the beginning memory area of the memory object. *dwNumberOfBytesToMap* specifies the number of bytes of the memory object to map. If *dwNumberOfBytesToMap* is zero, the entire memory area owned by the memory mapped object is mapped. On successful execution, *MapViewOfFile* call returns the starting address of the mapped view. If the function fails it returns *NULL*. A mapped view of the memory mapped object is unmapped by the API call *UnmapViewOfFile (LPCVOID lpBaseAddress)*. The *lpBaseAddress* parameter specifies a pointer to the base address of the mapped view of a memory object that is to be unmapped. This value must be identical to the value returned by a previous call to the *MapViewOfFile* function. Calling *UnmapViewOfFile* cleans up the committed physical storage in a process's virtual address space. In other words, it frees the virtual address space of the mapping object. Under Windows NT Kernel, a process can open an existing memory mapped object by calling the API *OpenFileMapping(DWORD dwDesiredAccess, BOOL bInheritHandle, LPCTSTR lpName)*. The parameter *dwDesiredAccess* specifies the read write access permissions for the memory mapped object. A value of *FILE_MAP_ALL_ACCESS* provides read-write access, whereas the value *FILE_MAP_READ* allocates only read access and *FILE_MAP_WRITE* allocates write only access. The parameter *bInheritHandle* specifies the handle inheritance. If this parameter is *TRUE*, the calling process inherits the handle of the existing object, otherwise not. The parameter *lpName* specifies the name of the existing memory mapped object which needs to be opened. Windows CE 5.0 does not support handle inheritance and hence the API call *OpenFileMapping* is not supported.

The following sample code illustrates the creation and accessing of memory mapped objects across multiple processes. The first piece of code illustrates the creation of a memory mapped object with name "memorymappedobject" and prints the address of the memory location where the memory is mapped within the virtual address space of Process 1.

```
##include "stdafx.h"
#include <stdio.h>
#include <windows.h>
//*******************************************************************
//Process 1: Creates the memory mapped object and maps it to
//Process 1's Virtual Address space
//*******************************************************************
void main() {
 //Define the handle to Memory mapped Object
 HANDLE hFileMap;
 //Define the handle to the view of Memory mapped Object
 LPBYTE hMapView;
 printf("//*********************************************\n");
 printf(" Process 1\n");
 printf("//*********************************************\n");
 //Create an 8 KB memory mapped object
 hFileMap = CreateFileMapping((HANDLE)-1,
                 NULL, // default security attributes
                 PAGE_READWRITE, // Read-Write Access
                 0, //Higher order 32 bits of the memory mapping object
                 0x2000, //Lower order 32 bits of the memory mapping object
```

```
                 TEXT("memorymappedobject")); // Memory mapped object name
  if (NULL == hFileMap)
  {
   printf("Memory mapped Object Creation Failed : Error Code : %d\n",
   GetLastError());
   //Memory mapped Object Creation failed. Return
   return;
  }
  //Map the memory mapped object to Process 1's address space
  hMapView = (LPBYTE)MapViewOfFile(hFileMap,
                  FILE_MAP_WRITE,
                  0, //Map the entire view
                  0,
                  0);
  if (NULL == hMapView)
  {
   printf("Mapping of Memory mapped view Failed : Error Code :%d\n",
   GetLastError());
   //Memory mapped view Creation failed. Return
   return;
  }
  else
  {
   //Successfully created the memory mapped view.
   //Print the start address of the mapped view
   printf("The memory is mapped to the virtual address starting at 0x%p\n",
  (void *)hMapView);
  }
  //Wait for user input to exit. Run Process 2 before providing
  //user input
  printf("Press any key to terminate Process 1");
  getchar();
  //Unmap the view
  UnmapViewOfFile(hMapView);
  //Close memory mapped object handle
  CloseHandle(hFileMap);
  return;
 }
```

The piece of code given below corresponds to Process 2. It illustrates the accessing of an existing memory mapped object (memory mapped object with name "memorymappedobject" created by Process 1) and prints the address of the memory location where the memory is mapped within the virtual address space of Process 2. To demonstrate the application, the program corresponding to Process 1 should be executed first and the program corresponding to Process 2 should be executed following Process 1.

```
#include "stdafx.h"
#include <stdio.h>
```

```
#include <windows.h>
//*******************************************************************
//Process 2: Opens the memory mapped object created by Process 1
//Maps the object to Process 2's virtual address space.
//*******************************************************************
void main() {
 //Define the handle for the Memory mapped Object
 HANDLE hChildFileMap;
 //Define the handle for the view of Memory mapped Object
 LPBYTE hChildMapView;
 printf("//*********************************************\n");
 printf(" Process 2\n");
 printf("//*********************************************\n");
 //Create an 8 KB memory mapped object
 hChildFileMap = CreateFileMapping(INVALID_HANDLE_VALUE,
                    NULL, // default security attributes
                    PAGE_READWRITE, // Read-Write Access
                    0, //Higher order 32 bits of the memory mapping object
                    0x2000, //Lower order 32 bits of the memory mapping object
                    TEXT("memorymappedobject")); // Memory mapped object name
 if ((NULL == hChildFileMap) || (INVALID_HANDLE_VALUE == hChildFileMap))
 {
  printf("Memory mapped Object Creation Failed : Error Code : %d\n",
  GetLastError());
  //Memory mapped Object Creation failed. Return
  return;
 }
 else if (ERROR_ALREADY_EXISTS == GetLastError())
  {
   //A memory mapped object with given name exists already.
   printf("The named memory mapped object is already existing\n");
  }
 //Map the memory mapped object to Process 2's address space
 hChildMapView = (LPBYTE) MapViewOfFile(hChildFileMap,
                      FILE_MAP_WRITE, //Read-Write access
                      0,
                      //Map the entire view
                      0,
                      0);
 if (NULL == hChildMapView)
 {
  printf("Mapping of Memory mapped view Failed : Error Code : %d\n",
  GetLastError());
  //Memory mapped view Creation failed. Return
  return;
 }
```

```
else
 {
  //Successfully created the memory mapped view.
  //Print the start address of the mapped view
  printf("The memory mapped view starts at memory location 0x%p\n",
 (void *) hChildMapView);
 }
 //Wait for user input to exit.
 printf("Press any key to terminate Process 2");
 getchar();
 //Unmap the view
 UnmapViewOfFile(hChildMapView);
 //Close memory mapped object handle
 CloseHandle(hChildFileMap);
 return;
}
```

The output of the above programs when executed on Windows 10 OS in the sequence, Process 1 is executed first and Process 2 is executed while Process 1 waits for the user input from the keyboard, is given in Fig. 10.19.

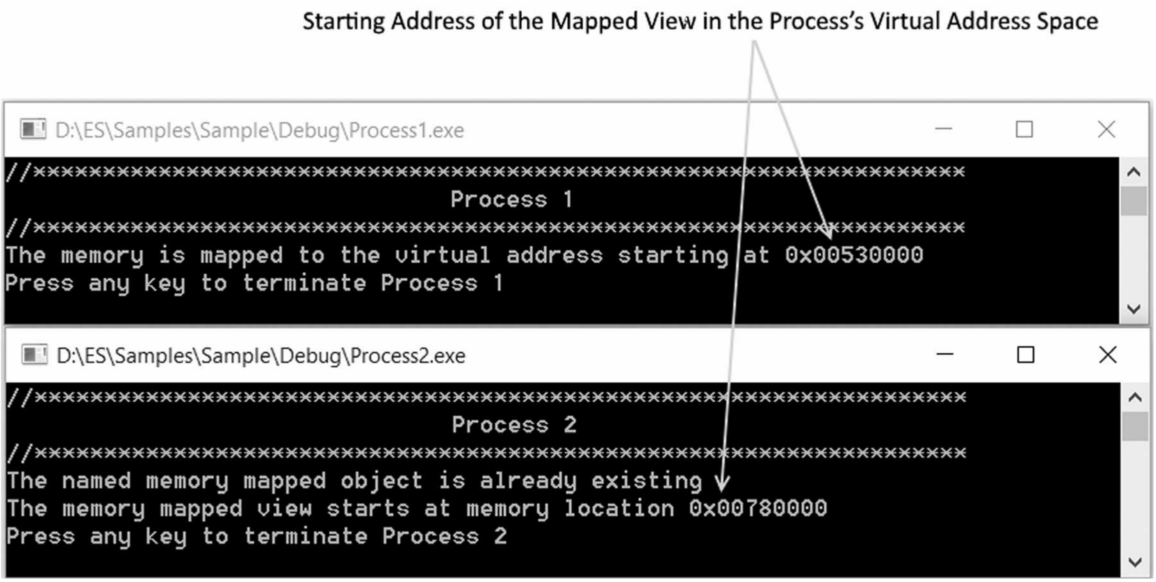


Fig. 10.19 Output of Win32 memory mapped object illustration program

The starting address of the memory mapped view in the virtual address space of the process is decided by the OS. However a process can request to map the view at a desired virtual address by using the *MapViewofFileEx* function. Depending on the availability of the address space the OS may grand the request or not. It is always safe to let the OS choose the address.

Reading and writing to a memory mapped area is same as any read write operation using pointers. The pointer returned by the API call *MapViewOfFile* can be used for this. The exercise of Read and Write operation is left to the readers. Proper care should be taken to avoid any conflicts that may arise due to the simultaneous read/write access of the shared memory area by multiple processes. This can be handled by applying various synchronisation techniques like events, mutex, semaphore, etc.

For using a memory mapped object across multiple threads of a process, it is not required for all the threads of the process to create/open the memory mapped object and map it to the thread's virtual address space. Since the thread's address space is part of the process's virtual address space, which contains the thread, only one thread, preferably the parent thread (main thread) is required to create the memory mapped object and map it to the process's virtual address space. The thread which creates the memory mapped object can share the pointer to the mapped memory area as global pointer and other threads of the process can use this pointer for reading and writing to the mapped memory area. If one thread of a process tries to create a memory mapped object with the same name as that of an existing mapping object, which is created by another thread of the same process, a new view of the mapping object is created at a different virtual address of the process. This is same as one process trying to create two views of the same memory mapped object☺.

10.7.2 Message Passing

Message passing is an (a)synchronous information exchange mechanism used for Inter Process/Thread Communication. The major difference between shared memory and message passing technique is that, through shared memory lots of data can be shared whereas only limited amount of info/data is passed through message passing. Also message passing is relatively fast and free from the synchronisation overheads compared to shared memory. Based on the message passing operation between the processes, message passing is classified into

10.7.2.1 Message Queue

Usually the process which wants to talk to another process posts the message to a First-In-First-Out (FIFO) queue called 'Message queue', which stores the messages temporarily in a system defined memory object, to pass it to the desired process (Fig. 10.20). Messages are sent and received through *send* (*Name of the process to which the message is to be sent, message*) and *receive* (*Name of the process from which the message is to be received, message*) methods. The messages are exchanged through a message queue. The implementation of the message queue, *send* and *receive* methods are OS kernel dependent. The Windows XP OS kernel maintains a single system message queue and one process/thread (Process and threads are used interchangeably here, since thread is the basic unit of process in windows) specific message queue. A thread which wants to communicate with another thread posts the message to the system message queue. The kernel picks up the message from the system message queue one at a time and examines the message for finding the destination thread and then posts the message to the message queue of the corresponding thread. For posting a message to a thread's message queue, the kernel fills a message structure *MSG* and copies it to the message queue of the thread. The message structure *MSG* contains the handle of the process/thread for which the message is intended, the message parameters, the time at which the message is posted, etc. A thread can simply post a message to another thread and can continue its operation or it may wait for a response from the thread to which the message is posted. The messaging mechanism is classified into synchronous and asynchronous based on the behaviour of the message posting thread. In asynchronous messaging, the message posting thread just posts the message to the queue and it will not wait for an acceptance (return) from the thread to which the message is posted, whereas in synchronous messaging, the thread which posts a message enters waiting state and waits for the message result from the thread to which the message is posted. The thread which invoked the send message becomes blocked and the scheduler will not pick it up for scheduling.

The *PostMessage (HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam)* or *PostThreadMessage (DWORD idThread, UINT Msg, WPARAM wParam, LPARAM lParam)* API is used by a thread in Windows for posting a message to its own message queue or to the message queue of another thread. *PostMessage* places a message at the end of a thread's message queue and returns immediately, without waiting for the thread to process the message. The function's parameters include a window handle, a message identifier, and two message parameters. The system copies these parameters to an *MSG* structure, and places the structure in the message queue. The *PostThreadMessage* is similar to *PostMessage*, except the first parameter is a thread identifier and it is used for posting a message to a specific thread message queue. The *SendMessage (HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam)* API call sends a message to the thread specified by the handle *hWnd* and waits for the callee thread to process the message. The thread which calls the *SendMessage* API enters waiting state and waits for the message result from the thread to which the message is posted. The thread which invoked the *SendMessage* API call becomes blocked and the scheduler will not pick it up for scheduling.

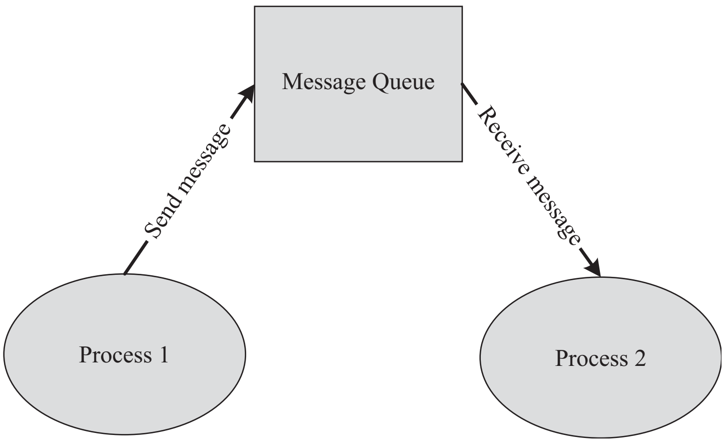


Fig. 10.20 Concept of message queue based indirect messaging for IPC

The Windows Embedded Compact operating system supports a special Point-to-Point Message queue implementation. The OS maintains a First In First Out (FIFO) buffer for storing the messages and each process can access this buffer for reading and writing messages. The OS also maintains a special queue, with single message storing capacity, for storing high priority messages (Alert messages). The creation and usage of message queues under Windows Embedded Compact OS is explained below.

The *CreateMsgQueue(LPCWSTR lpszName, LPMSGQUEUEOPTIONS lpOptions)* API call creates a message queue or opens a named message queue and returns a read only or write only handle to the message queue. A process can use this handle for reading or writing a message from/to of the message queue pointed by the handle. The parameter *lpszName* specifies the name of the message queue. If this parameter is *NULL*, an unnamed message queue is created. Processes can use the handle returned by the API call if the message queue is created without any name. If the message queue is created as named message queue, other processes can use the name of the message queue for opening the named message queue created by a process. Calling the *CreateMsgQueue* API with an existing named message queue as parameter returns a handle to the existing message queue. Under the Desktop Windows Operating Systems (Windows NT Kernel – Windows XP, Windows 8.1/10, etc.), each object type (viz. mutex, semaphores, events, memory maps, watchdog timers and message queues) share the same namespace and the same name is not allowed for creating any of this. Windows CE/Embedded Compact kernel maintains separate namespace for each and supports the same name across different objects. The *lpOptions* parameter points to a *MSGQUEUEOPTIONS* structure that sets the properties of the message queue. The member details of the *MSGQUEUEOPTIONS* structure is explained below.

```
typedef MSGQUEUEOPTIONS_OS{
  DWORD dwSize;
  DWORD dwFlags;
  DWORD dwMaxMessages;
  DWORD cbMaxMessage;
  BOOL bReadAccess;
} MSGQUEUEOPTIONS, FAR* LPMSGQUEUEOPTIONS, *PMSGQUEUEOPTIONS;
```

The members of the structure are listed below.

Member	Description
dwSize	Specifies the size of the structure in bytes
dwFlags	Describes the behaviour of the message queue. Set to MSGQUEUE_NOPRECOMMIT to allocate message buffers on demand and to free the message buffers after they are read, or set to MSGQUEUE_ALLOW_BROKEN to enable a read or write operation to complete even if there is no corresponding writer or reader present.
dwMaxMessages	Specifies the maximum number of messages to queue at any point of time. Set this value to zero to specify no limit on the number of messages to queue at any point of time.
cbMaxMessage	Specifies the maximum number of bytes in each message. This value must be greater than zero.
bReadAccess	Specifies the Read Write access to the message queue. Set to TRUE to request read access to the queue. Set to FALSE to request write access to the queue.

On successful execution, the *CreateMsgQueue* API call returns a 'Read Only' or 'Write Only' handle to the specified queue based on the *bReadAccess* member of the *MSGQUEUEOPTIONS* structure *lpOptions*. If the queue with specified name already exists, a new handle, which points to the existing queue, is created and a following call to *GetLastError* returns *ERROR_ALREADY_EXISTS*. If the function fails it returns *NULL*. A single call to the *CreateMsgQueue* creates the queue for either 'read' or 'write' access. The *CreateMsgQueue* API should be called twice with the *bReadAccess* member of the *MSGQUEUEOPTIONS* structure *lpOptions* set to TRUE and FALSE respectively in successive calls for obtaining 'Read only' and 'Write only' handles to the specified message queue. The handle returning by *CreateMsgQueue* API call is an *event* handle and, if it is a 'Read Only' access handle, it is signalled by the message queue if a new message is placed in the queue. The signal is reset on reading the message by *ReadMsgQueue* API call. A 'Write Only' access handle to the message queue is signalled when the queue is no longer full, i.e. when there is room for accommodating new messages. Processes can monitor the handles with the help of the wait functions, viz. *WaitForSingleObject* or *WaitForMultipleObjects*. The *OpenMsgQueue(HANDLE hSrcProc, HANDLE hMsgQ, LPMSGQUEUEOPTIONS lpOptions)* API call opens an existing named or unnamed message queue. The parameter *hSrcProc* specifies the process handle of the process that owns the message queue and *hMsgQ* specifies the handle of the existing message queue (Handle to the message queue returned by the CreateMsgQueue function). As in the case of *CreateMsgQueue*, the *lpOptions* parameter points to a *MSGQUEUEOPTIONS* structure that sets the properties of the message queue. On successful execution the *OpenMsgQueue* API call returns a handle to the message queue and *NULL* if it fails. Normally the *OpenMsgQueue* API is used for opening an unnamed message queue. The *WriteMsgQueue(HANDLE hMsgQ, LPVOID lpBuffer, DWORD cbDataSize, DWORD dwTimeout, DWORD dwFlags)* API call is used for writing a single message to the message queue pointed by the handle *hMsgQ*. *lpBuffer* points to a buffer that contains the message to be written to the message queue. The parameter *cbDataSize* specifies the number of bytes stored in the buffer pointed by *lpBuffer*, which forms a message. The parameter *dwTimeout* specifies the timeout interval in milliseconds for the message writing operation. A value of zero specifies the write operation to return immediately without blocking if the write operation cannot succeed. If the parameter is set to *INFINITE*, the write operation will block until it succeeds or the message queue signals the 'write only' handle indicating the availability of space for posting a message. The *dwFlags* parameter sets the priority of the message. If it is set to *MSGQUEUE_MSGALERT*, the message is posted to the queue as high priority or alert message. The Alert message is always placed in the front of the message queue. This function returns *TRUE* if it succeeds and *FALSE* otherwise.

The *ReadMsgQueue(HANDLE hMsgQ, LPVOID lpBuffer, DWORD cbBufferSize, LPDWORD lpNumberOfBytesRead, DWORD dwTimeout, DWORD* pdwFlags)* API reads a single message from the message queue. The parameter *hMsgQ* specifies a handle to the message queue from which the message needs to be read. *lpBuffer* points to a buffer for storing the message read from the queue. The parameter *cbBufferSize* specifies the size of the buffer pointed by *lpBuffer*, in bytes. *lpNumberOfBytesRead* specifies the number of bytes stored in the buffer. This is same as the number of bytes present in the message which is read from the message queue. *dwTimeout* specifies the timeout interval in milliseconds for the message reading operation. The timeout values and their meaning are same as that of the write message timeout parameter. The *dwFlags* parameter indicates the priority of the message. If the message read from the message queue is a high priority message (alert message), *dwFlags* is set to *MSGQUEUE_MSGALERT*. The function returns *TRUE* if it succeeds and *FALSE* otherwise. The *GetMsgQueueInfo (HANDLE hMsgQ, LPMSGQUEUEINFO lpInfo)* API call returns the information about a message queue specified by the handle *hMsgQ*. The message information is returned in a *MSGQUEUEINFO* structure pointed by *lpInfo*. The details of the *MSGQUEUEINFO* structure is explained below.

```
typedef MSGQUEUEINFO{
  DWORD dwSize;
  DWORD dwFlags;
  DWORD dwMaxMessages;
  DWORD cbMaxMessage;
  DWORD dwCurrentMessages;
  DWORD dwMaxQueueMessages;
  WORD wNumReaders;
  WORD wNumWriters;
} MSGQUEUEINFO, *PMSGQUEUEINFO, FAR* LPMSGQUEUEINFO;
```

The member variable details are listed below.

Member	Description
DwSize	Specifies the size of the buffer passed in.
dwFlags	Describes the behaviour of the message queue. It retrieves the *MSGQUEUEOPTIONS. dwFlags* passed when the message queue is created with *CreateMsgQueue* API call.
dwMaxMessages	Specifies the maximum number of messages to queue at any point of time. This reflects the *MSGQUEUEOPTIONS. dwMaxMessages* value passed when the message queue is created with *CreateMsgQueue* API call.

cbMaxMessage	Specifies the maximum number of bytes in each message. This reflects the *MSGQUEUEOPTIONS.cbMaxMessage* value passed when the message queue is created with *CreateMsgQueue* API call.
dwCurrentMessages	Specifies the number of messages currently existing in the specified message queue.
dwMaxQueueMessages	Specifies maximum number of messages that have ever been in the queue at one time.
wNumReaders	Specifies the number of readers (processes which opened the message queue for reading) subscribed to the message queue for reading.
wNumWriters	Specifies the number of writers (processes which opened the message queue for writing) subscribed to the message queue for writing.

The *GetMsgQueueInfo* API call returns *TRUE* if it succeeds and *FALSE* otherwise. The *CloseMsgQueue(HANDLE hMsgQ)* API call closes a message queue specified by the handle *hMsgQ*. If a process holds a 'read only' and 'write only' handle to the message queue, both should be closed for closing the message queue.

'Message queue' is the primary inter-task communication mechanism under VxWorks kernel. Message queues support two-way communication of messages of variable length. The two-way messaging between tasks can be implemented using one message queue for incoming messages and another one for outgoing messages. Messaging mechanism can be used for task-to task and task to Interrupt Service Routine (ISR) communication. We will discuss about the VxWorks' message queue implementation in a separate chapter.

10.7.2.2 Mailbox

Mailbox is an alternate form of 'Message queues' and it is used in certain Real-Time Operating Systems for IPC. Mailbox technique for IPC in RTOS is usually used for one way messaging. The task/thread which wants to send a message to other tasks/threads creates a mailbox for posting the messages. The threads which are interested in receiving the messages posted to the mailbox by the mailbox creator thread can subscribe to the mailbox. The thread which creates the mailbox is known as 'mailbox server' and the threads which subscribe to the mailbox are known as 'mailbox clients'. The mailbox server posts messages to the mailbox and notifies it to the clients which are subscribed to the mailbox. The clients read the message from the mailbox on receiving the notification. The mailbox creation, subscription, message reading and writing are achieved through OS kernel provided API calls. Mailbox and message queues are same in functionality. The only difference is in the number of messages supported by them. Both of them are used for passing data in the form of message(s) from a task to another task(s). Mailbox is used for exchanging a single message between two tasks or between an Interrupt Service Routine (ISR) and a task. Mailbox associates a pointer pointing to the mailbox and a wait list to hold the tasks waiting for a message to appear in the mailbox. The implementation of mailbox is OS kernel dependent. The MicroC/OS-II implements mailbox as a mechanism for inter-task communication. We will discuss about the mailbox based IPC implementation under MicroC/OS-II in a latter chapter. Figure 10.21 given below illustrates the mailbox based IPC technique.

10.7.2.3 Signalling

Signalling is a primitive way of communication between processes/threads. *Signals* are used for asynchronous notifications where one process/thread fires a signal, indicating the occurrence of a scenario which the other process(es)/thread(s) is waiting. Signals are not queued and they do not carry any data. The communication mechanisms used in RTX51 Tiny OS is an example for Signalling. The *os_send_signal* kernel call under RTX 51 sends a signal from one task to a specified task. Similarly the *os_wait* kernel call waits for a specified signal. Refer to the topic 'Round Robin Scheduling' under the section 'Priority based scheduling' for more details on Signalling in RTX51 Tiny OS. The VxWorks RTOS kernel also implements 'signals' for inter process communication. Whenever a specified signal occurs it is handled in a signal handler associated with the signal. We will discuss about the signal based IPC mechanism for VxWorks' kernel in a later chapter.

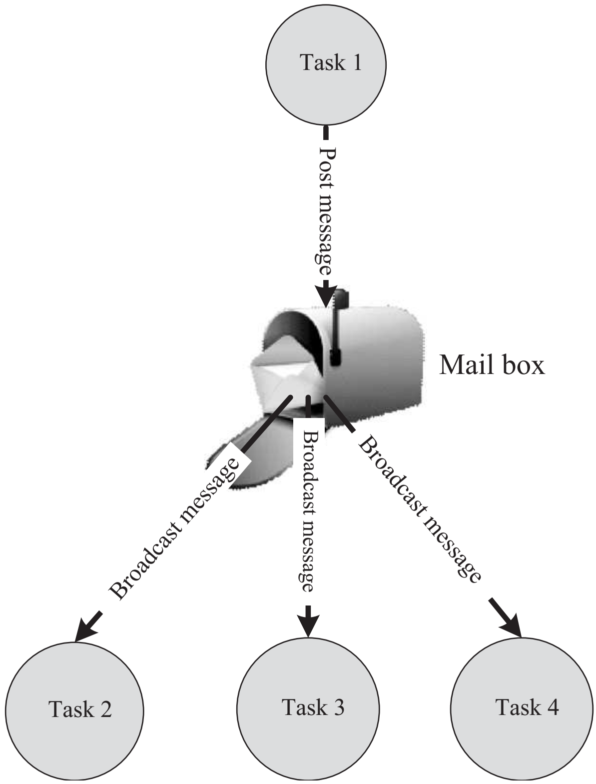
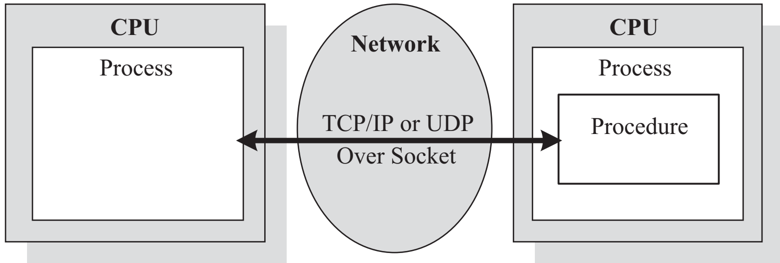


Fig. 10.21 Concept of Mailbox based indirect messaging for IPC

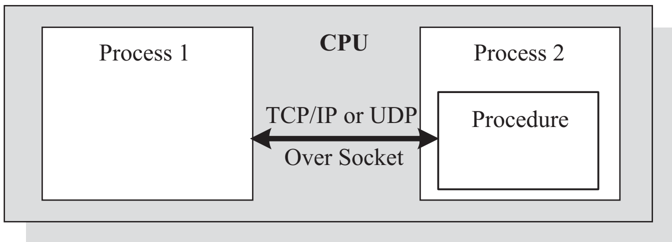
10.7.3 Remote Procedure Call (RPC) and Sockets

Remote Procedure Call or RPC (Fig. 10.22) is the Inter Process Communication (IPC) mechanism used by a process to call a procedure of another process running on the same CPU or on a different CPU which is interconnected in a network. In the object oriented language terminology RPC is also known as *Remote Invocation* or *Remote Method Invocation (RMI)*. RPC is mainly used for distributed applications like client-server applications. With RPC it is possible to communicate over a heterogeneous network (i.e. Network where Client and server applications are running on different Operating systems). The CPU/process containing the procedure which needs to be invoked remotely is known as server. The CPU/process which initiates an RPC request is known as client.

It is possible to implement RPC communication with different invocation interfaces. In order to make the RPC communication compatible across all platforms it should stick on to certain standard formats. Interface Definition Language (IDL) defines the interfaces for RPC. Microsoft Interface Definition Language (MIDL) is the IDL implementation from Microsoft for all Microsoft platforms. The RPC communication can be either Synchronous (Blocking) or Asynchronous (Non-blocking). In the Synchronous communication, the process which calls the remote procedure is blocked until it receives a response back from the other process. In asynchronous RPC calls, the calling process continues its execution while the remote process performs



Processes running on different CPUs which are networked



Processes running on same CPU

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

the execution of the procedure. The result from the remote procedure is returned back to the caller through mechanisms like callback functions.

On security front, RPC employs authentication mechanisms to protect the systems against vulnerabilities. The client applications (processes) should authenticate themselves with the server for getting access. Authentication mechanisms like IDs, public key cryptography (like DES, 3DES), etc. are used by the client for authentication. Without authentication, any client can access the remote procedure. This may lead to potential security risks.

Sockets are used for RPC communication. *Socket is a logical endpoint in a two-way communication link between two applications running on a network. A port number is associated with a socket so that the network layer of the communication channel can deliver the data to the designated application*. Sockets are of different types, namely, Internet sockets (INET), UNIX sockets, etc. The INET socket works on internet communication protocol. TCP/IP, UDP, etc. are the communication protocols used by INET sockets. INET sockets are classified into:

1. Stream sockets
2. Datagram sockets

Stream sockets are connection oriented and they use TCP to establish a reliable connection. On the other hand, Datagram sockets rely on UDP for establishing a connection. The UDP connection is unreliable when compared to TCP. The client-server communication model uses a socket at the client side and a socket at the server side. A port number is assigned to both of these sockets. The client and server should be aware of the port number associated with the socket. In order to start the communication, the client needs to send a connection request to the server at the specified port number. The client should be aware of the name of the server along with its port number. The server always listens to the specified port number on the network. Upon receiving a connection request from the client, based on the success of authentication, the server grants the connection request and a communication channel is established between the client and server. The client uses the host name and port number of server for sending requests and server uses the client's name and port number for sending responses.

If the client and server applications (both processes) are running on the same CPU, both can use the same host name and port number for communication. The physical communication link between the client and server uses network interfaces like Ethernet or Wi-Fi for data communication. The underlying implementation of *socket* is OS kernel dependent. Different types of OSs provide different socket interfaces. The following sample code illustrates the usage of socket for creating a client application under Windows OS. Winsock (Windows Socket 2) is the library implementing socket functions for Win32.

```
#include "stdafx.h"
#include <stdio.h>
#include <winsock2.h>
#include <Ws2tcpip.h>
//Specify the server address
#define SERVER "172.168.0.1"
//Specify the server port
#define PORT 5000
#pragma comment(lib, "Ws2_32.lib")
const int recvbuflen = 100;
char *sendbuf = "Hi from Client";
char recvbuffer[recvbuflen];

void main() {
 //***********************************************************
 // Initialise Winsock
 WSADATA wsaData;
 if (WSAStartup(MAKEWORD(2, 2), &wsaData) == NO_ERROR)
  printf("Winsock Initialisation succeeded...\n");
 else
 {
  printf("Winsock Initialisation failed...\n");
  return;
 }
 //***********************************************************
 // Create a SOCKET for connecting to server
 SOCKET MySocket;
 MySocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
 if (MySocket == INVALID_SOCKET)
 {
  printf("Socket Creation failed...\n");
  WSACleanup();
  return;
 }
 else
 {
  printf("Successfully created the socket...\n");
  //**********************************************************
  // Set the Socket type, IP address and port of the server
  sockaddr_in ServerParams;
  ServerParams.sin_family = AF_INET;
```

```
  //ServerParams.sin_addr.s_addr = inet_addr(SERVER);
  if (inet_pton(AF_INET, SERVER, &(ServerParams.sin_addr)) != 1)
  {
   printf("Converting IP Address failed...\n");
   WSACleanup();
   return;
  }
  ServerParams.sin_port = htons(PORT);
  //*********************************************************
  // Connect to server.
  if (connect(MySocket, (SOCKADDR*)& ServerParams, sizeof(ServerParams)) ==
  SOCKET_ERROR)
  {
   printf("Connecting to Server failed...Error code: %d\n", GetLastError());
   WSACleanup();
   return;
  }
  else
  {
   printf("Successfully Connected to the server...\n");
   //*********************************************************
   // Send command to server
   if (send(MySocket, sendbuf, (int)strlen(sendbuf), 0) == SOCKET_ERROR) {
    printf("Sending data to server failed...\n");
    closesocket(MySocket);
    WSACleanup();
    return;
   }
   else
   {
    printf("Successfully sent command to server...\n");
    //*********************************************************
    // Receive a data packet
    if (recv(MySocket, recvbuffer, recvbuflen, 0) > 0)
    printf("Successfully Received a packet...\n The received packet is %s\n",
     recvbuffer);
    else
      printf("No response from server...\n");
    //*********************************************************
    //Close Socket
    closesocket(MySocket);
    WSACleanup();
    return;
   }
  }
 }
 getchar();
}
```

The above application tries to connect to a server machine with IP address 172.168.0.1 and port number 5000. Change the values of *SERVER* and *PORT* to connect to a machine with different IP address and port number. If the connection is success, it sends the data “Hi from Client” to the server and waits for a response from the server and finally terminates the connection.

Under Windows, the socket function library *Winsock* should be initiated before using the socket related functions. The function *WSAStartup* performs this initiation. The *socket()* function call creates a socket. The socket type, connection type and protocols for communication are the parameters for socket creation. Here the socket type is *INET* (*AF_INET*) and connection type is stream socket (*SOCK_STREAM*). The protocol selected for communication is *TCP/IP* (*IPPROTO_TCP*). After creating the socket it is connected to a server. For connecting to server, the server address and port number should be indicated in the connection request. The *sockaddr_in* structure specifies the socket type, IP address and port of the server to be connected to. The *connect ()* function connects the socket with a specified server. If the server grants the connection request, the *connect ()* function returns success. The *send()* function is used for sending data to a server. It takes the socket name and data to be sent as parameters. Data from server is received using the function call *recv()*. It takes the socket name and details of buffer to hold the received data as parameters. The TCP/IP network stack expects network byte order (Big Endian: Higher order byte of the data is stored in lower memory address location) for data. The function *htons()* converts the byte order of an unsigned short integer to the network order. The *closesocket()* function closes the socket connection. On the server side, the server creates a socket using the function *socket()* and binds the socket with a port using the *bind()* function. It listens to the port bonded to the socket for any incoming connection request. The function *listen()* performs this. Upon receiving a connection request, the server accepts it. The function *accept()* performs the accepting operation. Now the connectivity is established. Server can receive and transmit data using the function calls *recv()* and *send()* respectively. The implementation of the server application is left to the readers as an exercise.

10.8 TASK SYNCHRONISATION

LO 8 State the need for task synchronisation in a multitasking environment

In a multitasking environment, multiple processes run concurrently (in pseudo parallelism) and share the system resources. Apart from this, each process has its own boundary wall and they communicate with each other with different IPC mechanisms including shared memory and variables. Imagine a situation where two processes try to access display hardware connected to the system or two processes try to access a shared memory area where one process tries to write to a memory location when the other process is trying to read from this. What could be the result in these scenarios? Obviously unexpected results. How these issues can be addressed? The solution is, make each process aware of the access of a shared resource either directly or indirectly. The act of making processes aware of the access of shared resources by each process to avoid conflicts is known as ‘*Task/Process Synchronisation*’. Various synchronisation issues may arise in a multitasking environment if processes are not synchronised properly. The following sections describe the major task communication synchronisation issues observed in multitasking and the commonly adopted synchronisation techniques to overcome these issues.

10.8.1 Task Communication/Synchronisation Issues

10.8.1.1 Racing

Let us have a look at the following piece of code:

```
#include "stdafx.h"
#include <windows.h>
```

```
#include <stdio.h>
//************************************************************
//counter is an integer variable and Buffer is a byte array shared
//between two processes Process A and Process B
char Buffer[10] = { 1,2,3,4,5,6,7,8,9,10 };
short int counter = 0;
//************************************************************
// Process A
void Process_A(void) {
 int i;
 for (i = 0; i<5; i++)
 {
  if (Buffer[i] > 0)
   counter++;
 }
}
//************************************************************
// Process B
void Process_B(void) {
 int j;
 for (j = 5; j<10; j++)
 {
  if (Buffer[j] > 0)
   counter++;
 }
}
//************************************************************
//Main Thread.
int main() {
 DWORD id;
 CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Process_A, (LPVOID)0, 0, &id);
 CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Process_B, (LPVOID)0, 0, &id);
 Sleep(100000);
 return 0;
}
```

From a programmer perspective the value of counter will be 10 at the end of execution of processes A & B. But 'it need not be always' in a real world execution of this piece of code under a multitasking kernel. The results depend on the process scheduling policies adopted by the OS kernel. Now let's dig into the piece of code illustrated above. The program statement counter++; looks like a single statement from a high level programming language ('C' language) perspective. The low level implementation of this statement is dependent on the underlying processor instruction set and the (cross) compiler in use. The low level implementation of the high level program statement counter++; under Windows XP operating system running on an Intel Centrino Duo processor is given below. The code snippet is compiled with Microsoft Visual Studio 6.0 compiler.

```
mov eax,dword ptr [ebp-4];Load counter in Accumulator
add eax,1 ; Increment Accumulator by 1
mov dword ptr [ebp-4],eax ;Store counter with Accumulator
```

Whereas the same high level program statement when compiled with Visual Studio 2013 on an Intel i7 dual core Processor running Windows 10 OS yields the following low level implementation for the program statement ***counter++***

```
mov  ax,word ptr ds:[00A08140h]
add  ax,1
mov  word ptr ds:[00A08140h],ax
```

In the first scenario, at the processor instruction level, the value of the variable counter is loaded to the Accumulator register (EAX register). The memory variable counter is represented using a pointer. The base pointer register (EBP register) is used for pointing to the memory variable counter. After loading the contents of the variable counter to the Accumulator, the Accumulator content is incremented by one using the add instruction. Finally the content of Accumulator is loaded to the memory location which represents the variable counter. Both the processes Process A and Process B contain the program statement counter++; Translating this into the machine instruction

Process A	Process B
mov eax,dword ptr [ebp-4]	mov eax,dword ptr [ebp-4]
add eax,1	add eax,1
mov dword ptr [ebp-4],eax	mov dword ptr [ebp-4],eax

Imagine a situation where a process switching (context switching) happens from Process A to Process B when Process A is executing the *counter++;* statement. Process A accomplishes the *counter++;* statement through three different low level instructions. Now imagine that the process switching happened at the point where Process A executed the low level instruction, 'mov eax,dword ptr [ebp-4]' and is about to execute the next instruction 'add eax,1'. The scenario is illustrated in Fig. 10.23.

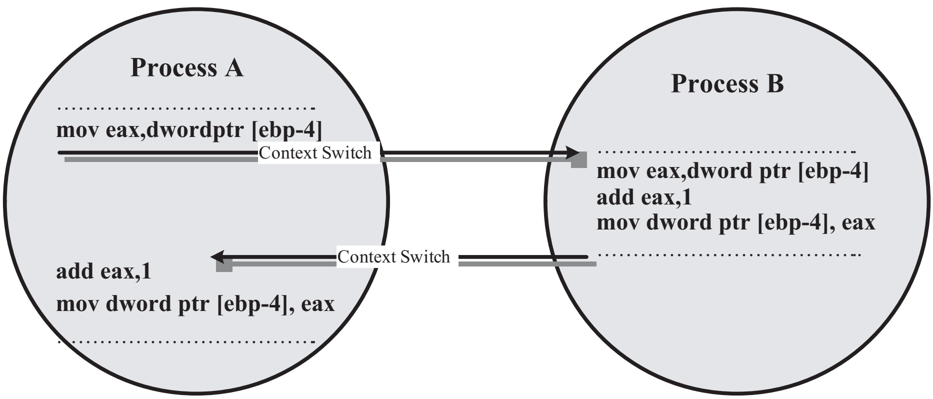


Fig. 10.23 Race condition

Process B increments the shared variable 'counter' in the middle of the operation where Process A tries to increment it. When Process A gets the CPU time for execution, it starts from the point where it got interrupted (If Process B is also using the same registers *eax* and *ebp* for executing *counter++;* instruction, the original content of these registers will be saved as part of the context saving and it will be retrieved back as part of context retrieval, when process A gets the CPU for execution. Hence the content of *eax* and *ebp* remains intact irrespective of context switching). Though the variable *counter* is incremented by Process B, Process A is unaware of it and it increments the variable with the old value. This leads to the loss of one increment for the variable *counter*. This problem occurs due to non-atomic§ operation on variables. This issue wouldn't have been occurred if the underlying actions corresponding to the program statement *counter++;* is finished in a single CPU execution cycle. The best way to avoid this situation is make the access and modification of shared variables mutually exclusive; meaning when one process accesses a shared variable, prevent the other processes from accessing it. We will discuss this technique in more detail under the topic '*Task Synchronisation techniques*' in a later section of this chapter.

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

10.8.1.2 Deadlock

A race condition produces incorrect results whereas a deadlock condition creates a situation where none of the processes are able to make any progress in their execution, resulting in a set of deadlocked processes. A situation very similar to our traffic jam issues in a junction as illustrated in Fig. 10.24.

Fig. 10.24 Deadlock visualisation

In its simplest form 'deadlock' is the condition in which a process is waiting for a resource held by another process which is waiting for a resource held by the first process (Fig. 10.25). To elaborate: Process A holds a resource *x* and it wants a resource *y* held by Process B. Process B is currently holding resource *y* and it wants the resource *x* which is currently held by Process A. Both hold the respective resources and they compete each other to get the resource held by the respective processes. The result of the competition is 'deadlock'. None of the competing process will be able to access the resources held by other processes since they are locked by the respective processes (If a mutual exclusion policy is implemented for shared resource access, the resource is locked by the process which is currently accessing it).

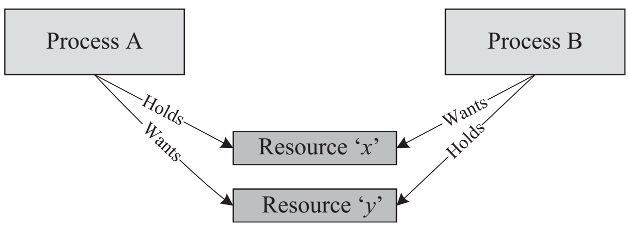


Fig. 10.25 Scenarios leading to deadlock

The different conditions favouring a deadlock situation are listed below.

Mutual Exclusion: The criteria that only one process can hold a resource at a time. Meaning processes should access shared resources with mutual exclusion. Typical example is the accessing of display hardware in an embedded device.

§ Atomic Operation: Operations which are non-interruptible.

Hold and Wait: The condition in which a process holds a shared resource by acquiring the lock controlling the shared access and waiting for additional resources held by other processes.

No Resource Preemption: The criteria that operating system cannot take back a resource from a process which is currently holding it and the resource can only be released voluntarily by the process holding it.

Circular Wait: A process is waiting for a resource which is currently held by another process which in turn is waiting for a resource held by the first process. In general, there exists a set of waiting process P0, P1 … P*n* with P0 is waiting for a resource held by P1 and P1 is waiting for a resource held by P0, …, P*n* is waiting for a resource held by P0 and P0 is waiting for a resource held by P*n* and so on… This forms a circular wait queue.

'Deadlock' is a result of the combined occurrence of these four conditions listed above. These conditions are first described by E. G. Coffman in 1971 and it is popularly known as *Coffman conditions*.

Deadlock Handling A smart OS may foresee the deadlock condition and will act proactively to avoid such a situation. Now if a deadlock occurred, how the OS responds to it? The reaction to deadlock condition by OS is nonuniform. The OS may adopt any of the following techniques to detect and prevent deadlock conditions.

Ignore Deadlocks: Always assume that the system design is deadlock free. This is acceptable for the reason the cost of removing a deadlock is large compared to the chance of happening a deadlock. UNIX is an example for an OS following this principle. A life critical system cannot pretend that it is deadlock free for any reason.

Detect and Recover: This approach suggests the detection of a deadlock situation and recovery from it. This is similar to the deadlock condition that may arise at a traffic junction. When the vehicles from different directions compete to cross the junction, deadlock (traffic jam) condition is resulted. Once a deadlock (traffic jam) is happened at the junction, the only solution is to back up the vehicles from one direction and allow the vehicles from opposite direction to cross the junction. If the traffic is too high, lots of vehicles may have to be backed up to resolve the traffic jam. This technique is also known as 'back up cars' technique (Fig. 10.26).

Operating systems keep a resource graph in their memory. The resource graph is updated on each resource request and release. A deadlock condition can be detected by analysing the resource graph by graph analyser algorithms. Once a deadlock condition is detected, the system can terminate a process or preempt the resource to break the deadlocking cycle.

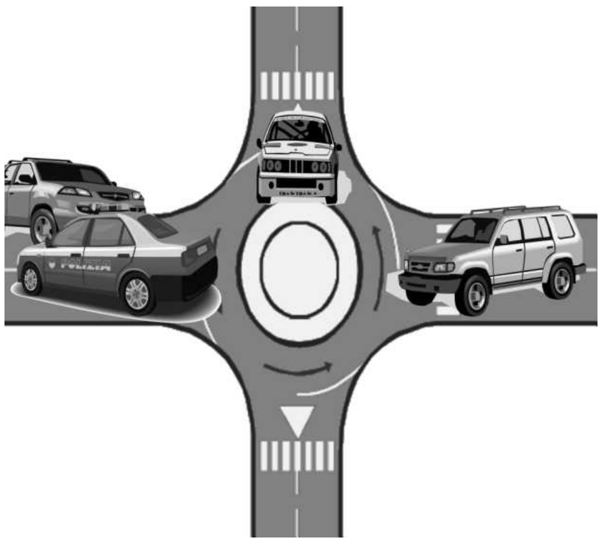


Fig. 10.26 'Back up cars' **technique for deadlock recovery**

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

Prevent Deadlocks: Prevent the deadlock condition by negating one of the four conditions favouring the deadlock situation.

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

This can be achieved by implementing the following set of rules/guidelines in allocating resources to processes.

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

- Ensure that resource preemption (resource releasing) is possible at operating system level. This can be achieved by implementing the following set of rules/guidelines in resources allocation and releasing.
  1. Release all the resources currently held by a process if a request made by the process for a new resource is not able to fulfil immediately.
  2. Add the resources which are preempted (released) to a resource list describing the resources which the process requires to complete its execution.
  3. Reschedule the process for execution only when the process gets its old resources and the new resource which is requested by the process.

Imposing these criterions may introduce negative impacts like low resource utilisation and starvation of processes.

Livelock The *Livelock* condition is similar to the deadlock condition except that a process in livelock condition changes its state with time. While in deadlock a process enters in wait state for a resource and continues in that state forever without making any progress in the execution, in a livelock condition a process always does something but is unable to make any progress in the execution completion. The livelock condition is better explained with the real world example, two people attempting to cross each other in a narrow corridor. Both the persons move towards each side of the corridor to allow the opposite person to cross. Since the corridor is narrow, none of them are able to cross each other. Here both of the persons perform some action but still they are unable to achieve their target, cross each other. We will make the livelock, the scenario more clear in a later section–*The Dining Philosophers' Problem*, of this chapter.

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

10.8.1.3 The Dining Philosophers' Problem

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

Five philosophers (It can be '*n*'. The number 5 is taken for illustration) are sitting around a round table, involved in eating and brainstorming (Fig. 10.27). At any point of time each philosopher will be in any one of the three states: eating, hungry or brainstorming. (While eating the philosopher is not involved in brainstorming and while brainstorming the philosopher is not involved in eating). For eating, each philosopher requires 2 forks. There are only 5 forks available on the dining table ('*n*' for '*n*' number of philosophers) and they are arranged in a fashion one fork in between two philosophers. The philosopher can only use the forks on his/her immediate left and right that too in the order pickup the left fork first and then the right fork. Analyse the situation and explain the possible outcomes of this scenario.

Let's analyse the various scenarios that may occur in this situation.

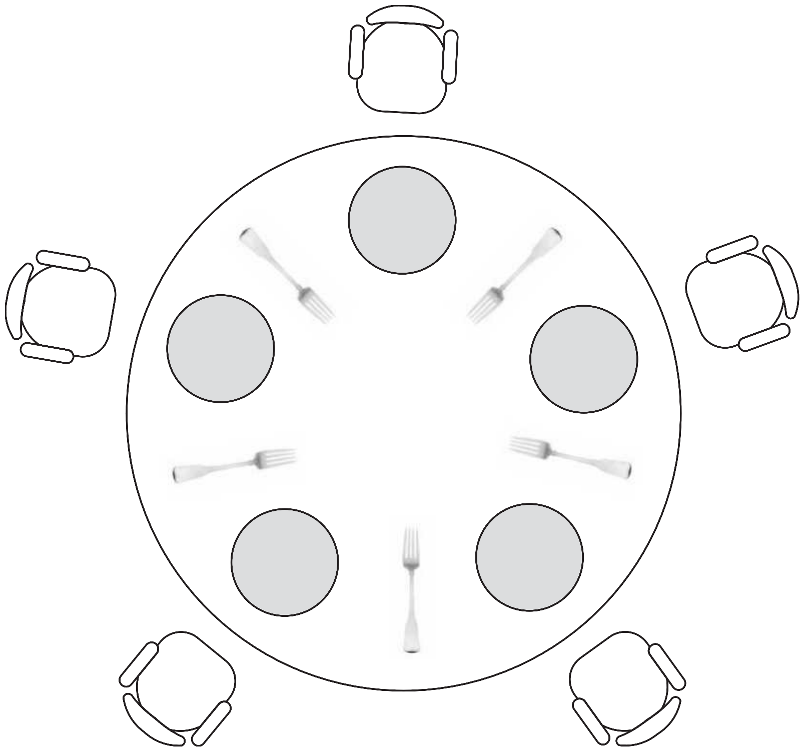


Fig. 10.27 Visualisation of the 'Dining Philosophers problem'

Scenario 1: All the philosophers involve in brainstorming together and try to eat together. Each philosopher picks up the left fork and is unable to proceed since two forks are required for eating the spaghetti present in the plate. Philosopher 1 thinks that Philosopher 2 sitting to the right of him/her will put the fork down and waits for it. Philosopher 2 thinks that Philosopher 3 sitting to the right of him/her will put the fork down and waits for it, and so on. This forms a circular chain of un-granted requests. If the philosophers continue in this state waiting for the fork from the philosopher sitting to the right of each, they will not make any progress in eating and this will result in *starvation* of the philosophers and *deadlock*.
Scenario 2: All the philosophers start brainstorming together. One of the philosophers is hungry and he/she picks up the left fork. When the philosopher is about to pick up the right fork, the philosopher sitting to his right also become hungry and tries to grab the left fork which is the right fork of his neighbouring philosopher who is trying to lift it, resulting in a '*Race condition*'.
Scenario 3: All the philosophers involve in brainstorming together and try to eat together. Each philosopher picks up the left fork and is unable to proceed, since two forks are required for eating the spaghetti present in the plate. Each of them anticipates that the adjacently sitting philosopher will put his/her fork down and waits for a fixed duration and after this puts the fork down. Each of them again tries to lift the fork after a fixed duration of time. Since all philosophers are trying to lift the fork at the same time, none of them will be able to grab two forks. This condition leads to *livelock* and *starvation* of philosophers, where each philosopher tries to do something, but they are unable to make any progress in achieving the target.

Figure 10.28 illustrates these scenarios.

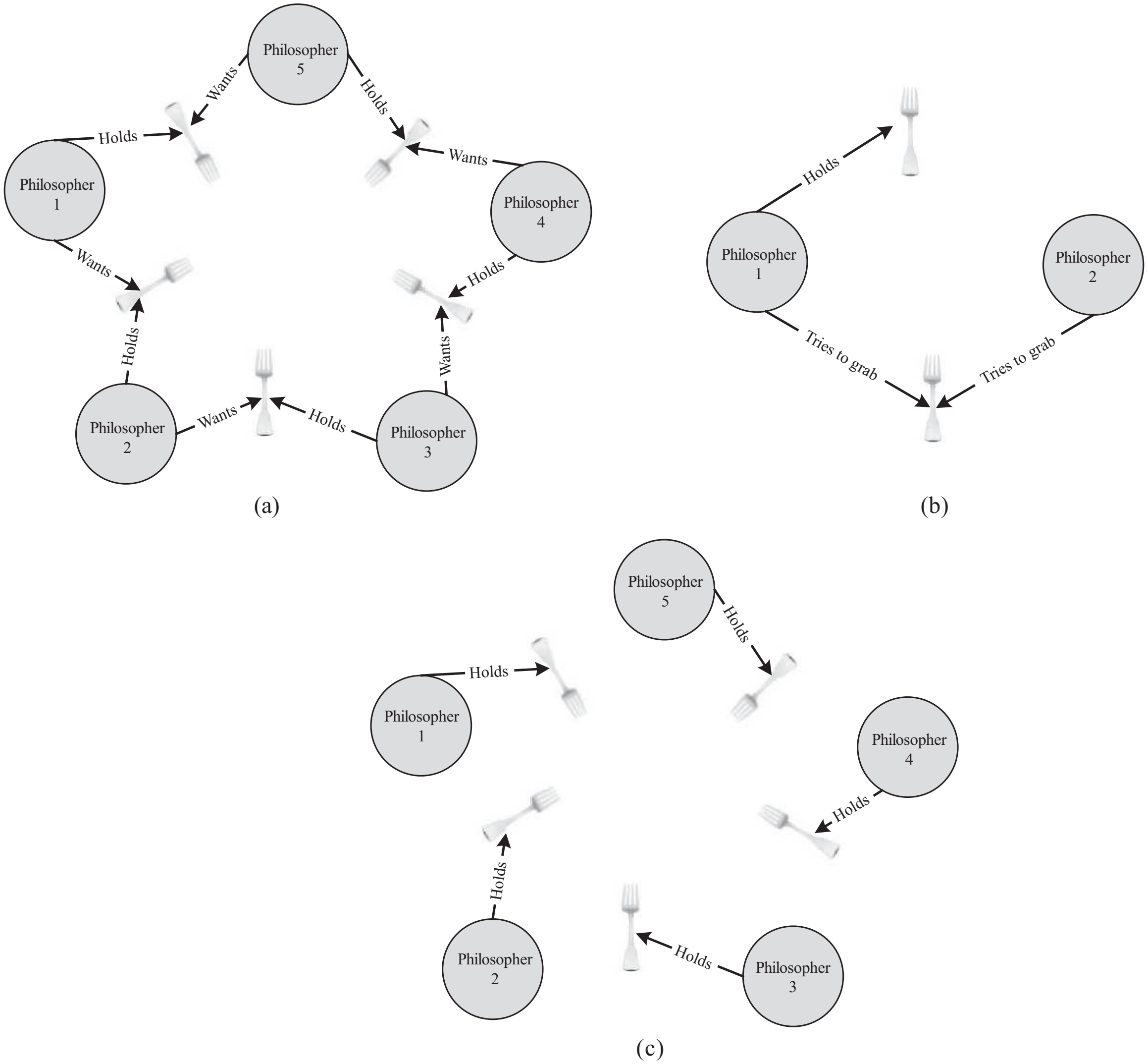


Fig. 10.28 The 'Real Problems' in the 'Dining Philosophers problem' (a) Starvation and Deadlock (b) Racing (c) Livelock and Starvation

Solution: We need to find out alternative solutions to avoid the *deadlock*, *livelock*, *racing* and *starvation* condition that may arise due to the concurrent access of forks by philosophers. This situation can be handled in many ways by allocating the forks in different allocation techniques including Round Robin allocation, FIFO allocation, etc. But the requirement is that the solution should be optimal, avoiding deadlock and starvation of the philosophers and allowing maximum number of philosophers to eat at a time. One solution that we could think of is:

- Imposing rules in accessing the forks by philosophers, like: The philosophers should put down the fork he/she already have in hand (left fork) after waiting for a fixed duration for the second fork (right fork) and should wait for a fixed time before making the next attempt.

This solution works fine to some extent, but, if all the philosophers try to lift the forks at the same time, a *livelock* situation is resulted.

Another solution which gives maximum concurrency that can be thought of is each philosopher acquires a semaphore (mutex) before picking up any fork. When a philosopher feels hungry he/she checks whether the philosopher sitting to the left and right of him is already using the fork, by checking the state of the associated semaphore. If the forks are in use by the neighbouring philosophers, the philosopher waits till the forks are available. A philosopher when finished eating puts the forks down and informs the philosophers sitting to his/her left and right, who are hungry (waiting for the forks), by signalling the semaphores associated with the forks. We will discuss about semaphores and mutexes at a latter section of this chapter. In the operating system context, the dining philosophers represent the processes and forks represent the resources. The dining philosophers' problem is an analogy of processes competing for shared resources and the different problems like racing, deadlock, starvation and livelock arising from the competition.

10.8.1.4 Producer-Consumer/Bounded Buffer Problem

Producer-Consumer problem is a common data sharing problem where two processes concurrently access a shared buffer with fixed size. A thread/process which produces data is called '*Producer thread/process*' and a thread/process which consumes the data produced by a producer thread/process is known as '*Consumer thread/process*'. Imagine a situation where the producer thread keeps on producing data and puts it into the buffer and the consumer thread keeps on consuming the data from the buffer and there is no synchronisation between the two. There may be chances where in which the producer produces data at a faster rate than the rate at which it is consumed by the consumer. This will lead to '*buffer overrun*' where the producer tries to put data to a full buffer. If the consumer consumes data at a faster rate than the rate at which it is produced by the producer, it will lead to the situation '*buffer under-run*' in which the consumer tries to read from an empty buffer. Both of these conditions will lead to inaccurate data and data loss. The following code snippet illustrates the producer-consumer problem

```
#include "stdafx.h"
#include <windows.h>
#include <stdio.h>
#define N 20       //Define buffer size as 20
int buffer[N];     //Shared buffer for producer & consumer

//*********************************************************************
//Producer thread
void producer_thread(void) {
 int x;
 while (true) {
  for (x = 0; x<N; x++)
  {
   //Fill buffer with random data
   buffer[x] = rand() % 1000;
   printf("Produced : Buffer[%d] = % 4d\n", x, buffer[x]);
   Sleep(25);
  }
 }
}
//*********************************************************************
//Consumer thread
```

```
void consumer_thread(void) {
 int y = 0, value;
 while (true) {
  for (y = 0; y<N; y++)
  {
   value = buffer[y];
   printf("Consumed : Buffer[%d] = % 4d\n", y, value);
   Sleep(20);
  }
 }
}
//*********************************************************************
//Main Thread
int main()
{
 DWORD thread_id;
 //Create Producer thread
 CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)producer_thread, NULL, 0,
 &thread_id);
 //Create Consumer thread
 CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)consumer_thread, NULL, 0,
 &thread_id);
 //Wait for some time and exit
 Sleep(500);
 return 0;
}
```

Here the 'producer thread' produces random numbers and puts it in a buffer of size 20. If the 'producer thread' fills the buffer fully it re-starts the filling of the buffer from the bottom. The 'consumer thread' consumes the data produced by the 'producer thread'. For consuming the data, the 'consumer thread' reads the buffer which is shared with the 'producer thread'. Once the 'consumer thread' consumes all the data, it starts consuming the data from the bottom of the buffer. These two threads run independently and are scheduled for execution based on the scheduling policies adopted by the OS. The different situations that may arise based on the scheduling of the 'producer thread' and 'consumer thread' is listed below.

1. 'Producer thread' is scheduled more frequently than the 'consumer thread': There are chances for overwriting the data in the buffer by the 'producer thread'. This leads to inaccurate data.
2. Consumer thread' is scheduled more frequently than the 'producer thread': There are chances for reading the old data in the buffer again by the 'consumer thread'. This will also lead to inaccurate data.

The output of the above program when executed on a Windows 10 machine is shown in Fig. 10.29.

The output shows that the consumer thread runs faster than the producer thread and most often leads to buffer under-run and thereby inaccurate data.

Note: It should be noted that the scheduling of the threads '*producer_thread*' and '*consumer_thread*' is OS kernel scheduling policy dependent and you may not get the same output all the time when you run this piece of code under Windows NT kernel (Say Windows 10 OS).

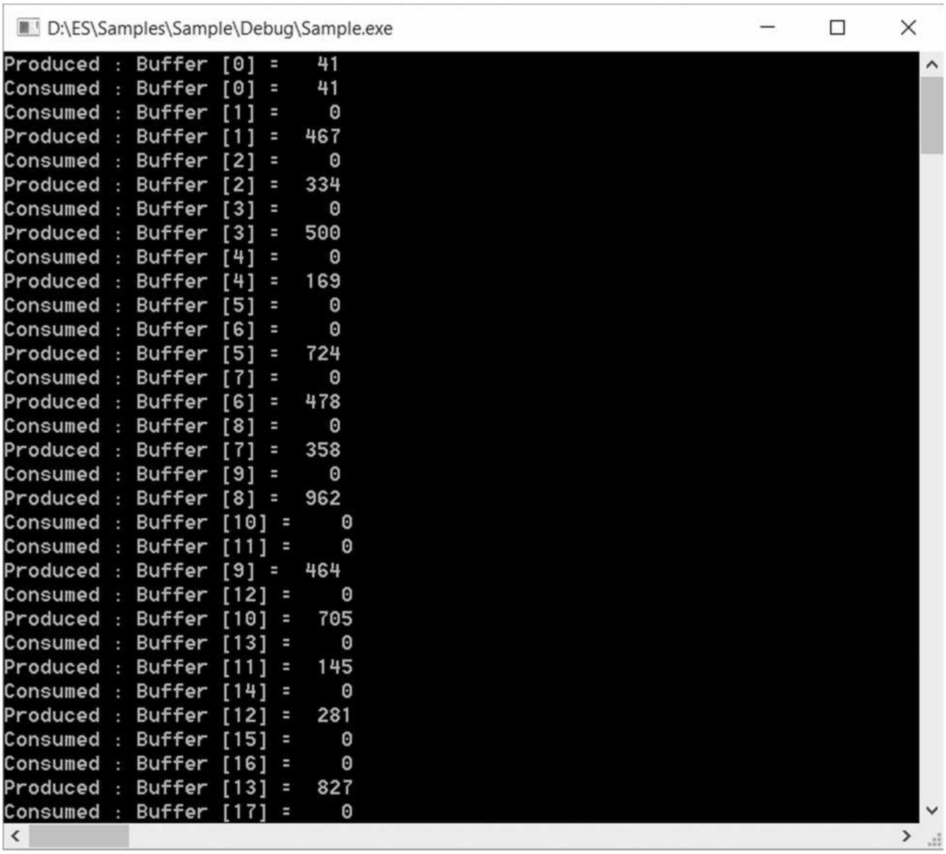


Fig. 10.29 Output of Win32 program illustrating producer consumer problem

The producer-consumer problem can be rectified in various methods. One simple solution is the '*sleep and wake-up*'. The '*sleep and wake-up*' can be implemented in various process synchronisation techniques like semaphores, mutex, monitors, etc. We will discuss it in a latter section of this chapter.

10.8.1.5 Readers-Writers Problem

The Readers-Writers problem is a common issue observed in processes competing for limited shared resources. The Readers-Writers problem is characterised by multiple processes trying to read and write shared data concurrently. A typical real-world example for the Readers-Writers problem is the banking system where one process tries to read the account information like available balance and the other process tries to update the available balance for that account. This may result in inconsistent results. If multiple processes try to read a shared data concurrently it may not create any impacts, whereas when multiple processes try to write and read concurrently it will definitely create inconsistent results. Proper synchronisation techniques should be applied to avoid the readers-writers problem. We will discuss about the various synchronisation techniques in a later section of this chapter.

10.8.1.6 Priority Inversion

Priority inversion is the byproduct of the combination of blocking based (lock based) process synchronisation and pre-emptive priority scheduling. '*Priority inversion*' is the condition in which a high priority task needs to wait for a low priority task to release a resource which is shared between the high priority task and the low priority task, and a medium priority task which doesn't require the shared resource continue its execution by preempting the low priority task (Fig. 10.30). Priority based preemptive scheduling technique ensures that a high priority task is always executed first, whereas the lock based process synchronisation mechanism (like mutex, semaphore, etc.) ensures that a process will not access a shared resource, which is currently in use by another process. The synchronisation technique is only interested in avoiding conflicts that may arise due to the concurrent access of the shared resources and not at all bothered about the priority of the process which tries to access the shared resource. In fact, the priority based preemption and lock based synchronisation are the two contradicting OS primitives. Priority inversion is better explained with the following scenario:

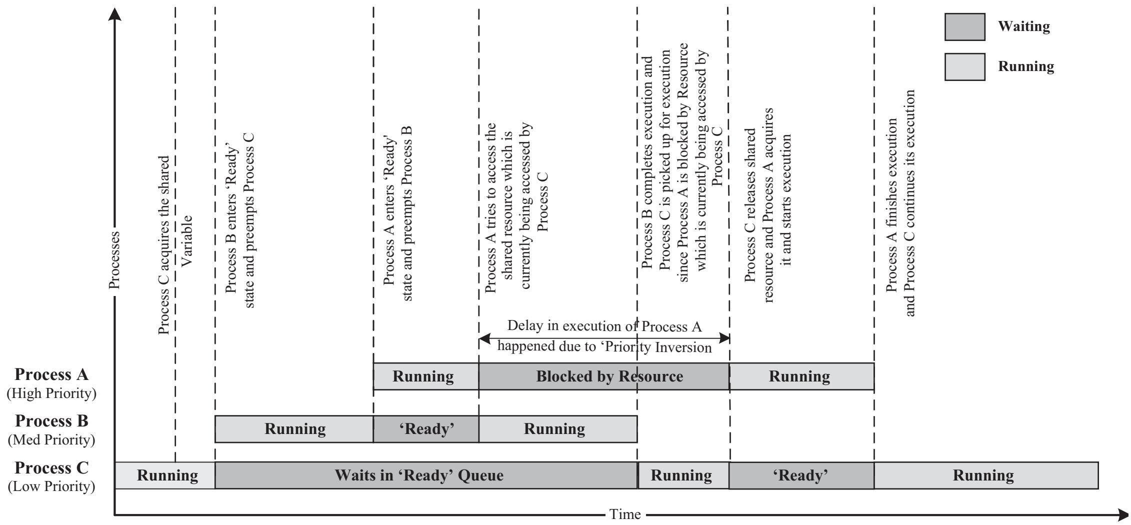


Fig. 10.30 Priority Inversion Problem

Let Process A, Process B and Process C be three processes with priorities High, Medium and Low respectively. Process A and Process C share a variable 'X' and the access to this variable is synchronised through a mutual exclusion mechanism like *Binary Semaphore S*. Imagine a situation where Process C is ready and is picked up for execution by the scheduler and 'Process C' tries to access the shared variable 'X'. 'Process C' acquires the 'Semaphore S' to indicate the other processes that it is accessing the shared variable 'X'. Immediately after 'Process C' acquires the 'Semaphore S', 'Process B' enters the 'Ready' state. Since 'Process B' is of higher priority compared to 'Process C', 'Process C' is preempted and 'Process B' starts executing. Now imagine 'Process A' enters the 'Ready' state at this stage. Since 'Process A' is of higher priority than 'Process B', 'Process B' is preempted and 'Process A' is scheduled for execution. 'Process A' involves accessing of shared variable 'X' which is currently being accessed by 'Process C'. Since 'Process C' acquired the semaphore for signalling the access of the shared variable 'X', 'Process A' will not be able to access it. Thus 'Process A' is put into blocked state (This condition is called Pending on resource). Now 'Process B' gets the CPU and it continues its execution until it relinquishes the CPU voluntarily or enters a wait state or preempted by another high priority task. The highest priority process 'Process A' has to wait till 'Process C' gets a chance to execute and release the semaphore. This produces unwanted delay in the execution of the high priority task which is supposed to be executed immediately when it was 'Ready'.

Priority inversion may be sporadic in nature but can lead to potential damages as a result of missing critical deadlines. Literally speaking, priority inversion 'inverts' the priority of a high priority task with that of a low priority task. Proper workaround mechanism should be adopted for handling the priority inversion problem. The commonly adopted priority inversion workarounds are:

Priority Inheritance: A low-priority task that is currently accessing (by holding the lock) a shared resource requested by a high-priority task temporarily '*inherits*' the priority of that high-priority task, from the moment the high-priority task raises the request. Boosting the priority of the low priority task to that of the priority of the task which requested the shared resource holding by the low priority task eliminates the preemption of the low priority task by other tasks whose priority are below that of the task requested the shared resource and thereby reduces the delay in waiting to get the resource requested by the high priority task. The priority of the low priority task which is temporarily boosted to high is brought to the original value when it releases the shared resource. Implementation of Priority inheritance workaround in the priority inversion problem discussed for Process A, Process B and Process C example will change the execution sequence as shown in Fig. 10.31.

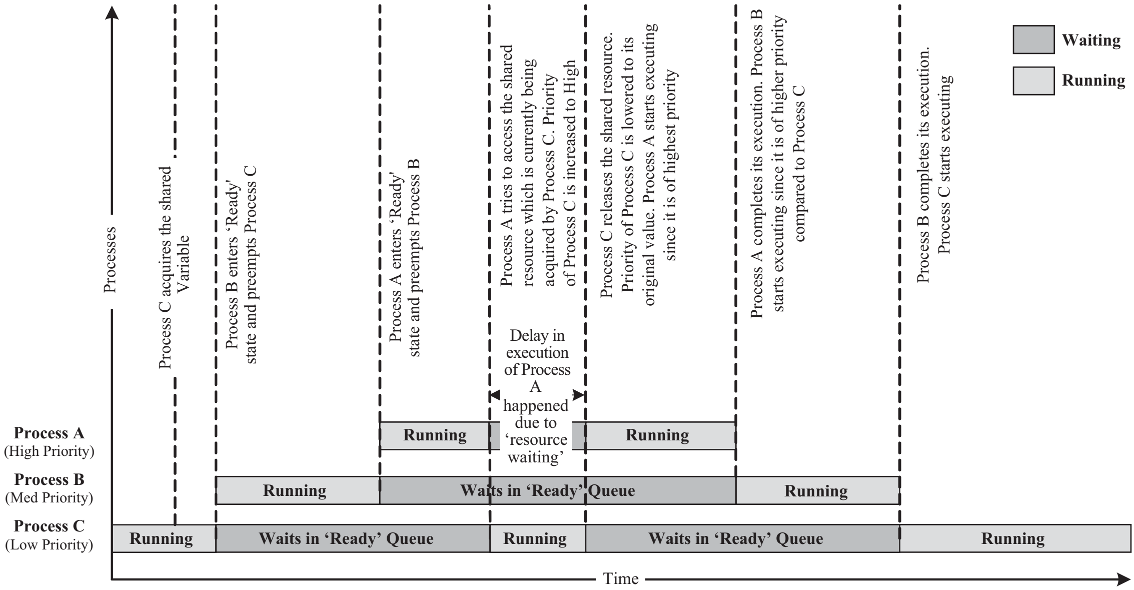


Fig. 10.31 Handling Priority Inversion Problem with Priority Inheritance

Priority inheritance is only a work around and it will not eliminate the delay in waiting the high priority task to get the resource from the low priority task. The only thing is that it helps the low priority task to continue its execution and release the shared resource as soon as possible. The moment, at which the low priority task releases the shared resource, the high priority task kicks the low priority task out and grabs the CPU – A true form of selfishness☺. Priority inheritance handles priority inversion at the cost of run-time overhead at scheduler. It imposes the overhead of checking the priorities of all tasks which tries to access shared resources and adjust the priorities dynamically.

Priority Ceiling: In '*Priority Ceiling*', a priority is associated with each shared resource. The priority associated to each resource is the priority of the highest priority task which uses this shared resource. This priority level is called 'ceiling priority'. Whenever a task accesses a shared resource, the scheduler elevates the priority of the task to that of the ceiling priority of the resource. If the task which accesses the shared resource is a low priority task, its priority is temporarily boosted to the priority of the highest priority task to which the resource is also shared. This eliminates the pre-emption of the task by other medium priority tasks leading to priority inversion. The priority of the task is brought back to the original level once the task completes the accessing of the shared resource. 'Priority Ceiling' brings the added advantage of sharing resources without the need for synchronisation techniques like locks. Since the priority of the task accessing a shared resource is boosted to the highest priority of the task among which the resource is shared, the concurrent access of shared resource is automatically handled. Another advantage of 'Priority Ceiling' technique is that all the overheads are at compile time instead of run-time. Implementation of 'priority ceiling' workaround in the priority inversion problem discussed for Process A, Process B and Process C example will change the execution sequence as shown in Fig. 10.32.

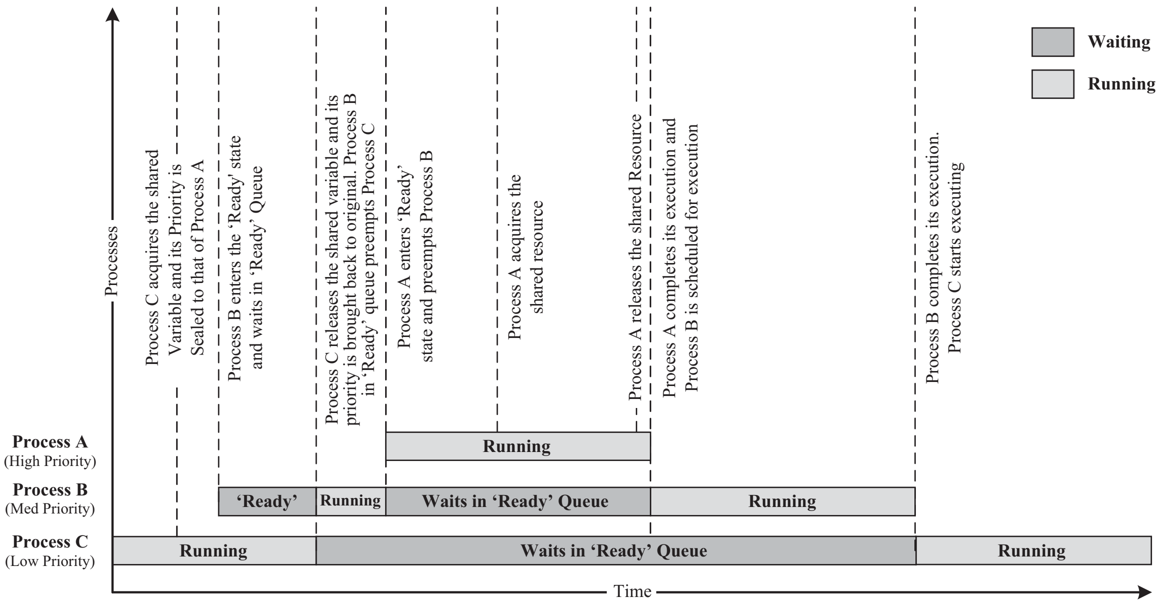


Fig. 10.32 Handling Priority Inversion Problem with Priority Ceiling

The biggest drawback of 'Priority Ceiling' is that it may produce *hidden priority inversion*. With 'Priority Ceiling' technique, the priority of a task is always elevated no matter another task wants the shared resources. This unnecessary priority elevation always boosts the priority of a low priority task to that of the highest priority tasks among which the resource is shared and other tasks with priorities higher than that of the low priority task is not allowed to preempt the low priority task when it is accessing a shared resource. This always gives the low priority task the luxury of running at high priority when accessing shared resources☺.

10.8.2 Task Synchronisation Techniques

So far we discussed about the various task/process synchronisation issues encountered in multitasking systems due to concurrent resource access. Now let's have a discussion on the various techniques used for synchronisation in concurrent access in multitasking. Process/Task synchronisation is essential for

1. Avoiding conflicts in resource access (racing, deadlock, starvation, livelock, etc.) in a multitasking environment.

2. Ensuring proper sequence of operation across processes. The producer consumer problem is a typical example for processes requiring proper sequence of operation. In producer consumer problem, accessing the shared buffer by different processes is not the issue, the issue is the writing process should write to the shared buffer only if the buffer is not full and the consumer thread should not read from the buffer if it is empty. Hence proper synchronisation should be provided to implement this sequence of operations.
3. Communicating between processes.

The code memory area which holds the program instructions (piece of code) for accessing a shared resource (like shared memory, shared variables, etc.) is known as '*critical section*'. In order to synchronise the access to shared resources, the access to the critical section should be exclusive. The exclusive access to critical section of code is provided through mutual exclusion mechanism. Let us have a look at how mutual exclusion is important in concurrent access. Consider two processes *Process* A and *Process B* running on a multitasking system. *Process A* is currently running and it enters its critical section. Before *Process A* completes its operation in the critical section, the scheduler preempts *Process A* and schedules *Process B* for execution (*Process B* is of higher priority compared to *Process A*). *Process B* also contains the access to the critical section which is already in use by *Process A*. If *Process B* continues its execution and enters the critical section which is already in use by *Process A*, a racing condition will be resulted. A mutual exclusion policy enforces mutually exclusive access of critical sections.

Mutual exclusions can be enforced in different ways. Mutual exclusion blocks a process. Based on the behaviour of the blocked process, mutual exclusion methods can be classified into two categories. In the following section we will discuss them in detail.

10.8.2.1 Mutual Exclusion through Busy Waiting/Spin Lock

'*Busy waiting*' is the simplest method for enforcing mutual exclusion. The following code snippet illustrates how 'Busy waiting' enforces mutual exclusion.

```
//Inside parent thread/main thread corresponding to a process
bool bFlag; //Global declaration of lock Variable.
bFlag= FALSE; //Initialise the lock to indicate it is available.
//.........................................................................................................................................
//Inside the child threads/threads of a process
while(bFlag == TRUE); //Check the lock for availability
bFlag=TRUE; //Lock is available. Acquire the lock
//Rest of the source code dealing with shared resource access
```

The 'Busy waiting' technique uses a lock variable for implementing mutual exclusion. Each process/thread checks this lock variable before entering the critical section. The lock is set to '***1***' by a process/thread if the process/thread is already in its critical section; otherwise the lock is set to '***0***'. The major challenge in implementing the lock variable based synchronisation is the non-availability of a single atomic instruction[¶] which combines the reading, comparing and setting of the lock variable. Most often the three different operations related to the locks, viz. the operation of Reading the lock variable, checking its present value and setting it are achieved with multiple low level instructions. The low level implementation of these operations are dependent on the underlying processor instruction set and the (cross) compiler in use. The low level implementation of the 'Busy waiting' code snippet, which we discussed earlier, under Windows 10 Operating system running on an Intel i7 dual core Processor is given below. The code snippet is compiled with Microsoft Visual Studio 2013 compiler.

¶ Atomic Instruction: Instruction whose execution is uninterruptible.

```
--- d:\es\samples\rev1\counter.cpp ------------------------------------
1:   #include "stdafx.h"

2:   #include <stdio.h>
3:   #include <windows.h>
4:
5:   int main()
6:   {
//Code memory     Opcode       Operand
00A21380          push         ebp
00A21381          mov          ebp,esp
00A21383          sub          esp,0CCh
00A21389          push         ebx
00A2138A          push         esi
00A2138B          push         edi
00A2138C          lea          edi,[ebp+FFFFFF34h]
00A21392          mov          ecx,33h
00A21397          mov          eax,0CCCCCCCCh
00A2139C          rep stos     dword ptr es:[edi]
7:   //Inside parent thread/ main thread corresponding to a process
8:   bool bFlag; //Global declaration of lock Variable.
9:   bFlag = FALSE; //Initialise the lock to indicate it is available.
00A2139E          mov          byte ptr [ebp-5],0
10:  //..........................................................................
11:  //Inside the child threads/ threads of a process
12:  while (bFlag == TRUE); //Check the lock for availability
00A213A2          movzx        eax,byte ptr [ebp-5]
00A213A6          cmp          eax,1
00A213A9          jne          main+2Dh (0A213ADh)
00A213AB          jmp          main+22h (0A213A2h)
13:  bFlag = TRUE; //Lock is available. Acquire the lock
00A213AD          mov          byte ptr [ebp-5],1
14:  //Rest of the source code dealing with shared resource access
```

The assembly language instructions reveals that the two high level instructions (*while(bFlag==false);* and *bFlag=true;*), corresponding to the operation of reading the lock variable, checking its present value and setting it is implemented in the processor level using five low level instructions. Imagine a hypothetical situation where 'Process 1' read the lock variable and tested it and found that the lock is available and it is about to set the lock for acquiring the critical section (Fig. 10.33). But just before 'Process 1' sets the lock variable, 'Process 2' preempts 'Process 1' and starts executing. 'Process 2' contains a critical section code and it tests the lock variable for its availability. Since 'Process 1' was unable to set the lock variable, its state is still '**0**' and 'Process 2' sets it and acquires the critical section. Now the scheduler preempts 'Process 2' and schedules 'Process 1' before 'Process 2' leaves the critical section. Remember, 'Process 1' was preempted at a point just before setting the lock variable ('Process 1' has already tested the lock variable just before it is preempted and found that the lock is available). Now 'Process 1' sets the lock variable and enters the critical section. It violates the mutual exclusion policy and may produce unpredicted results.

The above issue can be effectively tackled by combining the actions of reading the lock variable, testing its state and setting the lock into a single step. This can be achieved with the combined hardware and software support. Most of the processors support a single instruction '*Test and Set Lock (TSL)*' for testing and setting the lock variable. The '*Test and Set Lock (TSL)*' instruction call copies the value of the lock variable and sets it to a nonzero value. It should be noted that the implementation and usage of '*Test and Set Lock (TSL)*' instruction is processor architecture dependent. The *Intel 486* and the above family of processors support the '*Test and Set Lock (TSL)*' instruction with a special instruction *CMPXCHG*—Compare and Exchange. The usage of *CMPXCHG* instruction is given below.

```
CMPXCHG dest,src
```

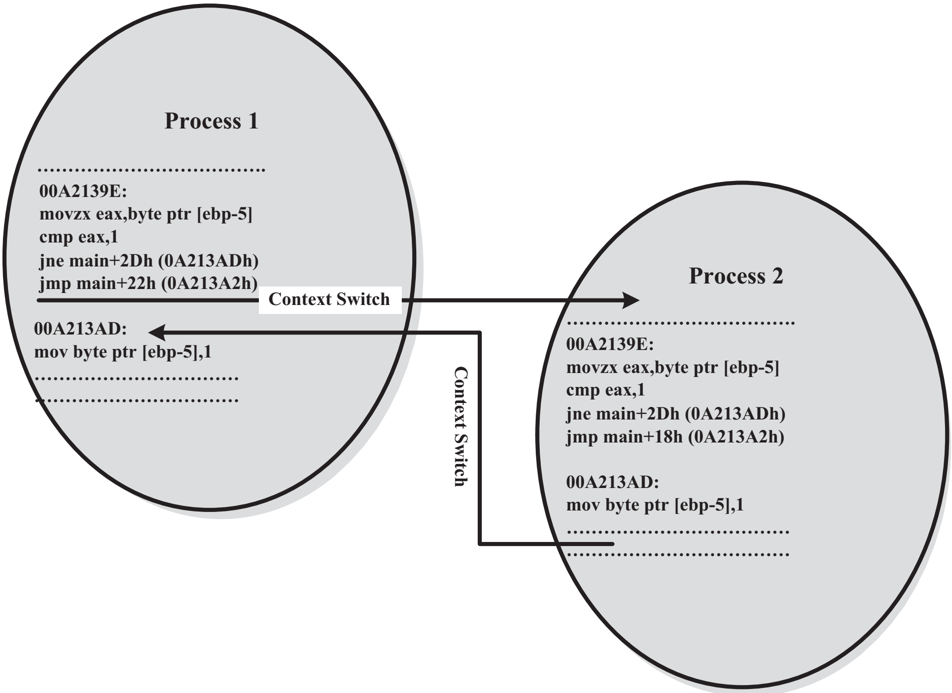


Fig. 10.33 Illustration of the issues with locks

This instruction compares the Accumulator (*EAX* register) with 'dest'. If the Accumulator and 'dest' contents are equal, 'dest' is loaded with 'src'. If not, the Accumulator is loaded with 'dest'. Executing this instruction changes the six status bits of the Program Control and Status register *EFLAGS*. The destination ('dest') can be a register or a memory location. The source ('src') is always a register. From a programmer's perspective the operation of *CMPXCHG* instruction can be viewed as:

```
if (accumulator == destination)
{
    ZF = 1; //Set the Zero Flag of EFLAGS Register
    destination = source;
}
else
{
    ZF = 0; //Reset the Zero Flag of EFLAGS Register
    accumulator = destination;
}
```

The process/thread checks the lock variable to see whether its state is '0' and sets its state to '1' if its state is '0', for acquiring the lock. To implement this at the 486 processor level, load the accumulator with '0' and a general purpose register with '1' and compare the memory location holding the lock variable with accumulator using *CMPXCHG* instruction. This instruction makes the accessing, testing and modification of the lock variable a single atomic instruction. How the *CMPXCHG* instruction support provided by the Intel® family of processors (486 and above) is made available to processes/threads is OS kernel implementation dependent. Let us see how this feature is implemented by Windows Operating systems. Windows Embedded Compact/Windows NT kernels support the compare and exchange hardware feature provided by Intel® family of processors, through the API call *InterlockedCompareExchange (LPLONG Destination, LONG Exchange, LONG Comperand)*. The variable *Destination* is the long pointer to the destination variable. The *Destination* variable should be of type '*long*'. The variable *Exchange* represents the exchange value. The value of *Destination* variable is replaced with the value of *Exchange* variable. The variable *Comperand* specifies the value which needs to be compared with the value of *Destination* variable. The function returns the initial value of the variable '*Destination*'. The following code snippet illustrates the usage of this API call for thread/process synchronisation.

```
//Inside parent thread/ main thread corresponding to a process
long bFlag; //Global declaration of lock Variable.
bFlag=0; //Initialise the lock to indicate it is available.
//................................................................................................................
//Inside the child threads/ threads of a process
//Check the lock for availability & acquire the lock if available.
while (InterlockedCompareExchange (&bFlag, 1, 0) == 1);

//Rest of the source code dealing with shared resource access
```

The *InterlockedCompareExchange* function is implemented as '*Compiler intrinsic function*'. The 'code for *Compiler intrinsic functions*' are inserted inline while compiling the code. This avoids the function call overhead and makes use of the built-in knowledge of the optimisation technique for intrinsic functions. The compiler can be instructed to use the intrinsic implementation for a function using the compiler directive *#pragma intrinsic (intrinsic-function-name)*. A sample implementation of the *InterlockedCompareExchange* interlocked intrinsic function for desktop Windows OS is given below.

```
#include "stdafx.h"
#include <intrin.h>
#include <windows.h>
long bFlag; //Global declaration of lock Variable.
//Declare InterlockedCompareExchange as intrinsic function
#pragma intrinsic(_InterlockedCompareExchange)
void child_thread(void)
{
//Inside the child thread of a process
//Check the lock for availability & acquire the lock if available.
//The lock can be set by any other threads
while (_InterlockedCompareExchange (&bFlag, 1, 0) == 1);
//Rest of the source code dealing with shared resource access
//..................................................................
return;
```

```
}
//.............................................................
int _tmain(int argc, _TCHAR* argv[])
{
//Inside parent thread/ main thread corresponding to a process
DWORD thread_id;
//Define handle to the child thread

HANDLE tThread;
//Initialise the lock to indicate it is available.
bFlag =0;
//Create child thread
tThread = CreateThread  (NULL,0,
                        (LPTHREAD_START_ROUTINE) child_thread,
                        NULL, 0, &thread_id);
if(NULL== tThread)
{
//Child thread creation failed.
printf ("Creation of Child thread failed. Error Code =
%d",GetLastError());
return -1;
}
//Wait for the completion of the child thread.
WaitForSingleObject(tThread,INFINITE);
return 0;
}
```

Note: Visual Studio 2005 or a later version of the compiler, which supports interlocked intrinsic functions, is required for compiling this application. The assembly code generated for the intrinsic interlocked function *while (_InterlockedCompareExchange (&bFlag, 1, 0) == 1);* when compiled using Visual Studio 2013 compiler, on Windows 10 platform running on an Intel® i7 Dual core processor is given below. It clearly depicts the usage of the cmpxchg instruction

```
//Inside the child thread of a process
//Check the lock for availability & acquire the lock if available.
//The lock can be set by any other threads
while (_InterlockedCompareExchange(&bFlag, 1, 0) == 1);
012013EE    mov         ecx,1
012013F3    mov         edx,1208130h
012013F8    xor         eax,eax
012013FA    lock cmpxchg dword ptr [edx],ecx
012013FE    cmp         eax,1
01201401    jne         child_thread+35h (01201405h)
01201403    jmp         child_thread+1Eh (012013EEh)
//Rest of the source code dealing with shared resource access
//.................................................................
```

The Intel 486 and above family of processors provide hardware level support for atomic execution of increment and decrement operations also. The *XADD* low level instruction implements atomic execution of increment and decrement operations. Windows Embedded Compact/NT kernel makes these features available to the users through a set of *Interlocked* function API calls. The API call *InterlockedIncrement (LPLONG lpAddend)* increments the value of the variable pointed by *lpAddend* and the API *InterlockedDecrement (LPLONG lpAddend)* decrements the value of the variable pointed by *lpAddend*.

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

The interlocked operations are the most efficient synchronisation primitives when compared to the classic lock based synchronisation mechanism. Interlocked function based synchronisation technique brings the following value adds.

- The interlocked operation is free from waiting. Unlike the mutex, semaphore and critical section synchronisation objects which may require waiting on the object, if they are not available at the time of request, the interlocked function simply performs the operation and returns immediately. This avoids the blocking of the thread which calls the interlocked function.
- The interlocked function call is directly converted to a processor specific instruction and there is no user mode to kernel mode transition as in the case of mutex, semaphore and critical section objects. This avoids the user mode to kernel mode transition delay and thereby increases the overall performance.

The types of interlocked operations supported by an OS are underlying processor hardware dependent and so they are limited in functionality. Normally the bit manipulation (Boolean) operations are not supported by interlocked functions. Also the interlocked operations are limited to integer or pointer variables only. This limits the possibility of extending the interlocked functions to variables of other types. Under windows operating systems, each process has its own virtual address space and so the interlocked functions can only be used for synchronising the access to a variable that is shared by multiple threads of a process (Multiple threads of a process share the same address space) (Intra Process Synchronisation). The interlocked functions can be extended for synchronising the access of the variables shared across multiple processes if the variable is kept in shared memory.

10.8.2.2 Mutual Exclusion through Sleep & Wakeup

The '*Busy waiting*' mutual exclusion enforcement mechanism used by processes makes the CPU always busy by checking the lock to see whether they can proceed. This results in the wastage of CPU time and leads to high power consumption. This is not affordable in embedded systems powered on battery, since it affects the battery backup time of the device. An alternative to '*busy waiting*' is the '*Sleep & Wakeup*' mechanism. When a process is not allowed to access the critical section, which is currently being locked by another process, the process undergoes '*Sleep*' and enters the '*blocked*' state. The process which is blocked on waiting for access to the critical section is awakened by the process which currently owns the critical section. The process which owns the critical section sends a wakeup message to the process, which is sleeping as a result of waiting for the access to the critical section, when the process leaves the critical section. The '*Sleep & Wakeup*' policy for mutual exclusion can be implemented in different ways. Implementation of this policy is OS kernel dependent. The following section describes the important techniques for '*Sleep & Wakeup*' policy implementation for mutual exclusion by Windows NT/CE OS kernels.

Semaphore Semaphore is a sleep and wakeup based mutual exclusion implementation for shared resource access. Semaphore is a system resource and the process which wants to access the shared resource can first acquire this system object to indicate the other processes which wants the shared resource that the shared resource is currently acquired by it. The resources which are shared among a process can be either for exclusive use by a process or for using by a number of processes at a time. The display device of an embedded system is a typical example for the shared resource which needs exclusive access by a process. The Hard disk (secondary storage) of a system is a typical example for sharing the resource among a limited number of multiple processes. Various processes can access the different sectors of the hard-disk concurrently. Based on the implementation of the sharing limitation of the shared resource, semaphores are classified into two; namely '*Binary Semaphore*' and '*Counting Semaphore*'. The binary semaphore provides exclusive access to shared resource by allocating the resource to a single process at a time and not allowing the other processes to access it when it is being owned by a process. The implementation of binary semaphore is OS kernel dependent. Under certain OS kernel it is referred as *mutex*. Unlike a binary semaphore, the '*Counting Semaphore*' limits the access of resources by a fixed number of processes/threads. '*Counting Semaphore*' maintains a count between zero and a maximum value. It limits the usage of the resource to the maximum value of the count supported by it. The state of the counting semaphore object is set to 'signalled' when the count of the object is greater than zero. The count associated with a '*Semaphore object*' is decremented by one when a process/thread acquires it and the count is incremented by one when a process/thread releases the '*Semaphore object*'. The state of the '*Semaphore object*' is set to non-signalled when the semaphore is acquired by the maximum number of processes/threads that the semaphore can support (i.e. when the count associated with the '*Semaphore object*' becomes zero). A real world example for the counting semaphore concept is the dormitory system for accommodation (Fig. 10.34). A dormitory contains a fixed number of beds (say 5) and at any point of time it can be shared by the maximum number of users supported by the dormitory. If a person wants to avail the dormitory facility, he/she can contact the dormitory caretaker for checking the availability. If beds are available in the dorm the caretaker will hand over the keys to the user. If beds are not available currently, the user can register his/her name to get notifications when a slot is available. Those who are availing the dormitory shares the dorm facilities like TV, telephone, toilet, etc. When a dorm user vacates, he/she gives the keys back to the caretaker. The caretaker informs the users, who booked in advance, about the dorm availability.

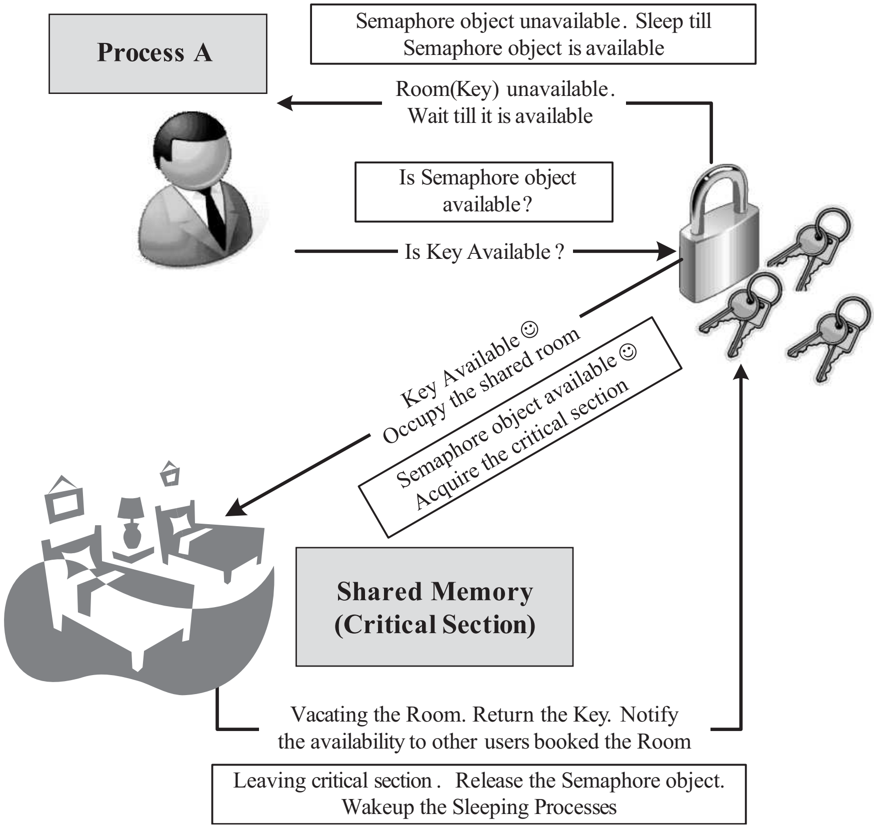


Fig. 10.34 The Concept of Counting Semaphore

The creation and usage of '*counting semaphore object*' is OS kernel dependent. Let us have a look at how we can implement semaphore based synchronisation for the '*Racing*' problem we discussed in the beginning, under the Windows kernel. The following code snippet explains the same.

```
##include "stdafx.h"
#include <stdio.h>
#include <windows.h>
#define MAX_SEMAPHORE_COUNT 1   //Make the semaphore object for exclusive use
#define thread_count 2          //No.of Child Threads
//*********************************************************************
//counter is an integer variable and Buffer is a byte array shared //between
two threads Process_A and Process_B
char Buffer[10] = { 1,2,3,4,5,6,7,8,9,10 };
short int counter = 0;
//Define the handle to Semaphore object
HANDLE hSemaphore;
//*********************************************************************
// Child Thread 1
void Process_A(void) {
```

```
 int i;
 for (i = 0; i<5; i++)
 {
  if (Buffer[i] > 0)
  {
   //Wait for the signaling of Semaphore object
   WaitForSingleObject(hSemaphore, INFINITE);
   //Semaphore is acquired
   counter++;
   printf("Process A : Counter = %d\n", counter);
   //Release the Semaphore Object
   if (!ReleaseSemaphore(
                     hSemaphore, // handle to semaphore
                     1, // increase count by one
                     NULL)) // not interested in previous count
   {
    //Semaphore Release failed. Print Error code & return.
    printf("Release Semaphore Failed with Error Code : %d\n",
    GetLastError());
    return;
   }
  }
 }
 return;
}
//*********************************************************************
// Child Thread 2
void Process_B(void) {
 int j;
 for (j = 5; j<10; j++)
 {
  if (Buffer[j] > 0)
  {
   //Wait for the signalling of Semaphore object
   WaitForSingleObject(hSemaphore, INFINITE);
   //Semaphore is acquired
   counter++;
   printf("Process B : Counter = %d\n", counter);
   //Release Semaphore
   if (!ReleaseSemaphore(
                     hSemaphore, // handle to semaphore
                     1, // increase count by one
                     NULL)) // not interested in previous count
   {
    //Semaphore Release failed. Print Error code &
    //return.
    printf("Release Semaphore Failed Error Code : %d\n", GetLastError());
    return;
```

```
    }
   }
  }
 return;
}
//*******************************************************************
// Main Thread
void main() {
 //Define HANDLE for child threads
 HANDLE child_threads[thread_count];
 DWORD thread_id;
 int i;
 //Create Semaphore object
 hSemaphore = CreateSemaphore(
             NULL, // default security attributes
             MAX_SEMAPHORE_COUNT, // initial count. Create as signaled
             MAX_SEMAPHORE_COUNT, // maximum count
            TEXT("Semaphore")); // Semaphore object with name "Semaphore"
 if (NULL == hSemaphore)
 {
  printf("Semaphore Object Creation Failed : Error Code : %d",
  GetLastError());
  //Semaphore Object Creation failed. Return
  return;
 }
 //Create Child thread 1
 child_threads[0] = CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)Process_A,
                                (LPVOID)0, 0, &thread_id);
 //Create Child thread 2
 child_threads[1] = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Process_B,
                                (LPVOID)0, 0, &thread_id);
 //Check the success of creation of child threads
 for (i = 0; i<thread_count; i++)
 {
  if (NULL == child_threads[i])
  {
   //Child thread creation failed.
   printf("Child thread Creation failed with Error Code : %d",
   GetLastError());
   return;
  }
 }
 // Wait for the termination of child threads
 WaitForMultipleObjects(thread_count, child_threads, TRUE, INFINITE);
 //Close handles of child threads
 for (i = 0; i < thread_count; i++)
  CloseHandle(child_threads[i]);
 //Close Semaphore object handle
```

```
 CloseHandle(hSemaphore);
 return;
}
```

Please refer to the Online Learning Centre for details on the various Win32 APIs used in the program for counting semaphore creation, acquiring, signalling, and releasing. The VxWorks and MicroC/OS-II Real-Time kernels also implements the Counting semaphore based task synchronisation/shared resource access. We will discuss them in detail in a later chapter.

Counting Semaphores are similar to *Binary Semaphores* in operation. The only difference between *Counting Semaphore* and *Binary Semaphore* is that *Binary Semaphore* can only be used for exclusive access, whereas *Counting Semaphores* can be used for both exclusive access (by restricting the maximum count value associated with the semaphore object to one (1) at the time of creation of the semaphore object) and limited access (by restricting the maximum count value associated with the semaphore object to the limited number at the time of creation of the semaphore object).

Binary Semaphore (Mutex) Binary Semaphore (Mutex) is a synchronisation object provided by OS for process/thread synchronisation. Any process/thread can create a '*mutex object*' and other processes/threads of the system can use this '*mutex object*' for synchronising the access to critical sections. Only one process/thread can own the '*mutex object*' at a time. The state of a mutex object is set to signalled when it is not owned by any process/thread, and set to non-signalled when it is owned by any process/thread. A real world example for the mutex concept is the hotel accommodation system (lodging system) Fig. 10.35. The rooms in a hotel are shared for the public. Any user who pays and follows the norms of the hotel can avail the

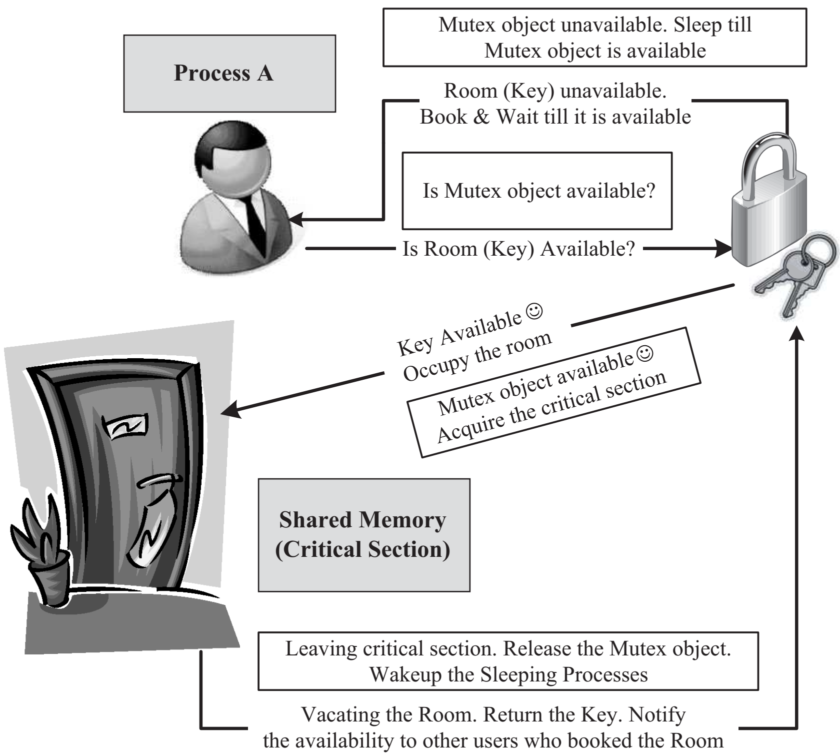


Fig. 10.35 The Concept of Binary Semaphore (Mutex)

rooms for accommodation. A person wants to avail the hotel room facility can contact the hotel reception for checking the room availability (see Fig. 10.35). If room is available the receptionist will handover the room key to the user. If room is not available currently, the user can book the room to get notifications when a room is available. When a person gets a room he/she is granted the exclusive access to the room facilities like TV, telephone, toilet, etc. When a user vacates the room, he/she gives the keys back to the receptionist. The receptionist informs the users, who booked in advance, about the room's availability.

Let's see how we can implement mutual exclusion with mutex object in the '*Racing*' problem example given under the section '*Racing*', under Windows kernel.

```
#include "stdafx.h"
#include <stdio.h>
#include <windows.h>
#define thread_count 2 //No.of Child Threads
//*******************************************************************
//counter is an integer variable and Buffer is a byte array shared
//between two
//threads Process_A and Process_B
char Buffer[10] = { 1,2,3,4,5,6,7,8,9,10 };
short int counter = 0;
//Define the handle to Mutex Object
HANDLE hMutex;
//*******************************************************************
// Child Thread 1
void Process_A(void) {
 int i;
 for (i = 0; i<5; i++)
 {
  if (Buffer[i] > 0)
  {
   //Wait for signaling of the Mutex object
   WaitForSingleObject(hMutex, INFINITE);
   //Mutex is acquired
   counter++;
   printf("Process A : Counter = %d\n", counter);
   //Release the Mutex Object
   if (!ReleaseMutex(hMutex)) // handle to Mutex Object
   {
    //Mutex object Releasing failed. Print Error code & return.
    printf("Release Mutex Failed with Error Code : %d\n", GetLastError());
    return;
   }
  }
 }
 return;
}
//*******************************************************************
// Child Thread 2
void Process_B(void) {
```

```
 int j;
 for (j = 5; j<10; j++)
 {
  if (Buffer[j] > 0)
  {
   //Wait for signaling of the Mutex object
   WaitForSingleObject(hMutex, INFINITE);
   //Mutex object is acquired
   counter++;
   printf("Process B : Counter = %d\n", counter);
   //Release Mutex object
   if (!ReleaseMutex(hMutex)) // handle to Mutex Object
   {
    //Mutex object Release failed. Print Error code & return.
    printf("Release Mutex Failed with Error Code : %d\n", GetLastError());
    return;
   }
  }
 }
 return;
}
//*********************************************************************
// Main Thread
void main() {
 //Define HANDLE for child threads
 HANDLE child_threads[thread_count];
 DWORD thread_id;
 int i;
 //Create Mutex object
 hMutex = CreateMutex(
                NULL, // default security attributes
                FALSE, // Not initial ownership
                TEXT("Mutex")); // Mutex object with name "Mutex"
 if (NULL == hMutex)
 {
  printf("Mutex Object Creation Failed : Error Code : %d",GetLastError());
  //Mutex Object Creation failed. Return
  return;
 }
 //Create Child thread 1
 child_threads[0] = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Process_A,
                              (LPVOID)0, 0, &thread_id);
 //Create Child thread 2
 child_threads[1] = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Process_B,
                              (LPVOID)0, 0, &thread_id);
 //Check the success of creation of child threads
 for (i = 0; i<thread_count; i++)
 {
```

```
   if (NULL == child_threads[i])
   {
    //Child thread creation failed.
   printf("Child thread Creation failed with Error Code : %d", GetLastError());
    return;
   }
  }
  // Wait for the termination of child threads
  WaitForMultipleObjects(thread_count, child_threads, TRUE, INFINITE);
  //Close child thread handles
  for (i = 0; i < thread_count; i++)
  {
   CloseHandle(child_threads[i]);
  }
  //Close Mutex object handle
  CloseHandle(hMutex);
  return;
 }
```

Please refer to the Online Learning Centre for details on the various Win32 APIs used in the program for mutex creation, acquiring, signalling, and releasing.

The mutual exclusion semaphore is a special implementation of the binary semaphore by certain real-time operating systems like VxWorks and MicroC/OS-II to prevent priority inversion problems in shared resource access. The mutual exclusion semaphore has an option to set the priority of a task owning it to the highest priority of the task which is being pended while attempting to acquire the semaphore which is already in use by a low priority task. This ensures that the low priority task which is currently holding the semaphore, when a high priority task is waiting for it, is not pre-empted by a medium priority task. This is the mechanism supported by the mutual exclusion semaphore to prevent priority inversion.

VxWorks kernel also supports binary semaphores for synchronising shared resource access. We will discuss about it in detail in a later chapter.

Critical Section Objects In Windows Embedded Compact, the '*Critical Section object*' is same as the '*mutex object*' except that '*Critical Section object*' can only be used by the threads of a single process (Intra process). The piece of code which needs to be made as '*Critical Section*' is placed at the '*Critical Section*' area by the process. The memory area which is to be used as the '*Critical Section*' is allocated by the process. The process creates a '*Critical Section*' area by creating a variable of type *CRITICAL_SECTION*. The *Critical Section*' must be initialised before the threads of a process can use it for getting exclusive access. The *InitialiseCriticalSection(LPCRITICAL_SECTION lpCriticalSection)* API initialises the critical section pointed by the pointer *lpCriticalSection* to the critical section. Once the critical section is initialised, all threads in the process can use it. Threads can use the API call *EnterCriticalSection (LPCRITICAL_SECTION lpCriticalSection)* for getting the exclusive ownership of the critical section pointed by the pointer *lpCriticalSection*. Calling the *EnterCriticalSection()* API blocks the execution of the caller thread if the critical section is already in use by other threads and the thread waits for the critical section object. Threads which are blocked by the *EnterCriticalSection()* 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. The API call *TryEnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection)* attempts to enter the critical section pointed by the pointer *lpCriticalSection* without blocking the caller thread. If the critical section is not in use by any other thread, the calling thread gets the ownership of the critical section. If the critical section is already in use by another thread, the

TryEnterCriticalSection() call indicates it to the caller thread by a specific return value and the thread resumes its execution. A thread can release the exclusive ownership of a critical section by calling the API *LeaveCriticalSection(LPCR-ITICAL_SECTION lpCriticalSection)*. The threads of a process can use the API *DeleteCriticalSection (LPCRITICAL_SECTION lpCriticalSection)* to release all resources used by a critical section object which was created by the process with the *CRITICAL_SECTION* variable.

Now let's have a look at the '*Racing*' problem we discussed under the section '*Racing*'. The racing condition can be eliminated by using a critical section object for synchronisation. The following code snippet illustrates the same.

```
##include "stdafx.h"
#include <stdio.h>
#include <windows.h>
//*******************************************************************
//counter is an integer variable and Buffer is a byte array shared
//between two threads
char Buffer[10] = { 1,2,3,4,5,6,7,8,9,10 };
short int counter = 0;
//Define the critical section
CRITICAL_SECTION CS;
//*******************************************************************
// Child Thread 1
void Process_A(void) {
 int i;
 for (i = 0; i<5; i++)
 {
  if (Buffer[i] > 0)
  {
   //Use critical section object for synchronisation
   EnterCriticalSection(&CS);
   counter++;
   LeaveCriticalSection(&CS);
  }
  printf("Process A : Counter = %d\n", counter);
 }
}
//*******************************************************************
// Child Thread 2
void Process_B(void) {
 int j;
 for (j = 5; j<10; j++)
 {
  if (Buffer[j] > 0)
  {
   //Use critical section object for synchronisation
   EnterCriticalSection(&CS);
   counter++;
   LeaveCriticalSection(&CS);
  }
```

```
   printf("Process B : Counter = %d\n", counter);
  }
 }
//***************************************************************
// Main Thread
int main() {
 DWORD id;
 //Initialise critical section object
 InitialiseCriticalSection(&CS);
 CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Process_A, (LPVOID)0, 0, &id);
 CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Process_B, (LPVOID)0, 0, &id);
 Sleep(100000);
 return 0;
}
```

Here the shared resource is the shared variable '*counter*'. The concurrent access to this variable by the threads '*Process_A*' and '*Process_B*' may create race condition and may produce incorrect results. The critical section object '*CS*' holds the piece of code corresponding to the access of the shared variable '*counter*' by each threads. This ensures that the memory area containing the low level instructions corresponding to the high level instruction '*counter++*' is accessed exclusively by threads '*Process_A*' and '*Process_B*' and avoids a race condition. The output of the above piece of code when executed on an Intel Centrino Duo processor running Windows XP OS is given in Fig. 10.36.

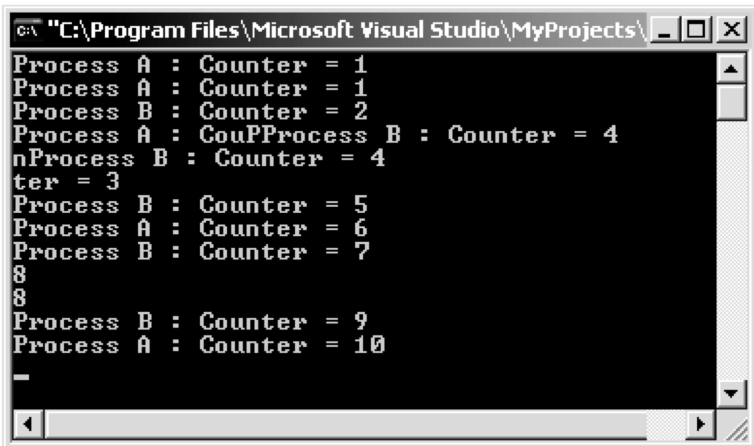


Fig. 10.36 Output of the Win32 application resolving racing condition through critical section object

The final value of '*counter*' is obtained as 10, which is the expected result for this piece of code. If you observe this output window you can see that the text is not outputted to the o/p window in the expected manner. The *printf ()* library routine used in this sample code is re-entrant and it can be preempted while in execution. That is why the outputting of text happened in a non expected way.

> **Note:** It should be noted that the scheduling of the threads '*Process_A*' and '*Process_B*' is OS kernel scheduling policy dependent and you may not get the same output all the time when you run this piece of code under **Windows XP**.

The critical section object makes the piece of code residing inside it non-reentrant. Now let's try the above piece of code by putting the *printf ()* library routine in the critical section object.

```
##include "stdafx.h"
#include <stdio.h>
#include <windows.h>
//*********************************************************************
//counter is an integer variable and Buffer is a byte array shared
//between two threads
char Buffer[10] = { 1,2,3,4,5,6,7,8,9,10 };
short int counter = 0;
//Define the critical section
CRITICAL_SECTION CS;
//*********************************************************************
// Child Thread 1
void Process_A(void) {
 int i;
 for (i = 0; i<5; i++)
 {
  if (Buffer[i] > 0)
  {
   //Use critical section object for synchronisation
   EnterCriticalSection(&CS);
   counter++;
   printf("Process A : Counter = %d\n", counter);
   LeaveCriticalSection(&CS);
  }
 }
}
//*********************************************************************
// Child Thread 2
void Process_B(void) {
 int j;
 for (j = 5; j<10; j++)
 {
  if (Buffer[j] > 0)
  {
   //Use critical section object for synchronisation
   EnterCriticalSection(&CS);
   counter++;
   printf("Process B : Counter = %d\n", counter);
   LeaveCriticalSection(&CS);
  }
 }
```

```
}
//*********************************************************************
// Main Thread
int main() {
 DWORD id;
 //Initialise critical section object
 InitialiseCriticalSection(&CS);
CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Process_A, (LPVOID)0, 0, &id);
CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Process_B, (LPVOID)0, 0, &id);
 Sleep(100000);
 return 0;
}
```

The output of the above piece of code when executed on a Windows 10 machine is given below.

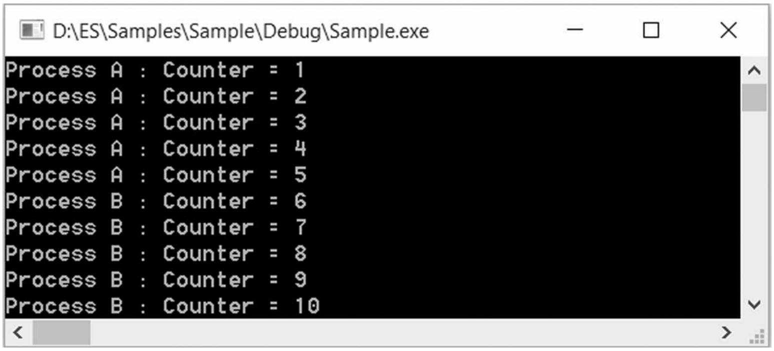


Fig. 10.37 Output of the Win32 application resolving racing condition through critical section object

Note: It should be noted that the scheduling of the threads '*Process_A*' and '*Process_B*' is OS kernel scheduling policy dependent and you may not get the same output all the time when you run this piece of code in Windows 10. The output of the above program when executed at three different instances of time is given shown in Fig. 10.38.

Events Event object is a synchronisation technique which uses the notification mechanism for synchronisation. In concurrent execution we may come across situations which demand the processes to wait for a particular sequence for its operations. A typical example of this is the producer consumer threads, where the consumer thread should wait for the consumer thread to produce the data and producer thread should wait for the consumer thread to consume the data before producing fresh data. If this sequence is not followed it will end up in producer-consumer problem. Notification mechanism is used for handling this scenario. Event objects are used for implementing notification mechanisms. A thread/process can wait for an event and another thread/process can set this event for processing by the waiting thread/process. The creation and handling of event objects for notification is OS kernel dependent. Please refer to the Online Learning Centre for information on the usage of 'Events' under Windows Kernel for process/thread synchronisation.

The MicroC/OS-II kernel also uses 'events' for task synchronisation. We will discuss it in a later chapter.

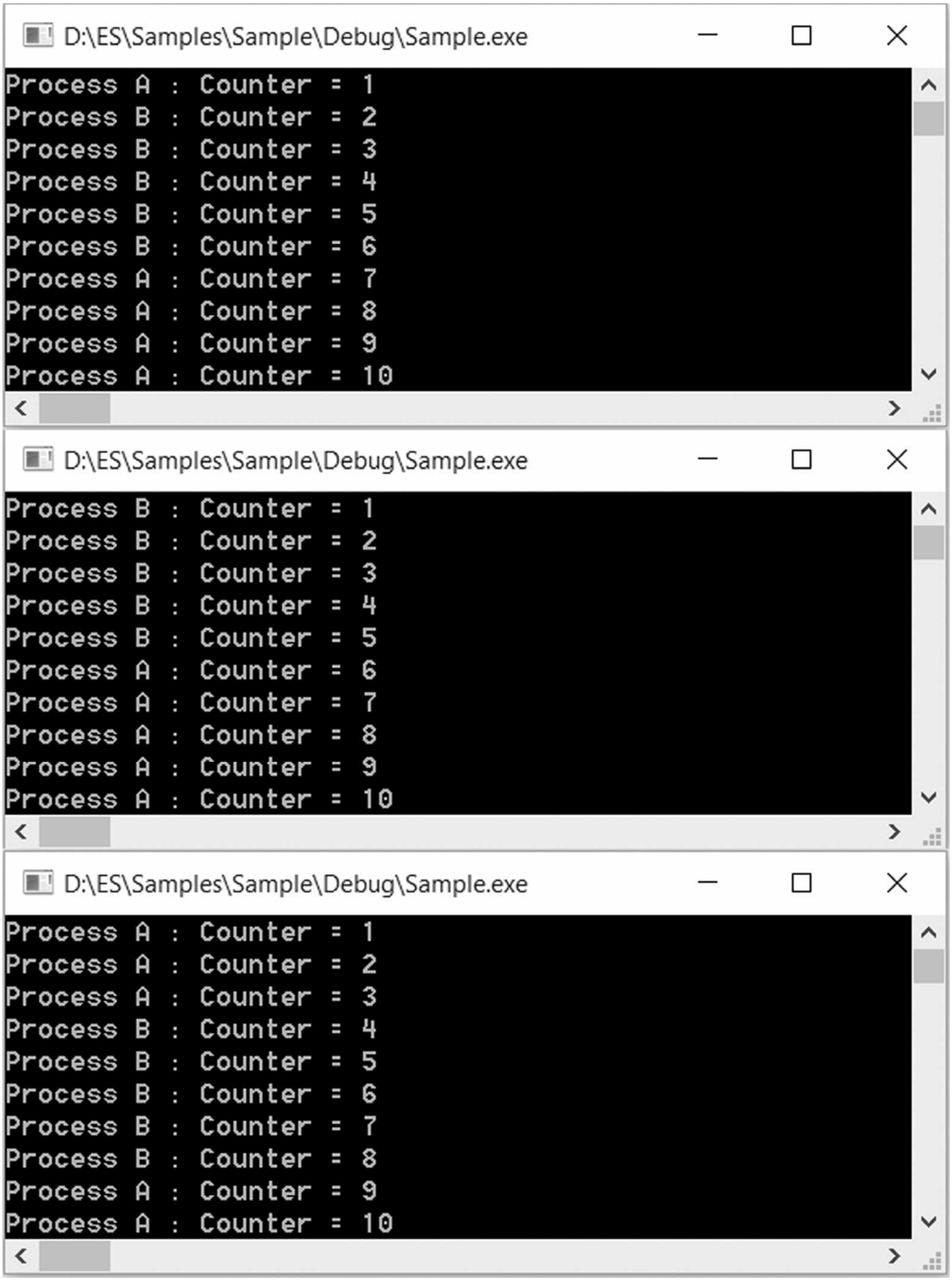


Fig. 10.38 Illustration of scheduler behaviour under Windows NT (E.g. Windows 10) kernel

10.9 DEVICE DRIVERS

LO 9 Analyse device drivers, their role in an operating system based embedded system design, the structure of a device driver, and interrupt handling inside device drivers

Device driver is a piece of software that acts as a bridge between the operating system and the hardware. In an operating system based product architecture, the user applications talk to the Operating System kernel for all necessary information exchange including communication with the hardware peripherals. The architecture of the OS kernel will not allow direct device access from the user application. All the device related access should flow through the OS kernel and the OS kernel routes it to the concerned hardware peripheral. OS provides interfaces in the form of Application Programming Interfaces (APIs) for accessing the hardware. The device driver abstracts the hardware from user applications. The topology of user applications and hardware interaction in an RTOS based system is depicted in Fig. 10.39.

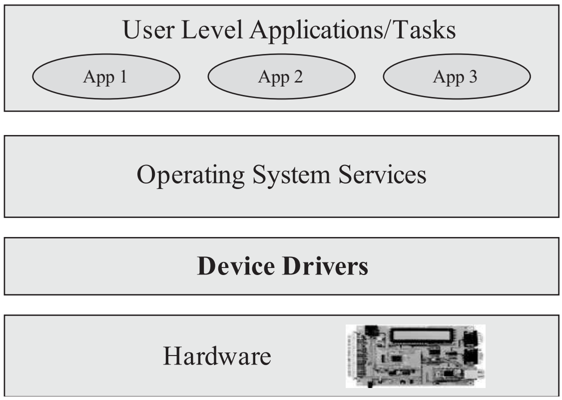


Fig. 10.39 Role of Device driver in Embedded OS based products

Device drivers are responsible for initiating and managing the communication with the hardware peripherals. They are responsible for establishing the connectivity, initialising the hardware (setting up various registers of the hardware device) and transferring data. An embedded product may contain different types of hardware components like Wi-Fi module, File systems, Storage device interface, etc. The initialisation of these devices and the protocols required for communicating with these devices may be different. All these requirements are implemented in drivers and a single driver will not be able to satisfy all these. Hence each hardware (more specifically each class of hardware) requires a unique driver component.

Certain drivers come as part of the OS kernel and certain drivers need to be installed on the fly. For example, the program storage memory for an embedded product, say NAND Flash memory requires a NAND Flash driver to read and write data from/to it. This driver should come as part of the OS kernel image. Certainly the OS will not contain the drivers for all devices and peripherals under the Sun. It contains only the necessary drivers to communicate with the onboard devices (Hardware devices which are part of the platform) and for certain set of devices supporting standard protocols and device class (Say USB Mass storage device or HID devices like Mouse/keyboard). If an external device, whose driver software is not available with OS kernel image, is connected to the embedded device (Say a medical device with custom USB class implementation is connected to the USB port of the embedded product), the OS prompts the user to instal its driver manually. Device drivers which are part of the OS image are known as 'Built-in drivers' or 'On-board drivers'. These drivers are loaded by the OS at the time of booting the device and are always kept in the RAM. Drivers which need to be installed for accessing a device are known as 'Installable drivers'. These drivers are loaded by the OS on a need basis. Whenever the device is connected, the OS loads the corresponding driver to memory. When the device is removed, the driver is unloaded from memory. The Operating system maintains a record of the drivers corresponding to each hardware.

The implementation of driver is OS dependent. There is no universal implementation for a driver. How the driver communicates with the kernel is dependent on the OS structure and implementation. Different Operating Systems follow different implementations.

It is very essential to know the hardware interfacing details like the memory address assigned to the device, the Interrupt used, etc. of on-board peripherals for writing a driver for that peripheral. It varies on the hardware design of the product. Some Real-Time operating systems like 'Windows CE' support a layered architecture for the driver which separates out the low level implementation from the OS specific interface. The low level implementation part is generally known as Platform Dependent Device (PDD) layer. The OS specific interface part is known as Model Device Driver (MDD) or Logical Device Driver (LDD). For a standard driver, for a specific operating system, the MDD/LDD always remains the same and only the PDD part needs to be modified according to the target hardware for a particular class of devices.

Most of the time, the hardware developer provides the implementation for all on board devices for a specific OS along with the platform. The drivers are normally shipped in the form of *Board Support Package*. The *Board Support Package* contains low level driver implementations for the onboard peripherals and OEM Adaptation Layer (OAL) for accessing the various chip level functionalities and a bootloader for loading the operating system. The OAL facilitates communication between the Operating System (OS) and the target device and includes code to handle interrupts, timers, power management, bus abstraction, generic I/O control codes (IOCTLs), etc. The driver files are usually in the form of a dll file. Drivers can run on either user space or kernel space. Drivers which run in user space are known as *user mode drivers* and the drivers which run in kernel space are known as *kernel mode drivers*. User mode drivers are safer than kernel mode drivers. If an error or exception occurs in a user mode driver, it won't affect the services of the kernel. On the other hand, if an exception occurs in the kernel mode driver, it may lead to the kernel crash. The way how a device driver is written and how the interrupts are handled in it are operating system and target hardware specific. However regardless of the OS types, a device driver implements the following:

1. Device (Hardware) Initialisation and Interrupt configuration
2. Interrupt handling and processing
3. Client interfacing (Interfacing with user applications)

The Device (Hardware) initialisation part of the driver deals with configuring the different registers of the device (target hardware). For example configuring the I/O port line of the processor as Input or output line and setting its associated registers for building a General Purpose IO (GPIO) driver. The interrupt configuration part deals with configuring the interrupts that needs to be associated with the hardware. In the case of the GPIO driver, if the intention is to generate an interrupt when the Input line is asserted, we need to configure the interrupt associated with the I/O port by modifying its associated registers. The basic Interrupt configuration involves the following.

1. Set the interrupt type (Edge Triggered (Rising/Falling) or Level Triggered (Low or High)), enable the interrupts and set the interrupt priorities.
2. Bind the Interrupt with an Interrupt Request (IRQ). The processor identifies an interrupt through IRQ. These IRQs are generated by the Interrupt Controller. In order to identify an interrupt the interrupt needs to be bonded to an IRQ.
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

LO 10 Discuss the different functional and non-functional requirements that need to be addressed in the selection of a Real-Time Operating System

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. **LO1**
- ✓ 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. **LO1**

- ✓ 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. **LO1**
- ✓ 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*). **LO2**
- ✓ 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. **LO2**
- ✓ 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* **LO3**
- ✓ 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. **LO3**
- ✓ 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*. **LO3**
- ✓ 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. **LO3**
- ✓ 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*. **LO4**
- ✓ *Co-operative* multitasking, *Preemptive* multitasking and *Non-preemptive* multitasking are the three important types of multitasking which exist in the Operating system context. **LO4**
- ✓ 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. **LO5**
- ✓ Job queue, Ready queue and Device queue are the important queues maintained by an operating system in association with CPU scheduling. **LO5**
- ✓ 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. **LO5**
- ✓ 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. **LO6**
- ✓ *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. **LO7**

- ✓ 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. **LO8**
- ✓ 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. **LO8**
- ✓ *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. **LO8**
- ✓ *Priority inheritance* and *Priority ceiling* are the two mechanisms for avoiding *Priority Inversion* in a multitasking environment. **LO8**
- ✓ 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. **LO8**
- ✓ *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. **LO8**
- ✓ *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. **LO8**
- ✓ *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. **LO9**
- ✓ Various functional and non-functional requirements need to be evaluated before the selection of an RTOS for an embedded design. **LO10**

Keywords

Operating System: A piece of software designed to manage and allocate system resources and execute other pieces of the software **[LO 1]**

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 **[LO 1]**

Kernel space: The primary memory area where the kernel applications are confined to run **[LO 1]**

User space: The primary memory area where the user applications are confined to run **[LO 1]**

Monolithic kernel: A kernel with all kernel services run in the kernel space under a single kernel thread **[LO 1]**

Microkernel: A kernel which incorporates only the essential services within the kernel space and the rest is installed as loadable modules called *servers* **[LO 1]**

Real-Time Operating System (RTOS): Operating system with a specialised kernel with a deterministic timing behaviour **[LO 2]**

Scheduler: OS kernel service which deals with the scheduling of processes/tasks **[LO 2]**

Hard Real-Time: Real-time operating systems that strictly adhere to the timing constraints for a task **[LO 2]**

Soft Real-Time: Real-time operating systems that does not guarantee meeting deadlines, but, offer the best effort to meet the deadline **[LO 2]**

Task/Job/Process: In the operating system context a task/process is a program, or part of it, in execution **[LO 3]**

Process Life Cycle: The different *states* through which a process traverses through during its journey from the newly created state to completed state **[LO 3]**

Thread: The primitive that can execute code. It is a single sequential flow of control within a process **[LO 3]**

Multiprocessing systems: Systems which contain multiple CPUs and are capable of executing multiple processes simultaneously **[LO 4]**

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 **[LO 4]**

Context switching: The act of switching CPU among the processes and changing the current execution context **[LO 4]**

Co-operative multitasking: Multitasking model in which a task/process gets a chance when the currently executing task relinquishes the CPU voluntarily **[LO 4]**

Preemptive multitasking: Multitasking model in which a currently running task/process is preempted to execute another task/process **[LO 4]**

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 **[LO 4]**

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 **[LO 5]**

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 **[LO 5]**

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 **[LO 5]**

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 **[LO 5]**

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 **[LO 5]**

Round Robin: Preemptive scheduling policy in which the currently running process is preempted based on time slice **[LO 5]**

Co-operating processes: Processes which share data for communicating among them **[LO 7]**

Inter Process/Task Communication (IPC): Mechanism for communicating between co-operating processes of a system **[LO 7]**

Shared memory: A memory sharing mechanism used for inter process communication **[LO 7]**

Message passing: IPC mechanism based on exchanging of messages between processes through a message queue or mailbox **[LO 7]**

Message queue: A queue for holding messages for exchanging between processes of a multitasking system **[LO 7]**

Mailbox: A special implementation of message queue under certain OS kernel, which supports only a single message **[LO 7]**

Signal: A form of asynchronous message notification **[LO 7]**

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 **[LO 7]**

Racing: The situation in which multiple processes compete (race) each other to access and manipulate shared data concurrently **[LO 8]**

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 **[LO 8]**

Livelock: A condition where a process always does something but is unable to make any progress in the execution completion **[LO 8]**

Starvation: The condition in which a process does not get the CPU or system resources required to continue its execution for a long time **[LO 8]**

Dining Philosophers' Problem: A real-life representation of the *deadlock*, *starvation*, *livelock* and *racing* issues in shared resource access in operating system context **[LO 8]**

Producer-Consumer problem: A common data sharing problem where two processes concurrently access a shared buffer with fixed size **[LO 8]**

Readers-Writers problem: A data sharing problem characterised by multiple processes trying to read and write shared data concurrently **[LO 8]**

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 **[LO 8]**

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 **[LO 8]**

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 **[LO 8]**

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 **[LO 8]**

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 **[LO 8]**

Semaphore: A system resource for implementing mutual exclusion in shared resource access or for restricting the access to the shared resource **[LO 8]**

Mutex: The *binary semaphore* implementation for exclusive resource access under certain OS kernel **[LO 8]**

Device driver: A piece of software that acts as a bridge between the operating system and the hardware **[LO 9]**

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 (c) Services (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? **[LO 1]**
2. What is kernel? What are the different functions handled by a general purpose kernel? **[LO 1]**
3. What is kernel space and user space? How is kernel space and user space interfaced? **[LO 1]**
4. What is monolithic and microkernel? Which one is widely used in real-time operating systems? **[LO 1]**
5. What is the difference between a General Purpose kernel and a Real-Time kernel? Give an example for both. **[LO 2]**

Real-Time Operating System (RTOS)

1. Explain the basic functions of a real-time kernel? **[LO 2]**
2. What is task control block (TCB)? Explain the structure of TCB. **[LO 3]**
3. Explain the difference between the memory management of general purpose kernel and real-time kernel. **[LO 2]**
4. What is virtual memory? What are the advantages and disadvantages of virtual memory? **[LO 2]**
5. Explain how 'accurate time management' is achieved in real-time kernel **[LO 2]**
6. What is the difference between 'Hard' and 'Soft' real-time systems? Give an example for 'Hard' and 'Soft' Real-Time kernels. **[LO 2]**

Tasks, Process and Threads

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

Multiprocessing and Multitasking

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

Task Scheduling

1. What is *task scheduling* in the operating system context? **[LO 5]**
2. Explain the various factors to be considered for the selection of a scheduling criteria. **[LO 5]**
3. Explain the different *queues* associated with process scheduling. **[LO 5]**
4. Explain the different types of *non-preemptive* scheduling algorithms. State the merits and de-merits of each. **[LO 5]**
5. Explain the different types of *preemptive* scheduling algorithms. State the merits and de-merits of each. **[LO 5]**
6. Explain *Round Robin (RR)* process scheduling with interrupts. **[LO 5]**

7. Explain *starvation* in the process scheduling context. Explain how starvation can be effectively tackled? **[LO 5]**
8. What is *IDLEPROCESS*? What is the significance of IDLEPROCESS in the process scheduling context? **[LO 5]**
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. **[LO 5]**
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. **[LO 4]**
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. **[LO 5]**
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. **[LO 5]**
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. **[LO 5]**
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. **[LO 5]**
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. **[LO 5]**
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. **[LO 5]**

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. **[LO 5]**

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. **[LO 5]**

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. **[LO 5]**

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. **[LO 5]**

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. **[LO 5]**

Task Communication and Synchronisation

1. Explain the various process interaction models in detail. **[LO 6]**
2. What is Inter Process Communication (IPC)? Give an overview of different IPC mechanisms adopted by various operating systems. **[LO 6, LO 7]**
3. Explain how multiple processes in a system co-operate. **[LO 6]**
4. Explain how multiple threads of a process co-operate. **[LO 6]**
5. Explain the shared memory based IPC. **[LO 7]**
6. Explain the concept of memory mapped objects for IPC. **[LO 7]**
7. Explain the handle sharing and name sharing based memory mapped object technique for IPC under Windows Operating System. **[LO 7]**
8. Explain the message passing technique for IPC. What are the merits and de-merits of message based IPC? **[LO 7]**
9. Explain the synchronous and asynchronous messaging mechanisms for IPC under Windows kernel. **[LO 7]**
10. Explain *Race condition* in detail, in relation to the shared resource access. **[LO 8]**
11. What is *deadlock*? What are the different conditions favouring deadlock? **[LO 8]**

12. Explain by *Coffman conditions*? **[LO 8]**
13. Explain the different methods of handling deadlocks. **[LO 8]**
14. Explain *livelock* in the resource sharing context. **[LO 8]**
15. Explain *starvation* in the resource sharing context. **[LO 8]**
16. Explain the *Dining Philosophers* problem in the process synchronisation context. **[LO 8]**
17. Explain the *Produces-consumer* problem in the inter process communication context. **[LO 8]**
18. Explain *bounded-buffer* problem in the interprocess communication context. **[LO 8]**
19. Explain *buffer overrun* and *buffer under-run*. **[LO 8]**
20. What is *priority inversion*? What are the different techniques adopted for handling priority inversion? **[LO 8]**
21. What are the merits and de-merits of *priority ceiling*? **[LO 8]**
22. Explain the different task communication synchronisation issues encountered in Interprocess Communication. **[LO 8]**
23. What is *task (process) synchronisation*? What is the role of process synchronisation in IPC? **[LO 8]**
24. What is *mutual exclusion* in the process synchronisation context? Explain the different mechanisms for mutual exclusion. **[LO 8]**
25. What are the merits and de-merits of *busy-waiting* (*spinlock*) based mutual exclusion? **[LO 8]**
26. Explain the *Test and Set Lock (TSL)* based mutual exclusion technique. Explain how *TSL* is implemented in Intel family of processors. **[LO 8]**
27. Explain the *interlocked functions* for lock based mutual exclusion under Windows OS. **[LO 8]**
28. Explain the advantages and limitations of *interlocked function* based synchronisation under Windows. **[LO 8]**
29. Explain the *sleep & wakeup* mechanism for mutual exclusion. **[LO 8]**
30. What are the merits and de-merits of *sleep & wakeup* mechanism based mutual exclusion? **[LO 8]**
31. What is *mutex*? **[LO 8]**
32. Explain the *mutex* based process synchronisation under Windows OS. **[LO 8]**
33. What is *semaphore*? Explain the different types of semaphores. Where is it used? **[LO 8]**
34. What is binary *semaphore*? Where is it used? **[LO 8]**
35. What is the difference between *mutex* and *semaphore*? **[LO 8]**
36. What is the difference between *semaphore* and *binary semaphore*? **[LO 8]**
37. What is the difference between *mutex* and *binary semaphore*? **[LO 8]**
38. Explain the *semaphore* based process synchronisation under Windows OS. **[LO 8]**
39. Explain the *critical section problem*? **[LO 8]**
40. What is *critical section*? What are the different techniques for controlling access to *critical section*?**[LO8]**
41. Explain the *critical section object* for process synchronisation. Why is critical section object based synchronisation fast? **[LO 8]**
42. Explain the *critical section object* based process synchronisation under Windows OS. **[LO 8]**
43. Explain the *Event* based synchronisation mechanism for IPC. **[LO 8]**

44. Explain the *Event object* based synchronisation mechanism for IPC under Windows OS. **[LO 8]**

45. What is a *device driver*? Explain its role in the OS context. **[LO 8]**

46. Explain the architecture of device drivers. **[LO 9]**

47. Explain the different functional and non-functional requirements that needs to be evaluated in the selection of an RTOS. **[LO 10]**

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 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 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 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 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.