# Module 8

# Industrial Embedded and Communication Systems

# Lesson 37

# Real-Time Operating Systems: Introduction and Process Management

## Instructional Objectives

After learning the lesson the student should be able to:

A. Describe the major functions of an Operating System
B. Define multi-tasking and describe its advantages
C. Describe the task states and transitions in the execution life cycle under a multi-tasking OS
D. Define the concept of preemptive priority scheduling
E. Describe common multi-tasking architectures of RTOS.
F. Describe the classification of computing tasks in terms of their timing constraints

## Introduction

Embedded Computing Applications exist in a spectacular range of size and complexity for areas such as home automation to cell phones, automobiles and industrial controllers. Most of these applications demand such functionality, performance and reliability from the software that simple and direct assembly language programming of the processors is clearly ruled out. Moreover, as a distinguishing feature from general purpose computing, a large part of the computation is "real-time" or time constrained and also reactive or "external event driven" since such systems generally interface strongly with the external environment through a variety of devices. Thus, an operating system is generally used. An operating system facilitates development of application program by making available a number of services, which, otherwise would have to be coded by the application program. The application programs "interface" with the hardware through a operating system services and functions. It is therefore important understand the basic features of such operating systems. In this chapter we present the fundamental concepts of a real-time operating system in a generic context. Most large industrial controllers employ such operating systems. Before we undertake the discussion, we briefly review the nature of computation in industrial automation systems.

## Nature of IA computation

Computation for industrial automation are typified by the following characteristics:

- The computation is reactive in nature, that is, the computation is carried out with respect to real-world physical signals and events. Therefore industrial computers must have extensive input-output subsystems that interface it with physical signals
- In general speed of computation is not very high. This is because dynamics for variables such as temperature are very slow. For electromechanical systems speed requirement is higher. However, there processing is often done by dedicated coprocessors.
- The algorithmic complexity of computation is generally not very high, although it is increasing due to the deployment of sophisticated control and signal processing algorithms. Therefore general-purpose processors are sufficient, almost in all cases.

- A large part of the computational tasks are repetitive or cyclic, that is their executions are invoked periodically in time. Many of these tasks are also real-time, in the sense that there are deadlines associated with their execution. When there are a large number of such tasks to be executed on a single processor, appropriate scheduling strategies are required. Since the number of tasks is mostly fixed, simple static scheduling policies are generally adequate.
- Although computational demands are not very high, the computation is critical in nature. Indeed, the cost ratios of industrial controllers and the plant equipment they control can well be in excess of one thousand. This requirement is coupled with the fact that such controllers are often destined to work in harsher industrial environments. Thus, reliability requirement of industrial control computers are very high. In fact, this is one of the main reasons that decide the high price of such systems compared to commercial computers of comparable or even higher data processing capability.

In this lesson the basic features of a Real-Time Operating System (RTOS) are introduced and motivated from the demands of typical real-time applications. Before the RTOS is discussed the typical functionalities of a general Operating System (OS) reviewed briefly. While the RTOS has some distinguishing features, it is also largely similar to general purpose operating systems. The discussion here is necessarily brief and user oriented, rather than designer oriented. For a detailed exposition on OS any standard textbook on the subject may be referred to.

## Operating Systems (OS) Basics

An Operating System is a collection of programs that provides an interface between application programs and the computer system (hardware). Its primary function is to provide application programmers with an abstraction of the system resources, such as memory, input-output and processor, which enhances the convenience, efficiency and correctness of their use. These programs or functions within the OS provide various kinds of services to the application programs. The application programs, in turn, call these programs to avail of such services. Thus the application programs can view the computer resources as abstract entities, (for example a block of memory can be used as a named sequential file with the abstract Open, Close, Read, Write operations) without need for knowing the low level hardware details (such as the addresses of the memory blocks). To get a better idea of such services provided by the OS, the user may like to refer to the DOS services for the IBM PC Compatibles.

The natural way to view computation in a typical modern computer system is in the form of a number of different programs, all of which, apparently, run in parallel. However, very often, all the programs in the system are executed on a single physical CPU or processor. Each program runs for a small continuous duration at a time, before it is stopped and another program begins to execute. If this is done rapidly enough, it appears as if all programs are running simultaneously. These programs very often perform independent computations, such as the programs executing in different windows on a PC. In real time systems, most often, such programs cooperate with each other, by exchanging data and synchronizing each other’s execution, to achieve the overall functionality and performance of the system.

## Types of Operating Systems

- Stand-Alone Operating systems
- Network Operating systems
- Embedded Operating Systems

## Stand-Alone Operating System

It is a complete operating system that works on a desktop or notebook computer. Examples of stand-alone operating systems are:

- *DOS*
- *Windows 2000 Professional*
- *Mac OS X*

## Network Operating System

It is an operating system that provides extensive support for computer networks. A network operating system typically resides on a server. Examples of a network operating system are:

- *Windows 2000 Server*
- *Unix*
- *Linux*
- *Solaris*

## Embedded Operating System

You can find this operating system on handheld computers and small devices, it resides on a ROM chip. Examples of embedded operating systems are:

- *Windows CE*
- *Pocket PC 2002*
- *Palm OS*

The above classification is based on the computing hardware environment towards which the OS is targetted. All three types can be either of a real-time or a non-real-time type. For example, VxWorks is an RTOS of the first category, RT-Linux and ARTS are of the second category and Windows CE of the third category.

## Point to Ponder 1

*A. What are the factors on which the execution time of a program depends on?*

*B. While a task is executing, is the CPU continuously busy?*

*C. Suppose there are two mutually independent programs A and B. If only program A (B) executes on the processor P, it takes $t_A$ ( $t_B$) units of time. If these programs are fired simultaneously and run under multi-programming, neglecting time related to program switching, would the execution time for both programs taken together be greater than, equal to or less than ($t_A + t_B$ )?*

## Real-Time Operating Systems

A Real-Time OS (RTOS) is an OS with special features that make it suitable for building real-time computing applications also referred to as Real-Time Systems (RTS). An RTS is a (computing) system where correctness of computing depends not only on the correctness of the logical result of the computation, but also on the result delivery time. An RTS is expected to respond in a timely, predictable way to unpredictable external stimuli.

Real-time systems can be categorized as Hard or Soft. For a Hard RTS, the system is taken to have failed if a computing deadline is not met. In a Soft RTS, a limited extent of failure in meeting deadlines results in degraded performance of the system and not catastrophic failure. The correctness and performance of an RTS is therefore not measured in terms of parameters such as, average number of transactions per second as in transactional systems such as databases.

A *Good* RTOS is one that enables bounded (predictable) behavior under all system load scenarios. Note however, that the RTOS, by itself cannot guarantee system correctness, but only is an enabling technology. That is, it provides the application programmer with facilities using which a correct application can be built. Speed, although important for meeting the overall requirements, does not by itself meet the requirements for an RTOS.

### Programs, Processes, Tasks and Threads

The above four terms are often found in literature on OS in similar contexts. All of them refer to a unit of computation. A ***program*** is a general term for a unit of computation and is typically used in the context of programming. A ***process*** refers to a program in execution. A process is an independently executable unit handled by an operating system. Sometimes, to ensure better utilisation of computational resources, a process is further broken up into ***threads***. Threads are sometimes referred to as lightweight processes because many threads can be run in parallel, that is, one at a time, for each process, without incurring significant additional overheads. A ***task*** is a generic term, which, refers to an independently schedulable unit of computation, and is used typically in the context of scheduling of computation on the processor. It may refer either to a process or a thread.

## Multitasking

A multitasking environment allows applications to be constructed as a set of independent tasks, each with a separate thread of execution and its own set of system resources. The inter-task communication facilities allow these tasks to synchronize and coordinate their activity. *Multitasking* provides the fundamental mechanism for an application to control and react to multiple, discrete real-world events and is therefore essential for many real-time applications. Multitasking creates the appearance of many threads of execution running concurrently when, in fact, the kernel interleaves their execution on the basis of a scheduling algorithm. This also leads to efficient utilisation of the CPU time and is essential for many embedded applications where processors are limited in computing speed due to cost, power, silicon area and other constraints. In a multi-tasking operating system it is assumed that the various tasks are to cooperate to serve the requirements of the overall system. Co-operation will require that the tasks communicate with each other and share common data in an orderly an disciplined manner, without creating undue contention and deadlocks. The way in which tasks communicate and share data is to be regulated such that communication or shared data access error is prevented and data, which is private to a task, is protected. Further, tasks may be dynamically created and terminated by other tasks, as and when needed. To realise such a system, the following major functions are to be carried out.

A. Process Management
   - interrupt handling
   - task scheduling and dispatch
     - create/delete, suspend/resume task
     - manage scheduling information
       - priority, scheduling policy, etc

B. Interprocess Communication and Synchronization
   - Code, data and device sharing
   - Synchronization, coordination and data exchange mechanisms
   - Deadlock and Livelock detection

C. Memory Management
   - dynamic memory allocation
   - memory locking
   - Services for file creation, deletion, reposition and protection

D. Input/Output Management
   - Handles request and release functions and read, write functions for a variety of peripherals

The following are important requirements that an OS must meet to be considered an RTOS in the contemporary sense.

A. The operating system must be multithreaded and preemptive. e.g. handle multiple threads and be able to preempt tasks if necessary.
B. The OS must support priority of tasks and threads.
C. A system of priority inheritance must exist. Priority inheritance is a mechanism to ensure that lower priority tasks cannot obstruct the execution of higher priority tasks.
D. The OS must support various types of thread/task synchronization mechanisms.
E. For predictable response :
   a. The time for every system function call to execute should be predictable and independent of the number of objects in the system.
   b. Non preemptable portions of kernel functions necessary for interprocess synchronization and communication are highly optimized, short and deterministic

c. Non-preemptable portions of the interrupt handler routines are kept small and deterministic
d. Interrupt handlers are scheduled and executed at appropriate priority
e. The maximum time during which interrupts are masked by the OS and by device drivers must be known.
f. The maximum time that device drivers use to process an interrupt, and specific IRQ information relating to those device drivers, must be known.
g. The interrupt latency (the time from interrupt to task run) must be predictable and compatible with application requirements

F. For fast response:
a. Run-time overhead is decreased by reducing the unnecessary context switch.
b. Important timings such as context switch time, interrupt latency, semaphore get/release latency must be minimum

## Point to Ponder: 2

*A. What are possible reasons for which execution time for a  can be unpredictable?*

*B. Why preemptive multi-tasking is such an important requirement for an RTOS?*

*C. Suppose there are two mutually independent tasks A and B. Let B be the higher priority task. If only task A (B) executes on the processor P, it takes $t_A$ ( $t_B$) units of time. If these tasks are run under multi-tasking, neglecting time related to task switching, can the execution time for task A be less than ($t_A + t_B$ ) ?*

## Process Management

On a computer system with only one processor, only one task can run at any given time, hence the other tasks must be in some state other than running. The number of other states, the names given to those states and the transition paths between the different states vary with the operating system. A typical state diagram is given in Figure 4 and the various states are described below.

## Task States

- ♦ Running: This is the task which has control of the CPU. It will normally be the task which has the highest current priority of the tasks which are ready to run.
- ♦ Ready: There may be several tasks in this state. The attributes of the task and the resources required to run it must be available for it to be placed in the 'ready' state.
- ♦ Waiting: The execution of tasks placed in this state has been suspended because the task requires some resources which is not available or because the task is waiting for some signal from the plant, e.g., input from the analog-to-digital converter, or the task is waiting for the elapse of time.
- ♦ New: The operating system is aware of the existence of this task, but the task has not been allocated a priority and a context and has not been included into the list of schedulable tasks.

- Terminated: The operating system has not as yet been made aware of the existence of this task, although it may be resident in the memory of the computer.

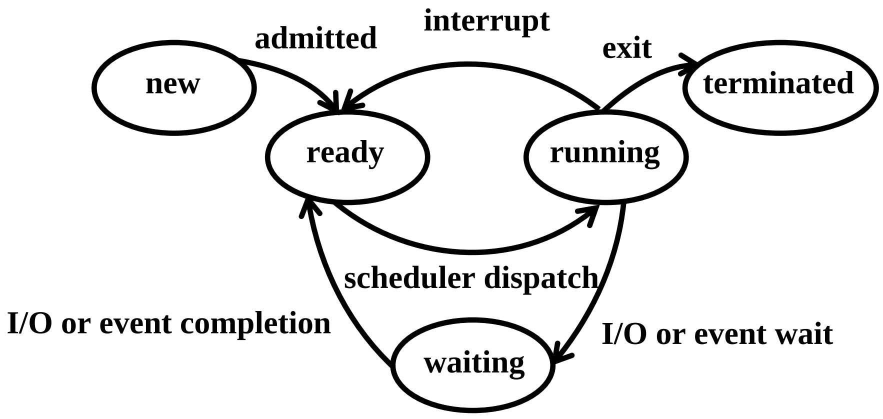

**Fig. 37.1 The various states a task can be in during its execution life cycle under an RTOS Task State Transitions**

When a task is “spawned”, either by the operating system, or another task, it is to be created, which involves loading it into the memory, creating and updating certain OS data structures such as the Task Control Block, necessary for running the task within the multi-tasking environment. During such times the task is in the new state. Once these are over, it enters the ready state where it waits. At this time it is within the view of the scheduler and is considered for execution according to the scheduling policy. A task is made to enter the running state from the ready state by the operating system dispatcher when the scheduler determines the task to be the one to be run according to its scheduling policy. While the task is running, it may execute a normal or abnormal exit according to the program logic, in which case it enters the terminated state and then removed from the view of the OS. Software or hardware interrupts may also occur while the task is running. In such a case, depending on the priority of the interrupt, the current task may be transferred to the ready state and wait for its next time allocation by the scheduler. Finally, a task may need to wait at times during its course of execution, either due to requirements of synchronization with other tasks or for completion of some service such as I/O that it has requested for. During such a time it is in the waiting state. Once the synchronization requirement is fulfilled, or the requested service is completed, it is returned to the ready state to again wait its turn to be scheduled.

## Point to Ponder: 3

A. *What are the situations under which a running task can go to the ready state?*

B. *What are the situations under which a running task can go to the waiting state?*

C. *Suppose there are two mutually independent tasks A and B, which are invoked periodically with periods with B as the higher priority task. If only task A (B) executes on the processor P, it takes $t_A$ ( $t_B$) units of time. The periods for A and B are $T_A$ and $T_B$, respectively, and $t_A \leq T_A$ and $t_B \leq T_B$. If these tasks are to be run under multi-tasking with negligible task switching time, state any one condition under which task preemption would be essential for the task dead lines to be met.*

D. *In the above case state a condition under which even with preemption, task deadlines cannot be met.*

## Task Control Functions

RTOSs provide functions to spawn, initialise and activate new tasks. They provide functions to gather information on existing tasks in the system, for task naming, checking of the state of a given task, setting options for task execution such as use of co-processor, specific memory models, as well as task deletion. Deletion often requires special precautions, especially with respect to semaphores, for shared memory tasks.

## Task Context

Whenever a task is switched its execution context, represented by the contents of the program counter, stack and registers, is saved by the operating system into a special data structure called a task control block so that the task can be resumed the next time it is scheduled. Similarly the context has to be restored from the task control block when the task state is set to running. The information related to a task stored in the TCB is shown below.

## Task Control Block

- Task ID: the unique identifier for a task
- Address Space : the address ranges of the data and code blocks of the task loaded in memory including statically and dynamically allocated blocks
- Task Context: includes the task's program counter(PC) , the CPU registers and (optionally) floating-point registers, a stack for dynamic variables and function calls, the stack pointer (SP), I/O device assignments, a delay timer, a time-slice timer and kernel control structures
- Task Parameters : includes task type, event list
- Scheduling Information : priority level, relative deadline, period, state
- Synchronization Information : semaphores, pipes, mailboxes, message queues, file handles etc.
- Parent and Child Tasks

## Task Scheduling and Dispatch

The basic purpose of task scheduling and dispatch in a real-time multi-tasking OS is to ensure that each task gets access to the CPU and other system resources in a manner that is necessary for successful and timely completion of all computation in the system. Secondly, it is desired that this is done efficiently from the point of view of resource utilisation as well as with correct synchronisation and protection of data and code for individual tasks against incorrect interference. Various task scheduling and dispatch models are in use to achieve the above. The appropriateness of a particular model depends on the application features. The major main task scheduling and dispatch models are described below.

## Cyclic Executive

This is the simplest of the models in which all the computing tasks are required to be run periodically in cycles. The computing sequence is static and therefore, a monolithic program called the Cyclic Executive runs the tasks in the required order in a loop. At times the execution sequence may also be determined in accordance with models such as an FSM. The execution sequences and the times allocated to each task are determined a priori and are not expected to vary significantly at run time. Such systems are often employed for controllers of industrial machines such as Programmable Logic Controllers that perform fixed set of tasks in fixed orders defined by the user. These systems have the advantage of being simple to develop and configure as well as faster than some of the more complex systems since task context switching is faster and less frequent. They are however suitable for static computing environments only and are custom developed using low level programming languages for specific hardware platforms.

## Coroutines

In this model of cooperative multitasking the set of tasks are distributed over a set of processes, called coroutines. These tasks mutually exchange program control rather than relinquish it to the operating system. Thus each task transfers control to one of the others after saving its data and control state. Note that the responsibility of scheduling, that is deciding which task gets control of the processor at a given time is left to the programmer, rather than the operating system. The task which is transferring control is often left in the waiting or blocked state. This model has now been adapted to a different form in Multithreading.

## Interrupts

In many cases task scheduling and dispatch needs to be made responsive to external signals or timing signals. In other cases running tasks may not be assumed to be transferring control to the dispatcher on their own. In such cases the facility of interrupts provided on all processors can be used for task switching. The various tasks in the system can be switched either by hardware or software interrupts. The interrupt handling routine would then transfer control to the task dispatcher. Interrupts through hardware may occur periodically, such as from a clock, or asynchronously by external devices. Interrupts can also occur by execution of special software instructions written in the code, or due to processing exceptions such as divide by zero errors. Interrupt-only systems are special case of foreground/background systems, described below, which are widely used in embedded systems.

## Foreground / Background

Typically, embedded and real-time applications involve a set of tasks, some of which are periodic and must be finished within deadlines. Others may be sporadic and may not have such deadlines associated with them. Foreground/background systems are common and simple solutions for such applications. Such systems involve a set of interrupt driven or real-time tasks called the *foreground* and a collection of non-interrupt driven tasks called the *background.* The foreground tasks run according to some real-time priority scheduling policy. The background tasks are preemptable by any foreground task.

## Real Time Operating Systems

This is the most complex model for real-time multi-tasking. The major features that distinguish it from the other ones described above are the following.

1. The explicit implementation of a scheduling policy in the form of a scheduler module. The scheduler is itself a task which executes every time an internal or external interrupt occurs and computes the decision on making state transitions for every application task in the system that has been spawned and has not yet been terminated. It computes this decision based on the current priority level of the tasks, the availability of the various resources of the system etc. The scheduler also computes the current priority levels of the tasks based on various factors such as deadlines, computational dependencies, waiting times etc.

2. Based on the decisions of the scheduler, the dispatcher actually effects the state transition of the tasks by
   a. saving the computational state or context of the currently executing task from the hardware environment.
   b. enabling the next task to run by loading the process context into the hardware environment.

It is also the responsibility of the dispatcher to make the short-term decisions in response to, e.g., interrupts from an input/output device or from the real-time clock.

The dispatcher/scheduler has two entry conditions:

1. The real-time clock interrupt and any interrupt which signals the completion of an input/output request
2. A task suspension due to a task delaying, completing or requesting an input/output transfer.

In response to the first condition the scheduler searches for work starting with the highest priority task and checking each task in priority order. Thus if tasks with a high repetition rate are given a high priority they will be treated as if they were clock-level tasks, i.e., they will be run first during each system clock period. In response to the second condition a search for work is started at the task with the next lowest priority to the task which has just been running. There cannot be another higher priority task ready to run since a higher priority task becoming ready always preempts a lower priority running task.

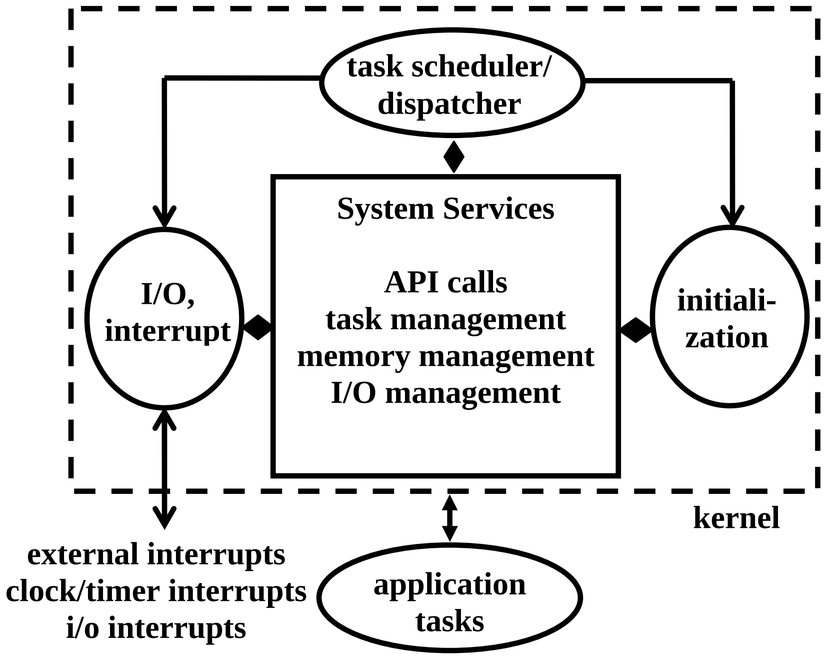

**Fig. 37.2 Structure of an RTOS Kernel**

The typical structure of an RTOS kernel showing the interaction between the System and the Application tasks.

## Point to Ponder: 4

A. *In what ways threads are similar to coroutines? In what ways are they different?*

B. *Under what situations is a cyclic executive adequate? When is it not and why not?*

C. *In what ways are interrupt service routines different from other tasks?*

D. *Can you give an example of a practical embedded application, for which an Foreground/ Background model of multi-tasking is adequate?*

E. *Under what situations is an full-featured RTOS an appropriate choice? Can you illustrate your answer with the example of a practical embedded application?*

## Priority Levels in a typical Real-Time Operating System

To be able to ensure that response to every event is generated by executing tasks within specific deadlines, it is essential to allocate the CPU and other computational resources to various tasks in accordance with their priorities. The priority of a process may be fixed or static. It may be calculated based on their computing time requirements, frequency of invocations or deadlines. However, it has been established that policies that allow task priorities to be adjusted dynamically are more efficient. The priority will depend on how quickly a task will have to respond to a particular event. An event may be some activity of the process or may be the elapsing of a specified amount of time. The set of tasks can often be categorised into three broad levels of priority as shown below. Tasks belonging to the same category are typically scheduled and dispatched using similar mechanisms.

***Interrupt Level*:** Tasks at this level require very fast response measured in milliseconds and occur very frequently. There is no scheduler at this level, since immediate execution follows an interrupt. Examples of this task include the real-time clock task. Obviously, to meet the deadlines of the other tasks in the system, the context switching and processing time requirements for these tasks are to be kept at the bare minimum level and must be highly predictable to make the whole system behaviour predictable. To ensure the former, often, all Interrupt Service Routines (ISRs) run in special common and fixed contexts, such as common stacks. To ensure the latter the interrupt service routines are sometimes divided into two parts. The first part executes immediately, spawns a new task for the remaining processing and returns to kernel. This new task gets executed as per the scheduling policy in course of time. The price to pay for fast execution of ISRs is often several constraints on the programming of ISRs which lacks many flexibilities compared to the programming of tasks. Priorities among tasks at this level are generally ensured through the hardware and software interrupt priority management system of the processor. There may also exist interrupt controllers that masks interrupts of lower priority in the presence of a higher priority one. The system clock and watchdog timers associated with them are tasks that execute at interrupt level. The dispatcher for the next level of priority is also a task at this level. The frequency of execution of this task depends on the frequency or period of the highest priority clock-level task.

***Hard Real-Time Level*:** At this level are the tasks which are periodic, such as the sampling and control tasks, and tasks which require accurate timing. The scheduling of these tasks is carried out based on the real-time system clock. A system clock device consists of a counter, a timer queue and an interrupt handler. Content of counter gives the current time, timer queue has pending timers associated with the clock device. Each system clock interrupt is known as a tick and represents the smallest time interval known to the system. Since most programs in real-time applications make use of time, virtual software clocks and delays can also be created by tasks and associated with the system clock device. The system clock device raises interrupts periodically and the kernel updates the software clock according to current time. Also every few clock cycles a new task gets dispatched according to the scheduling policy adopted. The lowest priority task at this level is the base level scheduler. Thus if at a clock level interrupt, the clock level scheduler finds no request for higher priority clock level tasks pending, the base level scheduler is dispatched.

***Soft/Non-Real Time Level*:** Tasks at this level are of soft or non-real-time in that they either have no deadlines to meet or are allowed a wide margin of error in their timing. These are therefore taken to be of low priority and executed only when no request for a higher priority task is pending. Tasks at this level may be allocated priorities or may all run at a single priority level - that of the base level scheduler in a round robin fashion. These tasks are typically initiated on demand rather that at some predetermined time interval. The demand may be a user input from a keypad, some process event such as the arrival of a packet or some particular requirement of the data being processed. Note that, since the base level scheduler is the lowest priority clock level task, the priorities of all base level tasks are lower than those at clock levels.

Each of the above priority levels can support multiple task priorities. For an RTOS to be compliant with the RT-POSIX standard, number of priority levels supported must be at least 32. Among commercial RTOS, the priority levels supported can be as low as 8 for Windows CE or 256 for VxWorks. For scheduling of equal priority threads, FIFO or Round-Robin policy is generally applied. Thread priorities be changed at run-time.

## Task Scheduling Management

Advanced multi-tasking RTOSs mostly use preemptive priority scheduling. These support more than one scheduling policy and often allow the user to set parameters associated with such policies, such as the time-slice in Round Robin scheduling where each task in the task queue is scheduled for up to a maximum time, set by the time-slice parameter, in a Round Robin manner. Task priorities can also be set. Hundred of priority levels are commonly available for scheduling. Specific tasks can also be indicated to be nonpremeptive.

### Point to Ponder: 5

A. *Give example of hard, soft and non-real-time computing tasks. In what ways are they different?*

B. *Can interrupt level priorities be seen as very high priority levels?*

C. *What happens if the user sets priorities in a manner that deadline would be violated for other tasks?*

## Lesson Summary

In this lesson, we have dealt with the following topics.

A. Basic Purpose of an Operating System

B. The features that characterise a real-time operating system

C. The major concepts involved in the Process Management function of the RTOS
   a. Task Dispatching
   b. Preemptive priority scheduling

D. Basic Styles of implementing the Process Management in RTOS's of varying complexities

## Answers, Remarks and Hints to Points to Ponder

## Point to Ponder: 1

*A. What are the factors on which the execution time of a task depends on?*

**Ans:** The execution time of a task can be divided into two types of activities, namely, CPU time and I/O time. In a multi-tasking operating system a third kind of time is added to this, namely the time spent by the task in waiting for the resources needed, that is CPU, or I/O. This may depend on a variety of factors, such as the other tasks running in the environment, priority, scheduling policies etc.

*B. While a task is executing, is the CPU continuously busy?*

**Ans:** The CPU is always doing something except during the times when it is fetching address or data from memory or devices. But since it may require sometime for the device to send the data, during that time, the CPU may do tasks other than the one for which it requested the device. In this sense the CPU may not be continuously busy.

*C. Suppose there are two mutually independent tasks A and B. If only task A (B) executes on the processor P, it takes $t_A$ ( $t_B$) units of time. If these tasks are fired simultaneously and run under multi-tasking, neglecting time related to task switching, would the execution time for both tasks taken together be greater than, equal to or less than ($t_A + t_B$ ) ?*

**Ans:** That depends on the scheduling policy. If the policy is non-preemptive, it will take time equal to *($t_A + t_B$ )*. If it is preemptive, however, it is expected to take less time than *($t_A + t_B$ )*.

## Point to Ponder: 2

*A. What are possible reasons for which execution time for a task can be unpredictable?*

**Ans:** Basically due to variations in its execution environment. This includes factors such as the other tasks which are executing at the same time, there priorities related to the task in question, the nature and frequency of interrupts coming from the environment etc. Note that, even apart from these environmental factors, the execution time of a task depends on the input data set used for the run. However, this cannot be termed "unpredictable".

*B. Why preemptive multi-tasking is such an important requirement for an RTOS?*

**Ans:** The first reason is that, without it, the concept of priority cannot be implemented properly. Thus without preemption, a task, that may have started when no other higher priority task was present, can block such higher priority tasks for long times, thus violating the principle of priority scheduling. Secondly, with a given computing speed the CPU utilization that can be realised with preemption cannot be utilised without it. Even if limited preemption, (namely that, only a task waiting for I/O is preempted) is used, a given set of tasks that is schedulable with respect to their deadlines under preemptive scheduling, may not be so without premption. It is for these reasons that it is an indispensable feature for an RTOS.

*C. Suppose there are two mutually independent tasks A and B. Let B be the higher priority task. If only task A (B) executes on the processor P, it takes $t_A$ ( $t_B$) units of time. If these tasks are run under multi-tasking, neglecting time related to task switching, can the execution time for task A be less than ($t_A + t_B$ ) ?*

**Ans:** Yes, because, while task B is waiting for I/O, it would be removed to a waiting queue and task A would be scheduled till the time the interrupt for I/O completion is received and task B is put into a ready queue.

## Point to Ponder: 3

*A. What are the situations under which a running task can go to the ready state?*

**Ans:** From the new state, after it is initialised. From the waiting or blocked state, after I/O completion. From the running state, after its allocated time-slice is spent.

*B. What are the situations under which a running task can go to the waiting state?*

**Ans:** If it is blocked on I/O or due to synchronization requirement with another task.

*C. Suppose there are two mutually independent tasks A and B, which are invoked periodically with periods $T_A$ ($T_B$). If only task A (B) executes on the processor P, it takes $t_A$ ($t_B$) units of time. Naturally, $t_A \leq T_A$ and $t_B \leq T_B$. If these tasks are to be run under multi-tasking with negligible task switching time, state any one condition under which task preemption would be essential for the task dead lines to be met.*

**Ans:** $t_B \leq T_A$.

*D. In the above case, let B be the higher priority task which can preempt the execution of task A whenever it is ready. State a condition under which even with preemption, task deadlines cannot be met.*

**Ans:** $T_A \geq \lceil T_A / t_B \rceil t_B + t_A$.

## Point to Ponder: 4

*A. State one factor in respect of which, threads are similar to coroutines. State one factor in respect of which they are different.*

**Ans:** In the sense that they do not involve process management overheads on the OS, such as creation and management of TCB, run-time synchronization management etc. They are different in the sense that, there is RTOS support for thread scheduling. Co-routines are entirely managed by the programmer.

*B. State one situation under which a cyclic executive is adequate? State one situation under which it is not and why not?*

**Ans:** When task execution times are predictable and the tasks can be run in a given fixed order without violating their deadlines, a cyclic executive is adequate. If the above conditions are not met, for example, if tasks are invoked by the environment asynchronously, a cyclic executive is not the choice.

*C. State one way in which are interrupt service routines different from other tasks?*

**Ans:** They are different in the sense that they receive an immediate response from the processor, if they are sensed (i.e. they are not masked). This is by the hardware design of the processor. Tasks on the other hand receive response from the CPU only when they are appropriate to be run under the scheduling policy.

*D. Can you give an example of a practical embedded application, for which an Foreground / Background model of multi-tasking is adequate?*

**Ans:** Consider an embedded CNC machine controller. The tasks in the machine can typically be divided into two categories. In the first category are real-time supervisory control tasks that generate the references for the two dimensional motion of the job as well as for the spindle speed. These references are generated by interpolating the cutting profiles programmed in the machine. On the other hand there are other tasks which are soft/non-real time, such as servicing the operator input console and updating the display. We do not need a full features RTOS here since the nature of tasks and the task loading level is fairly deterministic and predictable.

*E. Under what situations is a full-featured RTOS an appropriate choice? Can you illustrate your answer with the example of a practical embedded application?*

**Ans:** A full featured RTOS is the choice for a complex embedded system where the task loading can vary a lot and can consist of hard real-time tasks of various periods and deadlines. A typical application area is avionics, such as, say, for a fighter aircraft. The task loading levels can vary widely depending on mission profiles as well as emergency situations. Many of these tasks are mission critical. Also the expertise level of people building these is very high and therefore the complexity of such systems can be handled.

## Point to Ponder: 5

*A. Give example of hard, soft and non-real-time computing tasks. In what ways are they different?*

**Ans:** Consider a digital camera. Among the various tasks that are performed by the embedded processor, the shutter control functions which essentially compute the necessary exposure based on the light conditions and opens and closes the shutter is a hard real-time task. If the task does not execute in time, the exposure would change. On the other hand, the task that displays the picture on the display, as one reviews the pictures already taken, is a non-real-time task, because there is no fixed deadline for the task. Note that this task also must execute reasonably fast (otherwise the camera would not sell). On the other hand, the auto-focus task, on such a camera which allows one to zoom on a target may be classified as soft-real-time because if the camera does not focus exactly before the shutter is pressed, the performance of the camera would degrade in terms of photographic quality in some cases. Thus, the categorisation of the task among the three classes is actually intimately related to the criteria of performance that is decided for the system.

*B. Can interrupt level priorities be seen as a task at real-time level with very high priority?*

**Ans:** Interrupts differ from high priority real-time tasks, although in many cases a part of an interrupt service routine may actually be spawned as a real-time high priority task. Firstly, the interrupt response is built in into the hardware, whether it is generated through a hardware pin or a software instruction, and is automatic, once the interrupt is sensed. It is not computed using a scheduling policy as is the case for an RT task. Thus an interrupt does not have a deadline associated with it like an RT task. It may or may not be periodic.

*C. What happens if the user sets priorities in a manner that deadline would be violated for other tasks?*

**Ans:** RT systems generally do not admit of "users" who can set priorities of tasks at run-time. However, an application programmer can write code that spawns new tasks at specified priorities. If these are not set properly deadlines would indeed be failed and the system would be considered to have been implemented according to specifications. In fact, it is for this reasons that embedded systems are tested extensively to ascertain that the all deadlines can be met under all kinds of task scenarios.