Module 8: Real-time Operating Systems and Microkernels

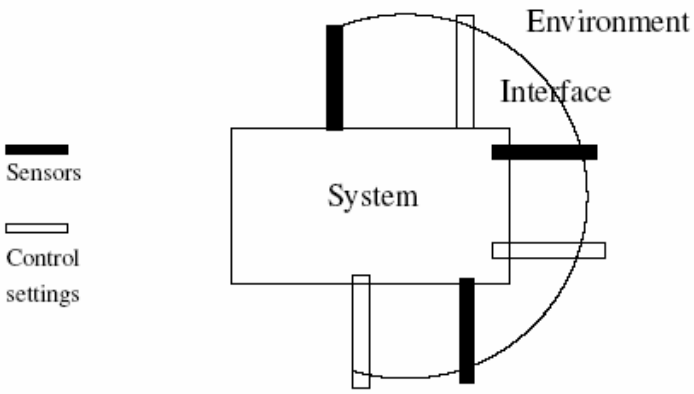
In some data processing applications system responses are meaningful, if these are within a certain stipulated time period. System responses that arrive later than the expected time are usually irrelevant or meaningless. In fact, sometimes, it's no better than being simply wrong. Therefore, the system response must be generated well within the stipulated time. This is true particularly of on-line stock trading, tele-ticketing and similar other transactions. These systems are generally recognized to be real-time systems. For interactive systems the responses ought to match human reaction times to be able to see the effects as well (as in on line banking or video games).

Real-time systems may be required in life critical applications such as patient monitoring systems. They may also be applied in safety critical systems such as reactor control systems in a power plant. Let us consider a safety critical application like anti-lock braking system (ABS), where control settings have to be determined in real-time. A passenger car driver needs to be able to control his automobile under adverse driving conditions.

In a car without ABS, the driver has to cleverly pump and release the brake pedal to prevent skids. Cars, with ABS control, regulate pumping cycle of brakes automatically. This is achieved by modifying the pressure applied on brake pedals by a driver in panic. A real-time control system gives a timely response. Clearly, what is a timely response is determined by the context of application. Usually, one reckons that a certain response is timely, if it allows for enough time to set the needed controller(s) appropriately, i.e. before it is too late. In safety critical, or life critical situations a delay may even result in a catastrophe. Operating systems are designed keeping in mind the context of use. As we have seen, the OS designers ensure high resource utilization and throughput in the general purpose computing context. However, for a system which both monitors and responds to events from its operative environment, the system responses are required to be timely. For such an OS, the minimalist kernel design is required. In fact, since all IO requires use of communications through kernel, it is important that kernel overheads are minimal. This has resulted in emergence of micro-kernels. Micro-kernels are minimal kernels which offer kernel services with minimum overheads. The kernels used in hard

real-time systems 1 are often micro-kernels. In this chapter, we shall cover the relevant issues and strategies to design an OS which can service real-time requirements.

8.1 Characteristics of real-time systems



In a real–time system the events in the environment are detected by sensors and the responses to these events are generated in a timely manner. A RTOS ensures that control settings (in response to an event) are achieved in real–time.

Figure 8.1: Operative environment for RTOS.

A typical real-time operating environment is shown in Figure 8.1. In this figure we note that the computer system has an interface which is embedded within its environment. The operating system achieves the desired extent of regulation as follows:

1. Sense an event: The system monitors its operative environment using some sensors. These sensors keep a tab on some measurable entity. Depending upon the context of use this entity may be a measure of temperature, or a stock price fluctuation or fluid level in a reservoir. These measurements may be periodic. In that case the system would accept an input periodically. In case the measurement of inputs is taken at specified times of operation then the OS may schedule its input at these specified times or it may be interrupted to accept the input. The input may even be measured only when an unusual deviation in the value of the monitored entity occurs. In these cases the input would certainly result in an interrupt. Regardless of the input mode, the system would have an input following a sensor reading (which is an event).
2. Process the data: The next important task is to process the data which has been most recently acquired. The data processing may be aimed at checking the health of the system. Usually it is to determine if some action is needed.

3. Decide on an action: Usually, the processing steps involving arriving at some decisions on control settings. For instance, if the stock prices cross some threshold, then one has to decide to buy or sell or do nothing. As another example, the action may be to open a valve a little more to increase inflow in case reservoir level drops.
4. Take a corrective action: In case, the settings need to be altered, the new settings are determined and control actuators are initiated. Note that the actions in turn affect the environment. It is quite possible that as a consequence, a new set of events get triggered. Also, it is possible that the corrective step requires a drastic and an immediate step. For instance, if an alarm is to be raised, then all the other tasks have to be suspended or pre-empted and an alarm raised immediately. Real-time systems quite often resort to pre-emption to prevent a catastrophe from happening.

The OS may be a bare-bone microkernel to ensure that input events are processed with minimum overhead. Usually, the sensor and monitoring instruments communicate with the rest of the system in interrupt mode. Device drivers are specifically tuned to service these inputs. In Section 8.2 we shall discuss the related design issues for micro-kernels and RTOS.

Why not use Unix or Windows? This is one very natural question to raise. Unix or Windows are operating systems that have been designed with no specific class of applications in mind. These are robust, (like all terrain vehicles), but not suitable for real-time operations (say Formula 1 cars). Their performance in real-time domain would be like that of an all terrain vehicle on a formula one race track. Note that the timeliness in response is crucial in real-time operations. General-purpose operating systems are designed to enhance throughput. Often it has considerable leeway in responding to events. Also, within a service type, the general-purpose OS cater to a very vast range of services. For example, just consider the print service. There is considerable leeway with regard to system response time. Additionally, the printer service may cater to a vast category of print devices which range from ink-jet to laser printing or from gray scale to color printing. In other words, the service rendering code is long. Additionally, it caters to a large selection in printer devices. This makes service rendering slow. Also, a few seconds of delay in printing matters very little, if at all. Real-time operative environments

usually have a fixed domain of operations in which events have fairly predictable patterns, but do need monitoring and periodic checks. For instance, a vessel in a chemical process will witness fairly predictable form of rise in temperature or pressure, but needs to be monitored. This means that the scheduling strategies would be event centered or time centered. In a general-purpose computing environment the events arise from multiple, and not necessarily predictable, sources. In real-time systems, the events are fairly well known and may even have a pattern. However, there is a stipulated response time. Within this context, development of scheduling algorithms for real-time systems is a major area of research.

A natural question which may be raised is: Can one modify a general purpose OS to meet real-time requirements. Sometimes a general-purpose OS kernel is stripped down to provide for the basic IO services. This kernel is called microkernel. Microkernels do meet RTOS application specific service requirements. This is what is done in Windows CE and Embedded Linux.

Note we have made two important points above. One relates to timeliness of response and the other relates to event-centric operation. Scheduling has to be organized to ensure timeliness under event-centric operation. This may have to be done at the expense of loss of overall throughput!!

8.1.1 Classification of Real-time Systems

The classification of real-time systems is usually based on the severity of the consequences of failing to meet time constraints. This can be understood as follows. Suppose a system requires a response to an event in time period T. Now we ask: what happens if the response is not received within the stipulated time period? The failure to meet the time constraint may result in different degrees of severity of consequences. In a life-critical or safety critical application, the failure may result in a disaster such as loss of life. A case in point is the shuttle Columbia's accident in early February 2 2003. Recall Kalpana Chawla, an aeronautics engineering Ph. D. was on board. During its descent, about 16 minutes from landing, the spacecraft temperature rose to dangerous levels resulting in a catastrophic end of the mission. Clearly, the rise in temperature as a space draft enters earth's atmosphere is anticipated. Spacecrafts have RTOS regulating the controllers to respond to such situations from developing. And yet the cooling system(s) in this case did not offer timely mitigation. Both in terms of loss of human life and the

cost of mission such a failure has the highest severity of consequences. Whereas in the case of an online stock trading, or a game show, it may mean a financial loss or a missed opportunity. In the case of a dropped packet in a video streaming application it would simply mean a glitch and a perhaps a temporary drop in the picture quality. The three examples of real-time system we have given here have different levels of severity in terms of timely response. The first one has life-threatening implication; the second case refers to a missed opportunity and finally, degraded picture quality in viewing. Associated with these are the broadly accepted categories | hard, firm and soft real-time systems.

Architecture of Real-time Systems: The basic architecture of such systems is simple. As shown in Figure 8.1, some sensors provide input from the operative environment and a computation determines the required control. Finally, an appropriate actuator is activated. However, since the consequence of failure to respond to events can be catastrophic, it is important to build in the following two features in the system.

(a) It should be a fault tolerant design.

(b) The scheduling policy must provide for pre-emptive action.

For a fault tolerant design, the strategies may include majority voting out of the faulty sensors. Systems like satellite guidance system, usually have back-up (or a hot-stand-by) system to fall back upon. This is because the cost of failure of a mission is simply too high. Designers of Airbus A-320 had pegged the figure of failure probability at lower than 10^{-10} for one hour period in flight

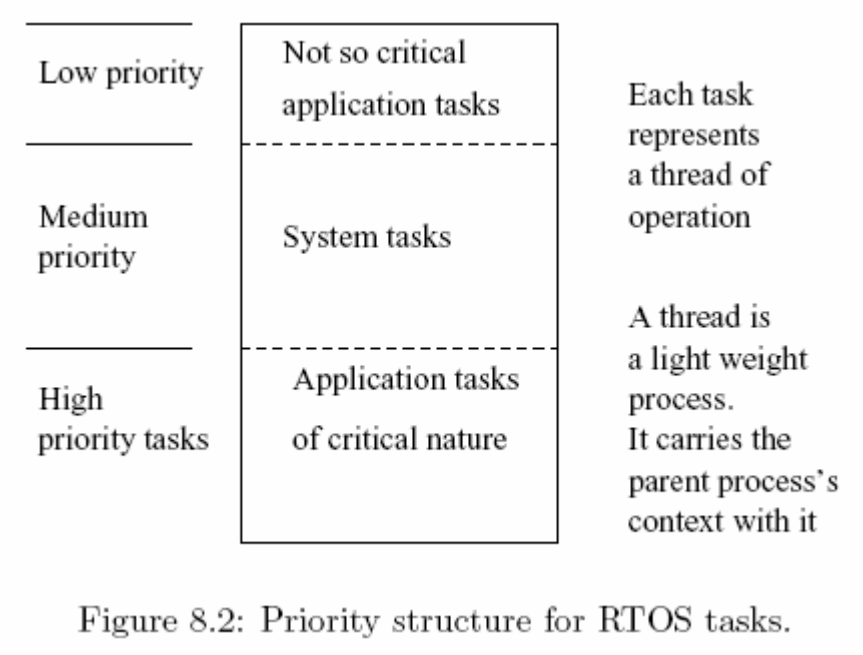


Figure 8.2: Priority structure for RTOS tasks.

As for design of scheduling policy, one first identifies the critical functions and not so critical functions within an operation. The scheduling algorithm ensures that the critical functions obtain high priority interrupts to elicit immediate responses. In Figure 8.2, we depict the priority structure for such a design.

A very detailed discussion on design of real-time systems is beyond the scope of this book. Yet, it is worth mentioning here that RTOS designers have two basic design orientations to consider. One is to think in terms of event-triggered operations and the other is to think of time-triggered operations. These considerations also determine its scheduling policy. The report prepared by Panzierri and his colleagues compares architectures based on these two considerations. The observation is that time-triggered architectures obtain greater predictability but end up wasting more resource cycles of operation due to more frequent pre-emptions. On the other hand, event-triggered system architectures seem to score in terms of their ability to adapt to a variety of operating scenarios. Event-triggered systems are generally better suited for asynchronous input events. The time-triggered systems are better suited for systems with periodic inputs. For now, let us examine micro-kernels which are at the heart of RTOS, event-triggered or time-triggered.

8.2 Microkernels and RTOS

As stated earlier, micro-kernels are kernels with bare-bone, minimal essentials. To understand the notion of “bare-bone minimal essentials", we shall proceed bottom up. Also, we shall take an embedded system design viewpoint.

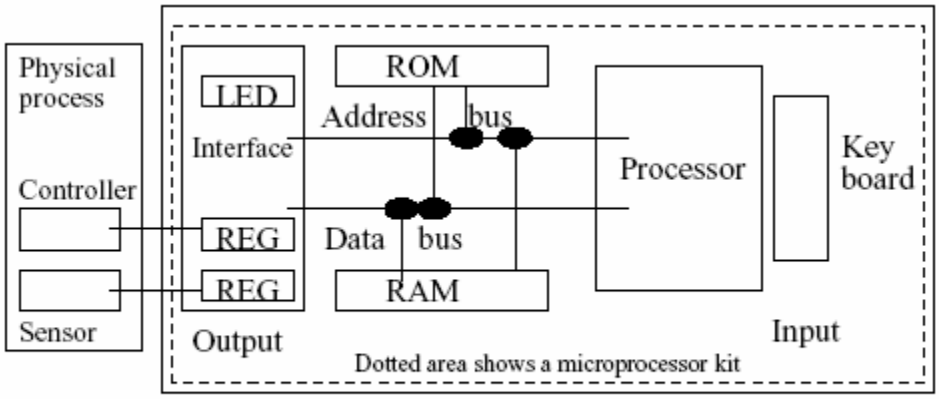


Figure 8.3: Embedded system architecture.

Let us first consider a microprocessor kit. The kit is shown in figure 8.3 within the dotted area. We can program the kit in machine language. The program can be directly stored in memory. On execution we observe output on LEDs. We also have some attached ROM. To simulate the operation of an embedded system input can be read from sensors and we can output to an interface to activate an actuator. We can use the timers and use these to periodically monitor a process. One can demonstrate the operation of an elevator control or a washing machine with these kits. We just write one program and may even do single steps through this program. Here there is no need to have an operating system. There is only one resident program.

Next, we move to an added level of complexity in interfaces. For an embedded system, input and output characterizations are very crucial. Many of the controls in embedded systems require a real-time clock. The need for real-time clocks arises from the requirement to periodically monitor (or regulate) process health. Also, abnormal state of any critical process variable needs to be detected. The timers, as also abnormal values of process state variables, generate interrupt. An example of process monitoring is shown in figure 8.4. As for the operational scenario, note that a serial controller is connected to two serial ports. Both the serial ports may be sending data. The system needs to regulate this traffic through the controller. For instance, in our example, regulating the operations of the serial controller is itself a task. In general, there may be more than one task each with

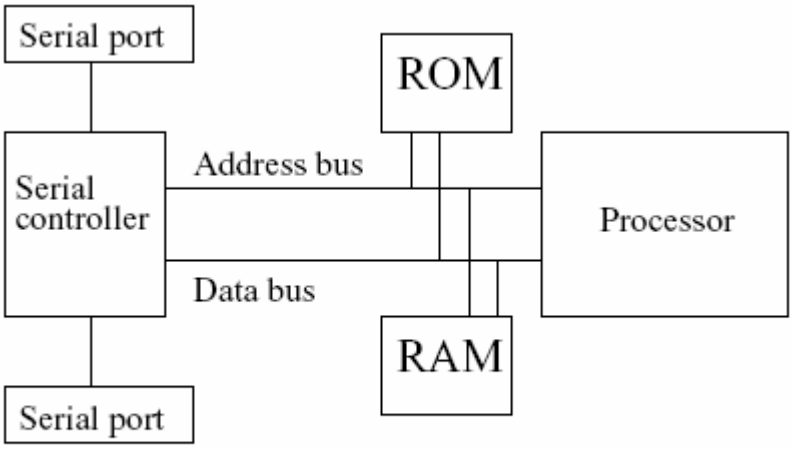


Figure 8.4: Embedded system example.

its own priority level to initiate an interrupt, or there is a timer to interrupt. Essentially, one or more interrupts may happen. Interrupts require two actions. One to store away the context of the running process. The other is to switch to an interrupt service routine (ISR). The ROM may store ISRs. Before switching to an ISR, the context (the status of the present program) can be temporarily stored in RAM. All these requirements translate to management of multiple tasks with their own priorities. And that establishes the need for an embedded operating system.

In addition, to fully meet control requirements, the present level of technology supports on-chip peripherals. Also, there may be more than one timer. Multiple timers enable monitoring multiple activities, each with a different period. There are also a number of ports to support inputs from multiple sensors and outputs to multiple controllers. All this because: a process in which this system is embedded usually has several periodic measurements to be made and several levels of priority of operations. Embedded systems may be even internet enabled. For instance, hand-held devices discussed in Section 8.2.1 are usually net enabled.

In Figure 8.5 we show a software view. The software view is that the device drivers are closely and directly tied to the peripherals. This ensures timely IO required by various tasks. The context of the applications define the tasks.

Typically, the IO may be using polling or an interrupt based IO. The IO may also be memory mapped. If it is memory mapped then the memory space is adequately allocated to offer IO mappings. Briefly, then the embedded system OS designers shall pay attention

to the device drivers and scheduling of tasks based on interrupt priority. The device driver functions in this context are the following.

- Do the initialization when started. May need to store an initial value in a register.
- Move the data from the device to the system. This is the most often performed task by the device driver.
- Bring the hardware to a safe state, if required. This may be needed when a recovery is required or the system needs to be reset.
- Respond to interrupt service routine. The interrupt service routine may need some status information. A typical embedded system OS is organized as a minimal system. Essentially, it is a system which has a microkernel at the core which is duly supported by a library of system call functions. The microkernel together with this library is capable of the following.
- Identify and create a task.
- Resource allocation and reallocation amongst tasks.
- Delete a task.
- Identify task state like running, ready-to-run, blocked for IO etc.
- To support task operations (launch, block, read-port, run etc.), i.e. should facilitate low level message passing (or signals communication).
- Memory management functions (allocation and de-allocation to processes).
- Support preemptive scheduling policy.
- Should have support to handle priority inversion3.

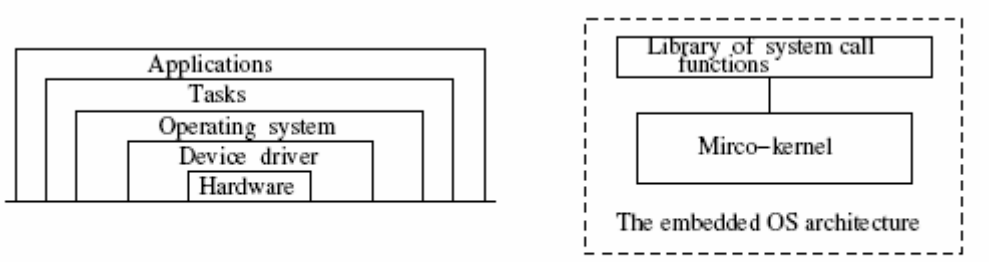


Figure 8.5: Software view of microkernel based OS.

Let us elaborate on some of these points. The allocation and de-allocation of main memory in a microkernel requires that there is a main memory management system. Also, the fact that we can schedule the operation of tasks means that it is essential to have a loader as a part of a microkernel. Usually, the micro-kernels are designed with a system

call \functions" library. These calls support, creation of a task, loading of a task, and communication via ports. Also, there may be a need to either suspend or kill tasks. When tasks are de-allocated, a resource reallocation may happen. This requires support for semaphores. Also, note that a support for critical section management is needed. With semaphores this can be provided for as well. In case the system operates in a distributed environment (tele-metered or internet environment), then a network support is also required. Here again, the support could be minimal so as to be able to communicate via a port. Usually in such systems, the ports are used as “mailboxes". Finally, hardware dependent features are supported via system calls.

When a task is created (or loaded), the task parameters in the system calls, include the size (in terms of main memory required), priority of the task, point of entry for task and a few other parameters to indicate the resources, ownership, or access rights. The microkernel needs to maintain some information about tasks like the state information of each task. This again is very minimal information like, running, runnable, blocked, etc. For periodic tasks we need to support the clock-based interrupt mechanism. We also have to support multiple interrupt levels. There are many advantages of a microkernel-based OS design. In particular, a microkernel affords portability. In fact, Carsten Ditze [10], argues that microkernel can be designed with minimal hardware dependence. The user services can be offered as set of library of system calls or utilities. In fact, Carsten advocates configuration management by suitably tailoring the library functions to meet the requirements of a real-time system. In brief, there are two critical factors in the microkernel design. One concerns the way we may handle nested interrupts with priority. The other concerns the way we may take care of scheduling. We studied interrupts in detail in the chapter on IO. Here, we focus on the consideration in the design of schedulers for real-time systems. An embedded OS veers around the device drivers and a microkernel with a library of system calls which supports real-time operations. One category of embedded systems are the hand-held devices. Next, we shall see the nature of operations of hand-held devices.

8.2.1 OS for Hand-held Devices

The first noticeable characteristic of hand-held devices is their size. Hand-held devices can be carried in person; this means that these device offer mobility. Mobile devices may be put in the category of phones, pagers, and personal digital assistants (PDAs). Each of

these devices have evolved from different needs and in different ways. Mobile phones came about as the people became more mobile themselves. There was this felt need to extend the capabilities of land-line communication. Clearly, the way was to go over air and support common phone capabilities for voice. The mobile phone today supports text via SMS and even offers internet connectivity. The pagers are ideal for text transmission and PDAs offer many of the personal productivity tools. Now a days there are devices which capture one or more of these capabilities in various combinations. In Figure 8.6 we depict architecture of a typical hand-held device. Typically, the hand-held devices have the following hard-ware base.

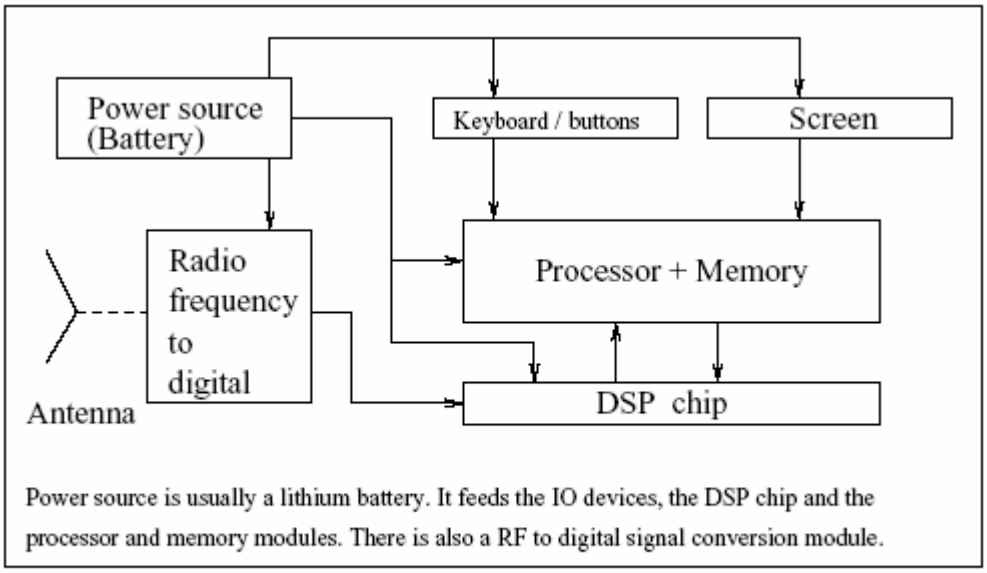


Figure 8.6: A typical hand-held device architecture. *

Microprocessor * Memory (persistent + volatile) * RF communication capability * IO units (keys + buttons / small screen (LCD)) * Power source (battery)

Today's enabling technologies offer sophisticated add-ons. For instance, the DSP (digital signal processing) chips allow MP3 players and digital cameras to be attached with PDAs and mobiles. This means that there are embedded and real-time applications being developed for hand-held devices. The signal processing capabilities are easily several MIPS (million instructions per second) and beyond. The IO may also allow use of stylus and touch screen capabilities. Now let us look at some of the design concerns for OS on hand-held devices. One of the main considerations in hand-held devices is to be able to operate while conserving power. Even though the lithium batteries are rechargeable,

these batteries drain in about 2 hours time. Another consideration is the flexibility in terms of IO.

These devices should be able to communicate using serial ports (or USB ports), infrared ports as well as modems. The OS should be able to service file transfer protocols. Also, the OS should have a small footprint, typically about 100K bytes with plug-in modules. Other design requirements include very low boot time and robustness. Another important facet of the operations is that hand-held devices hold a large amount of personal and enterprise information. This requires that the OS should have some minimal individual authentication before giving access to the device.

In some of the OSs, a memory management unit (MMU) is used to offer virtual memory operation. The MMU also determines if the data is in RAM. The usual architecture is microkernel supported by a library of functions just as we described in Section 8.2. An embedded Linux or similar capability OS is used in these devices. Microsoft too has some offerings around the Windows CE kernel.

The main consideration in scheduling for real-time systems is the associated predictability of response. To ensure the predictability of response, real-time systems resort to pre-emptive policies. This ensures that an event receives its due attention. So, one basic premise in real-time systems is that the scheduling must permit pre-emption. The obvious price is: throughput. The predictability requirement is regardless of the nature of input from the environment, which may be synchronous or asynchronous. Also, note that predictability does not require that the inputs be strictly periodic in nature (it does not rule out that case though). Predictability does tolerate some known extent of variability. The required adjustment to the variability is akin to the task of a wicket-keeper in the game of cricket. A bowler brings in certain variabilities in his bowling. The wicket-keeper is generally aware of the nature of variability the bowler beguiles. The wicket-keeper operates like a real-time system, where the input generator is the bowler,

who has the freedom to choose his ball line, length, and flight. Clearly, one has to bear the worst case in mind. In real-time systems too, the predictable situations require to cater for the worst case schedulability of tasks. Let us first understand this concept. Schedulability of a task is influenced by all the higher priority tasks that are awaiting scheduling. We can explain this as follows. Suppose we identify our current task as tc and the other higher priority tasks as t1; :::; tn where ti identifies the i-th task having

priority higher than tc. Now let us sum up the upper bounds for time of completion for all the higher priority tasks t1; :::; tn and to it add the time required for tc. If the total time is less than the period by which task tc must be completed, then we say that task tc meets the worst case schedulability consideration. Note that schedulability ensures predictability and offers an upper bound on acceptable time of completion for task tc. Above, we have emphasized pre-emption and predictability for real-time systems. We shall next examine some popular scheduling policies. The following three major scheduling policies are quite popular.

(a) Rate monotonic (or RM for short) scheduling policy.

(b) The earliest deadline first (or EDF for short) scheduling policy.

(c) The least laxity first (or LLF) scheduling policy.

We describe these policies in Sections 8.3.1, 8.3.2 and 8.3.3. A curious reader may wonder if the predictability (in terms of schedulability for timely response) could be guaranteed under all conditions. In fact, predictability does get affected when a lower priority task holds a mutually shared resource and blocks a higher priority task. This is termed as a case of priority inversion. The phenomenon of priority inversion may happen both under RM and EDF schedules. We shall discuss the priority inversion in section 8.3.4. We shall also describe strategies to overcome this problem.

8.3.1 Rate Monotonic Scheduling

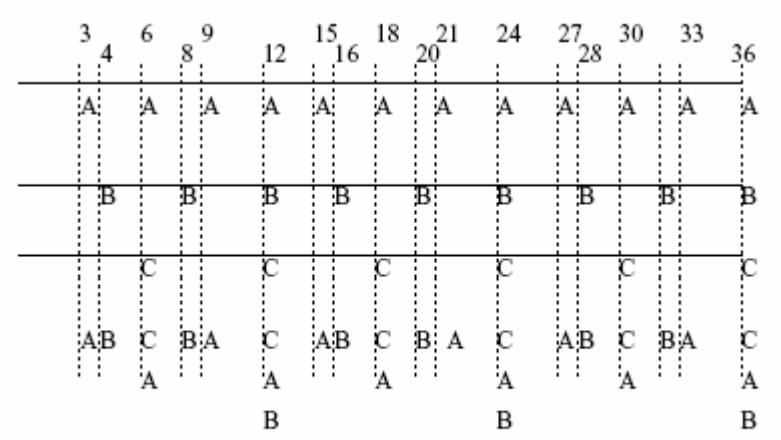


Figure 8.7: Rate monotonic scheduling for tasks.

Some real-time systems have tasks that require periodic monitoring and regulation. So the events are cyclic in nature. This cyclicity requires predictable event detection and consequent decision on control settings. For this category of real-time systems the

popular scheduling strategy is rate monotonic scheduling. Let us now examine it in some detail. The rate monotonic scheduling stipulates that all the tasks are known apriori. Also, known is their relative importance. This means that we know their orders of priority. Tasks with highest priority have the shortest periodicity. The tasks may be independent of each other. Armed with this information, and the known times of completion for each task, we find the least common multiple lcm of the task completion times. Let us denote the lcm as Trm. Now a schedule is drawn for the entire time period Trm such that each task satisfies the schedulability condition. The schedule so generated is the RM schedule. This schedule is then repeated with period Trm. As an example, consider that events A;B; and C happen with time periods 3, 4, and 6, and when an event occurs the system must respond to these. Then we need to draw up a schedule as shown in Figure 8.7. Note that at times 12, 24, and 36 all the three tasks need to be attended to while at time 21 only task A needs to be attended. This particular schedule is drawn taking its predictability into account. To that extent the RM policy ensures predictable performance. In theory, the rate monotonic scheduling is known to be an optimal policy when priorities are statically defined tasks.

8.3.2 Earliest Deadline First Policy

The EDF scheduling handles the tasks with dynamic priorities. The priorities are determined by the proximity of the deadline for task completion. Lower priorities are assigned when deadlines are further away. The highest priority is accorded to the task with the earliest deadline. Clearly, the tasks can be put in a queue like data-structure with the entries in the ascending order of the deadlines for completion. In case the inputs are periodic in nature, schedulability analysis is possible. However, in general the EDF policy can not guarantee optimal performance for the environment with periodic inputs. Some improvement can be seen when one incorporates some variations in EDF policy. For instance, one may account for the possibility of distinguishing the mandatory tasks from those that may be optional. Clearly, the mandatory tasks obtain schedules for execution by pre-emption whenever necessary. The EDF is known to utilize the processor better than the RM policy option.

8.3.3 Earliest Least Laxity First Policy

This is a policy option in which we try to see the slack time that may be available to start a task. This slack time is measured as follows:

slack time = dead line ¡ remaining processing time

The slack defines the laxity of the task. This policy has more overhead in general but has been found to be very useful for multi-media applications in multiprocessor environment.

8.3.4 Priority Inversion

Often priority inversion happens when a higher priority task gets blocked due to the fact that a mutually exclusive resource R is currently being held by a lower priority task. This may happen as follows. Suppose we have three tasks t1; t2, and t3 with priorities in the order of their index with task t1 having the highest priority. Now suppose there is a shared resource R which task t3 and task t1 share. Suppose t3 has obtained resource R and it is to now execute. Priority inversion occurs with the following plausible sequence of events.

1. Resource R is free and t3 seeks to execute. It gets resource R and t3 is executing.
2. Task t3 is executing. However, before it completes task t1 seeks to execute.
3. Task t3 is suspended and task t1 begins to execute till it needs resource R. It gets suspended as the mutually exclusive resource R is not available.
4. Task t3 is resumed. However, before it completes task t2 seeks to be scheduled.
5. Task t3 is suspended and task t2 begins executing.
6. Task t2 is completed. Task t1 still cannot begin executing as resource R is not available (held by task t3).
7. Task t3 resumes to finish its execution and releases resource R.
8. Blocked task t1 now runs to completion.

The main point to be noted in the above sequence is: even though the highest priority task t1 gets scheduled using a pre-emptive strategy, it completes later than the lower priority tasks t2 and t3!! Now that is priority inversion.

How to handle priority inversion: To ensure that priority inversion does not lead to missing deadlines, the following strategy is adopted [19]. In the steps described above, the task t1 blocks when it needs resource R which is with task t3. At that stage, we raise the priority of task t3, albeit temporarily to the level of task t1. This ensures that task t2 cannot get scheduled now. Task t3 is bound to complete and release the resource R. That would enable scheduling of the task t1 before task t2. This preserves the priority order and avoids the priority inversion. Consequently, the deadlines for task t1 can be adhered to with predictability.