Module 6

Embedded System Software

Lesson 28

Introduction to Real-Time Systems

Specific Instructional Objectives

At the end of this lesson, the student would be able to:

- Know what a Real-Time system is
- Get an overview of the various applications of Real-Time systems
- Visualize the basic model of a Real-Time system
- Identify the characteristics of a Real-Time system
- Understand the safety and reliability aspects of a Real-Time system
- Know how to achieve highly reliable software
- Get an overview of the software fault-tolerant techniques
- Classify the Real-Time tasks into different categories

1. Introduction

Commercial usage of computer dates back to a little more than fifty years. This brief period can roughly be divided into mainframe, PC, and post-PC eras of computing. The mainframe era was marked by expensive computers that were quite unaffordable by individuals, and each computer served a large number of users. The PC era saw the emergence of desktops which could be easily be afforded and used by the individual users. The post-PC era is seeing emergence of small and portable computers, and computers embedded in everyday applications, making an individual interact with several computers everyday.

Real-time and embedded computing applications in the first two computing era were rather rare and restricted to a few specialized applications such as space and defense. In the post-PC era of computing, the use of computer systems based on real-time and embedded technologies has already touched every facet of our life and is still growing at a pace that was never seen before. While embedded processing and Internet-enabled devices have now captured everyone's imagination, they are just a small fraction of applications that have been made possible by real-time systems. If we casually look around us, we can discover many of them often they are camouflaged inside simple looking devices. If we observe carefully, we can notice several gadgets and applications which have today become in- dispensable to our every day life, are in fact based on embedded real-time systems. For example, we have ubiquitous consumer products such as digital cameras, cell phones, microwave ovens, camcorders, video game sets; telecommunication domain products and applications such as set-top boxes, cable modems, voice over IP (VoIP), and video conferencing applications; office products such as fax machines, laser printers, and security systems. Besides, we encounter real-time systems in hospitals in the form of medical instrumentation equipments and imaging systems. There are also a large number of equipments and gadgets based on real-time systems which though we normally do not use directly, but never the less are still important to our daily life. A few examples of such systems are Internet routers, base stations in cellular systems, industrial plant automation systems, and industrial robots.

It can be easily inferred from the above discussion that in recent times real-time computers have become ubiquitous and have permeated large number of application areas. At present, the

computers used in real-time applications vastly outnumber the computers that are being used in conventional applications. According to an estimate [3], 70% of all processors manufactured world-wide are deployed in real-time embedded applications. While it is already true that an overwhelming majority of all processors being manufactured are getting deployed in real-time applications, what is more remarkable is the unmistakable trend of steady rise in the fraction of all processors manufactured world-wide finding their way to real-time applications.

Some of the reasons attributable to the phenomenal growth in the use of real-time systems in the recent years are the manifold reductions in the size and the cost of the computers, coupled with the magical improvements to their performance. The availability of computers at rapidly falling prices, reduced weight, rapidly shrinking sizes, and their increasing processing power have together contributed to the present scenario. Applications which not too far back were considered prohibitively expensive to automate can now be affordably automated. For instance, when microprocessors cost several tens of thousands of rupees, they were considered to be too expensive to be put inside a washing machine; but when they cost only a few hundred rupees, their use makes commercial sense.

The rapid growth of applications deploying real-time technologies has been matched by the evolutionary growth of the underlying technologies supporting the development of real-time systems. In this book, we discuss some of the core technologies used in developing real-time systems. However, we restrict ourselves to software issues only and keep hardware discussions to the bare minimum. The software issues that we address are quite expansive in the sense that besides the operating system and program development issues, we discuss the networking and database issues.

In this chapter, we restrict ourselves to some introductory and fundamental issues. In the next three chapters, we discuss some core theories underlying the development of practical real-time and embedded systems. In the subsequent chapter, we discuss some important features of commercial real-time operating systems. After that, we shift our attention to real-time communication technologies and databases.

1.1. What is Real-Time?

Real-time is a quantitative notion of time. Real-time is measured using a physical (real) clock. Whenever we quantify time using a physical clock, we deal with real time. An example use of this quantitative notion of time can be observed in a description of an automated chemical plant. Consider this: when the temperature of the chemical reaction chamber attains a certain predetermined temperature, say 250^{o}C, the system automatically switches off the heater within a predetermined time interval, say within 30 milliseconds. In this description of a part of the behavior of a chemical plant, the time value that was referred to denotes the readings of some physical clock present in the plant automation system.

In contrast to real time, logical time (also known as virtual time) deals with a qualitative notion of time and is expressed using event ordering relations such as before, after, sometimes, eventually, precedes, succeeds, etc. While dealing with logical time, time readings from a physical clock are not necessary for ordering the events. As an example, consider the following part of the behavior of library automation software used to automate the book-keeping activities of a college library: "After a *query book* command is given by the user, details of all matching

books are displayed by the software." In this example, the events "issue of query book command" and "display of results" are logically ordered in terms of which events follow the other. But, no quantitative expression of time was required. Clearly, this example behavior is devoid of any real-time considerations. We are now in a position to define what a real-time system is:

> A system is called a real-time system, when we need quantitative expression of time (i.e. real-time) to describe the behavior of the system.

Remember that in this definition of a real-time system, it is implicit that all quantitative time measurements are carried out using a physical clock. A chemical plant, whose part behavior description is - when temperature of the reaction chamber attains certain predetermined temperature value, say 250^{o}C, the system automatically switches off the heater within say 30 milliseconds - is clearly a real-time system. Our examples so far were restricted to the description of partial behavior of systems. The complete behavior of a system can be described by listing its response to various external stimuli. It may be noted that all the clauses in the description of the behavior of a real-time system need not involve quantitative measures of time. That is, large parts of a description of the behavior of a system may not have any quantitative expressions of time at all, and still qualify as a real-time system. Any system whose behavior can completely be described without using any quantitative expression of time is of course not a real-time system.

1.2. Applications of Real-Time Systems

Real-time systems have of late, found applications in wide ranging areas. In the following, we list some of the prominent areas of application of real-time systems and in each identified case, we discuss a few example applications in some detail. As we can imagine, the list would become very vast if we try to exhaustively list all areas of applications of real-time systems. We have therefore restricted our list to only a handful of areas, and out of these we have explained only a few selected applications to conserve space. We have pointed out the quantitative notions of time used in the discussed applications. The examples we present are important to our subsequent discussions and would be referred to in the later chapters whenever required.

1.2.1. Industrial Applications

Industrial applications constitute a major usage area of real-time systems. A few examples of industrial applications of real-time systems are: process control systems, industrial automation systems, SCADA applications, test and measurement equipments, and robotic equipments.

Example 1: Chemical Plant Control

Chemical plant control systems are essentially a type of process control application. In an automated chemical plant, a real-time computer periodically monitors plant conditions. The plant conditions are determined based on current readings of pressure, temperature, and chemical concentration of the reaction chamber. These parameters are sampled periodically. Based on the values sampled at any time, the automation system decides on the corrective actions necessary at that instant to maintain the chemical reaction at a certain rate.

Each time the plant conditions are sampled, the automation system should decide on the exact instantaneous corrective actions required such as changing the pressure, temperature, or chemical concentration and carry out these actions within certain predefined time bounds. Typically, the time bounds in such a chemical plant control application range from a few micro seconds to several milliseconds.

Example 2: Automated Car Assembly Plant

An automated car assembly plant is an example of a plant automation system. In an automated car assembly plant, the work product (partially assembled car) moves on a conveyor belt (see Fig. 28.1). By the side of the conveyor belt, several workstations are placed. Each workstation performs some specific work on the work product such as fitting engine, fitting door, fitting wheel, and spray painting the car, etc. as it moves on the conveyor belt. An empty chassis is introduced near the first workstation on the conveyor belt. A fully assembled car comes out after the work product goes past all the workstations. At each workstation, a sensor senses the arrival of the next partially assembled product. As soon as the partially assembled product is sensed, the workstation begins to perform its work on the work product. The time constraint imposed on the workstation computer is that the workstation must complete its work before the work product moves away to the next workstation. The time bounds involved here are typically of the order of a few hundreds of milliseconds.

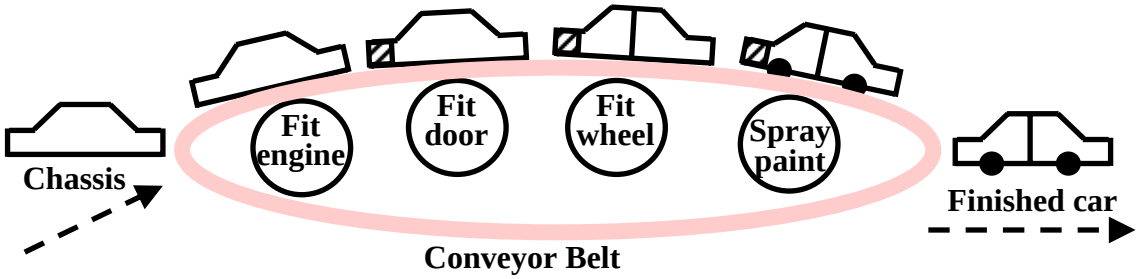


Fig. 28.1 Schematic Representation of an Automated Car Assembly Plant

Example 3: Supervisory Control And Data Acquisition (SCADA)

SCADA are a category of distributed control systems being used in many industries. A SCADA system helps monitor and control a large number of distributed events of interest. In SCADA systems, sensors are scattered at various geographic locations to collect raw data (called events of interest). These data are then processed and stored in a real-time database. The database models (or reflects) the current state of the environment. The database is updated frequently to make it a realistic model of the up-to-date state of the environment. An example of a SCADA application is an Energy Management System (EMS). An EMS helps to carry out load balancing in an electrical energy distribution network. The EMS senses the energy consumption at the distribution points and computes the load across different phases of power supply. It also helps dynamically balance the load. Another example of a SCADA system is a system that monitors and controls traffic in a computer network. Depending on the sensed load in different segments of the network, the SCADA system makes the router change its traffic routing policy dynamically. The time constraint in such a SCADA

application is that the sensors must sense the system state at regular intervals (say every few milliseconds) and the same must be processed before the next state is sensed.

1.2.2. Medical

A few examples of medical applications of real-time systems are: robots, MRI scanners, radiation therapy equipments, bedside monitors, and computerized axial tomography (CAT).

Example 4: Robot Used in Recovery of Displaced Radioactive Material

Robots have become very popular nowadays and are being used in a wide variety of medical applications. An application that we discuss here is a robot used in retrieving displaced radioactive materials. Radioactive materials such as Cobalt and Radium are used for treatment of cancer. At times during treatment, the radioactive Cobalt (or Radium) gets dislocated and falls down. Since human beings can not come near a radioactive material, a robot is used to restore the radioactive material to its proper position. The robot walks into the room containing the radioactive material, picks it up, and restores it to its proper position. The robot has to sense its environment frequently and based on this information, plan its path. The real-time constraint on the path planning task of the robot is that unless it plans the path fast enough after an obstacle is detected, it may collide with it. The time constraints involved here are of the order of a few milliseconds.

1.2.3. Peripheral equipments

A few examples of peripheral equipments that contain embedded real-time systems are: laser printers, digital copiers, fax machines, digital cameras, and scanners.

Example 5: Laser Printer

Most laser printers have powerful microprocessors embedded in them to control different activities associated with printing. The important activities that a microprocessor embedded in a laser printer performs include the following: getting data from the communication port(s), typesetting fonts, sensing paper jams, noticing when the printer runs out of paper, sensing when the user presses a button on the control panel, and displaying various messages to the user. The most complex activity that the microprocessor performs is driving the laser engine. The basic command that a laser engine supports is to put a black dot on the paper. However, the laser engine has no idea about the exact shapes of different fonts, font sizes, italic, underlining, boldface, etc. that it may be asked to print. The embedded microprocessor receives print commands on its input port and determines how the dots can be composed to achieve the desired document and manages printing the exact shapes through a series of dot commands issued to the laser engine. The time constraints involved here are of the order of a few milli seconds.

1.2.4. Automotive and Transportation

A few examples of automotive and transportation applications of real-time systems are: automotive engine control systems, road traffic signal control, air-traffic control, high-speed train control, car navigation systems, and MPFI engine control systems.

Example 6: Multi-Point Fuel Injection (MPFI) System

An MPFI system is an automotive engine control system. A conceptual diagram of a car embedding an MPFI system is shown in Fig.28.2. An MPFI is a real-time system that controls the rate of fuel injection and allows the engine to operate at its optimal efficiency. In older models of cars, a mechanical device called the carburetor was used to control the fuel injection rate to the engine. It was the responsibility of the carburetor to vary the fuel injection rate depending on the current speed of the vehicle and the desired acceleration. Careful experiments have suggested that for optimal energy output, the required fuel injection rate is highly nonlinear with respect to the vehicle speed and acceleration. Also, experimental results show that the precise fuel injection through multiple points is more effective than single point injection. In MPFI engines, the precise fuel injection rate at each injection point is determined by a computer. An MPFI system injects fuel into individual cylinders resulting in better 'power balance' among the cylinders as well as higher output from each one along with faster throttle response. The processor primarily controls the ignition timing and the quantity of fuel to be injected. The latter is achieved by controlling the duration for which the injector valve is open — popularly known as *pulse width*. The actions of the processor are determined by the data gleaned from sensors located all over the engine. These sensors constantly monitor the ambient temperature, the engine coolant temperature, exhaust temperature, emission gas contents, engine rpm (speed), vehicle road speed, crankshaft position, camshaft position, etc. An MPFI engine with even an 8-bit computer does a much better job of determining an accurate fuel injection rate for given values of speed and acceleration compared to a carburetor-based system. An MPFI system not only makes a vehicle more fuel efficient, it also minimizes pollution by reducing partial combustion.

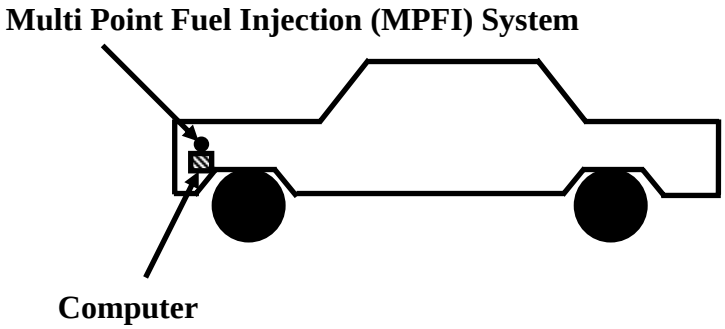


Fig. 28.2 A Real-Time System Embedded in an MPFI Car

1.2.5. Telecommunication Applications

A few example uses of real-time systems in telecommunication applications are: cellular systems, video conferencing, and cable modems.

Example 7: A Cellular System

Cellular systems have become a very popular means of mobile communication. A cellular system usually maps a city into cells. In each cell, a base station monitors the mobile handsets present in the cell. Besides, the base station performs several tasks such as locating a user, sending and receiving control messages to a handset, keeping track of

call details for billing purposes, and hand-off of calls as the mobile moves. Call hand-off is required when a mobile moves away from a base station. As a mobile moves away, its received signal strength (RSS) falls at the base station. The base station monitors this and as soon as the RSS falls below a certain threshold value, it hands-off the details of the on-going call of the mobile to the base station of the cell to which the mobile has moved. The hand-off must be completed within a sufficiently small predefined time interval so that the user does not feel any temporary disruption of service during the hand-off. Typically call hand-off is required to be achieved within a few milliseconds.

1.2.6. Aerospace

A few important use of real-time systems in aerospace applications are: avionics, flight simulation, airline cabin management systems, satellite tracking systems, and computer on-board an aircraft.

Example 8: Computer On-board an Aircraft

In many modern aircrafts, the pilot can select an "auto pilot" option. As soon as the pilot switches to the "auto pilot" mode, an on-board computer takes over all controls of the aircraft including navigation, take-off, and landing of the aircraft. In the "auto pilot" mode, the computer periodically samples velocity and acceleration of the aircraft. From the sampled data, the on-board computer computes X, Y, and Z co-ordinates of the current aircraft position and compares them with the pre-specified track data. Before the next sample values are obtained, it computes the deviation from the specified track values and takes any corrective actions that may be necessary. In this case, the sampling of the various parameters, and their processing need to be completed within a few micro seconds.

1.2.7. Internet and Multimedia Applications

Important use of real-time systems in multimedia and Internet applications include: video conferencing and multimedia multicast, Internet routers and switches.

Example 9: Video Conferencing

In a video conferencing application, video and audio signals are generated by cameras and microphones respectively. The data are sampled at a certain pre-specified frame rate. These are then compressed and sent as packets to the receiver over a network. At the receiver-end, packets are ordered, decompressed, and then played. The time constraint at the receiver-end is that the receiver must process and play the received frames at a predetermined constant rate. Thus if thirty frames are to be shown every minute, once a frame play-out is complete, the next frame must be played within two seconds.

1.2.8. Consumer Electronics

Consumer electronics area abounds numerous applications of real-time systems. A few sample applications of real-time systems in consumer electronics are: set-top boxes, audio equipment, Internet telephony, microwave ovens, intelligent washing machines, home security systems, air conditioning and refrigeration, toys, and cell phones.

Example 10: Cell Phones

Cell phones are possibly the fastest growing segment of consumer electronics. A cell phone at any point of time carries out a number of tasks simultaneously. These include: converting input voice to digital signals by deploying digital signal processing (DSP) techniques, converting electrical signals generated by the microphone to output voice signals, and sampling incoming base station signals in the control channel. A cell phone responds to the communications received from the base station within certain specified time bounds. For example, a base station might command a cell phone to switch the on-going communication to a specific frequency. The cell phone must comply with such commands from the base station within a few milliseconds.

1.2.9. Defense Applications

Typical defense applications of real-time systems include: missile guidance systems, anti-missile systems, satellite-based surveillance systems.

Example 11: Missile Guidance System

A guided missile is one that is capable of sensing the target and homes onto it. Homing becomes easy when the target emits either electrical or thermal radiation. In a missile guidance system, missile guidance is achieved by a computer mounted on the missile. The mounted computer computes the deviation from the required trajectory and effects track changes of the missile to guide it onto the target. The time constraint on the computer-based guidance system is that the sensing and the track correction tasks must be activated frequently enough to keep the missile from diverging from the target. The target sensing and track correction tasks are typically required to be completed within a few hundreds of microseconds or even lesser time depending on the speed of the missile and the type of the target.

1.2.10. Miscellaneous Applications

Besides the areas of applications already discussed, real-time systems have found numerous other applications in our every day life. An example of such an application is a railway reservation system.

Example 12: Railway Reservation System

In a railway reservation system, a central repository maintains the up-to-date data on booking status of various trains. Ticket booking counters are distributed across different geographic locations. Customers queue up at different booking counters and submit their reservation requests. After a reservation request is made at a counter, it normally takes only a few seconds for the system to confirm the reservation and print the ticket. A real-time constraint in this application is that once a request is made to the computer, it must print the ticket or display the seat unavailability message before the average human response time (about 20 seconds) expires, so that the customers do not notice any delay and get a feeling of having obtained instant results. However, as we discuss a little later (in Section 1.6), this application is an example of a category of applications that is in some aspects different from the other

discussed applications. For example, even if the results are produced just after 20 seconds, nothing untoward is going to happen - this may not be the case with the other discussed applications.

1.3. A Basic Model of a Real-Time System

We have already pointed out that this book confines itself to the software issues in real-time systems. However, in order to be able to see the software issues in a proper perspective, we need to have a basic conceptual understanding of the underlying hardware. We therefore in this section try to develop a broad understanding of high level issues of the underlying hardware in a real-time system. For a more detailed study of the underlying hardware issues, we refer the reader to [2]. Fig.28.3 shows a simple model of a real-time system in terms of its important functional blocks. Unless otherwise mentioned, all our subsequent discussions would implicitly assume such a model. Observe that in Fig. 28.3, the sensors are interfaced with the input conditioning block, which in turn is connected to the input interface. The output interface, output conditioning, and the actuator are interfaced in a complementary manner. In the following, we briefly describe the roles of the different functional blocks of a real-time system.

Sensor: A sensor converts some physical characteristic of its environment into electrical signals. An example of a sensor is a photo-voltaic cell which converts light energy into electrical energy. A wide variety of temperature and pressure sensors are also used. A temperature sensor typically operates based on the principle of a thermocouple. Temperature sensors based on many other physical principles also exist. For example, one type of temperature sensor employs the principle of variation of electrical resistance with temperature (called a *varistor*). A pressure sensor typically operates based on the *piezoelectricity* principle. Pressure sensors based on other physical principles also exist.

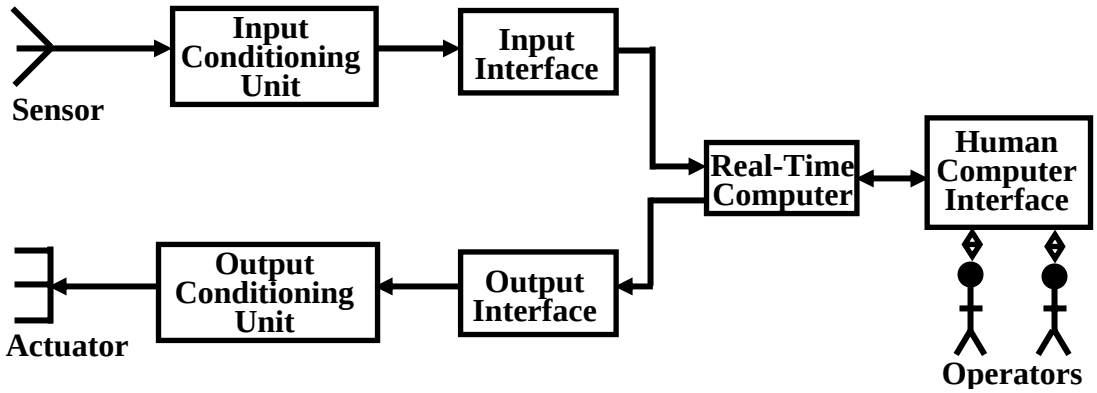


Fig. 28.3 A Model of a Real-Time System

Actuator: An actuator is any device that takes its inputs from the output interface of a computer and converts these electrical signals into some physical actions on its environment. The physical actions may be in the form of motion, change of thermal, electrical, pneumatic, or physical characteristics of some objects. A popular actuator is a motor. Heaters are also very commonly used. Besides, several hydraulic and pneumatic actuators are also popular.

Signal Conditioning Units: The electrical signals produced by a computer can rarely be used to directly drive an actuator. The computer signals usually need conditioning

before they can be used by the actuator. This is termed *output conditioning*. Similarly, *input conditioning* is required to be carried out on sensor signals before they can be accepted by the computer. For example, analog signals generated by a photo-voltaic cell are normally in the milli-volts range and need to be conditioned before they can be processed by a computer. The following are some important types of conditioning carried out on raw signals generated by sensors and digital signals generated by computers:

1. **Voltage Amplification:** Voltage amplification is normally required to be carried out to match the full scale sensor voltage output with the full scale voltage input to the interface of a computer. For example, a sensor might produce voltage in the millivolts range, whereas the input interface of a computer may require the input signal level to be of the order of a volt.
2. **Voltage Level Shifting:** Voltage level shifting is often required to align the voltage level generated by a sensor with that acceptable to the computer. For example, a sensor may produce voltage in the range -0.5 to +0.5 volt, whereas the input interface of the computer may accept voltage only in the range of 0 to 1 volt. In this case, the sensor voltage must undergo level shifting before it can be used by the computer.

3. **Frequency Range Shifting and Filtering:** Frequency range shifting is often used to reduce the noise components in a signal. Many types of noise occur in narrow bands and the signal must be shifted from the noise bands so that noise can be filtered out.

4. **Signal Mode Conversion:** A type of signal mode conversion that is frequently carried out during signal conditioning involves changing direct current into alternating current and vice-versa. Another type signal mode conversion that is frequently used is conversion of analog signals to a constant amplitude pulse train such that the pulse rate or pulse width is proportional to the voltage level. Conversion of analog signals to a pulse train is often necessary for input to systems such as transformer coupled circuits that do not pass direct current.

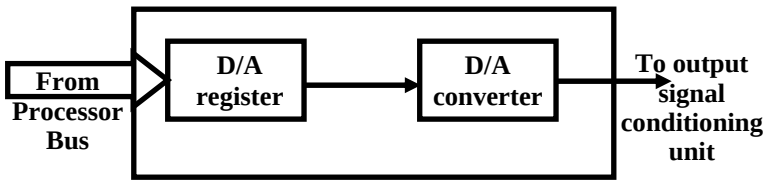


Fig. 28.4 An Output Interface

Interface Unit: Normally commands from the CPU are delivered to the actuator through an output interface. An output interface converts the stored voltage into analog form and then outputs this to the actuator circuitry. This of course would require the value generated to be written on a register (see Fig. 28.4). In an output interface, in order to produce an analog output, the CPU selects a data register of the output interface and writes the necessary data to it. The two main functional blocks of an output interface are shown in Fig. 28.4. The interface takes care of the buffering and the handshake control aspects. Analog to digital conversion is frequently deployed in an input interface. Similarly, digital to analog conversion is frequently used in an output interface.

In the following, we discuss the important steps of analog to digital signal conversion (ADC).

Analog to Digital Conversion: Digital computers can not process analog signals. Therefore, analog signals need to be converted to digital form. Analog signals can be converted to digital form using a circuitry whose block diagram is shown in Fig. 28.7. Using the block diagram shown in Fig. 28.7, analog signals are normally converted to digital form through the following two main steps:

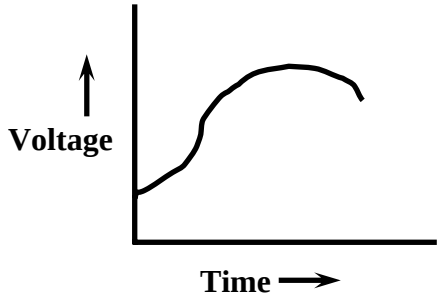


Fig. 28.5 Continuous Analog Voltage

- Sample the analog signal (shown in Fig. 28.5) at regular intervals. This sampling can be done by a capacitor circuitry that stores the voltage levels. The stored voltage levels can be made discrete. After sampling the analog signal (shown in Fig. 28.5), a step waveform as shown in Fig. 28.6 is obtained.

- Convert the stored value to a binary number by using an analog to digital converter (ADC) as shown in Fig. 28.7 and store the digital value in a register.

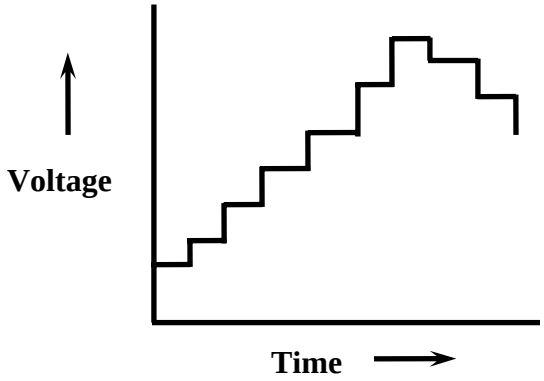


Fig. 28.6 Analog Voltage Converted to Discrete Form

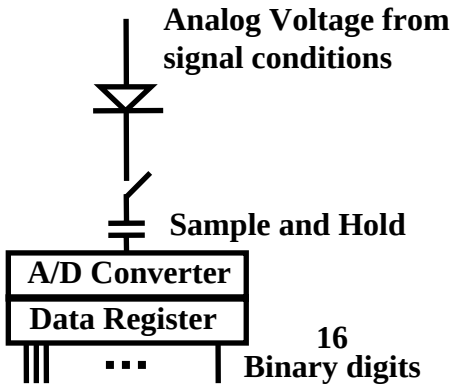


Fig. 28.7 Conversion of an Analog Signal to a 16 bit Binary Number

Digital to analog conversion can be carried out through a complementary set of operations. We leave it as an exercise to the reader to figure out the details of the circuitry that can perform the digital to analog conversion (DAC).

1.4. Characteristics of Real-Time Systems

We now discuss a few key characteristics of real-time systems. These characteristics distinguish real-time systems from non-real-time systems. However, the reader may note that all the discussed characteristics may not be applicable to every real-time system. Real-time systems cover such an enormous range of applications and products that a generalization of the characteristics into a set that is applicable to each and every system is difficult. Different categories of real-time systems may exhibit the characteristics that we identify to different extents or may not even exhibit some of the characteristics at all.

1. **Time constraints:** Every real-time task is associated with some time constraints. One form of time constraints that is very common is deadlines associated with tasks. A task deadline specifies the time before which the task must complete and produce the results. Other types of timing constraints are delay and duration (see Section 1.7). It is the responsibility of the real-time operating system (RTOS) to ensure that all tasks meet their respective time constraints. We shall examine in later chapters how an RTOS can ensure that tasks meet their respective timing constraints through appropriate task scheduling strategies.
2. **New Correctness Criterion:** The notion of correctness in real-time systems is different from that used in the context of traditional systems. In real-time systems, correctness implies not only logical correctness of the results, but the time at which the results are produced is important. A logically correct result produced after the deadline would be considered as an incorrect result.

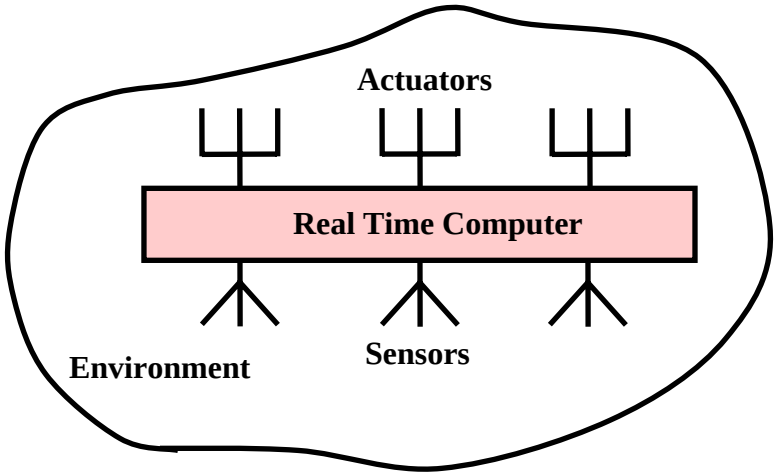


Fig. 28.8 A Schematic Representation of an Embedded Real-Time System

3. **Embedded:** A vast majority of real-time systems are embedded in nature [3]. An embedded computer system is physically "embedded" in its environment and often controls it. Fig. 28.8 shows a schematic representation of an embedded system. As shown in Fig. 28.8, the sensors of the real-time computer collect data from the environment, and pass them on to the real-time computer for processing. The computer, in turn passes information (processed data) to the actuators to carry out the necessary work on the environment, which results in controlling some characteristics of the environment. Several examples of embedded systems were discussed in Section 1.2. An example of an embedded system that we would often refer is the Multi-Point Fuel Injection (MPFI) system discussed in Example 6 of Sec. 1.2.

4. **Safety-Criticality:** For traditional non-real-time systems safety and reliability are independent issues. However, in many real-time systems these two issues are intricately bound together making them *safety-critical*. Note that a safe system is one that does not cause any damage even when it fails. A reliable system on the other hand, is one that can operate for long durations of time without exhibiting any failures. A safety-critical system is required to be highly reliable since any failure of the system can cause extensive damages. We elaborate this issue in Section 1.5.

5. **Concurrency:** A real-time system usually needs to respond to several independent events within very short and strict time bounds. For instance, consider a chemical plant automation system (see Example1 of Sec. 1.2), which monitors the progress of a chemical reaction and controls the rate of reaction by changing the different parameters of reaction such as pressure, temperature, chemical concentration. These parameters are sensed using sensors fixed in the chemical reaction chamber. These sensors may generate data asynchronously at different rates. Therefore, the real-time system must process data from all the sensors concurrently, otherwise signals may be lost and the system may malfunction. These systems can be considered to be non-deterministic, since the behavior of the system depends on the exact timing of its inputs. A non-deterministic computation is one in which two runs using the same set of input data can produce two distinct sets of output data in the two runs.

6. **Distributed and Feedback Structure:** In many real-time systems, the different components of the system are naturally distributed across widely spread geographic locations. In such

systems, the different events of interest arise at the geographically separate locations. Therefore, these events may often have to be handled locally and responses produced to them to prevent overloading of the underlying communication network. Therefore, the sensors and the actuators may be located at the places where events are generated. An example of such a system is a petroleum refinery plant distributed over a large geographic area. At each data source, it makes good design sense to locally process the data before being passed on to a central processor.

Many distributed as well as centralized real-time systems have a feedback structure as shown in Fig. 28.9. In these systems, the sensors usually sense the environment periodically. The sensed data about the environment is processed to determine the corrective actions necessary. The results of the processing are used to carry out the necessary corrective actions on the environment through the actuators, which in turn again cause a change to the required characteristics of the controlled environment, and so on.

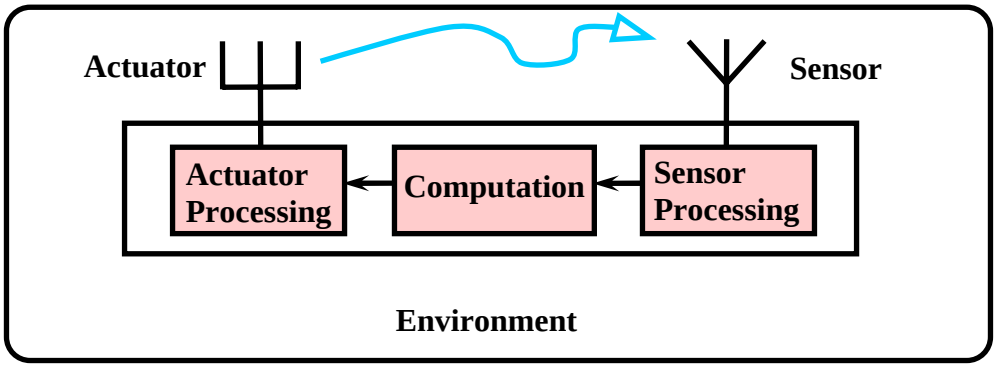


Fig. 28.9 Feedback Structure of Real-Time Systems

7. **Task Criticality:** Task criticality is a measure of the cost of failure of a task. Task criticality is determined by examining how critical are the results produced by the task to the proper functioning of the system. A real-time system may have tasks of very different criticalities. It is therefore natural to expect that the criticalities of the different tasks must be taken into consideration while designing for fault-tolerance. The higher the criticality of a task, the more reliable it should be made. Further, in the event of a failure of a highly critical task, immediate failure detection and recovery are important. However, it should be realized that task priority is a different concept and task criticality does not solely determine the task priority or the order in which various tasks are to be executed (these issues shall be elaborated in the later chapters).

8. **Custom Hardware:** A real-time system is often implemented on custom hardware that is specifically designed and developed for the purpose. For example, a cell phone does not use traditional microprocessors. Cell phones use processors which are tiny, supporting only those processing capabilities that are really necessary for cell phone operation and specifically designed to be power-efficient to conserve battery life. The capabilities of the processor used in a cell phone are substantially different from that of a general purpose processor. Another example is the embedded processor in an MPFI car. In this case, the processor used need not be a powerful general purpose processor such as a Pentium or an Athlon processor. Some of the most powerful computers used in MPFI engines are 16- or 32-bit processors running at approximately 40 MHz. However, unlike the conventional PCs, a processor used

in these car engines do not deal with processing frills such as screen-savers or a dozen of different applications running at the same time. All that the processor in an MPFI system needs to do is to compute the required fuel injection rate that is most efficient for a given speed and acceleration.

9. **Reactive:** Real-time systems are often *reactive*. A reactive system is one in which an on-going interaction between the computer and the environment is maintained. Ordinary systems compute functions on the input data to generate the output data (See Fig. 28.10 (a)). In other words, traditional systems compute the output data as some function φ of the input data. That is, output data can mathematically be expressed as: *output data* = φ(*input data*). For example, if some data I_1 is given as the input, the system computes O_1 as the result $O_1 = \varphi(I_1)$. To elaborate this concept, consider an example involving library automation software. In a library automation software, when the query book function is invoked and "Real-Time Systems" is entered as the input book name, then the software displays "Author name: R. Mall, Rack Number: 001, Number of Copies: 1".

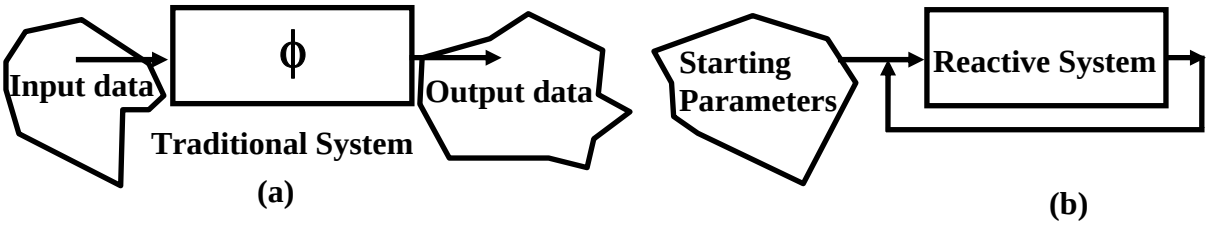


Fig. 28.10 Traditional versus Reactive Systems

In contrast to the traditional computation of the output as a simple function of the input data, real-time systems do not produce any output data but enter into an on-going interaction with their environment. In each interaction step, the results computed are used to carry out some actions on the environment. The reaction of the environment is sampled and is fed back to the system. Therefore the computations in a real-time system can be considered to be non-terminating. This reactive nature of real-time systems is schematically shown in the Fig. 28.10(b).

10. **Stability:** Under overload conditions, real-time systems need to continue to meet the deadlines of the most critical tasks, though the deadlines of non-critical tasks may not be met. This is in contrast to the requirement of *fairness* for traditional systems even under overload conditions.

11. **Exception Handling:** Many real-time systems work round-the-clock and often operate without human operators. For example, consider a small automated chemical plant that is set up to work non-stop. When there are no human operators, taking corrective actions on a failure becomes difficult. Even if no corrective actions can be immediate taken, it is desirable that a failure does not result in catastrophic situations. A failure should be detected and the system should continue to operate in a gracefully degraded mode rather than shutting off abruptly.

1.5. Safety and Reliability

In traditional systems, safety and reliability are normally considered to be independent issues. It is therefore possible to identify a traditional system that is safe and unreliable and systems that are reliable but unsafe. Consider the following two examples. Word-processing software may not be very reliable but is safe. A failure of the software does not usually cause any significant damage or financial loss. It is therefore an example of an unreliable but safe system. On the other hand, a hand gun can be unsafe but is reliable. A hand gun rarely fails. A hand gun is an unsafe system because if it fails for some reason, it can misfire or even explode and cause significant damage. It is an example of an unsafe but reliable system. These two examples show that for traditional systems, safety and reliability are independent concerns - it is therefore possible to increase the safety of a system without affecting its reliability and vice versa.

In real-time systems on the other hand, safety and reliability are coupled together. Before analyzing why safety and reliability are no longer independent issues in real-time systems, we need to first understand what exactly is meant by a fail-safe state.

> A fail-safe state of a system is one which if entered when the system fails, no damage would result.

To give an example, the fail-safe state of a word processing program is one where the document being processed has been saved onto the disk. All traditional non real-time systems do have one or more fail-safe states which help separate the issues of safety and reliability - even if a system is known to be unreliable, it can always be made to fail in a fail-safe state, and consequently it would still be considered to be a safe system.

If no damage can result if a system enters a fail-safe state just before it fails, then through careful transit to a fail-safe state upon a failure, it is possible to turn an extremely unreliable and unsafe system into a safe system. In many traditional systems this technique is in fact frequently adopted to turn an unreliable system into a safe system. For example, consider a traffic light controller that controls the flow of traffic at a road intersection. Suppose the traffic light controller fails frequently and is known to be highly unreliable. Though unreliable, it can still be considered safe if whenever a traffic light controller fails, it enters a fail-safe state where all the traffic lights are orange and blinking. This is a fail-safe state, since the motorists on seeing blinking orange traffic light become aware that the traffic light controller is not working and proceed with caution. Of course, a fail-safe state may not be to make all lights green, in which case severe accidents could occur. Similarly, all lights turned red is also not a fail-safe state - it may not cause accidents, but would bring all traffic to a stand still leading to traffic jams. However, in many real-time systems there are no fail-safe states. Therefore, any failure of the system can cause severe damages. Such systems are said to be safety-critical systems.

> A safety-critical system is one whose failure can cause severe damages.

An example of a safety-critical system is a navigation system on-board an aircraft. An on-board navigation system has no fail-safe states. When the computer on-board an aircraft fails, a fail-safe state may not be one where the engine is switched-off! In a safety-critical system, the absence of fail-safe states implies that safety can only be ensured through increased reliability. Thus, for safety-critical systems the issues of safety and reliability become interrelated - safety

can only be ensured through increased reliability. It should now be clear why safety-critical systems need to be highly reliable.

Just to give an example of the level of reliability required of safety-critical systems, consider the following. For any fly-by-wire aircraft, most of its vital parts are controlled by a computer. Any failure of the controlling computer is clearly not acceptable. The standard reliability requirement for such aircrafts is at most 1 failure per 109 flying hours (that is, a million years of continuous flying!). We examine how a highly reliable system can be developed in the next section.

1.5.1. How to Achieve High Reliability?

If you are asked by your organization to develop software which should be highly reliable, how would you proceed to achieve it? Highly reliable software can be developed by adopting all of the following three important techniques:

- **Error Avoidance:** For achieving high reliability, every possibility of occurrence of errors should be minimized during product development as much as possible. This can be achieved by adopting a variety of means: using well-founded software engineering practices, using sound design methodologies, adopting suitable CASE tools, and so on.

- **Error Detection and Removal:** In spite of using the best available error avoidance techniques, many errors still manage to creep into the code. These errors need to be detected and removed. This can be achieved to a large extent by conducting thorough reviews and testing. Once errors are detected, they can be easily fixed.

- **Fault-Tolerance:** No matter how meticulously error avoidance and error detection techniques are used, it is virtually impossible to make a practical software system entirely error-free. Few errors still persist even after carrying out thorough reviews and testing. Errors cause failures. That is, failures are manifestation of the errors latent in the system. Therefore to achieve high reliability, even in situations where errors are present, the system should be able to tolerate the faults and compute the correct results. This is called fault-tolerance. Fault-tolerance can be achieved by carefully incorporating redundancy.

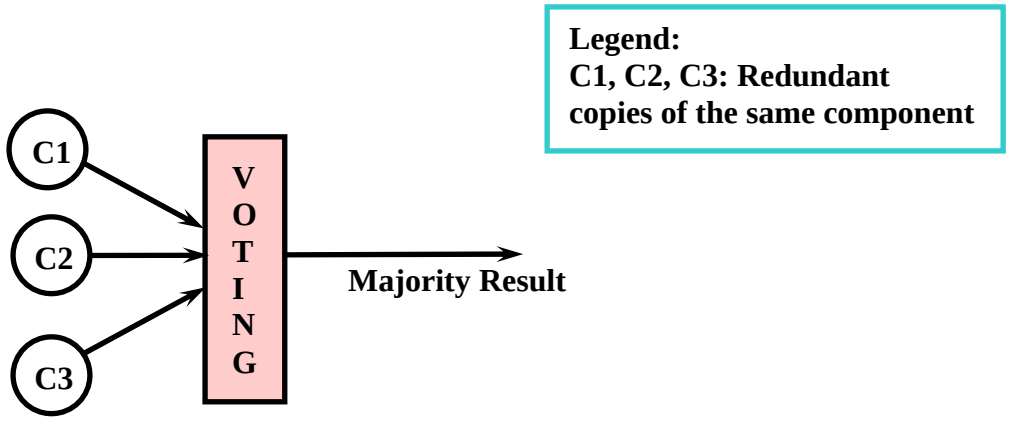


Fig. 28.11 Schematic Representation of TMR

It is relatively simple to design a hardware equipment to be fault-tolerant. The following are two methods that are popularly used to achieve hardware fault-tolerance:

- **Error Detection and Removal:** In spite of using the best available error avoidance techniques, many errors still manage to creep into the code. These errors need to be detected and removed. This can be achieved to a large extent by conducting thorough reviews and testing. Once errors are detected, they can be easily fixed.

- **Built In Self Test (BIST):** In BIST, the system periodically performs self tests of its components. Upon detection of a failure, the system automatically reconfigures itself by switching out the faulty component and switching in one of the redundant good components.

- **Triple Modular Redundancy (TMR):** In TMR, as the name suggests, three redundant copies of all critical components are made to run concurrently (see Fig. 28.11). Observe that in Fig. 28.11, C1, C2, and C3 are the redundant copies of the same critical component. The system performs voting of the results produced by the redundant components to select the majority result. TMR can help tolerate occurrence of only a single failure at any time. (Can you answer why a TMR scheme can effectively tolerate a single component failure only?). An assumption that is implicit in the TMR technique is that at any time only one of the three redundant components can produce erroneous results. The majority result after voting would be erroneous if two or more components can fail simultaneously (more precisely, before a repair can be carried out). In situations where two or more components are likely to fail (or produce erroneous results), then greater amounts of redundancies would be required to be incorporated. A little thinking can show that at least 2n+1 redundant components are required to tolerate simultaneous failures of n component.

As compared to hardware, software fault-tolerance is much harder to achieve. To investigate the reason behind this, let us first discuss the techniques currently being used to achieve software fault-tolerance. We do this in the following subsection.

1.6. Software Fault-Tolerance Techniques

Two methods are now popularly being used to achieve software fault-tolerance: N-version programming and recovery block techniques. These two techniques are simple adaptations of the basic techniques used to provide hardware fault-tolerance. We discuss these two techniques in the following.

N-Version Programming: This technique is an adaptation of the TMR technique for hardware fault-tolerance. In the N-version programming technique, independent teams develop N different versions (value of N depends on the degree of fault-tolerance required) of a software component (module). The redundant modules are run concurrently (possibly on redundant hardware). The results produced by the different versions of the module are subjected to voting at run time and the result on which majority of the components agree is accepted. The central idea behind this scheme is that independent teams would commit different types of mistakes, which would be eliminated when the results produced by them are subjected to voting. However, this scheme is not very successful in achieving fault-tolerance, and the problem can be attributed

to statistical correlation of failures. Statistical correlation of failures means that even though individual teams worked in isolation to develop the different versions of a software component, still the different versions fail for identical reasons. In other words, the different versions of a component show similar failure patterns. This does not mean that the different modules developed by independent programmers, after all, contain identical errors. The reason for this is not far to seek, programmers commit errors in those parts of a problem which they perceive to be difficult - and what is difficult to one team is usually difficult to all teams. So, identical errors remain in the most complex and least understood parts of a software component.

Recovery Blocks: In the recovery block scheme, the redundant components are called *try blocks*. Each try block computes the same end result as the others but is intentionally written using a different algorithm compared to the other try blocks. In N-version programming, the different versions of a component are written by different teams of programmers, whereas in recovery block different algorithms are used in different try blocks. Also, in contrast to the N-version programming approach where the redundant copies are run concurrently, in the recovery block approach they are (as shown in Fig. 28.12) run one after another. The results produced by a try block are subjected to an acceptance test (see Fig. 28.12). If the test fails, then the next try block is tried. This is repeated in a sequence until the result produced by a try block successfully passes the acceptance test. Note that in Fig. 28.12 we have shown acceptance tests separately for different try blocks to help understand that the tests are applied to the try blocks one after the other, though it may be the case that the same test is applied to each try block.

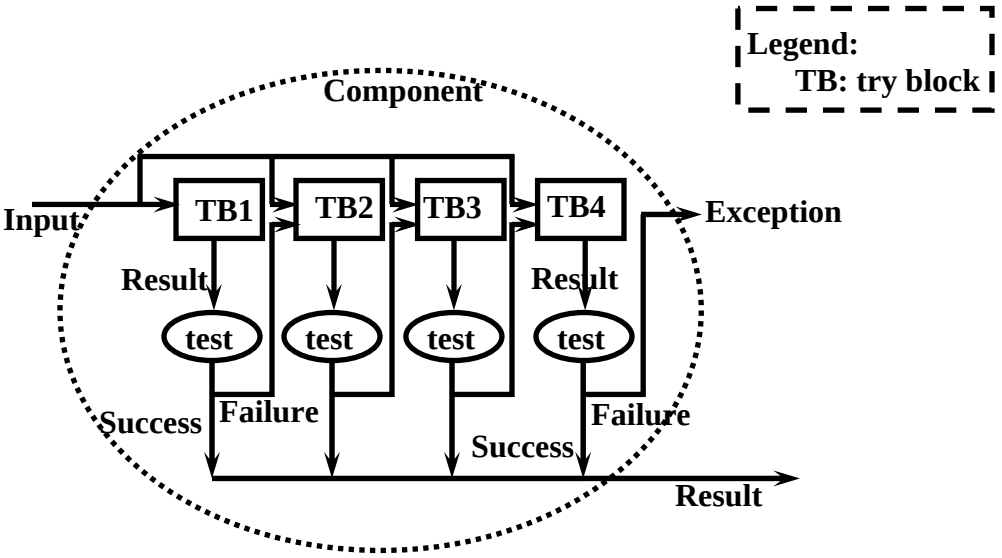


Fig. 28.12 A Software Fault-Tolerance Scheme Using Recovery Blocks

As was the case with N-version programming, the recovery blocks approach also does not achieve much success in providing effective fault-tolerance. The reason behind this is again statistical correlation of failures. Different try blocks fail for identical reasons as was explained in case of N-version programming approach. Besides, this approach suffers from a further limitation that it can only be used if the task deadlines are much larger than the task computation times (i.e. tasks have large laxity), since the different try blocks are put to execution one after the other when failures occur. The recovery block approach poses special difficulty when used with real-time tasks with very short slack time (i.e. short deadline and considerable execution time),

as the try blocks are tried out one after the other deadlines may be missed. Therefore, in such cases the later try-blocks usually contain only skeletal code.

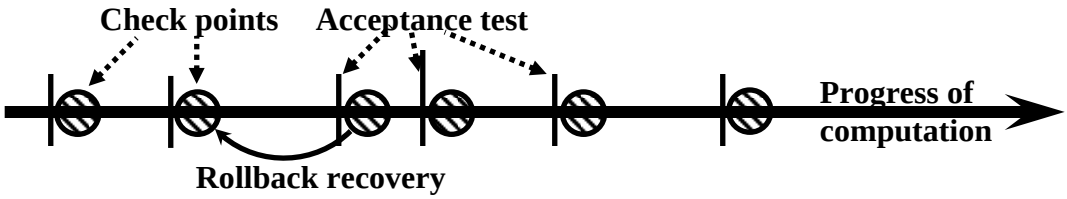


Fig. 28.13 Checkpointing and Rollback Recovery

Of course, it is possible that the later try blocks contain only skeletal code, produce only approximate results and therefore take much less time for computation than the first try block.

Checkpointing and Rollback Recovery: Checkpointing and roll-back recovery is another popular technique to achieve fault-tolerance. In this technique as the computation proceeds, the system state is tested each time after some meaningful progress in computation is made. Immediately after a state-check test succeeds, the state of the system is backed up on a stable storage (see Fig. 28.13). In case the next test does not succeed, the system can be made to roll-back to the last checkpointed state. After a rollback, from a checkpointed state a fresh computation can be initiated. This technique is especially useful, if there is a chance that the system state may be corrupted as the computation proceeds, such as data corruption or processor failure.

1.7. Types of Real-Time Tasks

We have already seen that a real-time task is one for which quantitative expressions of time are needed to describe its behavior. This quantitative expression of time usually appears in the form of a constraint on the time at which the task produces results. The most frequently occurring timing constraint is a deadline constraint which is used to express that a task is required to compute its results within some deadline. We therefore implicitly assume only deadline type of timing constraints on tasks in this section, though other types of constraints (as explained in **Sec**. ….) may occur in practice. Real-time tasks can be classified into the following three broad categories:

A real-time task can be classified into either hard, soft, or firm real-time task depending on the consequences of a task missing its deadline.

It is not necessary that all tasks of a real-time application belong to the same category. It is possible that different tasks of a real-time system can belong to different categories. We now elaborate these three types of real-time tasks.

1.7.1. Hard Real-Time Tasks

A hard real-time task is one that is constrained to produce its results within certain predefined time bounds. The system is considered to have failed whenever any of its hard real-time tasks does not produce its required results before the specified time bound.

An example of a system having hard real-time tasks is a robot. The robot cyclically carries out a number of activities including communication with the host system, logging all completed activities, sensing the environment to detect any obstacles present, tracking the objects of interest, path planning, effecting next move, etc. Now consider that the robot suddenly encounters an obstacle. The robot must detect it and as soon as possible try to escape colliding with it. If it fails to respond to it quickly (i.e. the concerned tasks are not completed before the required time bound) then it would collide with the obstacle and the robot would be considered to have failed. Therefore detecting obstacles and reacting to it are hard real-time tasks.

Another application having hard real-time tasks is an anti-missile system. An anti-missile system consists of the following critical activities (tasks). An anti-missile system must first detect all incoming missiles, properly position the anti-missile gun, and then fire to destroy the incoming missile before the incoming missile can do any damage. All these tasks are hard real-time in nature and the anti-missile system would be considered to have failed, if any of its tasks fails to complete before the corresponding deadlines.

Applications having hard real-time tasks are typically safety-critical (Can you think an example of a hard real-time system that is not safety-critical?[1]) This means that any failure of a real-time task, including its failure to meet the associated deadlines, would result in severe consequences. This makes hard real-time tasks extremely critical. Criticality of a task can range from extremely critical to not so critical. Task criticality therefore is a different dimension than hard or soft characterization of a task. Criticality of a task is a measure of the cost of a failure - the higher the cost of failure, the more critical is the task.

For hard real-time tasks in practical systems, the time bounds usually range from several micro seconds to a few milli seconds. It may be noted that a hard real-time task does not need to be completed within the shortest time possible, but it is merely required that the task must complete within the specified time bound. In other words, there is no reward in completing a hard real-time task much ahead of its deadline. This is an important observation and this would take a central part in our discussions on task scheduling in the next two chapters.

1.7.2. Firm Real-Time Tasks

Every firm real-time task is associated with some predefined deadline before which it is required to produce its results. However, unlike a hard real-time task, even when a firm real-time task does not complete within its deadline, the system does not fail. The late results are merely discarded. In other words, the utility of the results computed by a firm real-time task becomes zero after the deadline. Fig. 28.14 schematically shows the utility of the results produced by a firm real-time task as a function of time. In Fig. 28.14 it can be seen that if the response time of a task exceeds the specified deadline, then the utility of the results becomes zero and the results are discarded.

[1] Some computer games have hard real-time tasks; these are not safety-critical though. Whenever a timing constraint is not met, the game may fail, but the failure may at best be a mild irritant to the user.

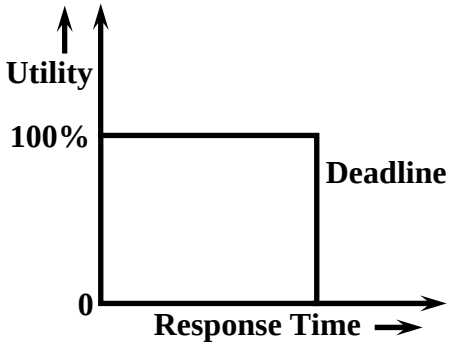


Fig. 28.14 Utility of Result of a Firm Real-Time Task with Time

Firm real-time tasks typically abound in multimedia applications. The following are two examples of firm real- time tasks:

- **Video conferencing:** In a video conferencing application, video frames and the accompanying audio are converted into packets and transmitted to the receiver over a network. However, some frames may get delayed at different nodes during transit on a packet-switched network due to congestion at different nodes. This may result in varying queuing delays experienced by packets traveling along different routes. Even when packets traverse the same route, some packets can take much more time than the other packets due to the specific transmission strategy used at the nodes. When a certain frame is being played, if some preceding frame arrives at the receiver, then this frame is of no use and is discarded. Due to this reason, when a frame is delayed by more than say one second, it is simply discarded at the receiver-end without carrying out any processing on it.

- **Satellite-based tracking of enemy movements:** Consider a satellite that takes pictures of an enemy territory and beams it to a ground station computer frame by frame. The ground computer processes each frame to find the positional difference of different objects of interest with respect to their position in the previous frame to determine the movements of the enemy. When the ground computer is overloaded, a new image may be received even before an older image is taken up for processing. In this case, the older image is of not much use. Hence the older images may be discarded and the recently received image could be processed.

For firm real-time tasks, the associated time bounds typically range from a few milli seconds to several hundreds of milli seconds.

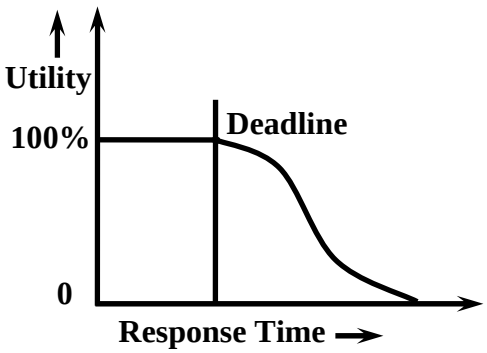


Fig. 28.15 Utility of the Results Produced by a Soft Real-Time Task as a Function of Time

1.7.3. Soft Real-Time Tasks

Soft real-time tasks also have time bounds associated with them. However, unlike hard and firm real-time tasks, the timing constraints on soft real-time tasks are not expressed as absolute values. Instead, the constraints are expressed either in terms of the average response times required.

An example of a soft real-time task is web browsing. Normally, after an URL (Uniform Resource Locater) is clicked, the corresponding web page is fetched and displayed within a couple of seconds on the average. However, when it takes several minutes to display a requested page, we still do not consider the system to have failed, but merely express that the performance of the system has degraded.

Another example of a soft real-time task is a task handling a request for a seat reservation in a railway reservation application. Once a request for reservation is made, the response should occur within 20 seconds on the average. The response may either be in the form of a printed ticket or an apology message on account of unavailability of seats. Alternatively, we might state the constraint on the ticketing task as: At least in case of 95% of reservation requests, the ticket should be processed and printed in less than 20 seconds.

Let us now analyze the impact of the failure of a soft real-time task to meet its deadline, by taking the example of the railway reservation task. If the ticket is printed in about 20 seconds, we feel that the system is working fine and get a feel of having obtained instant results. As already stated, missed deadlines of soft real-time tasks do not result in system failures. However, the utility of the results produced by a soft real-time task falls continuously with time after the expiry of the deadline as shown in Fig. 28.15. In Fig. 28.15, the utility of the results produced are 100% if produced before the deadline, and after the deadline is passed the utility of the results slowly falls off with time. For soft real-time tasks that typically occur in practical applications, the time bounds usually range from a fraction of a second to a few seconds.

1.7.4. Non-Real-Time Tasks

A non-real-time task is not associated with any time bounds. Can you think of any example of a non-real-time task? Most of the interactive computations you perform nowadays are handled by soft real-time tasks. However, about two or three decades back, when computers were not interactive almost all tasks were non-real-time. A few examples of non-real-time tasks are: batch processing jobs, e-mail, and back ground tasks such as event loggers. You may however argue that even these tasks, in the strict sense of the term, do have certain time bounds. For example, an e-mail is expected to reach its destination at least within a couple of hours of being sent. Similar is the case with a batch processing job such as pay-slip printing. What then really is the difference between a non-real-time task and a soft real-time task? For non-real-time tasks, the associated time bounds are typically of the order of a few minutes, hours or even days. In contrast, the time bounds associated with soft real-time tasks are at most of the order of a few seconds.

1.8. Exercises

1. State whether you consider the following statements to be TRUE or FALSE. Justify your answer in each case.
   a. A hard real-time application is made up of only hard real-time tasks.
   b. Every safety-critical real-time system has a fail-safe state.
   c. A deadline constraint between two stimuli can be considered to be a behavioral constraint on the environment of the system.
   d. Hardware fault-tolerance techniques can easily be adapted to provide software fault-tolerance.
   e. A good algorithm for scheduling hard real-time tasks must try to complete each task in the shortest time possible.
   f. All hard real-time systems are safety-critical in nature.
   g. Performance constraints on a real-time system ensure that the environment of the system is well-behaved.
   h. Soft real-time tasks are those which do not have any time bounds associated with them.
   i. Minimization of average task response times is the objective of any good hard real-time task-scheduling algorithm.
   j. It should be the goal of any good real-time operating system to complete every hard real-time task as ahead of its deadline as possible.
2. What do you understand by the term "real-time"? How is the concept of real-time different from the traditional notion of time? Explain your answer using a suitable example.
3. Using a block diagram show the important hardware components of a real-time system and their interactions. Explain the roles of the different components.
4. In a real-time system, raw sensor signals need to be preprocessed before they can be used by a computer. Why is it necessary to preprocess the raw sensor signals before they can be used by a computer? Explain the different types of preprocessing that are normally carried out on sensor signals to make them suitable to be used directly by a computer.
5. Identify the key differences between hard real-time, soft real-time, and firm real-time systems. Give at least one example of real-time tasks corresponding to these three categories. Identify the timing constraints in your tasks and justify why the tasks should be categorized into the categories you have indicated.
6. Give an example of a soft real-time task and a non-real-time task. Explain the key difference between the characteristics of these two types of tasks.
7. Draw a schematic model showing the important components of a typical hard real-time system. Explain the working of the input interface using a suitable schematic diagram. Explain using a suitable circuit diagram how analog to digital (ADC) conversion is achieved in an input interface.
8. Explain the check pointing and rollback recovery scheme to provide fault-tolerant real-time computing. Explain the types of faults it can help tolerate and the faults it can not tolerate. Explain the situations in which this technique is useful.
9. Answer the following questions concerning fault-tolerance of real-time systems.
   a. Explain why hardware fault-tolerance is easier to achieve compared to software fault-tolerance.
   b. Explain the main techniques available to achieve hardware fault-tolerance.

c. What are the main techniques available to achieve software fault-tolerance? What are the shortcomings of these techniques?

10. What do you understand by the "fail-safe" state of a system? Safety-critical real-time systems do not have a fail-safe state. What is the implication of this?
11. Is it possible to have an extremely safe but unreliable system? If your answer is affirmative, then give an example of such a system. If you answer in the negative, then justify why it is not possible for such a system to exist.
12. What is a safety-critical system? Give a few practical examples safety-critical hard real-time systems. Are all hard real-time systems safety-critical? If not, give at least one example of a hard real-time system that is not safety-critical.
13. Explain with the help of a schematic diagram how the recovery block scheme can be used to achieve fault- tolerance of real-time tasks. What are the shortcomings of this scheme? Explain situations where it can be satisfactorily be used and situations where it can not be used.
14. Identify and represent the timing constraints in the following air-defense system by means of an extended state machine diagram. Classify each constraint into either performance or behavioral constraint.
15. Every incoming missile must be detected within 0.2 seconds of its entering the radar coverage area. The intercept missile should be engaged within 5 seconds of detection of the target missile. The intercept missile should be fired after 0.1 seconds of its engagement but no later than 1 sec.
16. Represent a washing machine having the following specification by means of an extended state machine diagram. The wash-machine waits for the start switch to be pressed. After the user presses the start switch, the machine fills the wash tub with either hot or cold water depending upon the setting of the Hot Wash switch. The water filling continues until the high level is sensed. The machine starts the agitation motor and continues agitating the wash tub until either the preset timer expires or the user presses the stop switch. After the agitation stops, the machine waits for the user to press the start Drying switch. After the user presses the start Drying switch, the machine starts the hot air blower and continues blowing hot air into the drying chamber until either the user presses the stop switch or the preset timer expires.
17. Represent the timing constraints in a collision avoidance task in an air surveillance system as an extended finite state machine (EFSM) diagram. The collision avoidance task consists of the following activities.
    a. The first subtask named radar signal processor processes the radar signal on a signal processor to generate the track record in terms of the target's location and velocity within 100 mSec of receipt of the signal.
    b. The track record is transmitted to the data processor within 1 mSec after the track record is determined.
    c. A subtask on the data processor correlates the received track record with the track records of other targets that come close to detect potential collision that might occur within the next 500 mSec.
    d. If a collision is anticipated, then the corrective action is determined within 10 mSec by another subtask running on the data processor.
    e. The corrective action is transmitted to the track correction task within 25 mSec.
18. Consider the following (partial) specification of a real-time system:
    The velocity of a space-craft must be sampled by a computer on-board the space-craft at least once every second (the sampling event is denoted by S). After sampling the velocity, the current position is computed (denoted by event C) within 100msec. Concurrently, the

expected position of the space-craft is retrieved from the database within 200msec (denoted by event R). Using these data, the deviation from the normal course of the space-craft must be determined within 100 msec (denoted by event D) and corrective velocity adjustments must be carried out before a new velocity value is sampled in (the velocity adjustment event is denoted by A). Calculated positions must be transmitted to the earth station at least once every minute (position transmission event is denoted by the event T).
Identify the different timing constraints in the system. Classify these into either performance or behavioral constraints. Construct an EFSM to model the system.

19. Construct the EFSM model of a telephone system whose (partial) behavior is described below:
After lifting the receiver handset, the dial tone should appear within 20 seconds. If a dial tone can not be given within 20 seconds, then an idle tone is produced. After the dial tone appears, the first digit should to be dialed within 10 seconds and the subsequent five digits within 5 seconds of each other. If the dialing of any of the digits is delayed, then an idle tone is produced. The idle tone continues until the receiver handset is replaced.

Module 6

Embedded System Software

Lesson 29

Real-Time Task Scheduling – Part 1

Specific Instructional Objectives

At the end of this lesson, the student would be able to:

- Understand the basic terminologies associated with Real-Time task scheduling
- Classify the Real-Time tasks with respect to their recurrence
- Get an overview of the different types of schedulers
- Get an overview of the various ways of classifying scheduling algorithms
- Understand the logic of clock-driven scheduling
- Get an overview of table-driven schedulers
- Get an overview of cyclic schedulers
- Work out problems related to table-driven and cyclic schedulers
- Understand how a generalized task scheduler would be
- Compare table-driven and cyclic schedulers

1. Real-Time Task Scheduling

In the last Chapter we defined a real-time task as one that has some constraints associated with it. Out of the three broad classes of time constraints we discussed, deadline constraint on tasks is the most common. In all subsequent discussions we therefore implicitly assume only deadline constraints on real-time tasks, unless we mention otherwise.

Real-time tasks get generated in response to some events that may either be external or internal to the system. For example, a task might get generated due to an internal event such as a clock interrupt occurring every few milliseconds to periodically poll the temperature of a chemical plant. Another task might get generated due to an external event such as the user pressing a switch. When a task gets generated, it is said to have arrived or got released. Every real-time system usually consists of a number of real-time tasks. The time bounds on different tasks may be different. We had already pointed out that the consequences of a task missing its time bounds may also vary from task to task. This is often expressed as the criticality of a task.

In the last Chapter, we had pointed out that appropriate scheduling of tasks is the basic mechanism adopted by a real-time operating system to meet the time constraints of a task. Therefore, selection of an appropriate task scheduling algorithm is central to the proper functioning of a real-time system. In this Chapter we discuss some fundamental task scheduling techniques that are available. An understanding of these techniques would help us not only to satisfactorily design a real-time application, but also understand and appreciate the features of modern commercial real-time operating systems discussed in later chapters.

This chapter is organized as follows. We first introduce some basic concepts and terminologies associated with task scheduling. Subsequently, we discuss two major classes of task schedulers: clock-driven and event-driven. Finally, we explain some important issues that must be considered while developing practical applications.

1.1. Basic Terminologies

In this section we introduce a few important concepts and terminologies which would be useful in understanding the rest of this Chapter.

Task Instance: Each time an event occurs, it triggers the task that handles this event to run. In other words, a task is generated when some specific event occurs. Real-time tasks therefore normally recur a large number of times at different instants of time depending on the event occurrence times. It is possible that real-time tasks recur at random instants. However, most real-time tasks recur with certain fixed periods. For example, a temperature sensing task in a chemical plant might recur indefinitely with a certain period because the temperature is sampled periodically, whereas a task handling a device interrupt might recur at random instants. Each time a task recurs, it is called an instance of the task. The first time a task occurs, it is called the first instance of the task. The next occurrence of the task is called its second instance, and so on. The jth instance of a task Ti would be denoted as Ti(j). Each instance of a real-time task is associated with a deadline by which it needs to complete and produce results. We shall at times refer to task instances as processes and use these two terms interchangeably when no confusion arises.

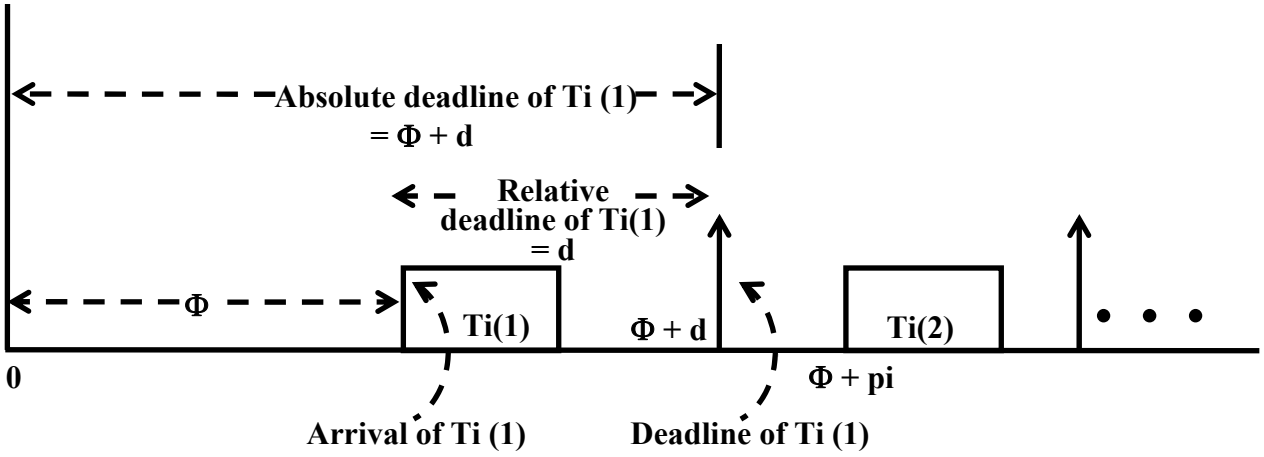


Fig. 29.1 Relative and Absolute Deadlines of a Task

Relative Deadline versus Absolute Deadline: The absolute deadline of a task is the absolute time value (counted from time 0) by which the results from the task are expected. Thus, absolute deadline is equal to the interval of time between the time 0 and the actual instant at which the deadline occurs as measured by some physical clock. Whereas, relative deadline is the time interval between the start of the task and the instant at which deadline occurs. In other words, relative deadline is the time interval between the arrival of a task and the corresponding deadline. The difference between relative and absolute deadlines is illustrated in Fig. 29.1. It can be observed from Fig. 29.1 that the relative deadline of the task Ti(1) is d, whereas its absolute deadline is $\phi + d$.

Response Time: The response time of a task is the time it takes (as measured from the task arrival time) for the task to produce its results. As already remarked, task instances get generated

due to occurrence of events. These events may be internal to the system, such as clock interrupts, or external to the system such as a robot encountering an obstacle.

The response time is the time duration from the occurrence of the event generating the task to the time the task produces its results.

For hard real-time tasks, as long as all their deadlines are met, there is no special advantage of completing the tasks early. However, for soft real-time tasks, average response time of tasks is an important metric to measure the performance of a scheduler. A scheduler for soft real-time tasks should try to execute the tasks in an order that minimizes the average response time of tasks.

Task Precedence: A task is said to precede another task, if the first task must complete before the second task can start. When a task Ti precedes another task Tj, then each instance of Ti precedes the corresponding instance of Tj. That is, if T1 precedes T2, then T1(1) precedes T2(1), T1(2) precedes T2(2), and so on. A precedence order defines a partial order among tasks. Recollect from a first course on discrete mathematics that a partial order relation is reflexive, antisymmetric, and transitive. An example partial ordering among tasks is shown in Fig. 29.2. Here T1 precedes T2, but we cannot relate T1 with either T3 or T4. We shall later use task precedence relation to develop appropriate task scheduling algorithms.

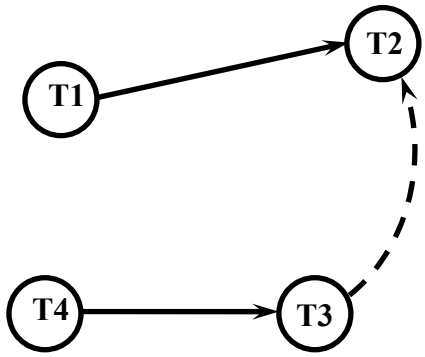


Fig. 29.2 Precedence Relation among Tasks

Data Sharing: Tasks often need to share their results among each other when one task needs to share the results produced by another task; clearly, the second task must precede the first task. In fact, precedence relation between two tasks sometimes implies data sharing between the two tasks (e.g. first task passing some results to the second task). However, this is not always true. A task may be required to precede another even when there is no data sharing. For example, in a chemical plant it may be required that the reaction chamber must be filled with water before chemicals are introduced. In this case, the task handling filling up the reaction chamber with water must complete, before the task handling introduction of the chemicals is activated. It is therefore not appropriate to represent data sharing using precedence relation. Further, data sharing may occur not only when one task precedes the other, but might occur among truly concurrent tasks, and overlapping tasks. In other words, data sharing among tasks does not necessarily impose any particular ordering among tasks. Therefore, data sharing relation among tasks needs to be represented using a different symbol. We shall represent data sharing among two tasks using a dashed arrow. In the example of data sharing among tasks represented in Fig. 29.2, T2 uses the results of T3, but T2 and T3 may execute concurrently. T2 may even

start executing first, after sometimes it may receive some data from T3, and continue its execution, and so on.

1.2. Types of Real-Time Tasks

Based on the way real-time tasks recur over a period of time, it is possible to classify them into three main categories: periodic, sporadic, and aperiodic tasks. In the following, we discuss the important characteristics of these three major categories of real-time tasks.

Periodic Task: A periodic task is one that repeats after a certain fixed time interval. The precise time instants at which periodic tasks recur are usually demarcated by clock interrupts. For this reason, periodic tasks are sometimes referred to as clock-driven tasks. The fixed time interval after which a task repeats is called the period of the task. If Ti is a periodic task, then the time from 0 till the occurrence of the first instance of Ti (i.e. Ti(1)) is denoted by ϕ_i, and is called the phase of the task. The second instance (i.e. Ti(2)) occurs at $\phi_i + p_i$. The third instance (i.e. $T_i(3)$) occurs at $\phi_i + 2 * p_i$ and so on. Formally, a periodic task Ti can be represented by a 4 tuple (ϕ_i, p_i, e_i, d_i) where p_i is the period of task, e_i is the worst case execution time of the task, and d_i is the relative deadline of the task. We shall use this notation extensively in future discussions.

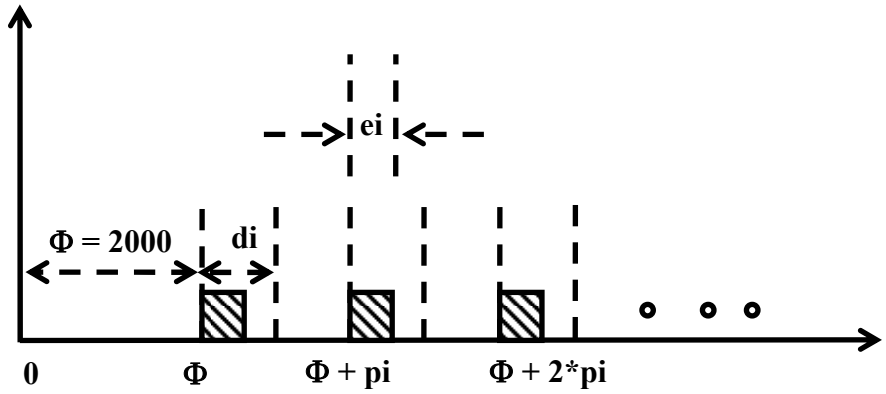


Fig. 29.3 Track Correction Task (2000mSec; p_i; e_i; d_i) of a Rocket

To illustrate the above notation to represent real-time periodic tasks, let us consider the track correction task typically found in a rocket control software. Assume the following characteristics of the track correction task. The track correction task starts 2000 milliseconds after the launch of the rocket, and recurs periodically every 50 milliseconds then on. Each instance of the task requires a processing time of 8 milliseconds and its relative deadline is 50 milliseconds. Recall that the phase of a task is defined by the occurrence time of the first instance of the task. Therefore, the phase of this task is 2000 milliseconds. This task can formally be represented as (2000 mSec, 50 mSec, 8 mSec, 50 mSec). This task is pictorially shown in Fig. 29.3. When the deadline of a task equals its period (i.e. pi=di), we can omit the fourth tuple. In this case, we can represent the task as Ti= (2000 mSec, 50 mSec, 8 mSec). This would automatically mean pi=di=50 mSec. Similarly, when $\phi_i = 0$, it can be omitted when no confusion arises. So, Ti = (20mSec; 100mSec) would indicate a task with $\phi_i = 0$, pi=100mSec, ei=20mSec, and di=100mSec. Whenever there is any scope for confusion, we shall explicitly write out the parameters Ti = (pi=50 mSecs, ei = 8 mSecs, di = 40 mSecs), etc.

A vast majority of the tasks present in a typical real-time system are periodic. The reason for this is that many activities carried out by real-time systems are periodic in nature, for example monitoring certain conditions, polling information from sensors at regular intervals to carry out certain action at regular intervals (such as drive some actuators). We shall consider examples of such tasks found in a typical chemical plant. In a chemical plant several temperature monitors, pressure monitors, and chemical concentration monitors periodically sample the current temperature, pressure, and chemical concentration values which are then communicated to the plant controller. The instances of the temperature, pressure, and chemical concentration monitoring tasks normally get generated through the interrupts received from a periodic timer. These inputs are used to compute corrective actions required to maintain the chemical reaction at a certain rate. The corrective actions are then carried out through actuators.

Sporadic Task: A sporadic task is one that recurs at random instants. A sporadic task Ti can be is represented by a three tuple:

$$T_i = (e_i, g_i, d_i)$$

where e_i is the worst case execution time of an instance of the task, g_i denotes the minimum separation between two consecutive instances of the task, d_i is the relative deadline. The minimum separation (g_i) between two consecutive instances of the task implies that once an instance of a sporadic task occurs, the next instance cannot occur before g_i time units have elapsed. That is, g_i restricts the rate at which sporadic tasks can arise. As done for periodic tasks, we shall use the convention that the first instance of a sporadic task T_i is denoted by $T_i(1)$ and the successive instances by $T_i(2)$, $T_i(3)$, etc.

Many sporadic tasks such as emergency message arrivals are highly critical in nature. For example, in a robot a task that gets generated to handle an obstacle that suddenly appears is a sporadic task. In a factory, the task that handles fire conditions is a sporadic task. The time of occurrence of these tasks can not be predicted.

The criticality of sporadic tasks varies from highly critical to moderately critical. For example, an I/O device interrupt, or a DMA interrupt is moderately critical. However, a task handling the reporting of fire conditions is highly critical.

Aperiodic Task: An aperiodic task is in many ways similar to a sporadic task. An aperiodic task can arise at random instants. However, in case of an aperiodic task, the minimum separation gi between two consecutive instances can be 0. That is, two or more instances of an aperiodic task might occur at the same time instant. Also, the deadline for an aperiodic tasks is expressed as either an average value or is expressed statistically. Aperiodic tasks are generally soft real-time tasks.

It is easy to realize why aperiodic tasks need to be soft real-time tasks. Aperiodic tasks can recur in quick succession. It therefore becomes very difficult to meet the deadlines of all instances of an aperiodic task. When several aperiodic tasks recur in a quick succession, there is a bunching of the task instances and it might lead to a few deadline misses. As already discussed, soft real-time tasks can tolerate a few deadline misses. An example of an aperiodic task is a logging task in a distributed system. The logging task can be started by different tasks running on different nodes. The logging requests from different tasks may arrive at the logger almost at the same time, or the requests may be spaced out in time. Other examples of aperiodic tasks include operator requests, keyboard presses, mouse movements, etc. In fact, all interactive commands issued by users are handled by aperiodic tasks.

1.3. Task Scheduling

Real-time task scheduling essentially refers to determining the order in which the various tasks are to be taken up for execution by the operating system. Every operating system relies on one or more task schedulers to prepare the schedule of execution of various tasks it needs to run. Each task scheduler is characterized by the scheduling algorithm it employs. A large number of algorithms for scheduling real-time tasks have so far been developed. Real-time task scheduling on uniprocessors is a mature discipline now with most of the important results having been worked out in the early 1970's. The research results available at present in the literature are very extensive and it would indeed be grueling to study them exhaustively. In this text, we therefore classify the available scheduling algorithms into a few broad classes and study the characteristics of a few important ones in each class.

1.3.1. A Few Basic Concepts

Before focusing on the different classes of schedulers more closely, let us first introduce a few important concepts and terminologies which would be used in our later discussions.

Valid Schedule: A valid schedule for a set of tasks is one where at most one task is assigned to a processor at a time, no task is scheduled before its arrival time, and the precedence and resource constraints of all tasks are satisfied.

Feasible Schedule: A valid schedule is called a feasible schedule, only if all tasks meet their respective time constraints in the schedule.

Proficient Scheduler: A task scheduler sch1 is said to be more proficient than another scheduler sch2, if sch1 can feasibly schedule all task sets that sch2 can feasibly schedule, but not vice versa. That is, sch1 can feasibly schedule all task sets that sch2 can, but there exists at least one task set that sch2 can not feasibly schedule, whereas sch1 can. If sch1 can feasibly schedule all task sets that sch2 can feasibly schedule and vice versa, then sch1 and sch2 are called equally proficient schedulers.

Optimal Scheduler: A real-time task scheduler is called optimal, if it can feasibly schedule any task set that can be feasibly scheduled by any other scheduler. In other words, it would not be possible to find a more proficient scheduling algorithm than an optimal scheduler. If an optimal scheduler can not schedule some task set, then no other scheduler should be able to produce a feasible schedule for that task set.

Scheduling Points: The scheduling points of a scheduler are the points on time line at which the scheduler makes decisions regarding which task is to be run next. It is important to note that a task scheduler does not need to run continuously, it is activated by the operating system only at the scheduling points to make the scheduling decision as to which task to be run next. In a clock-driven scheduler, the scheduling points are defined at the time instants marked by interrupts generated by a periodic timer. The scheduling points in an event-driven scheduler are determined by occurrence of certain events.

Preemptive Scheduler: A preemptive scheduler is one which when a higher priority task arrives, suspends any lower priority task that may be executing and takes up the higher priority task for execution. Thus, in a preemptive scheduler, it can not be the case that a higher priority task is ready and waiting for execution, and the lower priority task is executing. A preempted lower priority task can resume its execution only when no higher priority task is ready.

Utilization: The processor utilization (or simply utilization) of a task is the average time for which it executes per unit time interval. In notations: for a periodic task T_i, the utilization $u_i = e_i/p_i$, where e_i is the execution time and p_i is the period of T_i. For a set of periodic tasks {Ti}: the total utilization due to all tasks $U = \sum_{i=1}^{n} e_i/p_i$. It is the objective of any good scheduling algorithm to feasibly schedule even those task sets that have very high utilization, i.e. utilization approaching 1. Of course, on a uniprocessor it is not possible to schedule task sets having utilization more than 1.

Jitter: Jitter is the deviation of a periodic task from its strict periodic behavior. The arrival time jitter is the deviation of the task from arriving at the precise periodic time of arrival. It may be caused by imprecise clocks, or other factors such as network congestions. Similarly, completion time jitter is the deviation of the completion of a task from precise periodic points. The completion time jitter may be caused by the specific scheduling algorithm employed which takes up a task for scheduling as per convenience and the load at an instant, rather than scheduling at some strict time instants. Jitters are undesirable for some applications.

1.4. Classification of Real-Time Task Scheduling Algorithms

Several schemes of classification of real-time task scheduling algorithms exist. A popular scheme classifies the real-time task scheduling algorithms based on how the scheduling points are defined. The three main types of schedulers according to this classification scheme are: clock-driven, event-driven, and hybrid.

> The clock-driven schedulers are those in which the scheduling points are determined by the interrupts received from a clock. In the event-driven ones, the scheduling points are defined by certain events which precludes clock interrupts. The hybrid ones use both clock interrupts as well as event occurrences to define their scheduling points.

A few important members of each of these three broad classes of scheduling algorithms are the following:

1. Clock Driven
   - Table-driven
   - Cyclic
2. Event Driven
   - Simple priority-based
   - Rate Monotonic Analysis (RMA)
   - Earliest Deadline First (EDF)
3. Hybrid
   - Round-robin

Important members of clock-driven schedulers that we discuss in this text are table-driven and cyclic schedulers. Clock-driven schedulers are simple and efficient. Therefore, these are frequently used in embedded applications. We investigate these two schedulers in some detail in Sec. 2.5.

Important examples of event-driven schedulers are Earliest Deadline First (EDF) and Rate Monotonic Analysis (RMA). Event-driven schedulers are more sophisticated than clock-driven schedulers and usually are more proficient and flexible than clock-driven schedulers. These are more proficient because they can feasibly schedule some task sets which clock-driven schedulers cannot. These are more flexible because they can feasibly schedule sporadic and aperiodic tasks in addition to periodic tasks, whereas clock-driven schedulers can satisfactorily handle only periodic tasks. Event-driven scheduling of real-time tasks in a uniprocessor environment was a subject of intense research during early 1970's, leading to publication of a large number of research results. Out of the large number of research results that were published, the following two popular algorithms are the essence of all those results: Earliest Deadline First (EDF), and Rate Monotonic Analysis (RMA). If we understand these two schedulers well, we would get a good grip on real-time task scheduling on uniprocessors. Several variations to these two basic algorithms exist.

Another classification of real-time task scheduling algorithms can be made based upon the type of task acceptance test that a scheduler carries out before it takes up a task for scheduling. The acceptance test is used to decide whether a newly arrived task would at all be taken up for scheduling or be rejected. Based on the task acceptance test used, there are two broad categories of task schedulers:

- Planning-based
- Best effort

In planning-based schedulers, when a task arrives the scheduler first determines whether the task can meet its dead- lines, if it is taken up for execution. If not, it is rejected. If the task can meet its deadline and does not cause other already scheduled tasks to miss their respective deadlines, then the task is accepted for scheduling. Otherwise, it is rejected. In best effort schedulers, no acceptance test is applied. All tasks that arrive are taken up for scheduling and best effort is made to meet its deadlines. But, no guarantee is given as to whether a task's deadline would be met.

A third type of classification of real-time tasks is based on the target platform on which the tasks are to be run. The different classes of scheduling algorithms according to this scheme are:

- Uniprocessor
- Multiprocessor
- Distributed

Uniprocessor scheduling algorithms are possibly the simplest of the three classes of algorithms. In contrast to uniprocessor algorithms, in multiprocessor and distributed scheduling algorithms first a decision has to be made regarding which task needs to run on which processor and then these tasks are scheduled. In contrast to multiprocessors, the processors in a distributed system do not possess shared memory. Also in contrast to multiprocessors, there is no global up-to-date state information available in distributed systems. This makes uniprocessor scheduling algorithms that assume central state information of all tasks and processors to exist unsuitable for use in distributed systems. Further in distributed systems, the communication among tasks is through message passing. Communication through message passing is costly. This means that a scheduling algorithm should not incur too much communication over- head. So carefully designed distributed algorithms are normally considered suitable for use in a distributed system. In the following sections, we study the different classes of schedulers in more detail.

1.5. Clock-Driven Scheduling

Clock-driven schedulers make their scheduling decisions regarding which task to run next only at the clock interrupt points. Clock-driven schedulers are those for which the scheduling points are determined by timer interrupts. Clock- driven schedulers are also called off-line schedulers because these schedulers fix the schedule before the system starts to run. That is, the scheduler pre-determines which task will run when. Therefore, these schedulers incur very little run time overhead. However, a prominent shortcoming of this class of schedulers is that they can not satisfactorily handle aperiodic and sporadic tasks since the exact time of occurrence of these tasks can not be predicted. For this reason, this type of schedulers is also called static scheduler.

In this section, we study the basic features of two important clock-driven schedulers: table-driven and cyclic schedulers.

1.5.1. Table-Driven Scheduling

Table-driven schedulers usually pre-compute which task would run when, and store this schedule in a table at the time the system is designed or configured. Rather than automatic computation of the schedule by the scheduler, the application programmer can be given the freedom to select his own schedule for the set of tasks in the application and store the schedule in a table (called schedule table) to be used by the scheduler at run time.

An example of a schedule table is shown in Table 1. Table 1 shows that task T1 would be taken up for execution at time instant 0, T_2 would start execution 3 milliseconds afterwards, and so on. An important question that needs to be addressed at this point is what would be the size of the schedule table that would be required for some given set of periodic real-time tasks to be run on a system? An answer to this question can be given as follows: if a set ST = $\{T_i\}$ of n tasks is to be scheduled, then the entries in the table will replicate themselves after LCM (p_1, p_2, … ,p_n) time units, where p_1, p_2, …, p_n are the periods of T_1, T_2, ..., T_n. For example, if we have the following three tasks: (e1=5 msecs, p1=20 msecs), (e2=20 msecs, p2=100 msecs), (e3=30 msecs, p3=250 msecs); then, the schedule will repeat after every 1000 msecs. So, for any given task set, it is sufficient to store entries only for LCM (p_1, p_2, … ,p_n) duration in the schedule table. LCM (p_1, p_2, …, p_n) is called the major cycle of the set of tasks ST.

> A major cycle of a set of tasks is an interval of time on the time line such that in each major cycle, the different tasks recur identically.

In the reasoning we presented above for the computation of the size of a schedule table, one assumption that we implicitly made is that $\phi_i = 0$. That is, all tasks are in phase.

Task	***Start time in millisecs***
T_1	*0*
T_2	*3*
T_3	*10*
T_4	*12*
T_5	*17*

Table 29.1 An Example of a Table-Driven Schedule

However, tasks often do have non-zero phase. It would be interesting to determine what would be the major cycle when tasks have non-zero phase. The result of an investigation into this issue has been given as Theorem 2.1.

1.5.2. Theorem 1

The major cycle of a set of tasks ST = {T_1, T_2, … , T_n} is LCM ({p_1, p_2, … , p_n}) even when the tasks have arbitrary phasing.

Proof: As per our definition of a major cycle, even when tasks have non-zero phasing, task instances would repeat the same way in each major cycle. Let us consider an example in which the occurrences of a task Ti in a major cycle be as shown in Fig. 29.4. As shown in the example of Fig. 29.4, there are k-1 occurrences of the task Ti during a major cycle. The first occurrence of Ti starts ϕ time units from the start of the major cycle. The major cycle ends x time units after the last (i.e. (k-1)th) occurrence of the task Ti in the major cycle. Of course, this must be the same in each major cycle.

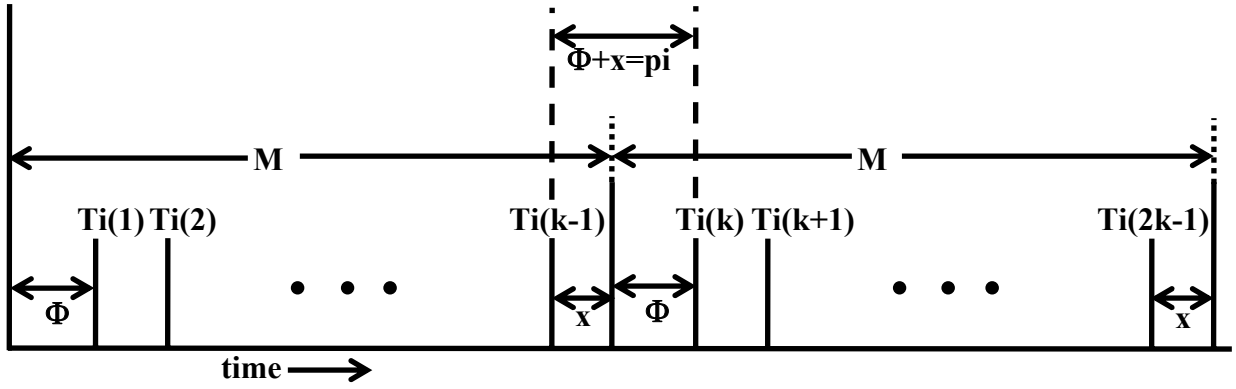


Fig. 29.4 Major Cycle When a Task T_i has Non-Zero Phasing

Assume that the size of each major cycle is M. Then, from an inspection of Fig. 29.4, for the task to repeat identically in each major cycle:

$$M = (k\text{-}1)p_i + \phi + x \qquad \ldots(2.1)$$

Now, for the task T_i to have identical occurrence times in each major cycle, $\phi + x$ must equal to p_i (see Fig. 29.4).

Substituting this in Expr. 2.1, we get, $M = (k\text{-}1) * p_i + p_i = k * p_i$ …(2.2)

So, the major cycle M contains an integral multiple of p_i. This argument holds for each task in the task set irrespective of its phase. Therefore M = LCM ({p_1, p_2, … , p_n}).

1.5.3. Cyclic Schedulers

Cyclic schedulers are very popular and are being extensively used in the industry. A large majority of all small embedded applications being manufactured presently are based on cyclic schedulers. Cyclic schedulers are simple, efficient, and are easy to program. An example application where a cyclic scheduler is normally used is a temperature controller. A

temperature controller periodically samples the temperature of a room and maintains it at a preset value. Such temperature controllers are embedded in typical computer-controlled air conditioners.

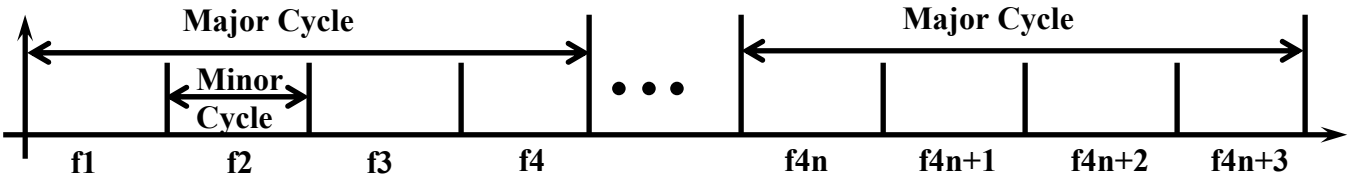


Fig. 29.5 Major and Minor Cycles in a Cyclic Scheduler

A cyclic scheduler repeats a pre-computed schedule. The pre-computed schedule needs to be stored only for one major cycle. Each task in the task set to be scheduled repeats identically in every major cycle. The major cycle is divided into one or more minor cycles (see Fig. 29.5). Each minor cycle is also sometimes called a frame. In the example shown in Fig. 29.5, the major cycle has been divided into four minor cycles (frames). The scheduling points of a cyclic scheduler occur at frame boundaries. This means that a task can start executing only at the beginning of a frame.

The frame boundaries are defined through the interrupts generated by a periodic timer. Each task is assigned to run in one or more frames. The assignment of tasks to frames is stored in a schedule table. An example schedule table is shown in Figure 29.6.

Task Number	***Frame Number***
T_3	f_1
T_1	f_2
T_3	f_3
T_4	f_4

Fig. 29.6 An Example Schedule Table for a Cyclic Scheduler

The size of the frame to be used by the scheduler is an important design parameter and needs to be chosen very carefully. A selected frame size should satisfy the following three constraints.

1. **Minimum Context Switching:** This constraint is imposed to minimize the number of context switches occurring during task execution. The simplest interpretation of this constraint is that a task instance must complete running within its assigned frame. Unless a task completes within its allocated frame, the task might have to be suspended and restarted in a later frame. This would require a context switch involving some processing overhead. To avoid unnecessary context switches, the selected frame size should be larger than the execution time of each task, so that when a task starts at a frame boundary it should be able to complete within the same frame. Formally, we can state this constraint as: $max(\{e_i\}) < F$ where e_i is the execution times of the of task T_i, and F is the frame size. Note that this constraint imposes a lower-bound on frame size, i.e., frame size F must not be smaller than $max(\{e_i\})$.

2. **Minimization of Table Size:** This constraint requires that the number of entries in the schedule table should be minimum, in order to minimize the storage requirement of the schedule table. Remember that cyclic schedulers are used in small embedded applications with a very small storage capacity. So, this constraint is important to the commercial success of a product. The number of entries to be stored in the schedule table can be minimized when the minor cycle squarely divides the major cycle. When the minor cycle squarely divides the major cycle, the major cycle contains an integral number of minor cycles (no fractional minor cycles). Unless the minor cycle squarely divides the major cycle, storing the schedule for one major cycle would not be sufficient, as the schedules in the major cycle would not repeat and this would make the size of the schedule table large. We can formulate this constraint as:

$$\lfloor M/F \rfloor = M/F \qquad \ldots(2.3)$$

In other words, if the floor of M/F equals M/F, then the major cycle would contain an integral number of frames.

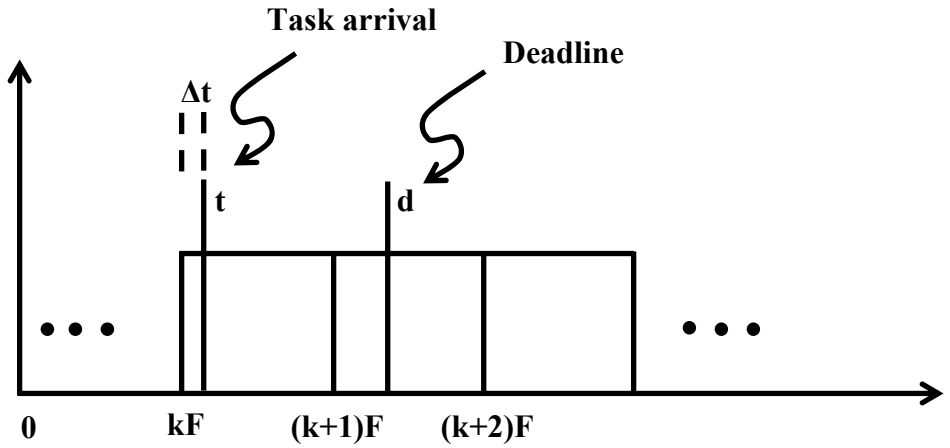


Fig. 29.7 Satisfaction of a Task Deadline

3. **Satisfaction of Task Deadline:** This third constraint on frame size is necessary to meet the task deadlines. This constraint imposes that between the arrival of a task and its deadline, there must exist at least one full frame. This constraint is necessary since a task should not miss its deadline, because by the time it could be taken up for scheduling, the deadline was imminent. Consider this: a task can only be taken up for scheduling at the start of a frame. If between the arrival and completion of a task, not even one frame exists, a situation as shown in Fig. 29.7 might arise. In this case, the task arrives sometimes after the kth frame has started. Obviously it can not be taken up for scheduling in the kth frame and can only be taken up in the k+1th frame. But, then it may be too late to meet its deadline since the execution time of a task can be up to the size of a full frame. This might result in the task missing its deadline since the task might complete only at the end of (k+1)th frame much after the deadline d has passed. We therefore need a full frame to exist between the arrival of a task and its deadline as shown in Fig. 29.8, so that task deadlines could be met.

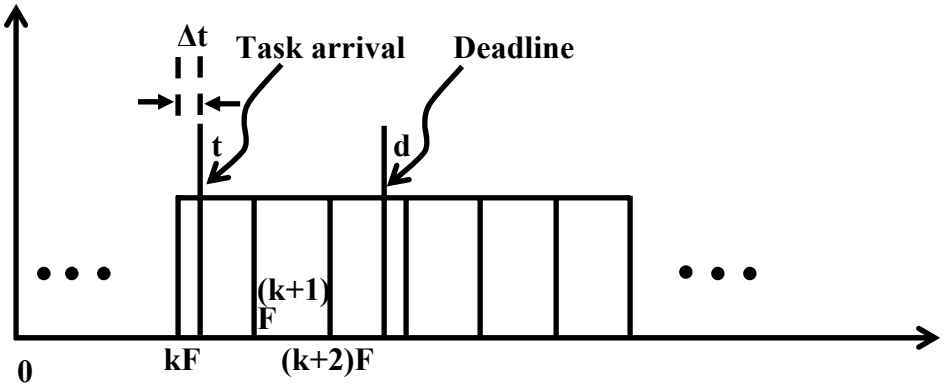


Fig. 29.8 A Full Frame Exists Between the Arrival and Deadline of a Task

More formally, this constraint can be formulated as follows: Suppose a task arises after Δt time units have passed since the last frame (see Fig. 29.8). Then, assuming that a single frame is sufficient to complete the task, the task can complete before its deadline iff $(2F - \Delta t) \leq d_i$, *or* $2F \leq (d_i + \Delta t)$.

…(2.4)

Remember that the value of Δt might vary from one instance of the task to another. The worst case scenario (where the task is likely to miss its deadline) occurs for the task instance having the minimum value of Δt, such that $\Delta t > 0$. This is the worst case scenario, since under this the task would have to wait the longest before its execution can start.

It should be clear that if a task arrives just after a frame has started, then the task would have to wait for the full duration of the current frame before it can be taken up for execution. If a task at all misses its deadline, then certainly it would be under such situations. In other words, the worst case scenario for a task to meet its deadline occurs for its instance that has the minimum separation from the start of a frame. The determination of the minimum separation value (i.e. $min(\Delta t)$) for a task among all instances of the task would help in determining a feasible frame size. We show by Theorem 2.2 that $min(\Delta t)$ is equal to $gcd(F, p_i)$. Consequently, this constraint can be written as:

$$\text{for every Ti,} \quad 2F - gcd(F, p_i) \leq d_i \qquad \ldots(2.5)$$

Note that this constraint defines an upper-bound on frame size for a task Ti, i.e., if the frame size is any larger than the defined upper-bound, then tasks might miss their deadlines. Expr. 2.5 defined the frame size, from the consideration of one task only. Now considering all tasks, the frame size must be smaller than $max(gcd(F, p_i)+d_i)/2$.

1.5.4. Theorem 2

The minimum separation of the task arrival from the corresponding frame start time (min(Δt)), considering all instances of a task T_i, is equal to $gcd(F, p_i)$.

Proof: Let $g = gcd(F, p_i)$, where gcd is the function determining the greatest common divisor of its arguments. It follows from the definition of gcd that g must squarely divide each of F and pi. Let T_i be a task with zero phasing. Now, assume that this Theorem is violated for certain integers m and n, such that the $T_i(n)$ occurs in the mth frame and the difference between

the start time of the mth frame and the nth task arrival time is less than g. That is, $0 < (m * F - n * p_i) < g$.

Dividing this expression throughout by g, we get:

$$0 < (m * F/g - n * p_i/g) < 1 \quad \ldots(2.6)$$

However, F/g and p_i/g are both integers because g is $gcd(F, p_i)$. Therefore, we can write $F/g = I_1$ and $p_i/g = I_2$ for some integral values I_1 and I_2. Substituting this in Expr 2.6, we get $0 < m*I_1 - n*I_2 < 1$. Since $m*I_1$ and $n*I_2$ are both integers, their difference cannot be a fractional value lying between 0 and 1. Therefore, this expression can never be satisfied.

It can therefore be concluded that the minimum time between a frame boundary and the arrival of the corresponding instance of T_i can not be less than $gcd(F, p_i)$.

For a given task set it is possible that more than one frame size satisfies all the three constraints. In such cases, it is better to choose the shortest frame size. This is because of the fact that the schedulability of a task set increases as more frames become available over a major cycle.

It should however be remembered that the mere fact that a suitable frame size can be determined does not mean that a feasible schedule would be found. It may so happen that there is not enough number of frames available in a major cycle to be assigned to all the task instances.

We now illustrate how an appropriate frame size can be selected for cyclic schedulers through a few examples.

1.5.5. Examples

Example 1: A cyclic scheduler is to be used to run the following set of periodic tasks on a uniprocessor: $T_1 = (e_1=1, p_1=4)$, $T_2 = (e_2=, p_2=5)$, $T_3 = (e_3=1, p_3=20)$, $T_4 = (e_4=2, p_4=20)$. Select an appropriate frame size.

Solution: For the given task set, an appropriate frame size is the one that satisfies all the three required constraints. In the following, we determine a suitable frame size F which satisfies all the three required constraints.

Constraint 1: Let F be an appropriate frame size, then max {ei, F}. From this constraint, we get $F \geq 1.5$.

Constraint 2: The major cycle M for the given task set is given by M = LCM(4,5,20) = 20. M should be an integral multiple of the frame size F, i.e., M mod F = 0. This consideration implies that F can take on the values 2, 4, 5, 10, 20. Frame size of 1 has been ruled out since it would violate the constraint 1.

Constraint 3: To satisfy this constraint, we need to check whether a selected frame size F satisfies the inequality: $2F - gcd(F, p_i) < d_i$ for each p_i.

Let us first try frame size 2.

For F = 2 and task T1:

$$2 * 2 - \gcd(2, 4) \leq 4 \equiv 4 - 2 \leq 4$$

Therefore, for p_1 the inequality is satisfied.

Let us try for F = 2 and task T2:

$$2 * 2 - \gcd(2, 5) \leq 5 \equiv 4 - 1 \leq 5$$

Therefore, for p_2 the inequality is satisfied.

Let us try for F = 2 and task T3:

$$2 * 2 - \gcd(2, 20) \leq 20 \equiv 4 - 2 \leq 20$$

Therefore, for p_3 the inequality is satisfied.
For F = 2 and task T4:

$$2 * 2 - \gcd(2, 20) \leq 20 \equiv 4 - 2 \leq 20$$

For p_4 the inequality is satisfied.
Thus, constraint 3 is satisfied by all tasks for frame size 2. So, frame size 2 satisfies all the three constraints. Hence, 2 is a feasible frame size.
Let us try frame size 4.
For F = 4 and task T1:

$$2 * 4 - \gcd(4, 4) \leq 4 \equiv 8 - 4 \leq 4$$

Therefore, for p_1 the inequality is satisfied.
Let us try for F = 4 and task T2:

$$2 * 4 - \gcd(4, 5) \leq 5 \equiv 8 - 1 \leq 5$$

For p2 the inequality is not satisfied. Therefore, we need not look any further. Clearly, F = 4 is not a suitable frame size.
Let us now try frame size 5, to check if that is also feasible.
For F = 5 and task T1, we have

$$2 * 5 - \gcd(5, 4) \leq 4 \equiv 10 - 1 \leq 4$$

The inequality is not satisfied for T1. We need not look any further. Clearly, F = 5 is not a suitable frame size.

Let us now try frame size 10.
For F = 10 and task T1, we have

$$2 * 10 - \gcd(10, 4) \leq 4 \equiv 20 - 2 \leq 4$$

The inequality is not satisfied for T1. We need not look any further. Clearly, F=10 is not a suitable frame size.
Let us try if 20 is a feasible frame size.
For F = 20 and task T1, we have

$$2 * 20 - \gcd(20, 4) \leq 4 \equiv 40 - 4 \leq 4$$

Therefore, F = 20 is also not suitable.
So, only the frame size 2 is suitable for scheduling.
Even though for Example 1 we could successfully find a suitable frame size that satisfies all the three constraints, it is quite probable that a suitable frame size may not exist for many problems. In such cases, to find a feasible frame size we might have to split the task (or a few tasks) that is (are) causing violation of the constraints into smaller sub-tasks that can be scheduled in different frames.

Example 2: Consider the following set of periodic real-time tasks to be scheduled by a cyclic scheduler: $T_1 = (e_1=1, p_1=4)$, $T_2 = (e_2=2, p_2=5)$, $T_3 = (e_3=5, p_3=20)$. Determine a suitable frame size for the task set.

Solution:
Using the first constraint, we have $F \geq 5$.
Using the second constraint, we have the major cycle M = LCM(4, 5, 20) = 20. So, the permissible values of F are 5, 10 and 20.
Checking for a frame size that satisfies the third constraint, we can find that no value of F is suitable. To overcome this problem, we need to split the task that is making the task-set not

schedulable. It is easy to observe that the task T3 has the largest execution time, and consequently due to constraint 1, makes the feasible frame sizes quite large.

We try splitting T3 into two or three tasks. After splitting T3 into three tasks, we have:

$$T_{3.1} = (20, 1, 20), T_{3.2} = (20, 2, 20), T_{3.3} = (20, 2, 20).$$

The possible values of F now are 2 and 4. We can check that now after splitting the tasks, F=2 and F=4 become feasible frame sizes.

It is very difficult to come up with a clear set of guidelines to identify the exact task that is to be split, and the parts into which it needs to be split. Therefore, this needs to be done by trial and error. Further, as the number of tasks to be scheduled increases, this method of trial and error becomes impractical since each task needs to be checked separately. However, when the task set consists of only a few tasks we can easily apply this technique to find a feasible frame size for a set of tasks otherwise not schedulable by a cyclic scheduler.

1.5.6. A Generalized Task Scheduler

We have already stated that cyclic schedulers are overwhelmingly popular in low-cost real-time applications. However, our discussion on cyclic schedulers was so far restricted to scheduling periodic real-time tasks. On the other hand, many practical applications typically consist of a mixture of several periodic, aperiodic, and sporadic tasks. In this section, we discuss how aperiodic and sporadic tasks can be accommodated by cyclic schedulers.

Recall that the arrival times of aperiodic and sporadic tasks are expressed statistically. Therefore, there is no way to assign aperiodic and sporadic tasks to frames without significantly lowering the overall achievable utilization of the system. In a generalized scheduler, initially a schedule (assignment of tasks to frames) for only periodic tasks is prepared. The sporadic and aperiodic tasks are scheduled in the slack times that may be available in the frames. Slack time in a frame is the time left in the frame after a periodic task allocated to the frame completes its execution. Non-zero slack time in a frame can exist only when the execution time of the task allocated to it is smaller than the frame size.

A sporadic task is taken up for scheduling only if enough slack time is available for the arriving sporadic task to complete before its deadline. Therefore, a sporadic task on its arrival is subjected to an acceptance test. The acceptance test checks whether the task is likely to be completed within its deadline when executed in the available slack times. If it is not possible to meet the task's deadline, then the scheduler rejects it and the corresponding recovery routines for the task are run. Since aperiodic tasks do not have strict deadlines, they can be taken up for scheduling without any acceptance test and best effort can be made to schedule them in the slack times available. Though for aperiodic tasks no acceptance test is done, but no guarantee is given for a task's completion time and best effort is made to complete the task as early as possible.

An efficient implementation of this scheme is that the slack times are stored in a table and during acceptance test this table is used to check the schedulability of the arriving tasks.

Another popular alternative is that the aperiodic and sporadic tasks are accepted without any acceptance test, and best effort is made to meet their respective deadlines.

Pseudo-code for a Generalized Scheduler: The following is the pseudo-code for a generalized cyclic scheduler we discussed, which schedules periodic, aperiodic and sporadic tasks. It is assumed that pre-computed schedule for periodic tasks is stored in a schedule table,

and if required the sporadic tasks have already been subjected to an acceptance test and only those which have passed the test are available for scheduling.

```
   cyclic-scheduler() {
   current-task T = Schedule-Table[k];
   k = k + 1;
   k = k mod N;       //N is the total number of tasks in the schedule
table
   dispatch-current-task(T);
   schedule-sporadic-tasks();    //Current task T completed  early,
                                 //  sporadic  tasks  can  be  taken
                              up
   schedule-aperiodic-tasks();   //At the end of the frame, the running
task
                                 //  is pre-empted if not complete
   idle();                       //No task to run, idle
   }
```

The cyclic scheduler routine cyclic-scheduler () is activated at the end of every frame by a periodic timer. If the current task is not complete by the end of the frame, then it is suspended and the task to be run in the next frame is dispatched by invoking the routine cyclic-scheduler(). If the task scheduled in a frame completes early, then any existing sporadic or aperiodic task is taken up for execution.

1.5.7. Comparison of Cyclic with Table-Driven Scheduling

Both table-driven and cyclic schedulers are important clock-driven schedulers. A scheduler needs to set a periodic timer only once at the application initialization time. This timer continues to give an interrupt exactly at every frame boundary. But in table-driven scheduling, a timer has to be set every time a task starts to run. The execution time of a typical real-time task is usually of the order of a few milliseconds. Therefore, a call to a timer is made every few mill Seconds. This represents a significant overhead and results in degraded system performance. Therefore, a cyclic scheduler is more efficient than a table-driven scheduler. This probably is a reason why cyclic schedulers are so overwhelmingly popular especially in embedded applications. However, if the overhead of setting a timer can be ignored, a table-driven scheduler is more proficient than a cyclic scheduler because the size of the frame that needs to be chosen should be at least as long as the size of the largest execution time of a task in the task set. This is a source of inefficiency, since this results in processor time being wasted in case of those tasks whose execution times are smaller than the chosen frame size.

1.6. Exercises

1. State whether the following assertions are True or False. Write one or two sentences to justify your choice in each case.
   a. Average response time is an important performance metric for real-time operating systems handling running of hard real-time tasks.
   b. Unlike table-driven schedulers, cyclic schedulers do not require to store a pre-computed schedule.

c. The minimum period for which a table-driven scheduler scheduling n periodic tasks needs to pre-store the schedule is given by $\max\{p_1, p_2, \ldots, p_n\}$, where p_i is the period of the task T_i.
d. A cyclic scheduler is more proficient than a pure table-driven scheduler for scheduling a set of hard real-time tasks.
e. A suitable figure of merit to compare the performance of different hard real-time task scheduling algorithms can be the average task response times resulting from each algorithm.
f. Cyclic schedulers are more proficient than table-driven schedulers.
g. While using a cyclic scheduler to schedule a set of real-time tasks on a uniprocessor, when a suitable frame size satisfying all the three required constraints has been found, it is guaranteed that the task set would be feasibly scheduled by the cyclic scheduler.
h. When more than one frame satisfies all the constraints on frame size while scheduling a set of hard real-time periodic tasks using a cyclic scheduler, the largest of these frame sizes should be chosen.
i. In table-driven scheduling of three periodic tasks T_1, T_2, T_3, the scheduling table must have schedules for all tasks drawn up to the time interval $[0, \max(p_1, p_2, p_3)]$, where p_i is the period of the task T_i.
j. When a set of hard real-time periodic tasks are being scheduled using a cyclic scheduler, if a certain frame size is found to be not suitable, then any frame size smaller than this would not also be suitable for scheduling the tasks.
k. When a set of hard real-time periodic tasks are being scheduled using a cyclic scheduler, if a candidate frame size exceeds the execution time of every task and squarely divides the major cycle, then it would be a suitable frame size to schedule the given set of tasks.
l. Finding an optimal schedule for a set of independent periodic hard real-time tasks without any resource- sharing constraints under static priority conditions is an NP-complete problem.

2. Real-time tasks are normally classified into periodic, aperiodic, and sporadic real-time task.
   a. What are the basic criteria based on which a real-time task can be determined to belong to one of the three categories?
   b. Identify some characteristics that are unique to each of the three categories of tasks.
   c. Give examples of tasks in practical systems which belong to each of the three categories.
3. What do you understand by an optimal scheduling algorithm? Is it true that the time complexity of an optimal scheduling algorithm for scheduling a set of real-time tasks in a uniprocessor is prohibitively expensive to be of any practical use? Explain your answer.
4. Suppose a set of three periodic tasks is to be scheduled using a cyclic scheduler on a uniprocessor. Assume that the CPU utilization due to the three tasks is less than 1. Also, assume that for each of the three tasks, the deadlines equals the respective periods. Suppose that we are able to find an appropriate frame size (without having to split any of the tasks) that satisfies the three constraints of minimization of context switches, minimization of schedule table size, and satisfaction of deadlines. Does this imply that it is possible to assert that we can feasibly schedule the three tasks using the cyclic scheduler? If you answer affirmatively, then prove your answer. If you answer negatively, then show an example involving three tasks that disproves the assertion.
5. Consider a real-time system which consists of three tasks T1, T2, and T3, which have been characterized in the following table.

Task	Phase mSec	Execution Time mSec	Relative Deadline mSec	Period mSec
T_1	20	10	20	20
T_2	40	10	50	50
T_3	70	20	80	80

If the tasks are to be scheduled using a table-driven scheduler, what is the length of time for which the schedules have to be stored in the pre-computed schedule table of the scheduler.

6. A cyclic real-time scheduler is to be used to schedule three periodic tasks T1, T2, and T3 with the following characteristics:

Task	Phase mSec	Execution Time mSec	Relative Deadline mSec	Period mSec
T_1	0	20	100	100
T_2	0	20	80	80
T_3	0	30	150	150

Suggest a suitable frame size that can be used. Show all intermediate steps in your calculations.

7. Consider the following set of three independent real-time periodic tasks.

Task	Start Time mSec	Processing Time mSec	Period mSec	Deadline mSec
T_1	20	25	150	100
T_2	40	10	50	30
T_3	60	50	200	150

Suppose a cyclic scheduler is to be used to schedule the task set. What is the major cycle of the task set? Suggest a suitable frame size and provide a feasible schedule (task to frame assignment for a major cycle) for the task set.

Module 6

Embedded System Software

Lesson 30

Real-Time Task Scheduling – Part 2

Specific Instructional Objectives

At the end of this lesson, the student would be able to:

- Get an introduction to event-driven schedulers
- Understand the basics of Foreground-Background schedulers
- Get an overview of Earliest Deadline First (EDF) Algorithm
- Work out solutions to problems based on EDF
- Know the shortcomings of EDF
- Get an overview of Rate Monotonic Algorithm (RMA)
- Know the necessary and sufficient conditions for a set of real-time tasks to be RMA-schedulable
- Work out solutions to problems based on EDF
- Infer the maximum achievable CPU utilization
- Understand the Advantages and Disadvantages of RMA
- Get an overview of Deadline Monotonic Algorithm (DMA)
- Understand the phenomenon of Context-Switching and Self-Suspension

1. Event-driven Scheduling – An Introduction

In this lesson, we shall discuss the various algorithms for event-driven scheduling. From the previous lesson, we may recollect the following points:
The clock-driven schedulers are those in which the scheduling points are determined by the interrupts received from a clock. In the event-driven ones, the scheduling points are defined by certain events which precludes clock interrupts. The hybrid ones use both clock interrupts as well as event occurrences to define their scheduling points

Cyclic schedulers are very efficient. However, a prominent shortcoming of the cyclic schedulers is that it becomes very complex to determine a suitable frame size as well as a feasible schedule when the number of tasks increases. Further, in almost every frame some processing time is wasted (as the frame size is larger than all task execution times) resulting in sub-optimal schedules. Event-driven schedulers overcome these shortcomings. Further, event-driven schedulers can handle aperiodic and sporadic tasks more proficiently. On the flip side, event-driven schedulers are less efficient as they deploy more complex scheduling algorithms. Therefore, event-driven schedulers are less suitable for embedded applications as these are required to be of small size, low cost, and consume minimal amount of power.

It should now be clear why event-driven schedulers are invariably used in all moderate and large-sized applications having many tasks, whereas cyclic schedulers are predominantly used in small applications. In event-driven scheduling, the scheduling points are defined by task completion and task arrival events. This class of schedulers is normally preemptive, i.e., when a higher priority task becomes ready, it preempts any lower priority task that may be running.

1.1. Types of Event Driven Schedulers

We discuss three important types of event-driven schedulers:

- Simple priority-based
- Rate Monotonic Analysis (RMA)
- Earliest Deadline First (EDF)

The simplest of these is the foreground-background scheduler, which we discuss next. In section 3.4, we discuss EDF and in section 3.5, we discuss RMA.

1.2. Foreground-Background Scheduler

A foreground-background scheduler is possibly the simplest priority-driven preemptive scheduler. In foreground-background scheduling, the real-time tasks in an application are run as fore- ground tasks. The sporadic, aperiodic, and non-real-time tasks are run as background tasks. Among the foreground tasks, at every scheduling point the highest priority task is taken up for scheduling. A background task can run when none of the foreground tasks is ready. In other words, the background tasks run at the lowest priority.

Let us assume that in a certain real-time system, there are n foreground tasks which are denoted as: T1,T2,...,Tn. As already mentioned, the foreground tasks are all periodic. Let T_B be the only background task. Let e_B be the processing time requirement of T_B. In this case, the completion time (ct_B) for the background task is given by:

$$ct_B = e_B / (1 - \sum_{i=1}^{n} e_i / p_i) \quad \ldots \quad (3.1/2.7)$$

This expression is easy to interpret. When any foreground task is executing, the background task waits. The average CPU utilization due to the foreground task T_i is e_i/p_i, since e_i amount of processing time is required over every p_i period. It follows that all foreground tasks together would result in CPU utilization of $\sum_{i=1}^{n} e_i / p_i$. Therefore, the average time available for execution of the background tasks in every unit of time is $1 - \sum_{i=1}^{n} e_i / p_i$. Hence, Expr. 2.7 follows easily. We now illustrate the applicability of Expr. 2.7 through the following three simple examples.

1.3. Examples

Example 1: Consider a real-time system in which tasks are scheduled using foreground-background scheduling. There is only one periodic foreground task T_f: (ϕ_f =0, p_f =50 msec, e_f =100 msec, d_f =100 msec) and the background task be T_B = (e_B =1000 msec). Compute the completion time for background task.

Solution: By using the expression (2.7) to compute the task completion time, we have

$$ct_B = 1000 / (1 - 50/100) = 2000 \text{ msec}$$

So, the background task T_B would take 2000 milliseconds to complete.

Example 2: In a simple priority-driven preemptive scheduler, two periodic tasks T1 and T2 and a background task are scheduled. The periodic task T1 has the highest priority and executes once every 20 milliseconds and requires 10 milliseconds of execution time each time. T2 requires 20 milliseconds of processing every 50 milliseconds. T3 is a background task and requires 100 milliseconds to complete. Assuming that all the tasks start at time 0, determine the time at which T3 will complete.

Solution: The total utilization due to the foreground tasks: $\sum_{i=1}^{2} e_i / p_i = 10/20 + 20/50 = 90/100$.

This implies that the fraction of time remaining for the background task to execute is given by:

$$1 - \sum_{i=1}^{2} e_i / p_i = 10/100.$$

Therefore, the background task gets 1 millisecond every 10 milliseconds. Thus, the background task would take $10*(100/1) = 1000$ milliseconds to complete.

Example 3: Suppose in Example 1, an overhead of 1 msec on account of every context switch is to be taken into account. Compute the completion time of T_B.

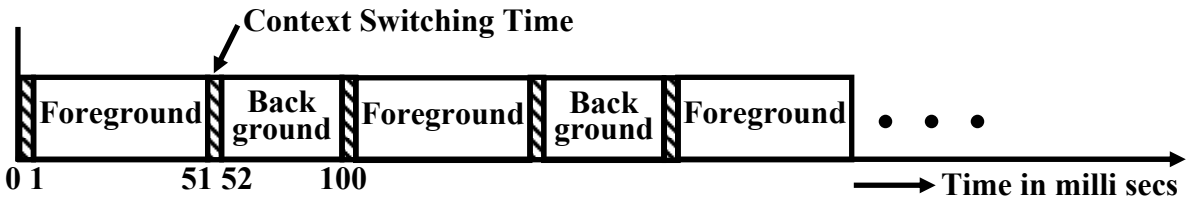


Fig. 30.1 Task Schedule for Example 3

Solution: The very first time the foreground task runs (at time 0), it incurs a context switching overhead of 1 msec. This has been shown as a shaded rectangle in Fig. 30.1. Subsequently each time the foreground task runs, it preempts the background task and incurs one context switch. On completion of each instance of the foreground task, the background task runs and incurs another context switch. With this observation, to simplify our computation of the actual completion time of T_B, we can imagine that the execution time of every foreground task is increased by two context switch times (one due to itself and the other due to the background task running after each time it completes). Thus, the net effect of context switches can be imagined to be causing the execution time of the foreground task to increase by 2 context switch times, i.e. to 52 milliseconds from 50 milliseconds. This has pictorially been shown in Fig. 30.1.

Now, using Expr. 2.7, we get the time required by the background task to complete:

$$1000/(1-52/100) = 2083.4 \text{ milliseconds}$$

In the following two sections, we examine two important event-driven schedulers: EDF (Earliest Deadline First) and RMA (Rate Monotonic Algorithm). EDF is the optimal dynamic priority real-time task scheduling algorithm and RMA is the optimal static priority real-time task scheduling algorithm.

1.4. Earliest Deadline First (EDF) Scheduling

In Earliest Deadline First (EDF) scheduling, at every scheduling point the task having the shortest deadline is taken up for scheduling. This basic principles of this algorithm is very intuitive and simple to understand. The schedulability test for EDF is also simple. A task set is schedulable under EDF, if and only if it satisfies the condition that the total processor utilization due to the task set is less than 1. For a set of periodic real-time tasks $\{T_1, T_2, \ldots, T_n\}$, EDF schedulability criterion can be expressed as:

$$\sum_{i=1}^{n} e_i / p_i = \sum_{i=1}^{n} u_i \leq 1$$

… (3.2/2.8)

where u_i is average utilization due to the task T_i and n is the total number of tasks in the task set. Expr. 3.2 is both a necessary and a sufficient condition for a set of tasks to be EDF schedulable.

EDF has been proven to be an optimal uniprocessor scheduling algorithm. This means that, if a set of tasks is not schedulable under EDF, then no other scheduling algorithm can feasibly schedule this task set. In the simple schedulability test for EDF (Expr. 3.2), we assumed that the period of each task is the same as its deadline. However, in practical problems the period of a task may at times be different from its deadline. In such cases, the schedulability test needs to be changed. If $p_i > d_i$, then each task needs e_i amount of computing time every $min(p_i, d_i)$ duration of time. Therefore, we can rewrite Expr. 3.2 as:

$$\sum_{i=1}^{n} e_i / min(p_i, d_i) \leq 1 \quad \ldots \quad (3.3/2.9)$$

However, if $p_i < d_i$, it is possible that a set of tasks is EDF schedulable, even when the task set fails to meet the Expr 3.3. Therefore, Expr 3.3 is conservative when $p_i < d_i$, and is not a necessary condition, but only a sufficient condition for a given task set to be EDF schedulable.

Example 4: Consider the following three periodic real-time tasks to be scheduled using EDF on a uniprocessor: $T_1 = (e_1=10, p_1=20)$, $T_2 = (e_2=5, p_2=50)$, $T_3 = (e_3=10, p_3=35)$. Determine whether the task set is schedulable.

Solution: The total utilization due to the three tasks is given by:

$$\sum_{i=1}^{3} e_i / p_i = 10/20 + 5/50 + 10/35 = 0.89$$

This is less than 1. Therefore, the task set is EDF schedulable.

Though EDF is a simple as well as an optimal algorithm, it has a few shortcomings which render it almost unusable in practical applications. The main problems with EDF are discussed in Sec. 3.4.3. Next, we discuss the concept of task priority in EDF and then discuss how EDF can be practically implemented.

1.4.1. Is EDF Really a Dynamic Priority Scheduling Algorithm?

We stated in Sec 3.3 that EDF is a dynamic priority scheduling algorithm. Was it after all correct on our part to assert that EDF is a dynamic priority task scheduling algorithm? If EDF were to be considered a dynamic priority algorithm, we should be able determine the precise priority value of a task at any point of time and also be able to show how it changes with time. If we reflect on our discussions of EDF in this section, EDF scheduling does not require any priority value to be computed for any task at any time. In fact, EDF has no notion of a priority value for a task. Tasks are scheduled solely based on the proximity of their deadline. However, the longer a task waits in a ready queue, the higher is the chance (probability) of being taken up for scheduling. So, we can imagine that a virtual priority value associated with a task keeps increasing with time until the task is taken up for scheduling. However, it is important to understand that in EDF the tasks neither have any priority value associated with them, nor does the scheduler perform any priority computations to determine the schedulability of a task at either run time or compile time.

1.4.2. Implementation of EDF

A naive implementation of EDF would be to maintain all tasks that are ready for execution in a queue. Any freshly arriving task would be inserted at the end of the queue. Every node in the

queue would contain the absolute deadline of the task. At every preemption point, the entire queue would be scanned from the beginning to determine the task having the shortest deadline. However, this implementation would be very inefficient. Let us analyze the complexity of this scheme. Each task insertion will be achieved in O(1) or constant time, but task selection (to run next) and its deletion would require O(n) time, where n is the number of tasks in the queue.

A more efficient implementation of EDF would be as follows. EDF can be implemented by maintaining all ready tasks in a sorted priority queue. A sorted priority queue can efficiently be implemented by using a heap data structure. In the priority queue, the tasks are always kept sorted according to the proximity of their deadline. When a task arrives, a record for it can be inserted into the heap in $O(\log_2 n)$ time where n is the total number of tasks in the priority queue. At every scheduling point, the next task to be run can be found at the top of the heap. When a task is taken up for scheduling, it needs to be removed from the priority queue. This can be achieved in O(1) time.

A still more efficient implementation of the EDF can be achieved as follows under the assumption that the number of distinct deadlines that tasks in an application can have are restricted. In this approach, whenever task arrives, its absolute deadline is computed from its release time and its relative deadline. A separate FIFO queue is maintained for each distinct relative deadline that tasks can have. The scheduler inserts a newly arrived task at the end of the corresponding relative deadline queue. Clearly, tasks in each queue are ordered according to their absolute deadlines.

To find a task with the earliest absolute deadline, the scheduler only needs to search among the threads of all FIFO queues. If the number of priority queues maintained by the scheduler is Q, then the order of searching would be O(1). The time to insert a task would also be O(1).

1.4.3. Shortcomings of EDF

In this subsection, we highlight some of the important shortcomings of EDF when used for scheduling real-time tasks in practical applications.

Transient Overload Problem: Transient overload denotes the overload of a system for a very short time. Transient overload occurs when some task takes more time to complete than what was originally planned during the design time. A task may take longer to complete due to many reasons. For example, it might enter an infinite loop or encounter an unusual condition and enter a rarely used branch due to some abnormal input values. When EDF is used to schedule a set of periodic real-time tasks, a task overshooting its completion time can cause some other task(s) to miss their deadlines. It is usually very difficult to predict during program design which task might miss its deadline when a transient overload occurs in the system due to a low priority task overshooting its deadline. The only prediction that can be made is that the task (tasks) that would run immediately after the task causing the transient overload would get delayed and might miss its (their) respective deadline(s). However, at different times a task might be followed by different tasks in execution. However, this lead does not help us to find which task might miss its deadline. Even the most critical task might miss its deadline due to a very low priority task overshooting its planned completion time. So, it should be clear that under EDF any amount of careful design will not guarantee that the most critical task would not miss its deadline under transient overload. This is a serious drawback of the EDF scheduling algorithm.

Resource Sharing Problem: When EDF is used to schedule a set of real-time tasks, unacceptably high overheads might have to be incurred to support resource sharing among the tasks without making tasks to miss their respective deadlines. We examine this issue in some detail in the next lesson.

Efficient Implementation Problem: The efficient implementation that we discussed in Sec. 3.4.2 is often not practicable as it is difficult to restrict the number of tasks with distinct deadlines to a reasonable number. The efficient implementation that achieves O(1) overhead assumes that the number of relative deadlines is restricted. This may be unacceptable in some situations. For a more flexible EDF algorithm, we need to keep the tasks ordered in terms of their deadlines using a priority queue. Whenever a task arrives, it is inserted into the priority queue. The complexity of insertion of an element into a priority queue is of the order $log_2 n$, where n is the number of tasks to be scheduled. This represents a high runtime overhead, since most real-time tasks are periodic with small periods and strict deadlines.

1.5. Rate Monotonic Algorithm(RMA)

We had already pointed out that RMA is an important event-driven scheduling algorithm. This is a static priority algorithm and is extensively used in practical applications. RMA assigns priorities to tasks based on their rates of occurrence. The lower the occurrence rate of a task, the lower is the priority assigned to it. A task having the highest occurrence rate (lowest period) is accorded the highest priority. RMA has been proved to be the optimal static priority real-time task scheduling algorithm.

In RMA, the priority of a task is directly proportional to its rate (or, inversely proportional to its period). That is, the priority of any task T_i is computed as: $priority = k / p_i$, where p_i is the period of the task T_i and k is a constant. Using this simple expression, plots of priority values of tasks under RMA for tasks of different periods can be easily obtained. These plots have been shown in Fig. 30.10(a) and Fig. 30.10(b). It can be observed from these figures that the priority of a task increases linearly with the arrival rate of the task and inversely with its period.

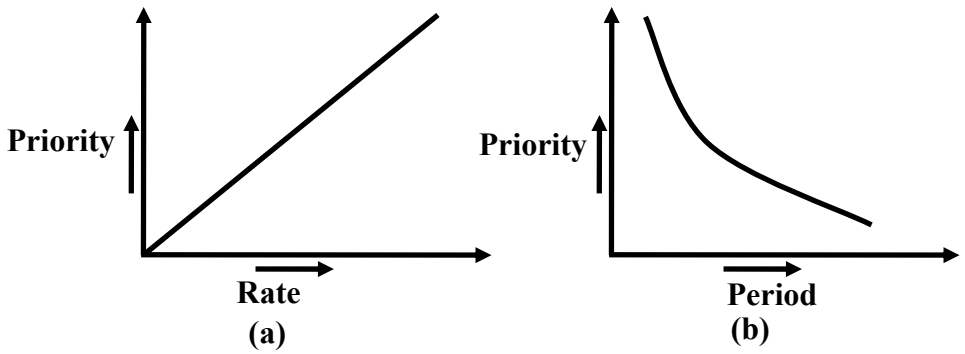


Fig. 30.2 Priority Assignment to Tasks in RMA

1.5.1. Schedulability Test for RMA

An important problem that is addressed during the design of a uniprocessor-based real-time system is to check whether a set of periodic real-time tasks can feasibly be scheduled under RMA. Schedulability of a task set under RMA can be determined from a knowledge of the

worst-case execution times and periods of the tasks. A pertinent question at this point is how can a system developer determine the worst-case execution time of a task even before the system is developed. The worst-case execution times are usually determined experimentally or through simulation studies.

The following are some important criteria that can be used to check the schedulability of a set of tasks set under RMA.

1.5.1.1 Necessary Condition

A set of periodic real-time tasks would not be RMA schedulable unless they satisfy the following necessary condition:

$$\sum_{i=1}^{n} e_i / p_i = \sum_{i=1}^{n} u_i \leq 1$$

where e_i is the worst case execution time and pi is the period of the task T_i, n is the number of tasks to be scheduled, and ui is the CPU utilization due to the task T_i. This test simply expresses the fact that the total CPU utilization due to all the tasks in the task set should be less than 1.

1.5.1.2 Sufficient Condition

The derivation of the sufficiency condition for RMA schedulability is an important result and was obtained by Liu and Layland in 1973. A formal derivation of the Liu and Layland's results from first principles is beyond the scope of this discussion. We would subsequently refer to the sufficiency as the Liu and Layland's condition. A set of n real-time periodic tasks are schedulable under RMA, if

$$\sum_{i=1}^{n} u_i \leq n(2^{1/n} - 1) \qquad (3.4/2.10)$$

where u_i is the utilization due to task T_i. Let us now examine the implications of this result. If a set of tasks satisfies the sufficient condition, then it is guaranteed that the set of tasks would be RMA schedulable.

Consider the case where there is only one task in the system, i.e. n = 1.

Substituting n = 1 in Expr. 3.4, we get,

$$\sum_{i=1}^{1} u_i \leq 1(2^{1/1} - 1) \text{ or } \sum_{i=1}^{1} u_i \leq 1$$

Similarly for n = 2, we get,

$$\sum_{i=1}^{2} u_i \leq 2(2^{1/2} - 1) \text{ or } \sum_{i=1}^{2} u_i \leq 0.828$$

For n = 3, we get,

$$\sum_{i=1}^{3} u_i \leq 3(2^{1/3} - 1) \text{ or } \sum_{i=1}^{3} u_i \leq 0.78$$

For $n \rightarrow \infty$, we get,

$$\sum_{i=1}^{\infty} u_i \leq 3(2^{1/\infty} - 1) \text{ or } \sum_{i=1}^{\infty} u_i \leq \infty.0$$

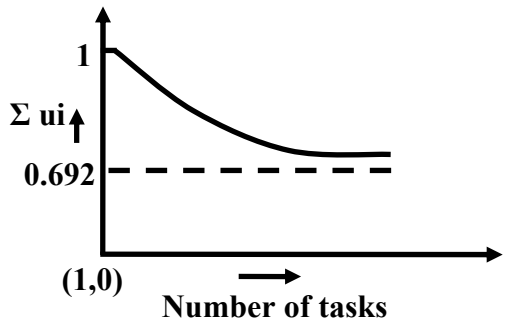


Fig. 30.3 Achievable Utilization with the Number of Tasks under RMA

Evaluation of Expr. 3.4 when $n \rightarrow \infty$ involves an indeterminate expression of the type $\infty.0$. By applying L'Hospital's rule, we can verify that the right hand side of the expression evaluates to $\log_e 2 = 0.692$. From the above computations, it is clear that the maximum CPU utilization that can be achieved under RMA is 1. This is achieved when there is only a single task in the system. As the number of tasks increases, the achievable CPU utilization falls and as $n \rightarrow \infty$, the achievable utilization stabilizes at $\log_e 2$, which is approximately 0.692. This is pictorially shown in Fig. 30.3. We now illustrate the applicability of the RMA schedulability criteria through a few examples.

1.5.2. Examples

Example 5: Check whether the following set of periodic real-time tasks is schedulable under RMA on a uniprocessor: $T_1 = (e_1=20, p_1=100)$, $T_2 = (e_2=30, p_2=150)$, $T_3 = (e_3=60, p_3=200)$.

Solution: Let us first compute the total CPU utilization achieved due to the three given tasks.

$$\sum_{i=1}^{3} u_i = 20/100 + 30/150 + 60/200 = 0.7$$

This is less than 1; therefore the necessary condition for schedulability of the tasks is satisfied. Now checking for the sufficiency condition, the task set is schedulable under RMA if Liu and Layland's condition given by Expr. 3.4 is satisfied Checking for satisfaction of Expr. 3.4, the maximum achievable utilization is given by:

$$3(2^{1/3} - 1) = 0.78$$

The total utilization has already been found to be 0.7. Now substituting these in Liu and Layland's criterion:

$$\sum_{i=1}^{3} u_i \leq 3(2^{1/3} - 1)$$

Therefore, we get $0.7 < 0.78$.
Expr. 3.4, a sufficient condition for RMA schedulability, is satisfied. Therefore, the task set is RMA-schedulable

Example 6: Check whether the following set of three periodic real-time tasks is schedulable under RMA on a uniprocessor: $T_1 = (e_1=20, p_1=100)$, $T_2 = (e_2=30, p_2=150)$, $T_3 = (e_3=90, p_3=200)$.

Solution: Let us first compute the total CPU utilization due to the given task set:

$$\sum_{i=1}^{3} u_i = 20/100 + 30/150 + 90/200 = 0.7$$

Now checking for Liu and Layland criterion:

$$\sum_{i=1}^{3} u_i \leq 0.78$$

Since 0.85 is not ≤ 0.78, the task set is not RMA-schedulable.

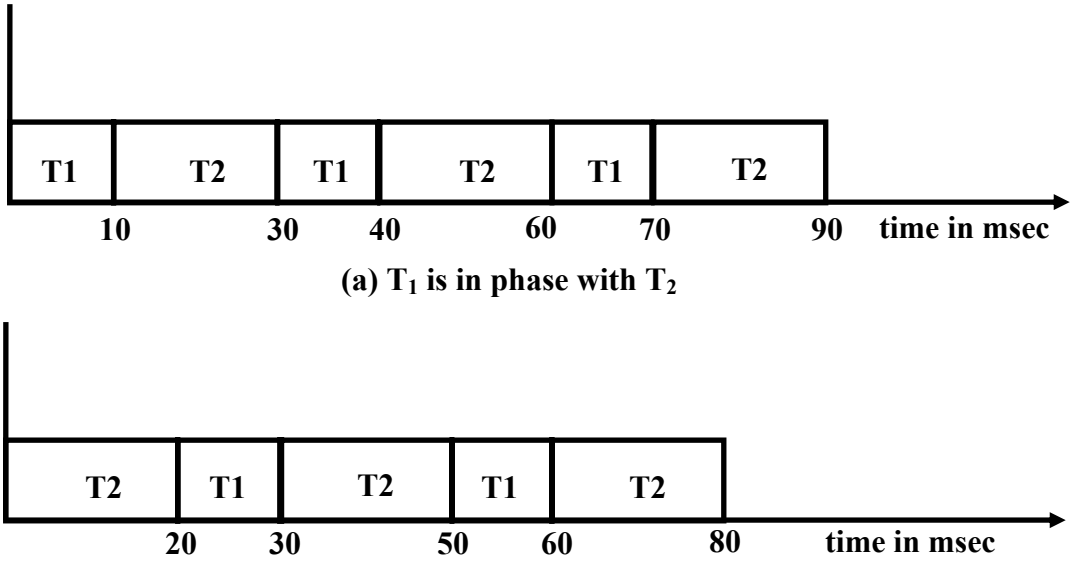
Liu and Layland test (Expr. 2.10) is pessimistic in the following sense.

> If a task set passes the Liu and Layland test, then it is guaranteed to be RMA schedulable. On the other hand, even if a task set fails the Liu and Layland test, it may still be RMA schedulable.

It follows from this that even when a task set fails Liu and Layland's test, we should not conclude that it is not schedulable under RMA. We need to test further to check if the task set is RMA schedulable. A test that can be per- formed to check whether a task set is RMA schedulable when it fails the Liu and Layland test is the Lehoczky's test. Lehoczky's test has been expressed as Theorem 3.

1.5.3. Theorem 3

A set of periodic real-time tasks is RMA schedulable under any task phasing, iff all the tasks meet their respective first deadlines under zero phasing.



(a) T_1 is in phase with T_2

(b) T_1 has a 20 msec phase with respect to T_2

Fig. 30.4 Worst Case Response Time for a Task Occurs When It is in Phase with Its Higher Priority Tasks

A formal proof of this Theorem is beyond the scope of this discussion. However, we provide an intuitive reasoning as to why Theorem 3 must be true. Intuitively, we can understand this result from the following reasoning. First let us try to understand the following fact.

The worst case response time for a task occurs when it is in phase with its higher

To see why this statement must be true, consider the following statement. Under RMA whenever a higher priority task is ready, the lower priority tasks can not execute and have to wait. This implies that, a lower priority task will have to wait for the entire duration of execution of each higher priority task that arises during the execution of the lower priority task. More number of instances of a higher priority task will occur, when a task is in phase with it, when it is in phase with it rather than out of phase with it. This has been illustrated through a simple example in Fig. 30.4. In Fig. 30.4(a), a higher priority task T1=(10,30) is in phase with a lower priority task T2=(60,120), the response time of T2 is 90 msec. However, in Fig. 30.4(b), when T1 has a 20 msec phase, the response time of T2 becomes 80. Therefore, if a task meets its first deadline under zero phasing, then they it will meet all its deadlines.

Example 7: Check whether the task set of Example 6 is actually schedulable under RMA.

Solution: Though the results of Liu and Layland's test were negative as per the results of Example 6, we can apply the Lehoczky test and observe the following:

For the task T1: $e_1 < p_1$ holds since 20 msec < 100 msec. Therefore, it would meet its first deadline (it does not have any tasks that have higher priority).

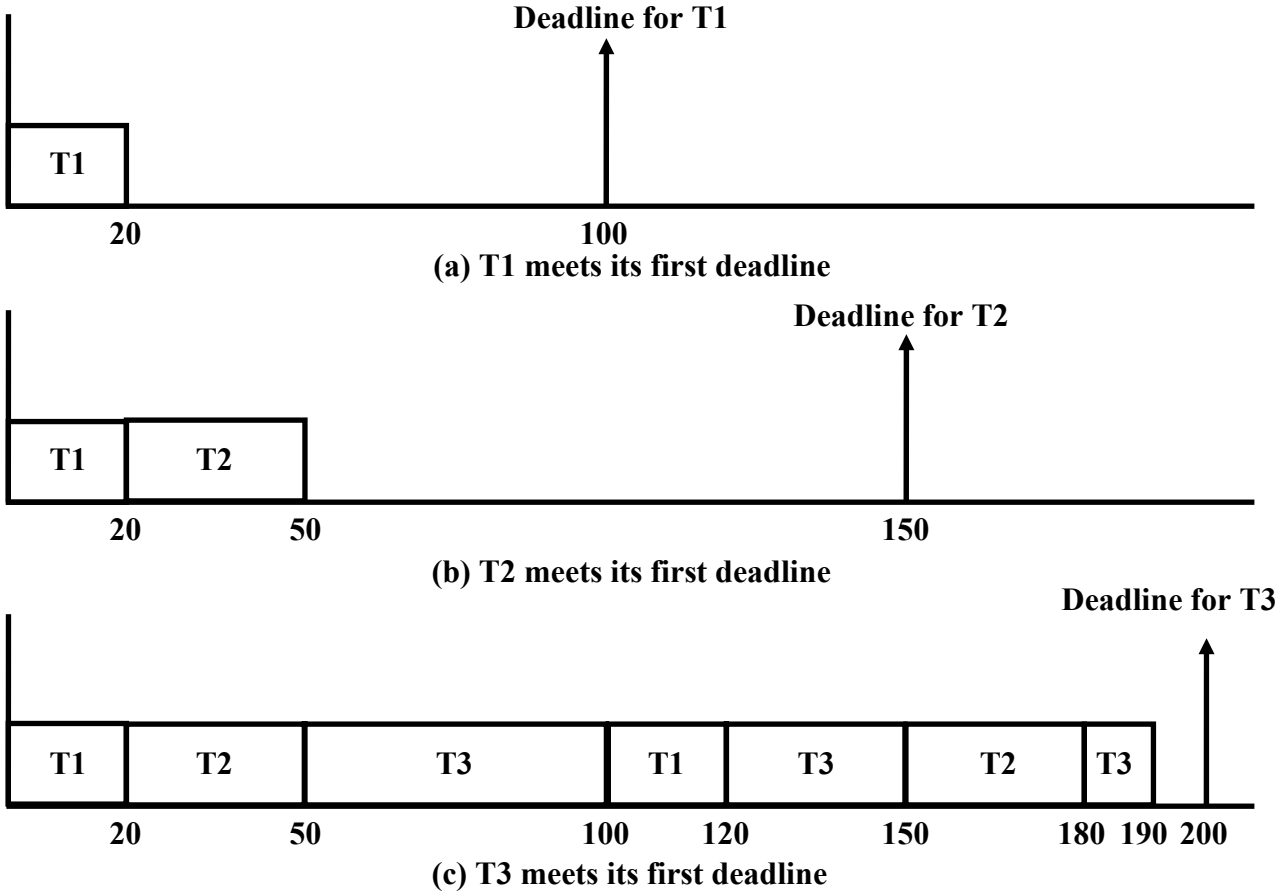


Fig. 30.5 Checking Lehoczky's Criterion for Tasks of Example 7

For the task T_2: T_1 is its higher priority task and considering 0 phasing, it would occur once before the deadline of T_2. Therefore, $(e_1 + e_2) < p_2$ holds, since 20 + 30 = 50 msec < 150 msec. Therefore, T_2 meets its first deadline.

For the task T_3: $(2e_1 + 2e_2 + e_3) < p_3$ holds, since 2∗20 + 2∗30 + 90 = 190msec < 200 msec.

We have considered 2∗e1 and 2∗e2 since T_1 and T_2 occur twice within the first deadline of T_3. Therefore, T_3 meets its first deadline. So, the given task set is schedulable under RMA. The schedulability test for T_3 has pictorially been shown in Fig. 30.5. Since all the tasks meet their first deadlines under zero phasing, they are RMA schedulable according to Lehoczky's results.

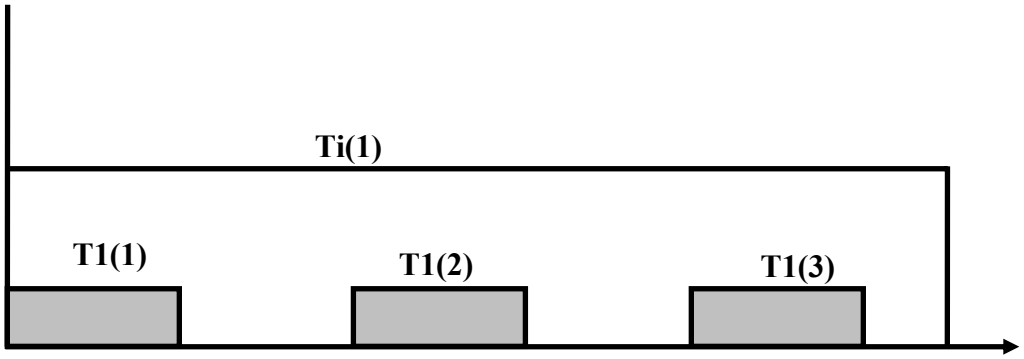


Fig. 30.6 Instances of T_1 over a single instance of T_i

Let us now try to derive a formal expression for this important result of Lehoczky. Let $\{T_1, T_2, \ldots, T_i\}$ be the set of tasks to be scheduled. Let us also assume that the tasks have been ordered in descending order of their priority. That is, task priorities are related as: $pr(T_1) > pr(T_2) > \ldots > pr(T_i)$, where $pr(T_i)$ denotes the priority of the task T_i. Observe that the task T_1 has the highest priority and task T_i has the least priority. This priority ordering can be assumed without any loss of generalization since the required priority ordering among an arbitrary collection of tasks can always be achieved by a simple renaming of the tasks. Consider that the task T_i arrives at the time instant 0. Consider the example shown in Fig. 30.6. During the first instance of the task T_i, three instances of the task T_1 have occurred. Each time T_1 occurs, T_i has to wait since T_1 has higher priority than T_i.

Let us now determine the exact number of times that T_1 occurs within a single instance of T_i. This is given by $\lceil p_i / p_1 \rceil$. Since T_1's execution time is e_1, then the total execution time required due to task T_1 before the deadline of T_i is $\lceil p_i / p_1 \rceil * e_1$. This expression can easily be generalized to consider the execution times all tasks having higher priority than T_i (i.e. T_1, T_2, …, T_{i-1}). Therefore, the time for which T_i will have to wait due to all its higher priority tasks can be expressed as:

$$\sum_{k=1}^{i-1} \lceil p_i / p_k \rceil * e_k \qquad \ldots(3.5/2.11)$$

Expression 3.5 gives the total time required to execute T_i's higher priority tasks for which Ti would have to wait. So, the task T_i would meet its first deadline, iff

$$e_i + \sum_{k=1}^{i-1} \lceil p_i / p_k \rceil * e_k \leq p_i \qquad \ldots(3.6/2.12)$$

That is, if the sum of the execution times of all higher priority tasks occurring before T_i's first deadline, and the execution time of the task itself is less than its period p_i, then T_i would complete before its first deadline. Note that in Expr. 3.6, we have implicitly assumed that the

task periods equal their respective deadlines, i.e. $p_i = d_i$. If $p_i < d_i$, then the Expr. 3.6 would need modifications as follows.

$$e_i + \sum_{k=1}^{i-1} \lceil d_i / p_k \rceil * e_k \leq d_i \quad \ldots(3.7/2.13)$$

Note that even if Expr. 3.7 is not satisfied, there is some possibility that the task set may still be schedulable. This might happen because in Expr. 3.7 we have considered zero phasing among all the tasks, which is the worst case. In a given problem, some tasks may have non-zero phasing. Therefore, even when a task set narrowly fails to meet Expr 3.7, there is some chance that it may in fact be schedulable under RMA. To understand why this is so, consider a task set where one particular task Ti fails Expr. 3.7, making the task set not schedulable. The task misses its deadline when it is in phase with all its higher priority task. However, when the task has non-zero phasing with at least some of its higher priority tasks, the task might actually meet its first deadline contrary to any negative results of the expression 3.7.

Let us now consider two examples to illustrate the applicability of the Lehoczky's results.

Example 8: Consider the following set of three periodic real-time tasks: T_1=(10,20), T_2=(15,60), T_3=(20,120) to be run on a uniprocessor. Determine whether the task set is schedulable under RMA.

Solution: First let us try the sufficiency test for RMA schedulability. By Expr. 3.4 (Liu and Layland test), the task set is schedulable if $\sum u_i \leq 0.78$.

$$\sum u_i = 10/20 + 15/60 + 20/120 = 0.91$$

This is greater than 0.78. Therefore, the given task set fails Liu and Layland test. Since Expr. 3.4 is a pessimistic test, we need to test further.

Let us now try Lehoczky's test. All the tasks T_1, T_2, T_3 are already ordered in decreasing order of their priorities.

Testing for task T_1:

Since e1 (10 msec) is less than d1 (20 msec), T_1 would meet its first deadline.

Testing for task T_2:

$$15 + \lceil 60/20 \rceil * 10 \leq 60 \text{ or } 15 + 30 = 45 \leq 60 \text{ msec}$$

The condition is satisfied. Therefore, T_2 would meet its first deadline.

Testing for Task T3:

$$20 + \lceil 120/20 \rceil * 10 + \lceil 120/60 \rceil * 15 = 20 + 60 + 30 = 110 \text{ msec}$$

This is less than T3's deadline of 120. Therefore T_3 would meet its first deadline.

Since all the three tasks meet their respective first deadlines, the task set is RMA schedulable according to Lehoczky's results.

Example 9: RMA is used to schedule a set of periodic hard real-time tasks in a system. Is it possible in this system that a higher priority task misses its deadline, whereas a lower priority task meets its deadlines? If your answer is negative, prove your denial. If your answer is affirmative, give an example involving two or three tasks scheduled using RMA where the lower priority task meets all its deadlines whereas the higher priority task misses its deadline.

Solution: Yes. It is possible that under RMA a higher priority task misses its deadline where as a lower priority task meets its deadline. We show this by constructing an example. Consider the following task set: T_1 = (e_1=15, p_1=20), T_2 = (e_2=6, p_2=35), T_3 = (e_3=3, p_3=100). For the given task set, it is easy to observe that $pr(T_1) > pr(T_2) > pr(T_3)$. That is, T1, T2, T3 are ordered in decreasing order of their priorities.

For this task set, T3 meets its deadline according to Lehoczky's test since

$$e_3 + \lceil p_3 / p_2 \rceil * e_2 + \lceil p_3 / p_1 \rceil * e_1 = 3 + (\lceil 100/35 \rceil * 6) + (\lceil 100/20 \rceil * 15)$$

$$= 3 + (3 * 6) + (5 * 15) = 96 \leq 100 \text{ msec.}$$

But, T_2 does not meet its deadline since

$$e_2 + \lceil p_2 / p_1 \rceil * e_1 = 6 + (\lceil 35/20 \rceil * 15) = 6 + (2 * 15) = 36 \text{ msec.}$$

This is greater than the deadline of T_2 (35 msec).

As a consequence of the results of Example 9, by observing that the lowest priority task of a given task set meets its first deadline, we can not conclude that the entire task set is RMA schedulable. On the contrary, it is necessary to check each task individually as to whether it meets its first deadline under zero phasing. If one finds that the lowest priority task meets its deadline, and concludes that the entire task set would be feasibly scheduled under RMA, he is likely to be flawed.

1.5.4. Achievable CPU Utilization

Liu and Layland's results (Expr. 3.4) bounded the CPU utilization below which a task set would be schedulable. It is clear from Expr. 3.4 and Fig. 30.10 that the Liu and Layland schedulability criterion is conservative and restricts the maximum achievable utilization due to any task set which can be feasibly scheduled under RMA to 0.69 when the number of tasks in the task set is large. However, (as you might have already guessed) this is a pessimistic figure. In fact, it has been found experimentally that for a large collection of tasks with independent periods, the maximum utilization below which a task set can feasibly be scheduled is on the average close to 88%.

For harmonic tasks, the maximum achievable utilization (for a task set to have a feasible schedule) can still be higher. In fact, if all the task periods are harmonically related, then even a task set having 100% utilization can be feasibly scheduled. Let us first understand when are the periods of a task set said to be harmonically related. The task periods in a task set are said to be harmonically related, iff for any two arbitrary tasks T_i and T_k in the task set, whenever $p_i > p_k$, it should imply that p_i is an integral multiple of p_k. That is, whenever $p_i > p_k$, it should be possible to express p_i as $n * p_k$ for some integer $n > 1$. In other words, p_k should squarely divide p_i. An example of a harmonically related task set is the following: T_1 = (5, 30), T2 = (8, 120), T3 = (12, 60).

It is easy to prove that a harmonically related task set with even 100% utilization can feasibly be scheduled.

1.5.5. Theorem 4

For a set of harmonically related tasks HS = {T_i}, the RMA schedulability criterion is given by $\sum_{i=1}^{n} u_i \leq 1$.

Proof: Let us assume that T_1, T_2, …, T_n be the tasks in the given task set. Let us further assume that the tasks in the task set T_1, T_2, …, T_n have been arranged in increasing order of their periods. That is, for any i and j, $pi < pj$ whenever $i < j$. If this relationship is not satisfied, then a simple renaming of the tasks can achieve this. Now, according to Expr. 3.6, a task T_i meets its deadline, if $e_i + \sum_{k=1}^{i-1} \lceil p_i / p_k \rceil * e_k \leq p_i$.

However, since the task set is harmonically related, p_i can be written as $m * p_k$ for some m. Using this, $\lceil p_i / p_k \rceil = p_i / p_k$. Now, Expr. 3.6 can be written as:

$$e_i + \sum_{k=1}^{i-1} (p_i / p_k) * e_k \leq p_i$$

For $T_i = T_n$, we can write, $e_n + \sum_{k=1}^{n-1} (p_n / p_k) * e_k \leq p_n$.

Dividing both sides of this expression by p_n, we get the required result.

Hence, the task set would be schedulable iff $\sum_{k=1}^{n} e_k / p_k \leq 1$ or $\sum_{i=1}^{n} u_i \leq 1$.

1.5.6. Advantages and Disadvantages of RMA

In this section, we first discuss the important advantages of RMA over EDF. We then point out some disadvantages of using RMA. As we had pointed out earlier, RMA is very commonly used for scheduling real-time tasks in practical applications. Basic support is available in almost all commercial real-time operating systems for developing applications using RMA. RMA is simple and efficient. RMA is also the optimal static priority task scheduling algorithm. Unlike EDF, it requires very few special data structures. Most commercial real-time operating systems support real-time (static) priority levels for tasks. Tasks having real-time priority levels are arranged in multilevel feedback queues (see Fig. 30.7). Among the tasks in a single level, these commercial real-time operating systems generally provide an option of either time-slicing and round-robin scheduling or FIFO scheduling.

RMA Transient Overload Handling: RMA possesses good transient overload handling capability. Good transient overload handling capability essentially means that, when a lower priority task does not complete within its planned completion time, it can not make any higher priority task to miss its deadline. Let us now examine how transient overload would affect a set of tasks scheduled under RMA. Will a delay in completion by a lower priority task affect a higher priority task? The answer is: ‘No’. A lower priority task even when it exceeds its planned execution time cannot make a higher priority task wait according to the basic principles of RMA – whenever a higher priority task is ready, it preempts any executing lower priority task. Thus, RMA is stable under transient overload and a lower priority task overshooting its completion time can not make a higher priority task to miss its deadline.

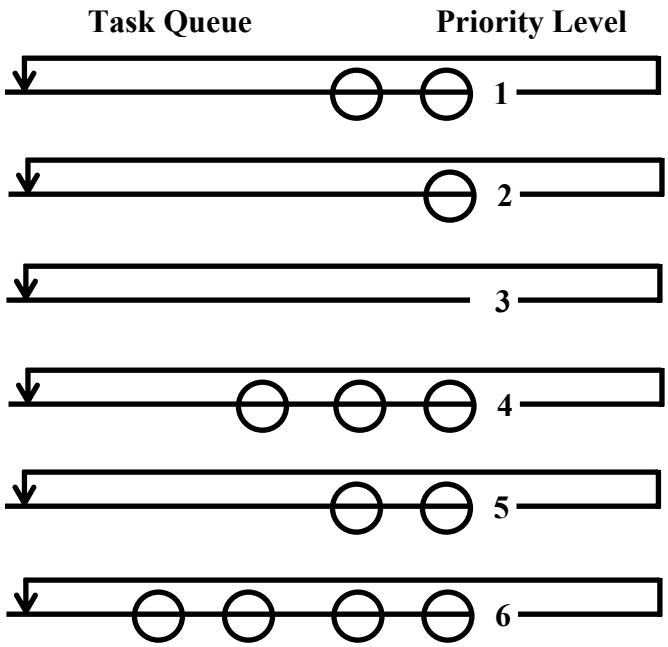


Fig. 30.7 Multi-Level Feedback Queue

The disadvantages of RMA include the following: It is very difficult to support aperiodic and sporadic tasks under RMA. Further, RMA is not optimal when task periods and deadlines differ.

1.6. Deadline Monotonic Algorithm (DMA)

RMA no longer remains an optimal scheduling algorithm for the periodic real-time tasks, when task deadlines and periods differ (i.e. $d_i \neq p_i$) for some tasks in the task set to be scheduled. For such task sets, Deadline Monotonic Algorithm (DMA) turns out to be more proficient than RMA. DMA is essentially a variant of RMA and assigns priorities to tasks based on their deadlines, rather than assigning priorities based on task periods as done in RMA. DMA assigns higher priorities to tasks with shorter deadlines. When the relative deadline of every task is proportional to its period, RMA and DMA produce identical solutions. When the relative deadlines are arbitrary, DMA is more proficient than RMA in the sense that it can sometimes produce a feasible schedule when RMA fails. On the other hand, RMA always fails when DMA fails. We now illustrate our discussions using an example task set that is DMA schedulable but not RMA schedulable.

Example 10: Is the following task set schedulable by DMA? Also check whether it is schedulable using RMA. T_1 = (e_1=10, p_1=50, d_1=35), T_2 = (e_2=15, p_2=100, d_1=20), T_3 = (e_3=20, p_3=200, d_1=200) [time in msec].

Solution: First, let us check RMA schedulability of the given set of tasks, by checking the Lehoczky's criterion. The tasks are already ordered in descending order of their priorities.
Checking for T_1:
10 msec < 35 msec. Hence, T_1 would meet its first deadline.
Checking for T_2:

(10 + 15) > 20 (exceeds deadline)

Thus, T_2 will miss its first deadline. Hence, the given task set can not be feasibly scheduled under RMA.

Now let us check the schedulability using DMA:

Under DMA, the priority ordering of the tasks is as follows: $pr(T_2) > pr(T_1) > pr(T_3)$.

Checking for T_2:

15 msec < 20 msec. Hence, T_2 will meet its first deadline.

Checking for T_1:

$$(15 + 10) < 35$$

Hence T_1 will meet its first deadline.

Checking for T_3:

$$(20 + 30 + 40) < 200$$

Therefore, T_3 will meet its deadline.

Therefore, the given task set is schedulable under DMA but not under RMA.

1.7. Context Switching Overhead

So far, while determining schedulability of a task set, we had ignored the overheads incurred on account of context switching. Let us now investigate the effect of context switching overhead on schedulability of tasks under RMA.

It is easy to realize that under RMA, whenever a task arrives, it preempts at most one task – the task that is currently running. From this observation, it can be concluded that in the worst-case, each task incurs at most two context switches under RMA. One when it preempts the currently running task. And the other when it completes possibly the task that was preempted or some other task is dispatched to run. Of course, a task may incur just one context switching overhead, if it does not preempt any task. For example, it arrives when the processor is idle or when a higher priority task was running. However, we need to consider two context switches for every task, if we try to determine the worst-case context switching overhead.

For simplicity we can assume that context switching time is constant, and equals 'c' milliseconds where 'c' is a constant. From this, it follows that the net effect of context switches is to increase the execution time e_i of each task T_i to at most $e_i + 2*c$. It is therefore clear that in order to take context switching time into consideration, in all schedulability computations, we need to replace e_i by $e_i + 2*c$ for each T_i.

Example 11: Check whether the following set of periodic real-time tasks is schedulable under RMA on a uniprocessor: $T_1 = (e_1=20, p_1=100)$, $T_2 = (e_2=30, p_2=150)$, $T_3 = (e_3=90, p_3=200)$. Assume that context switching overhead does not exceed 1 msec, and is to be taken into account in schedulability computations.

Solution: The net effect of context switches is to increase the execution time of each task by two context switching times. Therefore, the utilization due to the task set is:

$$\sum_{i=1}^{3} u_i = 22/100 + 32/150 + 92/200 = 0.89$$

Since $\sum_{i=1}^{3} u_i > 0.78$, the task set is not RMA schedulable according to the Liu and Layland test.

Let us try Lehoczky's test. The tasks are already ordered in descending order of their priorities.

Checking for task T_1:

$$22 < 100$$

The condition is satisfied; therefore T_1 meets its first deadline.
Checking for task T_2:

$$(22*2) + 32 < 150$$

The condition is satisfied; therefore T_2 meets its first deadline.
Checking for task T_3:

$$(22*2) + (32*2) + 90 < 200.$$

The condition is satisfied; therefore T_3 meets its first deadline.
Therefore, the task set can be feasibly scheduled under RMA even when context switching overhead is taken into consideration.

1.8. Self Suspension

A task might cause its self-suspension, when it performs its input/output operations or when it waits for some events/conditions to occur. When a task self suspends itself, the operating system removes it from the ready queue, places it in the blocked queue, and takes up the next eligible task for scheduling. Thus, self-suspension introduces an additional scheduling point, which we did not consider in the earlier sections. Accordingly, we need to augment our definition of a scheduling point given in Sec. 2.3.1 (lesson 2).

> In event-driven scheduling, the scheduling points are defined by task completion, task arrival, and self-suspension events.

Let us now determine the effect of self-suspension on the schedulability of a task set. Let us consider a set of periodic real-time tasks $\{T_1, T_2, \ldots, T_n\}$, which have been arranged in the increasing order of their priorities (or decreasing order of their periods). Let the worst case self-suspension time of a task T_i is b_i. Let the delay that the task T_i might incur due to its own self-suspension and the self-suspension of all higher priority tasks be bt_i. Then, bt_i can be expressed as:

$$bt_i = b_i + \sum_{k=1}^{i-1} min(e_k, b_k) \qquad \ldots(3.8/2.15)$$

Self-suspension of a higher priority task T_k may affect the response time of a lower priority task T_i by as much as its execution time e_k if $e_k < b_k$. This worst case delay might occur when the higher priority task after self-suspension starts its execution exactly at the time instant the lower priority task would have otherwise executed. That is, after self-suspension, the execution of the higher priority task overlaps with the lower priority task, with which it would otherwise not have overlapped. However, if $e_k > b_k$, then the self suspension of a higher priority task can delay a lower priority task by at most b_k, since the maximum overlap period of the execution of a higher priority task due to self-suspension is restricted to b_k.

Note that in a system where some of the tasks are non preemptable, the effect of self suspension is much more severe than that computed by Expr.3.8. The reason is that, every time a processor self suspends itself, it loses the processor. It may be blocked by a non-preemptive lower priority task after the completion of self-suspension. Thus, in a non-preemptable scenario, a task incurs delays due to self-suspension of itself and its higher priority tasks, and also the delay caused due to non-preemptable lower priority tasks. Obviously, a task can not get delayed due to the self-suspension of a lower priority non-preemptable task.

The RMA task schedulability condition of Liu and Layland (Expr. 3.4) needs to change when we consider the effect of self-suspension of tasks. To consider the effect of self-suspension in Expr. 3.4, we need to substitute e_i by $(e_i + bt_i)$. If we consider the effect of self-suspension on task completion time, the Lehoczky criterion (Expr. 3.6) would also have to be generalized:

$$e_i + bt_i + \sum_{k=1}^{i-1} \lceil p_i / p_k \rceil * e_k \leq p_i \quad \ldots (3.9/2.16)$$

We have so far implicitly assumed that a task undergoes at most a single self suspension. However, if a task undergoes multiple self-suspensions, then expression 3.9 we derived above, would need to be changed. We leave this as an exercise for the reader.

Example 14: Consider the following set of periodic real-time tasks: $T_1 = (e_1=10, p_1=50)$, $T_2 = (e_2=25, p_2=150)$, $T_3 = (e_3=50, p_3=200)$ [all in msec]. Assume that the self-suspension times of T_1, T_2, and T_3 are 3 msec, 3 msec, and 5 msec, respectively. Determine whether the tasks would meet their respective deadlines, if scheduled using RMA.

Solution: The tasks are already ordered in descending order of their priorities. By using the generalized Lehoczky's condition given by Expr. 3.9, we get:
For T_1 to be schedulable:

$$(10 + 3) < 50$$

Therefore T_1 would meet its first deadline.
For T_2 to be schedulable:

$$(25 + 6 + 10*3) < 150$$

Therefore, T_2 meets its first deadline.
For T_3 to be schedulable:

$$(50 + 11 + (10*4 + 25*2)) < 200$$

This inequality is also satisfied. Therefore, T_3 would also meet its first deadline.
It can therefore be concluded that the given task set is schedulable under RMA even when self-suspension of tasks is considered.

1.9. Self Suspension with Context Switching Overhead

Let us examine the effect of context switches on the generalized Lehoczky's test (Expr.3.9) for schedulability of a task set, which takes self-suspension by tasks into account. In a fixed priority preemptable system, each task preempts at most one other task if there is no self suspension. Therefore, each task suffers at most two context switches – one context switch when it starts and another when it completes. It is easy to realize that any time when a task self-suspends, it causes at most two additional context switches. Using a similar reasoning, we can determine that when each task is allowed to self-suspend twice, additional four context switching overheads are incurred. Let us denote the maximum context switch time as *c*. The effect of a single self-suspension of tasks is to effectively increase the execution time of each task T_i in the worst case from e_i to $(e_i + 4*c)$. Thus, context switching overhead in the presence of a single self-suspension of tasks can be taken care of by replacing the execution time of a task T_i by $(e_i + 4*c)$ in Expr. 3.9. We can easily extend this argument to consider two, three, or more self-suspensions.

1.10. Exercises

1. State whether the following assertions are True or False. Write one or two sentences to justify your choice in each case.
   a. When RMA is used for scheduling a set of hard real-time periodic tasks, the upper bound on achievable utilization improves as the number in tasks in the system being developed increases.

b. If a set of periodic real-time tasks fails Lehoczky's test, then it can safely be concluded that this task set can not be feasibly scheduled under RMA.
c. A time-sliced round-robin scheduler uses preemptive scheduling.
d. RMA is an optimal static priority scheduling algorithm to schedule a set of periodic real-time tasks on a non-preemptive operating system.
e. Self-suspension of tasks impacts the worst case response times of the individual tasks much more adversely when preemption of tasks is supported by the operating system compared to the case when preemption is not supported.
f. When a set of periodic real-time tasks is being scheduled using RMA, it can not be the case that a lower priority task meets its deadline, whereas some higher priority task does not.
g. EDF (Earliest Deadline First) algorithm possesses good transient overload handling capability.
h. A time-sliced round robin scheduler is an example of a non-preemptive scheduler.
i. EDF algorithm is an optimal algorithm for scheduling hard real-time tasks on a uniprocessor when the task set is a mixture of periodic and aperiodic tasks.
j. In a non-preemptable operating system employing RMA scheduling for a set of real-time periodic tasks, self-suspension of a higher priority task (due to I/O etc.) may increase the response time of a lower priority task.
k. The worst-case response time for a task occurs when it is out of phase with its higher priority tasks.
l. Good real-time task scheduling algorithms ensure fairness to real-time tasks while scheduling.

2. State whether the following assertions are True or False. Write one or two sentences to justify your choice in each case.
   a. The EDF algorithm is optimal for scheduling real-time tasks in a uniprocessor in a non-preemptive environment.
   b. When RMA is used to schedule a set of hard real-time periodic tasks in a uniprocessor environment, if the processor becomes overloaded any time during system execution due to overrun by the lowest priority task, it would be very difficult to predict which task would miss its deadline.
   c. While scheduling a set of real-time periodic tasks whose task periods are harmonically related, the upper bound on the achievable CPU utilization is the same for both EDF and RMA algorithms.
   d. In a non-preemptive event-driven task scheduler, scheduling decisions are made only at the arrival and completion of tasks.
   e. The following is the correct arrangement of the three major classes of real-time scheduling algorithms in ascending order of their run-time overheads.
      - static priority preemptive scheduling algorithms
      - table-driven algorithms
      - dynamic priority algorithms
   f. While scheduling a set of independent hard real-time periodic tasks on a uniprocessor, RMA can be as proficient as EDF under some constraints on the task set.
   g. RMA should be preferred over the time-sliced round-robin algorithm for scheduling a set of soft real-time tasks on a uniprocessor.

h. Under RMA, the achievable utilization of a set of hard real-time periodic tasks would drop when task periods are multiples of each other compared to the case when they are not.

i. RMA scheduling of a set of real-time periodic tasks using the Liu and Layland criterion might produce infeasible schedules when the task periods are different from the task deadlines.

3. What do you understand by scheduling point of a task scheduling algorithm? How are the scheduling points determined in (i) clock-driven, (ii) event-driven, (iii) hybrid schedulers? How will your definition of scheduling points for the three classes of schedulers change when (a) self-suspension of tasks, and (b) context switching overheads of tasks are taken into account.
4. What do you understand by jitter associated with a periodic task? How are these jitters caused?
5. Is EDF algorithm used for scheduling real-time tasks a dynamic priority scheduling algorithm? Does EDF compute any priority value of tasks any time? If you answer affirmatively, then explain when is the priority computed and how is it computed. If you answer in negative, then explain the concept of priority in EDF.
6. What is the sufficient condition for EDF schedulability of a set of periodic tasks whose period and deadline are different? Construct an example involving a set of three periodic tasks whose period differ from their respective deadlines such that the task set fails the sufficient condition and yet is EDF schedulable. Verify your answer. Show all your intermediate steps.
7. A preemptive static priority real-time task scheduler is used to schedule two periodic tasks T1 and T2 with the following characteristics:

Task	**Phase** mSec	**Execution Time** mSec	**Relative Deadline** mSec	**Period** mSec
T_1	0	10	20	20
T_2	0	20	50	50

Assume that T1 has higher priority than T2. A background task arrives at time 0 and would require 1000mSec to complete. Compute the completion time of the background task assuming that context switching takes no more than 0.5 mSec.

8. Assume that a preemptive priority-based system consists of three periodic foreground tasks T1, T2, and T3 with the following characteristics:

Task	**Phase** mSec	**Execution Time** mSec	**Relative Deadline** mSec	**Period** mSec
T_1	0	20	100	100
T_2	0	30	150	150
T_3	0	30	300	300

T1 has higher priority than T2 and T2 has higher priority than T3. A background task Tb arrives at time 0 and would require 2000mSec to complete. Compute the completion time of the background task Tb assuming that context switching time takes no more than 1 mSec.

9. Consider the following set of four independent real-time periodic tasks.

Task	**Start Time** msec	**Processing Time** msec	**Period** msec
T_1	20	25	150
T_2	40	10	50
T_3	20	15	50
T_4	60	50	200

Assume that task T3 is more critical than task T2. Check whether the task set can be feasibly scheduled using RMA.

10. What is the worst case response time of the background task of a system in which the background task requires 1000 msec to complete? There are two foreground tasks. The higher priority foreground task executes once every 100mSec and each time requires 25mSec to complete. The lower priority foreground task executes once every 50 msec and requires 15 msec to complete. Context switching requires no more than 1 msec.
11. Construct an example involving more than one hard real-time periodic task whose aggregate processor utilization is 1, and yet schedulable under RMA.
12. Determine whether the following set of periodic tasks is schedulable on a uniprocessor using DMA (Deadline Monotonic Algorithm). Show all intermediate steps in your computation.

Task	**Start Time** mSec	**Processing Time** mSec	**Period** mSec	**Deadline** mSec
T_1	20	25	150	140
T_2	60	10	60	40
T_3	40	20	200	120
T_4	25	10	80	25

13. Consider the following set of three independent real-time periodic tasks.

Task	**Start Time** mSec	**Processing Time** mSec	**Period** mSec	**Deadline** mSec
T_1	20	25	150	100
T_2	60	10	50	30
T_3	40	50	200	150

Determine whether the task set is schedulable on a uniprocessor using EDF. Show all intermediate steps in your computation.

14. Determine whether the following set of periodic real-time tasks is schedulable on a uniprocessor using RMA. Show the intermediate steps in your computation. Is RMA optimal when the task deadlines differ from the task periods?

Task	Start Time mSec	Processing Time mSec	Period mSec	Deadline mSec
T_1	20	25	150	100
T_2	40	7	40	40
T_3	60	10	60	50
T_4	25	10	30	20

15. Construct an example involving two periodic real-time tasks which can be feasibly scheduled by both RMA and EDF, but the schedule generated by RMA differs from that generated by EDF. Draw the two schedules on a time line and highlight how the two schedules differ. Consider the two tasks such that for each task:
    a. the period is the same as deadline
    b. period is different from deadline
16. Can multiprocessor real-time task scheduling algorithms be used satisfactorily in distributed systems. Explain the basic difference between the characteristics of a real-time task scheduling algorithm for multiprocessors and a real-time task scheduling algorithm for applications running on distributed systems.
17. Construct an example involving a set of hard real-time periodic tasks that are not schedulable under RMA but could be feasibly scheduled by DMA. Verify your answer, showing all intermediate steps.
18. Three hard real-time periodic tasks T_1 = (50, 100, 100), T_2 = (70, 200, 200), and T_3 = (60, 400, 400) [time in msec] are to be scheduled on a uniprocessor using RMA. Can the task set be feasibly be scheduled? Suppose context switch overhead of 1 millisecond is to be taken into account, determine the schedulability.
19. Consider the following set of three real-time periodic tasks.

Task	Start Time mSec	Processing Time mSec	Period mSec	Deadline mSec
T_1	20	25	150	100
T_2	40	10	50	50
T_3	60	50	200	200

    a. Check whether the three given tasks are schedulable under RMA. Show all intermediate steps in your computation.
    b. Assuming that each context switch incurs an overhead of 1 msec, determine whether the tasks are schedulable under RMA. Also, determine the average context switching overhead per unit of task execution.
    c. Assume that T1, T2, and T3 self-suspend for 10 msec, 20 msec, and 15 msec respectively. Determine whether the task set remains schedulable under RMA. The context switching overhead of 1 msec should be considered in your result. You can assume that each task undergoes self-suspension only once during each of its execution.
    d. Assuming that T1 and T2 are assigned the same priority value, determine the additional delay in response time that T2 would incur compared to the case when they are assigned distinct priorities. Ignore the self-suspension times and the context switch overhead for this part of the question.

Module 6

Embedded System Software

Lesson 31

Concepts in Real-Time Operating Systems

Specific Instructional Objectives

At the end of this lesson, the student would be able to:

- Know the clock and time services provided by a Real-Time OS
- Get an overview of the features that a Real-Time OS is required to support
- Investigate Unix as a Real-Time operating System
- Know the shortcomings on traditional Unix in Real-Time applications
- Know the different approaches taken to make Unix suitable for real-time applications
- Investigate Windows as a Real-Time operating System
- Know the features of Windows NT desirable for Real-Time applications
- Know the shortcomings of Windows NT
- Compare Windows with Unix OS

1. Introduction

In the last three lessons, we discussed the important real-time task scheduling techniques. We highlighted that timely production of results in accordance to a physical clock is vital to the satisfactory operation of a real-time system. We had also pointed out that real-time operating systems are primarily responsible for ensuring that every real-time task meets its timeliness requirements. A real-time operating system in turn achieves this by using appropriate task scheduling techniques. Normally real-time operating systems provide flexibility to the programmers to select an appropriate scheduling policy among several supported policies. Deployment of an appropriate task scheduling technique out of the supported techniques is therefore an important concern for every real-time programmer. To be able to determine the suitability of a scheduling algorithm for a given problem, a thorough understanding of the characteristics of various real-time task scheduling algorithms is important. We therefore had a rather elaborate discussion on real-time task scheduling techniques and certain related issues such as sharing of critical resources and handling task dependencies.

In this lesson, we examine the important features that a real-time operating system is expected to support. We start by discussing the time service supports provided by the real-time operating systems, since accurate and high precision clocks are very important to the successful operation any real- time application. Next, we point out the important features that a real-time operating system needs to support. Finally, we discuss the issues that would arise if we attempt to use a general purpose operating system such as UNIX or Windows in real-time applications.

1.1. Time Services

Clocks and time services are among some of the basic facilities provided to programmers by every real-time operating system. The time services provided by an operating system are based on a software clock called the system clock maintained by the operating system. The system clock is maintained by the kernel based on the interrupts received from the hardware clock. Since hard real-time systems usually have timing constraints in the micro seconds range, the

system clock should have sufficiently fine resolution[1] to support the necessary time services. However, designers of real-time operating systems find it very difficult to support very fine resolution system clocks. In current technology, the resolution of hardware clocks is usually finer than a nanosecond (contemporary processor speeds exceed 3GHz). But, the clock resolution being made available by modern real-time operating systems to the programmers is of the order of several milliseconds or worse. Let us first investigate why real-time operating system designers find it difficult to maintain system clocks with sufficiently fine resolution. We then examine various time services that are built based on the system clock, and made available to the real-time programmers.

The hardware clock periodically generates interrupts (often called time service interrupts). After each clock interrupt, the kernel updates the software clock and also performs certain other work (explained in Sec 4.1.1). A thread can get the current time reading of the system clock by invoking a system call supported by the operating system (such as the POSIX clock-gettime()). The finer the resolution of the clock, the more frequent need to be the time service interrupts and larger is the amount of processor time the kernel spends in responding to these interrupts. This overhead places a limitation on how fine is the system clock resolution a computer can support. Another issue that caps the resolution of the system clock is the response time of the clock-gettime() system call is not deterministic. In fact, every system call (or for that matter, a function call) has some associated jitter. The problem gets aggravated in the following situation. The jitter is caused on account of interrupts having higher priority than system calls. When an interrupt occurs, the processing of a system call is stalled. Also, the preemption time of system calls can vary because many operating systems disable interrupts while processing a system call. The variation in the response time (jitter) introduces an error in the accuracy of the time value that the calling thread gets from the kernel. Remember that jitter was defined as the difference between the worst-case response time and the best case response time (see Sec. 2.3.1). In commercially available operating systems, jitters associated with system calls can be several milliseconds. A software clock resolution finer than this error, is therefore not meaningful.

We now examine the different activities that are carried out by a handler routine after a clock interrupt occurs. Subsequently, we discuss how sufficient fine resolution can be provided in the presence of jitter in function calls.

1.1.1. Clock Interrupt Processing

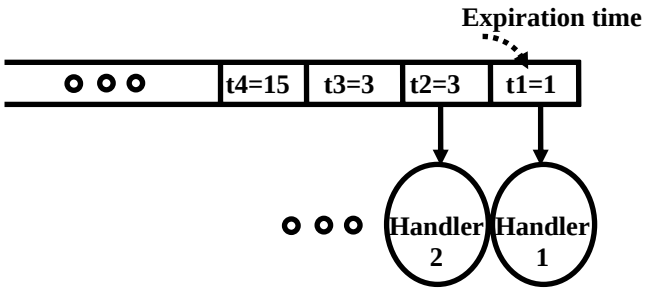


Fig. 31.1 Structure of a Timer Queue

[1] Clock resolution denotes the time granularity provided by the clock of a computer. It corresponds to the duration of time that elapses between two successive clock ticks.

Each time a clock interrupt occurs, besides incrementing the software clock, the handler routine carries out the following activities:

Process timer events: Real-time operating systems maintain either per-process timer queues or a single system-wide timer queue. The structure of such a timer queue has been shown in Fig. 31.1. A timer queue contains all timers arranged in order of their expiration times. Each timer is associated with a handler routine. The handler routine is the function that should be invoked when the timer expires. At each clock interrupt, the kernel checks the timer data structures in the timer queue to see if any timer event has occurred. If it finds that a timer event has occurred, then it queues the corresponding handler routine in the ready queue.

Update ready list: Since the occurrence of the last clock event, some tasks might have arrived or become ready due to the fulfillment of certain conditions they were waiting for. The tasks in the wait queue are checked, the tasks which are found to have become ready, are queued in the ready queue. If a task having higher priority than the currently running task is found to have become ready, then the currently running task is preempted and the scheduler is invoked.

Update execution budget: At each clock interrupt, the scheduler decrements the time slice (budget) remaining for the executing task. If the remaining budget becomes zero and the task is not complete, then the task is preempted, the scheduler is invoked to select another task to run.

1.1.2. Providing High Clock Resolution

We had pointed out in Sec. 4.1 that there are two main difficulties in providing a high resolution timer. First, the overhead associated with processing the clock interrupt becomes excessive. Secondly, the jitter associated with the time lookup system call (clock-gettime()) is often of the order of several milliseconds. Therefore, it is not useful to provide a clock with a resolution any finer than this. However, some real-time applications need to deal with timing constraints of the order of a few nanoseconds. Is it at all possible to support time measurement with nanosecond resolution? A way to provide sufficiently fine clock resolution is by mapping a hardware clock into the address space of applications. An application can then read the hardware clock directly (through a normal memory read operation) without having to make a system call. On a Pentium processor, a user thread can be made to read the Pentium time stamp counter. This counter starts at 0 when the system is powered up and increments after each processor cycle. At today's processor speed, this means that during every nanosecond interval, the counter increments several times.

However, making the hardware clock readable by an application significantly reduces the portability of the application. Processors other than Pentium may not have a high resolution counter, and certainly the memory address map and resolution would differ.

1.1.3. Timers

We had pointed out that timer service is a vital service that is provided to applications by all real-time operating systems. Real-time operating systems normally support two main types of timers: periodic timers and aperiodic (or one shot) timers. We now discuss some basic concepts about these two types of timers.

Periodic Timers: Periodic timers are used mainly for sampling events at regular intervals or performing some activities periodically. Once a periodic timer is set, each time after it expires the corresponding handler routine is invoked, it gets reinserted into the timer queue. For example, a periodic timer may be set to 100 msec and its handler set to poll the temperature sensor after every 100 msec interval.

Aperiodic (or One Shot) Timers: These timers are set to expire only once. Watchdog timers are popular examples of one shot timers.

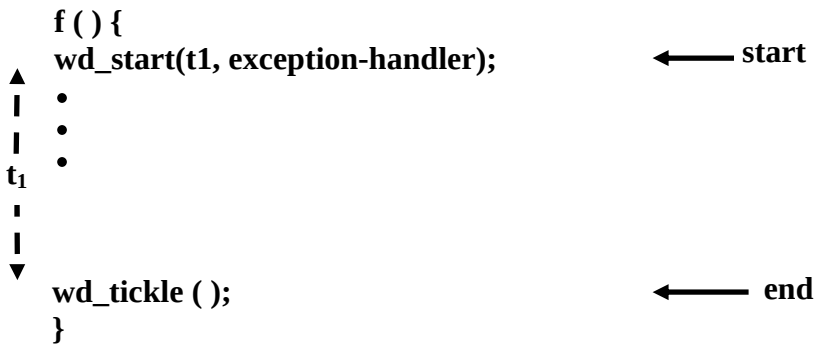


Fig. 31.2 Use of a Watchdog Timer

Watchdog timers are used extensively in real-time programs to detect when a task misses its deadline, and then to initiate exception handling procedures upon a deadline miss. An example use of a watchdog timer has been illustrated in Fig. 31.2. In Fig. 31.2, a watchdog timer is set at the start of a certain critical function f*()* through a *wd_start(t1)* call. The *wd_start(t1)* call sets the watch dog timer to expire by the specified deadline (t_1) of the starting of the task. If the function *f()* does not complete even after t_1 time units have elapsed, then the watchdog timer fires, indicating that the task deadline must have been missed and the exception handling procedure is initiated. In case the task completes before the watchdog timer expires (i.e. the task completes within its deadline), then the watchdog timer is reset using a *wd_ tickle()* call.

1.2. Features of a Real-Time Operating System

Before discussing about commercial real-time operating systems, we must clearly understand the features normally expected of a real-time operating system and also let us compare different real-time operating systems. This would also let us understand the differences between a traditional operating system and a real-time operating system. In the following, we identify some important features required of a real-time operating system, and especially those that are normally absent in traditional operating systems.

Clock and Timer Support: Clock and timer services with adequate resolution are one of the most important issues in real-time programming. Hard real-time application development often requires support of timer services with resolution of the order of a few microseconds. And even finer resolution may be required in case of certain special applications. Clocks and timers are a vital part of every real-time operating system. On the other hand, traditional operating systems often do not provide time services with sufficiently high resolution.

Real-Time Priority Levels: A real-time operating system must support static priority levels. A priority level supported by an operating system is called static, when once the programmer assigns a priority value to a task, the operating system does not change it by itself. Static priority levels are also called *real-time priority levels*. This is because, as we discuss in section 4.3, all traditional operating systems dynamically change the priority levels of tasks from programmer assigned values to maximize system throughput. Such priority levels that are changed by the operating system dynamically are obviously not static priorities.

Fast Task Preemption: For successful operation of a real-time application, whenever a high priority critical task arrives, an executing low priority task should be made to instantly yield the CPU to it. The time duration for which a higher priority task waits before it is allowed to execute is quantitatively expressed as the corresponding *task preemption time.* Contemporary real-time operating systems have task preemption times of the order of a few micro seconds. However, in traditional operating systems, the worst case task preemption time is usually of the order of a second. We discuss in the next section that this significantly large latency is caused by a non-preemptive kernel. It goes without saying that a real-time operating system needs to have a preemptive kernel and should have task preemption times of the order of a few micro seconds.

Predictable and Fast Interrupt Latency: Interrupt latency is defined as the time delay between the occurrence of an interrupt and the running of the corresponding ISR (Interrupt Service Routine). In real-time operating systems, the upper bound on interrupt latency must be bounded and is expected to be less than a few micro seconds. The way low interrupt latency is achieved, is by performing bulk of the activities of ISR in a deferred procedure call (DPC). A DPC is essentially a task that performs most of the ISR activity. A DPC is executed later at a certain priority value. Further, support for nested interrupts are usually desired. That is, a real-time operating system should not only be preemptive while executing kernel routines, but should be preemptive during interrupt servicing as well. This is especially important for hard real-time applications with sub-microsecond timing requirements.

Support for Resource Sharing Among Real-Time Tasks: If real- time tasks are allowed to share critical resources among themselves using the traditional resource sharing techniques, then the response times of tasks can become unbounded leading to deadline misses. This is one compelling reason as to why every commercial real-time operating system should at the minimum provide the basic priority inheritance mechanism. Support of priority ceiling protocol (PCP) is also desirable, if large and moderate sized applications are to be supported.

Requirements on Memory Management: As far as general-purpose operating systems are concerned, it is rare to find one that does not support virtual memory and memory protection features. However, embedded real-time operating systems almost never support these features. Only those that are meant for large and complex applications do. Real-time operating systems for large and medium sized applications are expected to provide virtual memory support, not only to meet the memory demands of the heavy weight tasks of the application, but to let the memory demanding non-real-time applications such as text editors, e-mail software, etc. to also run on the same platform. Virtual memory reduces the average memory access time, but degrades the worst-case memory access time. The penalty of using virtual memory is the overhead associated with storing the address translation table and performing the virtual to physical address translations. Moreover, fetching pages from the secondary memory on demand incurs significant latency. Therefore, operating systems supporting virtual memory must provide the real-time

applications with some means of controlling paging, such as memory locking. Memory locking prevents a page from being swapped from memory to hard disk. In the absence of memory locking feature, memory access times of even critical real-time tasks can show large jitter, as the access time would greatly depend on whether the required page is in the physical memory or has been swapped out.

Memory protection is another important issue that needs to be carefully considered. Lack of support for memory protection among tasks leads to a single address space for the tasks. Arguments for having only a single address space include simplicity, saving memory bits, and light weight system calls. For small embedded applications, the overhead of a few Kilo Bytes of memory per process can be unacceptable. However, when no memory protection is provided by the operating system, the cost of developing and testing a program without memory protection becomes very high when the complexity of the application increases. Also, maintenance cost increases as any change in one module would require retesting the entire system.

Embedded real-time operating systems usually do not support virtual memory. Embedded real-time operating systems create physically contiguous blocks of memory for an application upon request. However, memory fragmentation is a potential problem for a system that does not support virtual memory. Also, memory protection becomes difficult to support a non-virtual memory management system. For this reason, in many embedded systems, the kernel and the user processes execute in the same space, i.e. there is no memory protection. Hence, a system call and a function call within an application are indistinguishable. This makes debugging applications difficult, since a run away pointer can corrupt the operating system code, making the system "freeze".

Additional Requirements for Embedded Real-Time Operating Systems: Embedded applications usually have constraints on cost, size, and power consumption. Embedded real-time operating systems should be capable of diskless operation, since many times disks are either too bulky to use, or increase the cost of deployment. Further, embedded operating systems should minimize total power consumption of the system. Embedded operating systems usually reside on ROM. For certain applications which require faster response, it may be necessary to run the real-time operating system on a RAM. Since the access time of a RAM is lower than that of a ROM, this would result in faster execution. Irrespective of whether ROM or RAM is used, all ICs are expensive. Therefore, for real-time operating systems for embedded applications it is desirable to have as small a foot print (memory usage) as possible. Since embedded products are typically manufactured large scale, every rupee saved on memory and other hardware requirements impacts millions in profit.

1.3. Unix as a Real-Time Operating System

Unix is a popular general purpose operating system that was originally developed for the mainframe computers. However, UNIX and its variants have now permeated to desktop and even handheld computers. Since UNIX and its variants inexpensive and are widely available, it is worthwhile to investigate whether Unix can be used in real-time applications. This investigation would lead us to some significant findings and would give us some crucial insights into the current Unix-based real-time operating systems that are currently commercially available.

The traditional UNIX operating system suffers from several shortcomings when used in real-time applications.

We elaborate these problems in the following two subsections.

The two most troublesome problems that a real-time programmer faces while using Unix for real-time applications include non-preemptive Unix kernel and dynamically changing priority of tasks.

1.3.1. Non-Preemptive Kernel

One of the biggest problems that real-time programmers face while using Unix for real-time application development is that Unix kernel cannot be preempted. That is, all interrupts are disabled when any operating system routine runs. To set things in proper perspective, let us elaborate this issue.

Application programs invoke operating system services through *system calls*. Examples of system calls include the operating system services for creating a process, interprocess communication, I/O operations, etc. After a system call is invoked by an application, the arguments given by the application while invoking the system call are checked. Next, a special instruction called a trap (or a software interrupt) is executed. As soon as the trap instruction is executed, the handler routine changes the processor state from *user mode* to *kernel mode* (or *supervisor mode*), and the execution of the required kernel routine starts. The change of mode during a system call has schematically been depicted in Fig. 31.3.

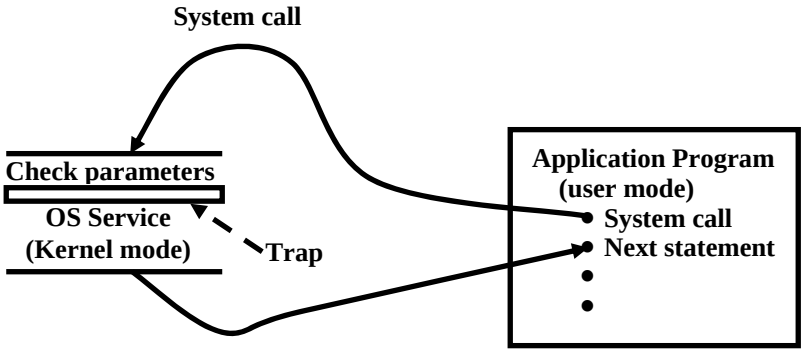


Fig. 31.3 Invocation of an Operating System Service through System Call

At the risk of digressing from the focus of this discussion, let us understand an important operating systems concept. Certain operations such as handling devices, creating processes, file operations, etc., need to be done in the kernel mode only. That is, application programs are prevented from carrying out these operations, and need to request the operating system (through a system call) to carry out the required operation. This restriction enables the kernel to enforce discipline among different programs in accessing these objects. In case such operations are not performed in the kernel mode, different application programs might interfere with each other's operation. An example of an operating system where all operations were performed in user mode is the once popular operating system DOS (though DOS is nearly obsolete now). In DOS, application programs are free to carry out any operation in user mode[2], including crashing the system by deleting the system files. The instability this can bring about is clearly unacceptable in real-time environment, and is usually considered insufficient in general applications as well.

[2] In fact, in DOS there is only one mode of operation, i.e. kernel mode and user mode are indistinguishable.

A process running in kernel mode cannot be preempted by other processes. In other words, the Unix kernel is *non-preemptive*. On the other hand, the Unix system does preempt processes running in the user mode. A consequence of this is that even when a low priority process makes a system call, the high priority processes would have to wait until the system call completes. The longest system calls may take up to several hundreds of milliseconds to complete. Worst-case preemption times of several hundreds of milliseconds can easily cause, high priority tasks with short deadlines of the order of a few milliseconds to miss their deadlines.

Let us now investigate, why the Unix kernel was designed to be non-preemptive in the first place. Whenever an operating system routine starts to execute, all interrupts are disabled. The interrupts are enabled only after the operating system routine completes. This was a very efficient way of preserving the integrity of the kernel data structures. It saved the overheads associated with setting and releasing locks and resulted in lower average task preemption times. Though a non-preemptive kernel results in worst-case task response time of upto a second, it was acceptable to Unix designers. At that time, the Unix designers did not foresee usage of Unix in real-time applications. Of course, it could have been possible to ensure correctness of kernel data structures by using locks at appropriate places rather than disabling interrupts, but it would have resulted in increasing the average task preemption time. In Sec. 4.4.4 we investigate how modern real-time operating systems make the kernel preemptive without unduly increasing the task preemption time.

1.3.2. Dynamic Priority Levels

In Unix systems real-time tasks can not be assigned static priority values. Soon after a programmer sets a priority value, the operating system alters it. This makes it very difficult to schedule real-time tasks using algorithms such as RMA or EDF, since both these schedulers assume that once task priorities are assigned, it should not be altered by any other parts of the operating system. It is instructive to understand why Unix dynamically changes the priority values of tasks in the first place.

Unix uses round-robin scheduling with multilevel feedback. This scheduler arranges tasks in multilevel queues as shown in Fig. 31.4. At every preemption point, the scheduler scans the multilevel queue from the top (highest priority) and selects the task at the head of the first non-empty queue. Each task is allowed to run for a fixed time quantum (or time slice) at a time. Unix normally uses one second time slice. That is, if the running process does not block or complete within one second of its starting execution, it is preempted and the scheduler selects the next task for dispatching. Unix system however allows configuring the default one second time slice during **system generation**. The kernel preempts a process that does not complete within its assigned time quantum, recomputes its priority, and inserts it back into one of the priority queues depending on the recomputed priority value of the task.

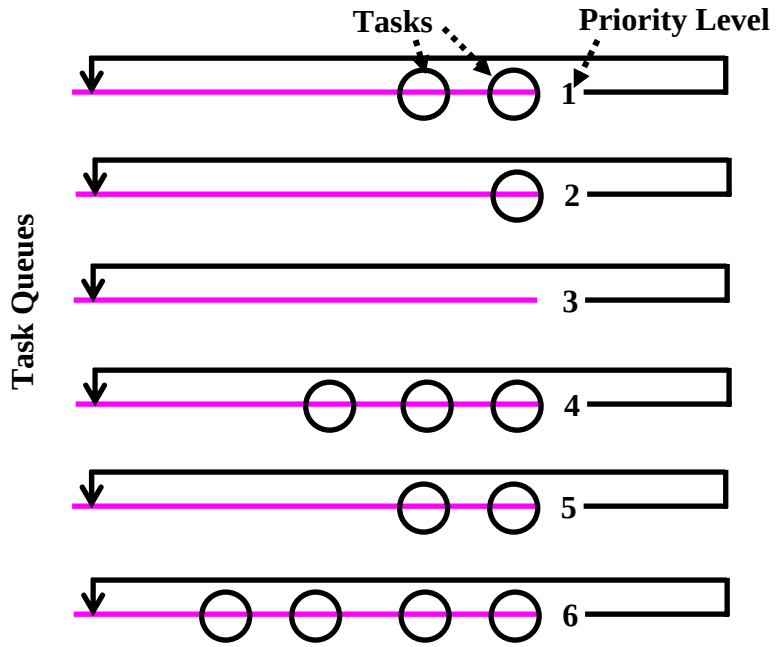


Fig. 31.4 Multi-Level Feedback Queues

Unix periodically computes the priority of a task based on the type of the task and its execution history. The priority of a task (T_i) is recomputed at the end of its j-th time slice using the following two expressions:

$$Pr(T_i, j) = Base(T_i) + CPU(T_i, j) + nice(T_i) \quad \ldots(4.1)$$

$$CPU(T_i, j) = U(T_i, j-1) / 2 + CPU(T_i, j-1) / 2 \quad \ldots(4.2)$$

where $Pr(T_i, j)$ is the priority of the task T_i at the end of its j-th time slice; $U(T_i, j)$ is the utilization of the task T_i for its j-th time slice, and $CPU(T_i, j)$ is the weighted history of CPU utilization of the task T_i at the end of its j-th time slice. $Base(T_i)$ is the base priority of the task T_i and $nice(T_i)$ is the nice value associated with T_i. User processes can have non-negative nice values. Thus, effectively the nice value lowers the priority value of a process (i.e. being nice to the other processes).

Expr. 4.2 has been recursively defined. Unfolding the recursion, we get:

$$CPU(T_i, j) = U(T_i, j-1) / 2 + U(T_i, j-2) / 4 + \quad \ldots\ldots(4.3)$$

It can be easily seen from Expr. 4.3 that, in the computation of the weighted history of CPU utilization of a task, the activity (i.e. processing or I/O) of the task in the immediately concluded interval is given the maximum weightage. If the task used up CPU for the full duration of the slice (i.e. 100% CPU utilization), then $CPU(T_i, j)$ gets a higher value indicating a lower priority. Observe that the activities of the task in the preceding intervals get progressively lower weightage. It should be clear that $CPU(T_i, j)$ captures the weighted history of CPU utilization of the task T_i at the end of its j-th time slice.

Now, substituting Expr 4.3 in Expr. 4.1, we get:

$$Pr(T_i, j) = Base(T_i) + U(T_i, j-1) / 2 + U(T_i, j-2) / 4 + \ldots + nice(T_i) \quad \ldots (4.4)$$

The purpose of the base priority term in the priority computation expression (Expr. 4.4) is to divide all tasks into a set of fixed bands of priority levels. The values of $U(T_i, j)$ and nice components are restricted to be small enough to prevent a process from migrating from its assigned band. The bands have been designed to optimize I/O, especially

block I/O. The different priority bands under Unix in decreasing order of priorities are: swapper, block I/O, file manipulation, character I/O and device control, and user processes. Tasks performing block I/O are assigned the highest priority band. To give an example of block I/O, consider the I/O that occurs while handling a page fault in a virtual memory system. Such block I/O use DMA-based transfer, and hence make efficient use of I/O channel. Character I/O includes mouse and keyboard transfers. The priority bands were designed to provide the most effective use of the I/O channels.

Dynamic re-computation of priorities was motivated from the following consideration. Unix designers observed that in any computer system, I/O is the bottleneck. Processors are extremely fast compared to the transfer rates of I/O devices. I/O devices such as keyboards are necessarily slow to cope up with the human response times. Other devices such as printers and disks deploy mechanical components that are inherently slow and therefore can not sustain very high rate of data transfer. Therefore, effective use of the I/O channels is very important to increase the overall system throughput. The I/O channels should be kept as busy as possible for letting the interactive tasks to get good response time. To keep the I/O channels busy, any task performing I/O should not be kept waiting for CPU. For this reason, as soon as a task blocks for I/O, its priority is increased by the priority re-computation rule given in Expr. 4.4. However, if a task makes full use of its last assigned time slice, it is determined to be computation-bound and its priority is reduced. Thus the basic philosophy of Unix operating system is that the interactive tasks are made to assume higher priority levels and are processed at the earliest. This gives the interactive users good response time. This technique has now become an accepted way of scheduling soft real-time tasks across almost all available general purpose operating systems.

We can now state from the above observations that the overall effect of re-computation of priority values using Expr. 4.4 as follows:

In Unix, I/O intensive tasks migrate to higher and higher priorities, whereas CPU-intensive tasks seek lower priority levels.

No doubt that the approach taken by Unix is very appropriate for maximizing the average task throughput, and does indeed provide good average responses time to interactive (soft real-time) tasks. In fact, almost every modern operating system does very similar dynamic re-computation of the task priorities to maximize the overall system throughput and to provide good average response time to the interactive tasks. However, for hard real-time tasks, dynamic shifting of priority values is clearly not appropriate.

1.3.3. Other Deficiencies of Unix

We have so far discussed two glaring shortcomings of Unix in handling the requirements of real-time applications. We now discuss a few other deficiencies of Unix that crop up while trying to use Unix in real-time applications.

Insufficient Device Driver Support: In Unix, (remember that we are talking of the original Unix System V) device drivers run in kernel mode. Therefore, if support for a new device is to be added, then the driver module has to be linked to the kernel modules – necessitating a system generation step. As a result, providing support for a new device in an already deployed application is cumbersome.

Lack of Real-Time File Services: In Unix, file blocks are allocated as and when they are requested by an application. As a consequence, while a task is writing to a file, it may encounter an error when the disk runs out of space. In other words, no guarantee is given that disk space would be available when a task writes a block to a file. Traditional file writing approaches also result in slow writes since required space has to be allocated before writing a block. Another problem with the traditional file systems is that blocks of the same file may not be contiguously located on the disk. This would result in read operations taking unpredictable times, resulting in jitter in data access. In real-time file systems significant performance improvement can be achieved by storing files contiguously on the disk. Since the file system pre-allocates space, the times for read and write operations are more predictable.

Inadequate Timer Services Support: In Unix systems, real-time timer support is insufficient for many hard real-time applications. The clock resolution that is provided to applications is 10 milliseconds, which is too coarse for many hard real-time applications.

1.4. Unix-based Real-Time Operating Systems

We have already seen in the previous section that traditional Unix systems are not suitable for being used in hard real-time applications. In this section, we discuss the different approaches that have been undertaken to make Unix suitable for real-time applications.

1.4.1. Extensions To The Traditional Unix Kernel

A naive attempted in the past to make traditional Unix suitable for real-time applications was by adding some real-time capabilities over the basic kernel. These additionally implemented capabilities included real-time timer support, a real-time task scheduler built over the Unix scheduler, etc. However, these extensions do not address the fundamental problems with the Unix system that were pointed out in the last section; namely, non-preemptive kernel and dynamic priority levels. No wonder that superficial extensions to the capabilities of the Unix kernel without addressing the fundamental deficiencies of the Unix system would fall wide short of the requirements of hard real-time applications.

1.4.2. Host-Target Approach

Host-target operating systems are popularly being deployed in embedded applications. In this approach, the real- time application development is done on a host machine. The host machine is either a traditional Unix operating system or an Windows system. The real-time application is developed on the host and the developed application is downloaded onto a target board that is to be embedded in a real-time system. A ROM-resident small real-time kernel is used in the target board. This approach has schematically been shown in Fig. 31.5.

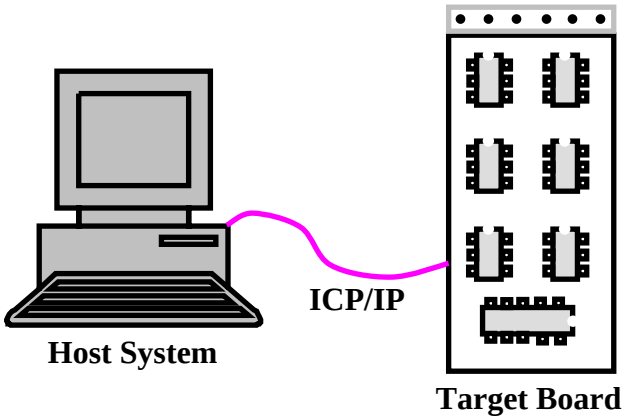


Fig. 31.5 Schematic Representation of a Host-Target System

The main idea behind this approach is that the real-time operating system running on the target board be kept as small and simple as possible. This implies that the operating system on the target board would lack virtual memory management support, neither does it support any utilities such as compilers, program editors, etc. The processor on the target board would run the real-time operating system.

The host system must have the program development environment, including compilers, editors, library, cross-compilers, debuggers etc. These are memory demanding applications that require virtual memory support. The host is usually connected to the target using a serial port or a TCP/IP connection (see Fig. 31.5). The real-time program is developed on the host. It is then cross-compiled to generate code for the target processor. Subsequently, the executable module is downloaded to the target board. Tasks are executed on the target board and the execution is controlled at the host side using a symbolic cross-debugger. Once the program works successfully, it is fused on a ROM or flash memory and becomes ready to be deployed in applications.

Commercial examples of host-target real-time operating systems include PSOS, VxWorks, and VRTX. We examine these commercial products in lesson 5. We would point out that these operating systems, due to their small size, limited functionality, and optimal design achieve much better performance figures than full-fledged operating systems. For example, the task preemption times of these systems are of the order of few microseconds compared to several hundreds of milliseconds for traditional Unix systems.

1.4.3. Preemption Point Approach

We have already pointed out that one of the major shortcomings of the traditional Unix V code is that during a system call, all interrupts are masked(disabled) for the entire duration of execution of the system call. This leads to unacceptable worst case task response time of the order of second, making Unix-based systems unacceptable for most hard real-time applications.

An approach that has been taken by a few vendors to improve the real-time performance of non-preemptive kernels is the introduction of preemption points in system routines. Preemption points in the execution of a system routine are the instants at which the kernel data structures are consistent. At these points, the kernel can safely be preempted to make way for any waiting higher priority real-time tasks without corrupting any kernel data structures. In this approach, when the execution of a system call reaches a preemption point, the kernel checks to see if any higher priority tasks have become ready. If there is at least one, it preempts

the processing of the kernel routine and dispatches the waiting highest priority task immediately. The worst-case preemption latency in this technique therefore becomes the longest time between two consecutive preemption points. As a result, the worst-case response times of tasks are now several folds lower than those for traditional operating systems without preemption points. This makes the preemption point-based operating systems suitable for use in many categories hard real-time applications, though still not suitable for applications requiring preemption latency of the order of a few micro seconds or less. Another advantage of this approach is that it involves only minor changes to be made to the kernel code. Many operating systems have taken the preemption point approach in the past, a prominent example being HP-UX.

1.4.4. Self-Host Systems

Unlike the host-target approach where application development is carried out on a separate host system machine running traditional Unix, in self-host systems a real-time application is developed on the same system on which the real-time application would finally run. Of course, while deploying the application, the operating system modules that are not essential during task execution are excluded during deployment to minimize the size of the operating system in the embedded application. Remember that in host-target approach, the target real-time operating system was a lean and efficient system that could only run the application but did not include program development facilities; program development was carried out on the host system. This made application development and debugging difficult and required cross-compiler and cross-debugger support. Self-host approach takes a different approach where the real-time application is developed on the full-fledged operating system, and once the application runs satisfactorily it is fused on the target board on a ROM or flash memory along with a stripped down version of the same operating system.

Most of the self-host operating systems that are available now are based on micro-kernel architecture. Use of microkernel architecture for a self-host operating system entails several advantages. In microkernel architecture, only the core functionalities such as interrupt handling and process management are implemented as kernel routines. All other functionalities such as memory management, file management, device management, etc are implemented as add-on modules which operate in user mode. As a result, it becomes very easy to configure the operating system. Also, the micro kernel is lean and therefore becomes much more efficient. A monolithic operating system binds most drivers, file systems, and protocol stacks to the operating system kernel and all kernel processes share the same address space. Hence a single programming error in any of these components can cause a fatal kernel fault. In microkernel-based operating systems, these components run in separate memory-protected address spaces. So, system crashes on this count are very rare, and microkernel-based operating systems are very reliable.

We had discussed earlier that any Unix-based system has to overcome the following two main shortcomings of the traditional Unix kernel in order to be useful in hard real-time applications: non-preemptive kernel and dynamic priority values. We now examine how these problems are overcome in self-host systems.

Non-preemptive kernel: We had identified the genesis of the problem of non-preemptive Unix kernel in Sec.4.3.1. We had remarked that in order to preserve the integrity of the kernel data structures, all interrupts are disabled as long as a system call does not complete. This was

done from efficiency considerations and worked well for non-real-time and uniprocessor applications.

Masking interrupts during kernel processing makes to even very small critical routines to have worst case response times of the order of a second. Further, this approach would not work in multiprocessor environments. In multiprocessor environments masking the interrupts for one processor does not help, as the tasks running on other processors can still corrupt the kernel data structure.

It is now clear that in order to make the kernel preemptive, locks must be used at appropriate places in the kernel code. In fully preemptive Unix systems, normally two types of locks are used: kernel-level locks, and spin locks.

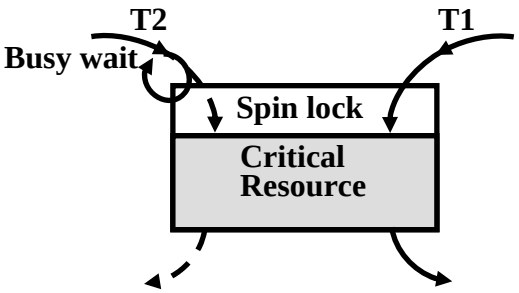


Fig. 31.6 Operation of a Spin Lock

A kernel-level lock is similar to a traditional lock. When a task waits for a kernel level lock to be released, it is blocked and undergoes a context switch. It becomes ready only after the required lock is released by the holding task and becomes available. This type of locks is inefficient when critical resources are required for short durations of the order of a few milliseconds or less. In some situations such context switching overheads are not acceptable. Consider that some task requires the lock for carrying out very small processing (possibly a single arithmetic operation) on some critical resource. Now, if a kernel level lock is used, another task requesting the lock at that time would be blocked and a context switch would be incurred, also the cache contents, pages of the task etc. may be swapped. Here a context switching time is comparable to the time for which a task needs a resource even greater than it. In such a situation, a spin lock would be appropriate. Now let us understand the operation of a spin lock. A spin lock has been schematically shown in Fig. 31.6. In Fig. 31.6, a critical resource is required by the tasks T_1 and T_2 for very short times (comparable to a context switching time). This resource is protected by a spin lock. The task T_1 has acquired the spin lock guarding the resource. Meanwhile, the task T_2 requests the resource. When task T_2 cannot get access to the resource, it just busy waits (shown as a loop in the figure) and does not block and suffer context switch. T_2 gets the resource as soon as T_1 relinquishes the resource.

Real-Time Priorities: Let us now examine how self-host systems address the problem of dynamic priority levels of the traditional Unix systems. In Unix based real-time operating systems, in addition to dynamic priorities, real-time and idle priorities are supported. Fig. 31.7 schematically shows the three available priority levels.

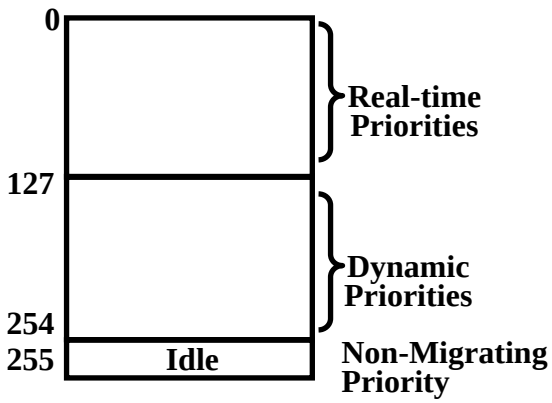


Fig. 31.7 Priority Changes in Self-host Unix Systems

Idle(Non-Migrating): This is the lowest priority. The task that runs when there are no other tasks to run (idle), runs at this level. Idle priorities are static and are not recomputed periodically.

Dynamic: Dynamic priorities are recomputed periodically to improve the average response time of soft real-time tasks. Dynamic re-computation of priorities ensures that I/O bound tasks migrate to higher priorities and CPU-bound tasks operate at lower priority levels. As shown in Fig. 31.7, dynamic priority levels are higher than the idle priority, but are lower than the real-time priorities.

Real-Time: Real-time priorities are static priorities and are not recomputed. Hard real-time tasks operate at these levels. Tasks having real-time priorities operate at higher priorities than the tasks with dynamic priority levels.

1.5. Windows As A Real-Time Operating System

Microsoft's Windows operating systems are extremely popular in desktop computers. Windows operating systems have evolved over the years last twenty five years from the naive DOS (Disk Operating System). Microsoft developed DOS in the early eighties. Microsoft kept on announcing new versions of DOS almost every year and kept on adding new features to DOS in the successive versions. DOS evolved to the Windows operating systems, whose main distinguishing feature was a graphical front-end. As several new versions of Windows kept on appearing by way of upgrades, the Windows code was completely rewritten in 1998 to develop the Windows NT system. Since the code was completely rewritten, Windows NT system was much more stable (does not crash) than the earlier DOS-based systems. The later versions of Microsoft's operating systems were descendants of the Windows NT; the DOS-based systems were scrapped. Fig. 31.8 shows the genealogy of the various operating systems from the Microsoft stable. Because stability is a major requirement for hard real-time applications, we consider only the Windows NT and its descendants in our study and do not include the DOS line of products.

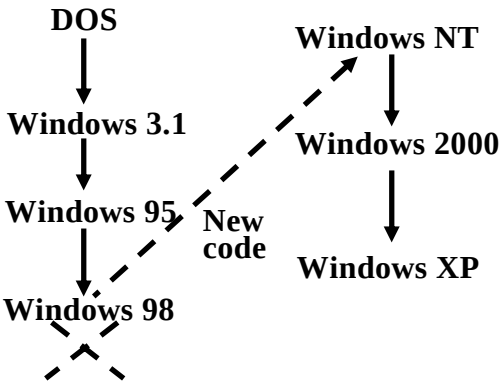


Fig. 31.8 Genealogy of Operating Systems from Microsoft's Stable

An organization owning Windows NT systems might be interested to use it for its real-time applications on account of either cost saving or convenience. This is especially true in prototype application development and also when only a limited number of deployments are required. In the following, we critically analyze the suitability of Windows NT for real-time application development. First, we highlight some features of Windows NT that are very relevant and useful to a real-time application developer. In the subsequent subsection, we point out some of the lacuna of Windows NT when used in real-time application development.

1.5.1. Features of Windows NT

Windows NT has several features which are very desirable for real-time applications such as support for multithreading, real-time priority levels, and timer. Moreover, the clock resolutions are sufficiently fine for most real-time applications.

Windows NT supports 32 priority levels (see Fig. 31.9). Each process belongs to one of the following priority classes: idle, normal, high, real-time. By default, the priority class at which an application runs is normal. Both normal and high are variable type where the priority is recomputed periodically. NT uses priority-driven pre- emptive scheduling and threads of real-time priorities have precedence over all other threads including kernel threads. Processes such as screen saver use priority class idle. NT lowers the priority of a task (belonging to variable type) if it used all of its last time slice. It raises the priority of a task if it blocked for I/O and could not use its last time slice in full. However, the change of a task from its base priority is restricted to ±2.

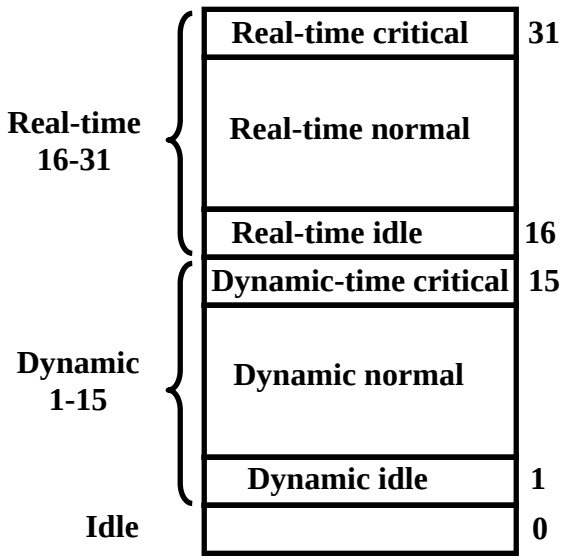


Fig. 31.9 Task Priorities in Windows NT

1.5.2. Shortcomings of Windows NT

In spite of the impressive support that Windows provides for real-time program development as discussed in Section 4.5.1, a programmer trying to use Windows in real-time system development has to cope up with several problems. Of these, the following two main problems are the most troublesome.

1. **Interrupt Processing:** Priority level of interrupts is always higher than that of the user-level threads; including the threads of real-time class. When an interrupt occurs, the handler routine saves the machine's state and makes the system execute an Interrupt Service Routine (ISR). Only critical processing is performed in ISR and the bulk of the processing is done as a Deferred Procedure *Call(DPC)*. DPCs for various interrupts are queued in the DPC queue in a FIFO manner. While this separation of ISR and DPC has the advantage of providing quick response to further interrupts, it has the disadvantage of maintaining the all DPCs at the same priorities. A DPC can not be preempted by another DPC but by an interrupt. DPCs are executed in FIFO order at a priority lower than the hardware interrupt priorities but higher than the priority of the scheduler/dispatcher. Further, it is not possible for a user-level thread to execute at a priority higher than that of ISRs or DPCs. Therefore, even ISRs and DPCs corresponding to very low priority tasks can preempt real-time processes. Therefore, the potential blocking of real-time tasks due to DPCs can be large. For example, interrupts due to page faults generated by low priority tasks would get processed faster than real-time processes. Also, ISRs and DPCs generated due to key board and mouse interactions would operate at higher priority levels compared to real-time tasks. If there are processes doing network or disk I/O, the effect of system-wide FIFO queues may lead to unbounded response times for even real-time threads.

These problems have been avoided by Windows CE operating system through a priority inheritance mechanism.

2. **Support for Resource Sharing Protocols:** We had discussed in Chapter 3 that unless appropriate resource sharing protocols are used, tasks while accessing shared resources may suffer unbounded priority inversions leading to deadline misses and even system failure. Windows NT does not provide any support (such as priority inheritance, etc.) to support real-time tasks to share critical resource among themselves. This is a major shortcoming of Windows NT when used in real-time applications.
   Since most real-time applications do involve resource sharing among tasks we outline below the possible ways in which user-level functionalities can be added to the Windows NT system.
   The simplest approach to let real-time tasks share critical resources without unbounded priority inversions is as follows. As soon as a task is successful in locking a non-preemptable resource, its priority can be raised to the highest priority (31). As soon as a task releases the required resource, its priority is restored. However, we know that this arrangement would lead to large inheritance-related inversions.
   Another possibility is to implement the priority ceiling protocol (PCP). To implement this protocol, we need to restrict the real-time tasks to have even priorities (i.e. 16, 18, ..., 30). The reason for this restriction is that NT does not support FIFO scheduling among equal priority tasks. If the highest priority among all tasks needing a resource is $2*n$, then the ceiling priority of the resource is $2*n+1$. In Unix, FIFO option among equal priority tasks is available; therefore all available priority levels can be used.

1.6. Windows vs Unix

Table 31.1 Windows NT versus Unix

Real-Time Feature	Windows NT	Unix V
DPCs	Yes	No
Real-Time priorities	Yes	No
Locking virtual memory	Yes	Yes
Timer precision	1 msec	10 msec
Asynchronous I/O	Yes	No

Though Windows NT has many of the features desired of a real-time operating system, its implementation of DPCs together its lack of protocol support for resource sharing among equal priority tasks makes it unsuitable for use in safety-critical real-time applications. A comparison of the extent to which some of the basic features required for real-time programming are provided by Windows NT and Unix V is indicated in Table 1. With careful programming, Windows NT may be useful for applications that can tolerate occasional deadline misses, and have deadlines of the order of hundreds of milliseconds than microseconds. Of course, to be used in such applications, the processor utilization must be kept sufficiently low and priority inversion control must be provided at the user level.

1.7. Exercises

1. State whether the following assertions are True or False. Justify your answer in each case.
   a. When RMA is used for scheduling a set of hard real-time periodic tasks, the upper bound on achievable utilization improves as the number in tasks in the system being developed increases.
   b. Under the Unix operating system, computation intensive tasks dynamically gravitate towards higher priorities.
   c. Normally, task switching time is larger than task preemption time.
   d. Suppose a real-time operating system does not support memory protection, then a procedure call and a system call are indistinguishable in that system.
   e. Watchdog timers are typically used to start certain tasks at regular intervals.
   f. For the memory of same size under segmented and virtual addressing schemes, the segmented addressing scheme would in general incur lower memory access jitter compared to the virtual addressing scheme.
2. Even though clock frequency of modern processors is of the order of several GHz, why do many modern real-time operating systems not support nanosecond or even microsecond resolution clocks? Is it possible for an operating system to support nanosecond resolution clocks in operating systems at present? Explain how this can be achieved.
3. Give an example of a real-time application for which a simple segmented memory management support by the RTOS is preferred and another example of an application for which virtual memory management support is essential. Justify your choices.
4. Is it possible to meet the service requirements of hard real-time applications by writing additional layers over the Unix System V kernel? If your answer is “no”, explain the reason. If your answer is “yes”, explain what additional features you would implement in the external layer of Unix System V kernel for supporting hard real-time applications.
5. Briefly indicate how Unix dynamically recomputes task priority values. Why is such re-computation of task priorities required? What are the implications of such priority re-computations on real-time application development?
6. Why is Unix V non-preemptive in kernel mode? How do fully preemptive kernels based on Unix (e.g. Linux) overcome this problem? Briefly describe an experimental set up that can be used to determine the preemptability of different operating systems by high-priority real-time tasks when a low priority task has made a system call.
7. Explain how interrupts are handled in Windows NT. Explain how the interrupt processing scheme of Windows NT makes it unsuitable for hard real-time applications. How has this problem been overcome in WinCE?
8. Would you recommend Unix System V to be used for a few real-time tasks for running a data acquisition application? Assume that the computation time for these tasks is of the order of few hundreds of milliseconds and the deadline of these tasks is of the order of several tens of seconds. Justify your answer.
9. Explain the problems that you would encounter if you try to develop and run a hard real-time system on the Windows NT operating system.
10. Briefly explain why the traditional Unix kernel is not suitable to be used in a multiprocessor environments. Define a spin lock and a kernel-level lock and explain their use in realizing a preemptive kernel.

11. What do you understand by a microkernel-based operating system? Explain the advantages of a microkernel- based real-time operating system over a monolithic operating system.
12. What is the difference between a self-host and a host-target based embedded operating system? Give at least one example of a commercial operating system from each category. What problems would a real-time application developer might face while using RT-Linux for developing hard real-time applications?
13. What are the important features required in a real-time operating system? Analyze to what extent these features are provided by Windows NT and Unix V.

Module 6

Embedded System Software

Lesson 32

Commercial Real-Time Operating Systems

Specific Instructional Objectives

At the end of this lesson, the student would be able to:

- Get an understanding of open software
- Know the historical background under which POSIX was developed
- Get an overview of POSIX
- Understand the Real-Time POSIX standard
- Get an insight into the features of some of the popular Real-Time OS: PSOS, VRTX, VxWorks, QNX, μC/OS-II, RT-Linux, Lynx, Windows CE

1. Introduction

Many real-time operating systems are at present available commercially. In this lesson, we analyze some of the popular real-time operating systems and investigate why these popular systems cannot be used across all applications. We also examine the POSIX standards for RTOS and their implications.

1.1. POSIX

POSIX stands for Portable Operating System Interface. "X" has been suffixed to the abbreviation to make it sound Unix-like. Over the last decade, POSIX has become an important standard in the operating systems area including real-time operating systems. The importance of POSIX can be gauzed from the fact that nowadays it has become uncommon to come across a commercial operating system that is not POSIX-compliant. POSIX started as an open software initiative. Since POSIX has now become overwhelmingly popular, we discuss the POSIX requirements on real-time operating systems. We start with a brief introduction to open software movement and then trace the historical events that have led to the emergence of POSIX. Subsequently, we highlight the important requirements of real-time POSIX.

1.2. Open Software

An *open system* is a vendor neutral environment, which allows users to intermix hardware, software, and networking solutions from different vendors. Open systems are based on open standards and are not copyrighted, saving users from expensive intellectual property right (IPR) law suits. The most important characteristics of open systems are: interoperability and portability. Interoperability means systems from multiple vendors can exchange information among each other. A system is portable if it can be moved from one environment to another without modifications. As part of the open system initiative, open software movement has become popular.

Advantages of open software include the following: It reduces cost of development and time to market a product. It helps increase the availability of add-on software packages. It enhances the ease of programming. It facilitates easy integration of separately developed modules. POSIX is an off-shoot of the open software movement.

Open Software standards can be divided into three categories:

Open Source: Provides portability at the source code level. To run an application on a new platform would require only compilation and linking. ANSI and POSIX are important open source standards.

Open Object: This standard provides portability of unlinked object modules across different platforms. To run an application in a new environment, relinking of the object modules would be required.

Open Binary: This standard provides complete software portability across hardware platforms based on a common binary language structure. An open binary product can be portable at the executable code level. At the moment, no open binary standards.

The main goal of POSIX is application portability at the source code level. Before we discuss about RT-POSIX, let us explore the historical background under which POSIX was developed.

1.3. Genesis of POSIX

Before we discuss the different features of the POSIX standard in the next subsection, let us understand the historical developments that led to the development of POSIX.

Unix was originally developed by AT&T Bell Labs. Since AT&T was primarily a telecommunication company, it felt that Unix was not commercially important for it. Therefore, it distributed Unix source code free of cost to several universities. UCB (University of California at Berkeley) was one of the earliest recipient of Unix source code.

AT&T later got interested in computers, realized the potential of Unix and started developing Unix further and came up with Unix V. Meanwhile, UCB had incorporated TCP/IP into Unix through a large DARPA (Defense Advanced Research Project Agency of USA) project and had come up with BSD 4.3 and C Shell. With this, the commercial importance of Unix started to grow rapidly. As a result, many vendors implemented and extended Unix services in different ways: IBM with its AIX, HP with its HP-UX, Sun with its Solaris, Digital with its Ultrix, and SCO with SCO-Unix. Since there were so many variants of Unix, portability of applications across Unix platforms became a problem. It resulted in a situation where a program written on one Unix platform would not run on another platform.

The need for a standard Unix was recognized by all. The first effort towards standardization of Unix was taken by AT&T in the form of its SVID (System V Interface Definition). However, BSD and other vendors ignored this initiative. The next initiative was taken under ANSI/IEEE, which yielded POSIX.

1.4. Overview of POSIX

POSIX is an off-shoot of the open software movement, and portability of applications across different variants of Unix operating systems was the major concern. POSIX standard defines only interfaces to operating system services and the semantics of these services, but does not specify how exactly the services are to be implemented. For example, the standard does not specify whether an operating system kernel must be single threaded or multithreaded or at what priority level the kernel services are to be executed.

The POSIX standard has several parts. The important parts of POSIX and the aspects that they deal with, are the following:

Open Source: Provides portability at the source code level. To run an application on a new platform would require only compilation and linking. ANSI and POSIX are important open source standards.

- **POSIX 1 :** system interfaces and system call parameters
- **POSIX 2 :** shells and utilities
- **POSIX 3 :** test methods for verifying conformance to POSIX
- **POSIX 4 :** real-time extensions

1.5. Real-Time POSIX Standard

POSIX.4 deals with real-time extensions to POSIX and is also known as POSIX-RT. For an operating system to be POSIX-RT compliant, it must meet the different requirements specified in the POSIX-RT standard. The main requirements of the POSIX-RT are the following:

- **Execution scheduling:** An operating system to be POSIX-RT compliant must provide support for real-time (static) priorities.
- **Performance requirements on system calls:** It specifies the worst case execution times required for most real-time operating services.
- **Priority levels:** The number of priority levels supported should be at least 32.
- **Timers:** Periodic and one shot timers (also called watch dog timer) should be supported. The system clock is called CLOCK REALTIME when the system supports real-time POSIX.
- **Real-time files:** Real-time file system should be supported. A real-time file system can pre-allocate storage for files and should be able to store file blocks contiguously on the disk. This enables to have predictable delay in file access in virtual memory system.
- **Memory locking:** Memory locking should be supported. POSIX-RT defines the operating system services: mlockall() to lock all pages of a process, mlock() to lock a range of pages, and mlockpage() to lock only the current page. The unlock services are munlockall(), munlock(), and munlockpage. Memory locking services have been introduced to support deterministic memory access.
- **Multithreading support:** Real-time threading support is mandated. Real-time threads are schedulable entities of a real-time application that have individual timeliness constraints and may have collective timeliness constraints when belonging to a runnable set of threads.

1.6. A Survey of Contemporary Real-Time Operating Systems

In this section, we briefly survey the important feature of some of the popular real-time operating systems that are being used in commercial applications.

1.6.1. PSOS

PSOS is a popular real-time operating system that is being primarily used in embedded applications. It is available from Wind River Systems, a large player in the real-time operating system arena. It is a host-target type of real- time operating system. PSOS is being used in

several commercial embedded products. An example application of PSOS is in the base stations of the cellular systems.

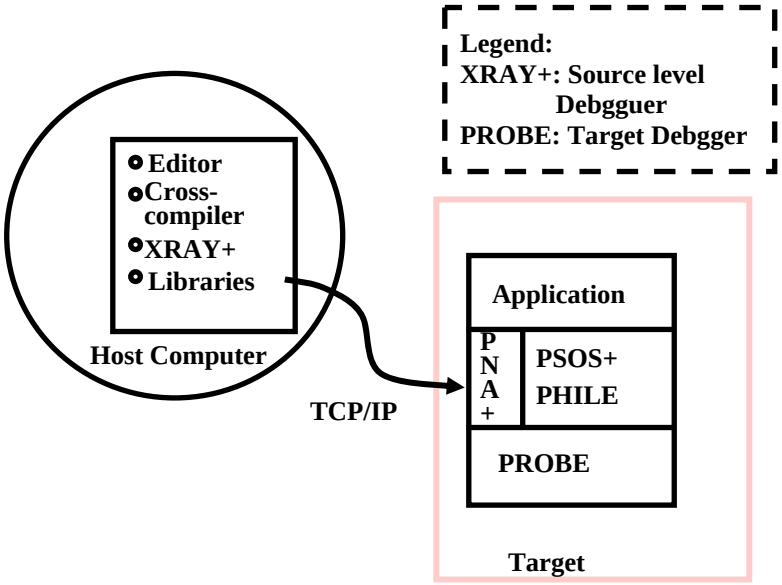


Fig. 32.1 PSOS-based Development of Embedded Software

PSOS-based application development has schematically been shown in Fig. 32.1. The host computer is typically a desktop. Both Unix and Windows hosts are supported. The target board contains the embedded processor, ROM, RAM, etc. The host computer runs the editor, cross-compiler, source-level debugger, and library routines. On the target board PSOS+, and other optional modules such as PNA+, PHILE, and PROBE are installed on a RAM. PNA+ is the network manager. It provides TCP/IP communication over Ethernet and FDDI. It conforms to Unix 4.3 (BSD) socket syntax and is compatible with other TCP/IP-based networking standards such as ftp and NFS. Using these, PNA+ provides efficient downloading and debugging communication between the target and the host. PROBE+ is the target debugger and XRAY+ is the source-level debugger. The application development is done on the host machine and is downloaded to the target board. The application is debugged using the source debugger (XRAY+). Once the application runs satisfactorily, it is fused on a ROM and installed on the target board.

We now highlight some important features of PSOS. PSOS consists of 32 priority levels. In the minimal configuration, the foot print of the operating system is only 12KBytes. For sharing critical resources among real-time tasks, it supports priority inheritance and priority ceiling protocols. It support segmented memory management. It allocates tasks to memory regions. A memory region is a physically contiguous block of memory. A memory region is created by the operating system in response to a call from an application.

In most modern operating systems, the control jumps to the kernel when an interrupt occurs. PSOS takes a different approach. The device drivers are outside the kernel and can be loaded and removed at the run time. When an interrupt occurs, the processor jumps directly to the ISR (interrupt service routine) pointed to by the vector table.

The intention is not only to gain speed, but also to give the application developer complete control over interrupt handling.

1.6.2. VRTX

VRTX is a POSIX-RT compliant operating system from Mentor Graphics. VRTX has been certified by the US FAA (Federal Aviation Agency) for use in mission and life critical applications such as avionics. VRTX has two multitasking kernels: VRTXsa and VRTXmc.

VRTXsa is used for large and medium applications. It supports virtual memory. It has a POSIX-compliant library and supports priority inheritance. Its system calls are deterministic and fully preemptable. VRTXmc is optimized for power consumption and ROM and RAM sizes. It has therefore a very small foot print. The kernel typically requires only 4 to 8 Kbytes of ROM and 1KBytes of RAM. It does not support virtual memory. This version is targeted for cell phones and other small hand-held devices.

1.6.3. VxWorks

VxWorks is a product from Wind River Systems. It is host-target system. The host can be either a Windows or a Unix machine. It supports most POSIX-RT functionalities. VxWorks comes with an integrated development environment (IDE) called Tornado. In addition to the standard support for program development tools such as editor, cross-compiler, cross-debugger, etc. Tornado contains VxSim and WindView. VxSim simulates a VxWorks target for use as a prototyping and testing environment. WindView provides debugging tools for the simulator environment. VxMP is the multiprocessor version of VxWorks.

VxWorks was deployed in the Mars Pathfinder which was sent to Mars in 1997. Pathfinder landed in Mars, responded to ground commands, and started to send science and engineering data. However, there was a hitch: it repeatedly reset itself. Remotely using trace generation, logging, and debugging tools of VxWorks, it was found that the cause was unbounded priority inversion. The unbounded priority inversion caused real-time tasks to miss their deadlines, and as a result, the exception handler reset the system each time. Although VxWorks supports priority inheritance, using the remote debugging tool, it was found to have been disabled in the configuration file. The problem was fixed by enabling it.

1.6.4. QNX

QNX is a product from QNX Software System Ltd. QNX Neutrino offers POSIX-compliant APIs and is implemented using microkernel architecture.

The microkernel architecture of QNX is shown in Fig. 32.2. Because of the fine grained scalability of the micro- kernel architecture, it can be configured to a very small size – a critical advantage in high volume devices, where even a 1% reduction in memory costs can return millions of dollars in profit.

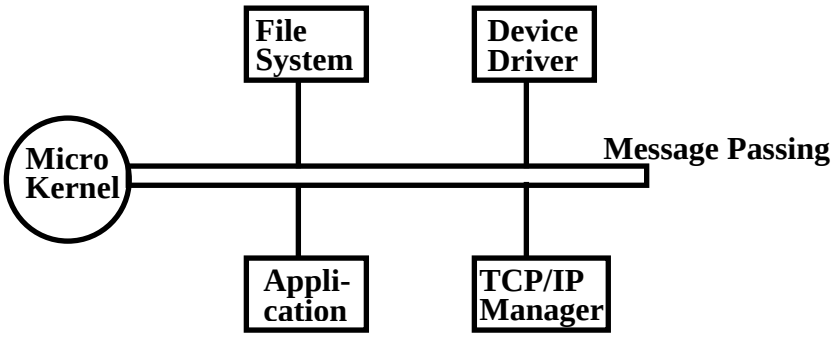


Fig. 32.2 Microkernel Architecture of QNX

1.6.5. μC/OS-II

μC/OS-II is a free RTOS, easily available on Internet. It is written in ANSI C and contains small portion of assembly code. The assembly language portion has been kept to a minimum to make it easy to port it to different processors. To date, μC/OS-II has been ported to over 100 different processor architectures ranging from 8-bit to 64-bit microprocessors, microcontrollers, and DSPs. Some important features of μC/OS-II are highlighted in the following.

- μC/OS-II was designed so that the programmer can use just a few of the offered services or select the entire range of services. This allows the programmer to minimize the amount of memory needed by μC/OS-II on a per-product basis.
- μC/OS-II has a fully preemptive kernel. This means that μC/OS-II always ensures that the highest priority task that is ready would be taken up for execution.
- μC/OS-II allows up to 64 tasks to be created. Each task operates at a unique priority level. There are 64 priority levels. This means that round-robin scheduling is not supported. The priority levels are used as the PID (Process Identifier) for the tasks.
- μC/OS-II uses a partitioned memory management. Each memory partition consists of several fixed sized blocks. A task obtains memory blocks from the memory partition and the task must create a memory partition before it can be used. Allocation and deallocation of fixed-sized memory blocks is done in constant time and is deterministic. A task can create and use multiple memory partitions, so that it can use memory blocks of different sizes.
- μC/OS-II has been certified by Federal Aviation Administration (FAA) for use in commercial aircraft by meeting the demanding requirements of its standard for software used in avionics. To meet the requirements of this standard it was demonstrated through documentation and testing that it is robust and safe.

1.6.6. RT Linux

Linux is by large a free operating system. It is robust, feature rich, and efficient. Several real-time implementations of Linux (RT-Linux) are available. It is a self-host operating system (see Fig. 32.3). RT-Linux runs along with a Linux system. The real-time kernel sits between the hardware and the Linux system. The RT kernel intercepts all interrupts generated by the hardware. Fig. 32.12 schematically shows this aspect. If an interrupt is to cause a real-time task

to run, the real-time kernel preempts Linux, if Linux is running at that time, and lets the real-time task run. Thus, in effect Linux runs as a task of RT-Linux.

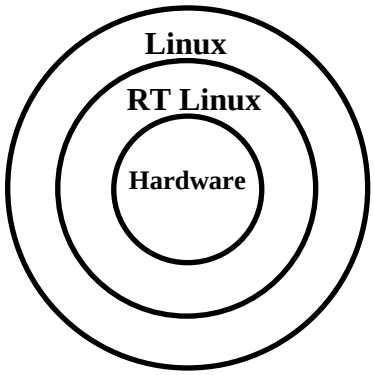


Fig. 32.3 Structure of RT Linux

The real-time applications are written as loadable kernel modules. In essence, real-time applications run in the kernel space.

In the approach taken by RT Linux, there are effectively two independent kernels: real-time kernel and Linux kernel. Therefore, this approach is also known as the *dual kernel approach* as the real-time kernel is implemented outside the Linux kernel. Any task that requires deterministic scheduling is run as a real-time task. These tasks preempt Linux whenever they need to execute and yield the CPU to Linux only when no real-time task is ready to run.

Compared to the microkernel approach, the following are the shortcomings of the dual-kernel approach.

- **Duplicated Coding Efforts:** Tasks running in the real-time kernel can not make full use of the Linux system services – file systems, networking, and so on. In fact, if a real-time task invokes a Linux service, it will be subject to the same preemption problems that prohibit Linux processes from behaving deterministically. As a result, new drivers and system services must be created specifically for the real-time kernel – even when equivalent services already exist for Linux.
- **Fragile Execution Environment:** Tasks running in the real-time kernel do not benefit from the MMU-protected environment that Linux provides to the regular non-real-time processes. Instead, they run unprotected in the kernel space. Consequently, any real-time task that contains a coding error such as a corrupt C pointer can easily cause a fatal kernel fault. This is serious problem since many embedded applications are safety-critical in nature.
- **Limited Portability:** In the dual kernel approach, the real-time tasks are not Linux processes at all; but programs written using a small subset of POSIX APIs. To aggravate the matter, different implementations of dual kernels use different APIs. As a result, real-time programs written using one vendor's RT-Linux version may not run on another's.
- **Programming Difficulty:** RT-Linux kernels support only a limited subset of POSIX APIs. Therefore, application development takes more effort and time.

1.6.7. Lynx

Lynx is a self host system. The currently available version of Lynx (Lynx 3.0) is a microkernel-based real-time operating system, though the earlier versions were based on monolithic design. Lynx is fully compatible with Linux. With Lynx's binary compatibility, a Linux program's binary image can be run directly on Lynx. On the other hand, for other Linux compatible operating systems such as QNX, Linux applications need to be recompiled in order to run on them. The Lynx microkernel is 28KBytes in size and provides the essential services in scheduling, interrupt dispatch, and synchronization. The other services are provided as kernel plug-ins (KPIs). By adding KPIs to the microkernel, the system can be configured to support I/O, file systems, sockets, and so on. With full configuration, it can function as a multipurpose Unix machine on which both hard and soft real-time tasks can run. Unlike many embedded real-time operating systems, Lynx supports memory protection.

1.6.8. Windows CE

Windows CE is a stripped down version of Windows, and has a minimum footprint of 400KBytes only. It provides 256 priority levels. To optimize performance, all threads are run in the kernel mode. The timer accuracy is 1 msec for sleep and wait related APIs. The different functionalities of the kernel are broken down into small non-preemptive sections. So, during system call preemption is turned off for only short periods of time. Also, interrupt servicing is preemptable. That is, it supports nested interrupts. It uses memory management unit (MMU) for virtual memory management.

Windows CE uses a priority inheritance scheme to avoid priority inversion problem present in Windows NT. Normally, the kernel thread handling the page fault (i.e. DPC) runs at priority level higher than NORMAL (refer Sec. 4.5.2). When a thread with priority level NORMAL suffers a page fault, the priority of the corresponding kernel thread handling this page fault is raised to the priority of the thread causing the page fault. This ensures that a thread is not blocked by another lower priority thread even when it suffers a page fault.

1.6.9. Exercises

1. State whether the following statements are True or False. Justify your answer in each case.
   a. In real-time Linux (RT-Linux), real-time processes are scheduled at priorities higher than the kernel processes.
   b. EDF scheduling of tasks is commonly supported in commercial real-time operating systems such as PSOS and VRTX.
   c. POSIX 1003.4 (real-time standard) requires that real-time processes be scheduled at priorities higher than kernel processes.
   d. POSIX is an attempt by ANSI/IEEE to enable executable files to be portable across different Unix machines.
2. What is the difference between block I/O and character I/O? Give examples of each. Which type of I/O is accorded higher priority by Unix? Why?
3. List four important features that a POSIX 1003.4 (Real-Time standard) compliant operating system must support. Is preemptability of kernel processes required by POSIX 1003.4? Can a Unix-based operating system using the preemption-point technique claim to be POSIX 1003.4 compliant? Explain your answers.

4. Suppose you are the manufacturer of small embedded components used mainly in consumer electronics goods such as automobiles, MP3 players, and computer-based toys. Would you prefer to use PSOS, WinCE, or RT-Linux in your embedded component? Explain the reasons behind your answer.
5. What is the difference between a system call and a function call? What problems, if any, might arise if the system calls are invoked as procedure calls?
6. Explain how a real-time operating system differs from a traditional operating system. Name a few real-time operating systems that are commercially available.
7. What is open software? Does an open software mandate portability of the executable files across different platforms? Name an open software standard for real-time operating systems. What is the advantage of using an open software operating system for real-time application development? What are the pros and cons of using an open software product in program development compared to a proprietary product?
8. Identify at least four important advantages of using VxWorks as the operating system for real-time applications compared to using Unix V.3.
9. What is an open source standard? How is it different from open object and open binary standards? Give some examples of popular open source software products.
10. Can multithreading result in faster response times (compared to single threaded tasks) even in uniprocessor systems? Explain your answer and identify the reasons to support your answer.

References (Lessons 24 - 28)

1. C.M. Krishna and Shin K.G., Real-Time Systems, Tata McGraw-Hill, 1999.
2. Philip A. Laplante, Real-Time System Design and Analysis, Prentice Hall of India, 1996.
3. Jane W.S. Liu, Real-Time Systems, Pearson Press, 2000.
4. Alan C. Shaw, Real-Time Systems and Software, John Wiley and Sons, 2001.
5. C. SivaRam Murthy and G. Manimaran, Resource Management in Real-Time Systems and Networks, MIT Press, 2001.
6. B. Dasarathy, Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods for Validating Them, IEEE Transactions on Software Engineering, January 1985, Vol. 11, No. 1, pages 80-86.
7. Lui Sha, Ragunathan Rajkumar, John P. Lehoczky, Priority inheritance protocols: An approach to real-time synchronization,, IEEE Transactions on Computers, 1990, Vol. 39, pages 1175-1185.

Module 7

Software Engineering Issues

Lesson 33

Introduction to Software Engineering

Specific Instructional Objectives

At the end of this lesson, the student would be able to:

- Get an introduction to software engineering
- Understand the need for software engineering principles
- Identify the causes of and solutions for software crisis
- Differentiate a piece of program from a software product
- Understand the evolution of software design techniques over last 50 years
- Identify the features of a structured program and its advantages
- Identify the features of various design techniques
- Differentiate between the exploratory style and modern styles of software development
- Explain what a life cycle model is
- Understand the need for a software life cycle model
- Identify the different phases of the classical waterfall model and related activities
- Identify the phase-entry and phase-exit criteria of each phase
- Explain what a prototype is
- Explain the need for prototype development
- State the activities carried out during each phase of a spiral model

1. Introduction

With the advancement of technology, computers have become more powerful and sophisticated. The more powerful a computer is, the more sophisticated programs it can run. Thus, programmers have been tasked to solve larger and more complex problems. They have coped with this challenge by innovating and by building on their past programming experience. All those past innovations and experience of writing good quality programs in efficient and cost-effective ways have been systematically organized into a body of knowledge. This body of knowledge forms the basis of software engineering principles. Thus, we can view software engineering as a systematic collection of past experience. The experience is arranged in the form of methodologies and guidelines.

1.1. The Need for Software Engineering

Alternatively, software engineering can be viewed an engineering approach to software development. A small program can be written without using software engineering principles. But if one wants to develop a large software product, then software engineering principles are indispensable to achieve a good quality software cost effectively. These definitions can be elaborated with the help of a building construction analogy.

Suppose you have a friend who asked you to build a small wall as shown in fig. 33.1. You would be able to do that using your common sense. You will get building materials like bricks; cement etc. and you will then build the wall.

Fig. 33.1 A Small Wall

But what would happen if the same friend asked you to build a large multistoried building as shown in fig. 33.2?

Fig. 33.2 A Multistoried Building

You don't have a very good idea about building such a huge complex. It would be very difficult to extend your idea about a small wall construction into constructing a large building. Even if you tried to build a large building, it would collapse because you would not have the requisite knowledge about the strength of materials, testing, planning, architectural design, etc. Building a small wall and building a large building are entirely different ball games. You can use your intuition and still be successful in building a small wall, but building a large building requires knowledge of civil, architectural and other engineering principles.

Without using software engineering principles it would be difficult to develop large programs. In industry it is usually needed to develop large programs to accommodate multiple functions. A problem with developing such large commercial programs is that the complexity and difficulty levels of the programs increase exponentially with their sizes as shown in fig. 33.3. For example, a program of size 1,000 lines of code has some complexity. But a program with 10,000 LOC is not just 10 times more difficult to develop, but may as well turn out to be 100 times more difficult unless software engineering principles are used. In such situations software engineering techniques come to the rescue. Software engineering helps to reduce programming complexity. Software engineering principles use two important techniques to reduce problem complexity: abstraction and decomposition.

The principle of abstraction (in fig. 33.4) implies that a problem can be simplified by omitting irrelevant details. Once the simpler problem is solved then the omitted details can be taken into consideration to solve the next lower level abstraction, and so on.

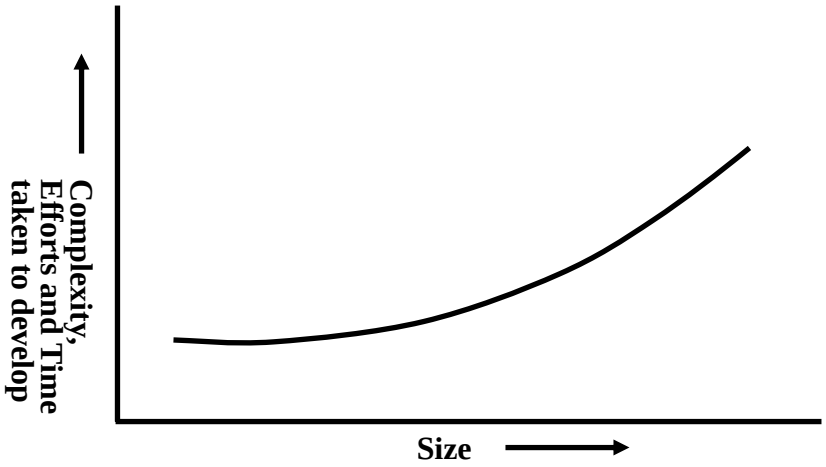


Fig. 33.3 Increase in development time and effort with problem size

1.1.1. Abstraction and Decomposition

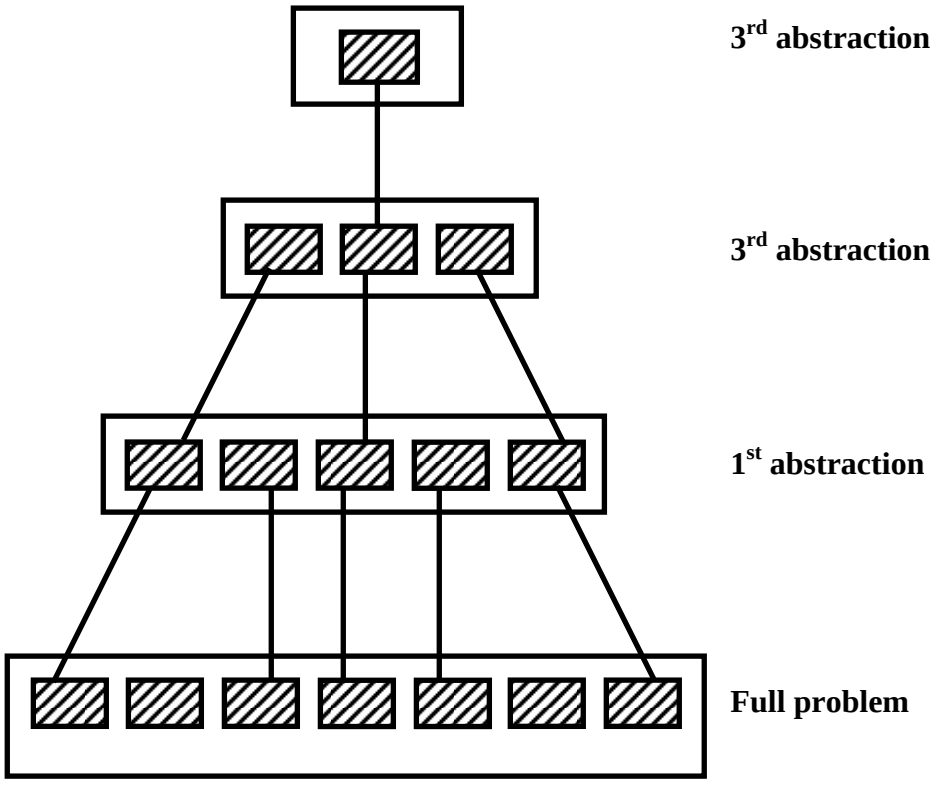


Fig. 33.4 A hierarchy of abstraction

The other approach to tackle problem complexity is decomposition. In this technique, a complex problem is divided into several smaller problems and then the smaller problems are solved one by one. However, in this technique any random decomposition of a problem into smaller parts will not help. The problem has to be decomposed such that each component of the decomposed problem can be solved independently and then the solution of the different

components can be combined to get the full solution. A good decomposition of a problem as shown in fig. 33.5 should minimize interactions among various components. If the different subcomponents are interrelated, then the different components cannot be solved separately and the desired reduction in complexity will not be realized.

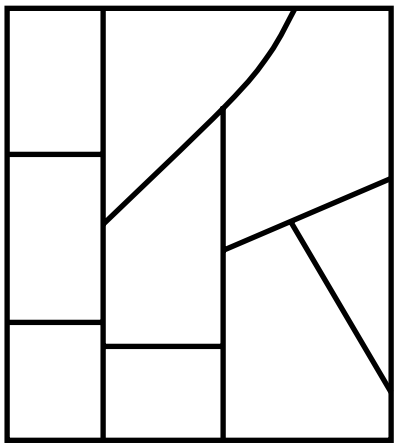


Fig. 33.5 Decomposition of a large problem into a set of smaller problems

1.2. The Software Crisis

Software engineering appears to be among the few options available to tackle the present software crisis.

To explain the present software crisis in simple words, consider the following. The expenses that organizations all around the world are incurring on software purchases compared to those on hardware purchases have been showing a worrying trend over the years (as shown in fig. 33.6)

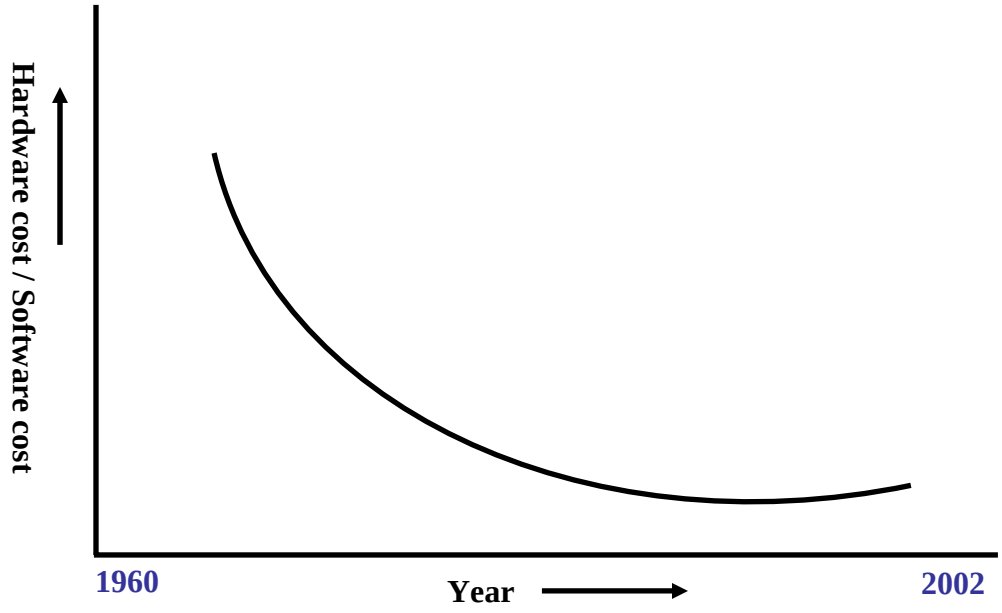


Fig. 33.6 Change in the relative cost of hardware and software over time

Organizations are spending larger and larger portions of their budget on software. Not only are the software products turning out to be more expensive than hardware, but they also present a host of other problems to the customers: software products are difficult to alter, debug, and enhance; use resources non-optimally; often fail to meet the user requirements; are far from being reliable; frequently crash; and are often delivered late. Among these, the trend of increasing software costs is probably the most important symptom of the present software crisis. Remember that the cost we are talking of here is not on account of increased features, but due to ineffective development of the product characterized by inefficient resource usage, and time and cost over-runs.

There are many factors that have contributed to the making of the present software crisis. Factors are larger problem sizes, lack of adequate training in software engineering, increasing skill shortage, and low productivity improvements.

It is believed that the only satisfactory solution to the present software crisis can possibly come from a spread of software engineering practices among the engineers, coupled with further advancements to the software engineering discipline itself.

1.3. Program vs. Software Product

Programs are developed by individuals for their personal use. They are therefore, small in size and have limited functionality but software products are extremely large. In case of a program, the programmer himself is the sole user but on the other hand, in case of a software product, most users are not involved with the development. In case of a program, a single developer is involved but in case of a software product, a large number of developers are involved. For a program, the user interface may not be very important, because the programmer is the sole user. On the other hand, for a software product, user interface must be carefully designed and implemented because developers of that product and users of that product are totally different. In case of a program, very little documentation is expected, but a software product must be well documented. A program can be developed according to the programmer's individual style of development, but a software product must be developed using the accepted software engineering principles.

2. Evolution of Program Design Techniques

During the 1950s, most programs were being written in assembly language. These programs were limited to about a few hundreds of lines of assembly code, i.e. were very small in size. Every programmer developed programs in his own individual style - based on his intuition. This type of programming was called Exploratory Programming.

The next significant development which occurred during early 1960s in the area computer programming was the high-level language programming. Use of high-level language programming reduced development efforts and development time significantly. Languages like FORTRAN, ALGOL, and COBOL were introduced at that time.

2.1. Structured Programming

As the size and complexity of programs kept on increasing, the exploratory programming style proved to be insufficient. Programmers found it increasingly difficult not only to write cost-effective and correct programs, but also to understand and maintain programs written by others. To cope with this problem, experienced programmers advised other programmers to pay particular attention to the design of the program's control flow structure (in late 1960s). In the late 1960s, it was found that the "GOTO" statement was the main culprit which makes control structure of a program complicated and messy. At that time most of the programmers used assembly languages extensively. They considered use of "GOTO" statements in high-level languages were very natural because of their familiarity with JUMP statements which are very frequently used in assembly language programming. So they did not really accept that they can write programs without using GOTO statements, and considered the frequent use of GOTO statements inevitable. At this time, **Dijkstra [1968]** published his (now famous) article "GOTO Statements Considered Harmful". Expectedly, many programmers were enraged to read this article. They published several counter articles highlighting the advantages and inevitable use of GOTO statements. But, soon it was conclusively proved that only three programming constructs – sequence, selection, and iteration – were sufficient to express any programming logic. This formed the basis of the structured programming methodology.

2.1.1. Features of Structured Programming

A structured program uses three types of program constructs i.e. selection, sequence and iteration. Structured programs avoid unstructured control flows by restricting the use of GOTO statements. A structured program consists of a well partitioned set of modules. Structured programming uses single entry, single-exit program constructs such as if-then-else, do-while, etc. Thus, the structured programming principle emphasizes designing neat control structures for programs.

2.1.2. Advantages of Structured Programming

Structured programs are easier to read and understand. Structured programs are easier to maintain. They require less effort and time for development. They are amenable to easier debugging and usually fewer errors are made in the course of writing such programs.

2.2. Data Structure-Oriented Design

After structured programming, the next important development was data structure-oriented design. Programmers argued that for writing a good program, it is important to pay more attention to the design of data structure, of the program rather than to the design of its control structure. Data structure-oriented design techniques actually help to derive program structure from the data structure of the program. Example of a very popular data structure-oriented design technique is Jackson's Structured Programming (JSP) methodology, developed by Michael Jackson in the1970s.

2.3. Data Flow-Oriented Design

Next significant development in the late 1970s was the development of data flow-oriented design technique. Experienced programmers stated that to have a good program structure, one has to study how the data flows from input to the output of the program. Every program reads data and then processes that data to produce some output. Once the data flow structure is identified, then from there one can derive the program structure.

2.4. Object-Oriented Design

Object-oriented design (1980s) is the latest and very widely used technique. It has an intuitively appealing design approach in which natural objects (such as employees, pay-roll register, etc.) occurring in a problem are first identified. Relationships among objects (such as composition, reference and inheritance) are determined. Each object essentially acts as a data hiding entity.

2.5. Changes in Software Development Practices

An important difference is that the exploratory software development style is based on error correction while the software engineering principles are primarily based on error prevention. Inherent in the software engineering principles is the realization that it is much more cost-effective to prevent errors from occurring than to correct them as and when they are detected. Even when errors occur, software engineering principles emphasize detection of errors as close to the point where the errors are committed as possible. In the exploratory style, errors are detected only during the final product testing. In contrast, the modern practice of software development is to develop the software through several well-defined stages such as requirements specification, design, coding, testing, etc., and attempts are made to detect and fix as many errors as possible in the same phase in which they occur.

In the exploratory style, coding was considered synonymous with software development. For instance, exploratory programming style believed in developing a working system as quickly as possible and then successively modifying it until it performed satisfactorily.

In the modern software development style, coding is regarded as only a small part of the overall software development activities. There are several development activities such as design and testing which typically require much more effort than coding.

A lot of attention is being paid to requirements specification. Significant effort is now being devoted to develop a clear specification of the problem before any development activity is started.

Now, there is a distinct design phase where standard design techniques are employed.

Periodic reviews are being carried out during all stages of the development process. The main objective of carrying out reviews is phase containment of errors, i.e. detect and correct errors as soon as possible. Defects are usually not detected as soon as they occur, rather they are noticed much later in the life cycle. Once a defect is detected, we have to go back to the phase

where it was introduced and rework those phases - possibly change the design or change the code and so on.

Today, software testing has become very systematic and standard testing techniques are available. Testing activity has also become all encompassing in the sense that test cases are being developed right from the requirements specification stage.

There is better visibility of design and code. By visibility we mean production of good quality, consistent and standard documents during every phase. In the past, very little attention was paid to producing good quality and consistent documents. In the exploratory style, the design and test activities, even if carried out (in whatever way), were not documented satisfactorily. Today, consciously good quality documents are being developed during product development. This has made fault diagnosis and maintenance smoother.

Now, projects are first thoroughly planned. Project planning normally includes preparation of various types of estimates, resource scheduling, and development of project tracking plans. Several techniques and tools for tasks such as configuration management, cost estimation, scheduling, etc. are used for effective software project management.

Several metrics are being used to help in software project management and software quality assurance.

3. Software Life Cycle Model

A software life cycle model (also called process model) is a descriptive and diagrammatic representation of the software life cycle. A life cycle model represents all the activities required to make a software product transit through its life cycle phases. It also captures the order in which these activities are to be undertaken. In other words, a life cycle model maps the different activities performed on a software product from its inception to its retirement. Different life cycle models may map the basic development activities to phases in different ways. Thus, no matter which life cycle model is followed, the basic activities are included in all life cycle models though the activities may be carried out in different orders in different life cycle models. During any life cycle phase, more than one activity may also be carried out. For example, the design phase might consist of the structured analysis activity followed by the structured design activity.

3.1. The Need for a Life Cycle Model

The development team must identify a suitable life cycle model for the particular project and then adhere to it. Without using a particular life cycle model, the development of a software product would not be in a systematic and disciplined manner. When a software product is being developed by a team there must be a clear understanding among team members about when and what to do. Otherwise it would lead to chaos and project failure. Let us try to illustrate this problem using an example. Suppose a software development problem is divided into several parts and the parts are assigned to the team members. From then on, suppose the team members are allowed the freedom to develop the parts assigned to them in whatever way they like. It is possible that one member might start writing the code for his part, another might decide to

prepare the test documents first, and some other engineer might begin with the design phase of the parts assigned to him. This would be one of the perfect recipes for project failure.

A software life cycle model defines entry and exit criteria for every phase. A phase can start only if its phase-entry criteria have been satisfied. So without a software life cycle model, the entry and exit criteria for a phase cannot be recognized. Without models (such as classical waterfall model, iterative waterfall model, prototyping model, evolutionary model, spiral model etc.), it becomes difficult for software project managers to monitor the progress of the project.

Many life cycle models have been proposed so far. Each of them has some advantages as well as some disadvantages. A few important and commonly used life cycle models are as follows:

- Classical Waterfall Model
- Iterative Waterfall Model
- Prototyping Model
- Evolutionary Model
- Spiral Model

3.2. Classical Waterfall Model

The classical waterfall model is intuitively the most obvious way to develop software. Though the classical waterfall model is elegant and intuitively obvious, we will see that it is not a practical model in the sense that it can not be used in actual software development projects. Thus, we can consider this model to be a *theoretical way of developing software*. But all other life cycle models are essentially derived from the classical waterfall model. So, in order to be able to appreciate other life cycle models, we must first learn the classical waterfall model.

Classical waterfall model divides the life cycle into the following phases as shown in fig. 33.7:

- Feasibility study
- Requirements analysis and specification
- Design
- Coding and unit testing
- Integration and system testing
- Maintenance

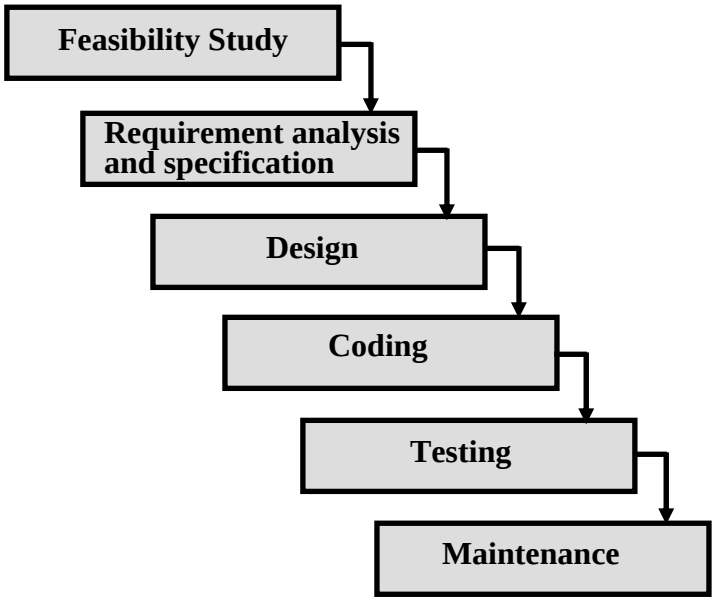


Fig. 33.7 Classical Waterfall Model

3.2.1. Feasibility Study

The main aim of feasibility study is to determine whether it would be financially and technically feasible to develop the product

- At first project managers or team leaders try to have a rough understanding of what is required to be done by visiting the client side. They study different input data to the system and output data to be produced by the system. They study what kind of processing is needed to be done on these data and they look at the various constraints on the behaviour of the system.
- After they have an overall understanding of the problem, they investigate the different solutions that are possible. Then they examine each of the solutions in terms of what kinds of resources are required, what would be the cost of development and what would be the development time for each solution.
- Based on this analysis, they pick the best solution and determine whether the solution is feasible financially and technically. They check whether the customer budget would meet the cost of the product and whether they have sufficient technical expertise in the area of development.

The following is an example of a feasibility study undertaken by an organization. It is intended to give one a feel of the activities and issues involved in the feasibility study phase of a typical software project.

Case Study

A mining company named Galaxy Mining Company Ltd. (GMC) has mines located at various places in India. It has about fifty different mine sites spread across eight states. The company employs a large number of mines at each mine site. Mining being a risky profession, the company intends to operate a special provident fund, which would exist in addition to the standard provident fund that the miners already enjoy. The main objective of having the special provident fund (SPF) would be to quickly distribute some compensation before the standard provident amount is paid. According to this scheme, each mine site would deduct SPF instalments from each miner every month and deposit the same with the CSPFC (Central Special Provident Fund Commissioner). The CSPFC will maintain all details regarding the SPF instalments collected from the miners. GMC employed a reputed software vendor Adventure Software Inc. to undertake the task of developing the software for automating the maintenance of SPF records of all employees. GMC realized that besides saving manpower on bookkeeping work, the software would help in speedy settlement of claim cases. GMC indicated that the amount it could afford for this software to be developed and installed was 1 million rupees.

Adventure Software Inc. deputed their project manager to carry out the feasibility study. The project manager discussed the matter with the top managers of GMC to get an overview of the project. He also discussed the issues involved with the several field PF officers at various mine sites to determine the exact details of the project. The project manager identified two broad approaches to solve the problem. One was to have a central database which could be accessed and updated via a satellite connection to various mine sites. The other approach was to have local databases at each mine site and to update the central database periodically through a dial-up connection. These periodic updates could be done on a daily or hourly basis depending on the delay acceptable to GMC in invoking various functions of the software. The project manager found that the second approach was very affordable and more fault-tolerant as the local mine sites could still operate even when the communication link to the central database temporarily failed. The project manager quickly analyzed the database functionalities required, the user-interface issues, and the software handling communication with the mine sites. He arrived at a cost to develop from the analysis. He found that the solution involving maintenance of local databases at the mine sites and periodic updating of a central database was financially and technically feasible. The project manager discussed his solution with the GMC management and found that the solution was acceptable to them as well.

3.2.2. Requirements Analysis and Specification

The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document them properly. This phase consists of two distinct activities, namely

- Requirements gathering and analysis, and
- Requirements specification

The goal of the requirements gathering activity is to collect all relevant information from the customer regarding the product to be developed with a view to clearly understand the customer requirements and weed out the incompleteness and inconsistencies in these requirements.

The requirements analysis activity is begun by collecting all relevant data regarding the product to be developed from the users of the product and from the customer through interviews and discussions. For example, to perform the requirements analysis of a business accounting software required by an organization, the analyst might interview all the accountants of the organization to ascertain their requirements. The data collected from such a group of users usually contain several contradictions and ambiguities, since each user typically has only a partial and incomplete view of the system. Therefore it is necessary to identify all ambiguities and contradictions in the requirements and resolve them through further discussions with the customer. After all ambiguities, inconsistencies, and incompleteness have been resolved and all the requirements properly understood, the requirements specification activity can start. During this activity, the user requirements are systematically organized into a Software Requirements Specification (SRS) document.

The customer requirements identified during the requirements gathering and analysis activity are organized into an SRS document. The important components of this document are functional requirements, the non-functional requirements, and the goals of implementation.

3.2.3. Design

The goal of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language. In technical terms, during the design phase the software architecture is derived from the SRS document. Two distinctly different approaches are available: the traditional design approach and the object-oriented design approach.

Traditional design approach: Traditional design consists of two different activities; first a structured analysis of the requirements specification is carried out where the detailed structure of the problem is examined. This is followed by a structured design activity. During structured design, the results of structured analysis are transformed into the software design.

Object-oriented design approach: In this technique, various objects that occur in the problem domain and the solution domain are first identified, and the different relationships that exist among these objects are identified. The object structure is further refined to obtain the detailed design.

3.2.4. Coding and Unit Testing

The purpose of the coding and unit testing phase (sometimes called the implementation phase) of software development is to translate the software design into source code. Each component of the design is implemented as a program module. The end-product of this phase is a set of program modules that have been individually tested.

During this phase, each module is unit tested to determine the correct working of all the individual modules. It involves testing each module in isolation as this is the most efficient way to debug the errors identified at this stage.

3.2.5. Integration and System Testing

Integration of different modules is undertaken once they have been coded and unit tested. During the integration and system testing phase, the modules are integrated in a planned manner.

The different modules making up a software product are almost never integrated in one shot. Integration is normally carried out incrementally over a number of steps. During each integration step, the partially integrated system is tested and a set of previously planned modules are added to it. Finally, when all the modules have been successfully integrated and tested, system testing is carried out. The goal of system testing is to ensure that the developed system conforms to the requirements laid out in the SRS document. System testing usually consists of three different kinds of testing activities:

- α – testing: It is the system testing performed by the development team.
- β – testing: It is the system testing performed by a friendly set of customers.
- Acceptance testing: It is the system testing performed by the customer himself after product delivery to determine whether to accept or reject the delivered product.

System testing is normally carried out in a planned manner according to the system test plan document. The system test plan identifies all testing-related activities that must be performed, specifies the schedule of testing, and allocates resources. It also lists all the test cases and the expected outputs for each test case.

3.2.6. Maintenance

Maintenance of a typical software product requires much more than the effort necessary to develop the product itself. Many studies carried out in the past confirm this and indicate that the relative effort of development of a typical software product to its maintenance effort is roughly in the 40:60 ratio. Maintenance involves performing any one or more of the following three kinds of activities:

- Correcting errors that were not discovered during the product development phase. This is called corrective maintenance.
- Improving the implementation of the system, and enhancing the functionalities of the system according to the customer's requirements. This is called perfective maintenance.
- Porting the software to work in a new environment. For example, porting may be required to get the software to work on a new computer platform or with a new operating system. This is called adaptive maintenance.

3.2.7. Shortcomings of the Classical Waterfall Model

The classical waterfall model is an idealistic one since it assumes that no development error is ever committed by the engineers during any of the life cycle phases. However, in practical development environments, the engineers do commit a large number of errors in almost every phase of the life cycle. The source of the defects can be many: oversight, wrong assumptions, use of inappropriate technology, communication gap among the project engineers, etc. These defects usually get detected much later in the life cycle. For example, a design defect might go unnoticed till we reach the coding or testing phase. Once a defect is detected, the engineers need to go back to the phase where the defect had occurred and redo some of the work done during that phase and the subsequent phases to correct the defect and its effect on the later phases. Therefore, in any practical software development work, it is not possible to strictly follow the classical waterfall model.

3.2.8. Phase-Entry and Phase-Exit Criteria

At the start of the feasibility study, project managers or team leaders try to understand what the actual problem is, by visiting the client side. At the end of that phase, they pick the best solution and determine whether the solution is feasible financially and technically.

At the start of requirements analysis and specification phase, the required data is collected. After that requirement specification is carried out. Finally, SRS document is produced.

At the start of design phase, context diagram and different levels of DFDs are produced according to the SRS document. At the end of this phase module structure (structure chart) is produced.

During the coding phase each module (independently compilation unit) of the design is coded. Then each module is tested independently as a stand-alone unit and debugged separately. After this each module is documented individually. The end product of the implementation phase is a set of program modules that have been tested individually but not tested together.

After the implementation phase, different modules which have been tested individually are integrated in a planned manner. After all the modules have been successfully integrated and tested, system testing is carried out.

Software maintenance denotes any changes made to a software product after it has been delivered to the customer. Maintenance is inevitable for almost any kind of product. However, most products need maintenance due to the wear and tear caused by use.

3.3. Prototyping Model

A prototype is a toy implementation of the system. A prototype usually exhibits limited functional capabilities, low reliability, and inefficient performance compared to the actual software. A prototype is usually built using several shortcuts. The shortcuts might involve using inefficient, inaccurate, or dummy functions. The shortcut implementation of a function, for example, may produce the desired results by using a table look-up instead of performing the actual computations. A prototype usually turns out to be a very crude version of the actual system.

3.3.1. The Need for a Prototype

There are several uses of a prototype. An important purpose is to illustrate the input data formats, messages, reports, and the interactive dialogues to the customer. This is a valuable mechanism for gaining better understanding of the customer's needs.

- how screens might look like
- how the user interface would behave
- how the system would produce outputs, etc.

This is something similar to what the architectural designers of a building do; they show a prototype of the building to their customer. The customer can evaluate whether he likes it or not and the changes that he would need in the actual product. A similar thing happens in the case of a software product and its prototyping model.

32.

3.4. Spiral Model

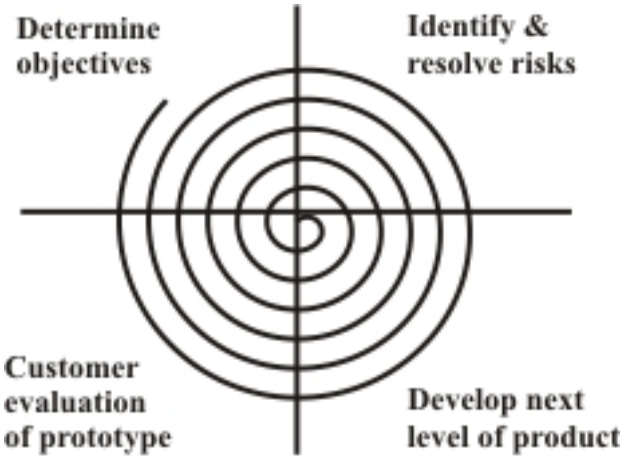


Fig. 33.8 Spiral Model

The Spiral model of software development is shown in fig. 33.8. The diagrammatic representation of this model appears like a spiral with many loops. The exact number of loops in the spiral is not fixed. Each loop of the spiral represents a phase of the software process. For example, the innermost loop might be concerned with feasibility study; the next loop with requirements specification; the next one with design, and so on. Each phase in this model is split into four sectors (or quadrants) as shown in fig. 33.8. The following activities are carried out during each phase of a spiral model.

First quadrant (Objective Setting):
- During the first quadrant, we need to identify the objectives of the phase.
- Examine the risks associated with these objectives

Second quadrant (Risk Assessment and Reduction):
- A detailed analysis is carried out for each identified project risk.
- Steps are taken to reduce the risks. For example, if there is a risk that the requirements are inappropriate, a prototype system may be developed

Third quadrant (Objective Setting):
- Develop and validate the next level of the product after resolving the identified risks.

Fourth quadrant (Objective Setting):
- Review the results achieved so far with the customer and plan the next iteration around the spiral.
- With each iteration around the spiral, progressively a more complete version of the software gets built.

3.4.1. A Meta Model

The spiral model is called a meta-model since it encompasses all other life cycle models. Risk handling is inherently built into this model. The spiral model is suitable for development of

technically challenging software products that are prone to several kinds of risks. However, this model is much more complex than the other models. This is probably a factor deterring its use in ordinary projects.

3.5. Comparison of Different Life Cycle Models

The classical waterfall model can be considered as the basic model and all other life cycle models as embellishments of this model. However, the classical waterfall model can not be used in practical development projects, since this model supports no mechanism to handle the errors committed during any of the phases.

This problem is overcome in the iterative waterfall model. The iterative waterfall model is probably the most widely used software development model evolved so far. This model is simple to understand and use. However, this model is suitable only for well-understood problems; it is not suitable for very large projects and for projects that are subject to many risks.

The prototyping model is suitable for projects for which either the user requirements or the underlying technical aspects are not well understood. This model is especially popular for development of the user-interface part of the projects.

The evolutionary approach is suitable for large problems which can be decomposed into a set of modules for incremental development and delivery. This model is also widely used for object-oriented development projects. Of course, this model can only be used if the incremental delivery of the system is acceptable to the customer.

The spiral model is called a meta-model since it encompasses all other life cycle models. Risk handling is inherently built into this model. The spiral model is suitable for development of technically challenging software products that are prone to several kinds of risks. However, this model is much more complex than the other models. This is probably a factor deterring its use in ordinary projects.

The different software life cycle models can be compared from the viewpoint of the customer. Initially, customer confidence in the development team is usually high irrespective of the development model followed. During the long development process, customer confidence normally drops, as no working product is immediately visible. Developers answer customer queries using technical slang, and delays are announced. This gives rise to customer resentment. On the other hand, an evolutionary approach lets the customer experiment with a working product much earlier than the monolithic approaches. Another important advantage of the incremental model is that it reduces the customer's trauma of getting used to an entirely new system. The gradual introduction of the product via incremental phases provides time to the customer to adjust to the new product. Also, from the customer's financial viewpoint, incremental development does not require a large upfront capital outlay. The customer can order the incremental versions as and when he can afford them.

3.6. Exercises

1. Mark the following as True or False. Justify your answer.
   a. All software engineering principles are backed by either scientific basis or theoretical proof.
   b. There are well defined steps through which a problem is solved using an exploratory style.
   c. Evolutionary life cycle model is ideally suited for development of very small software products typically requiring a few months of development effort.

d. Prototyping life cycle model is the most suitable one for undertaking a software development project susceptible to schedule slippage.
e. Spiral life cycle model is not suitable for products that are vulnerable to a large number of risks.

2. For the following, mark all options which are true.
   a. Which of the following problems can be considered to be contributing to the present software crisis?
      - large problem size
      - lack of rapid progress of software engineering
      - lack of intelligent engineers
      - shortage of skilled manpower
   b. Which of the following are essential program constructs (i.e. it would not be possible to develop programs for any given problem without using the construct)?
      - Sequence
      - Selection
      - Jump
      - Iteration
   c. In a classical waterfall model, which phase precedes the design phase?
      - Coding and unit testing
      - Maintenance
      - Requirements analysis and specification
      - Feasibility study
   d. Among development phases of software life cycle, which phase typically consumes the maximum effort?
      - Requirements analysis and specification
      - Design
      - Coding
      - Testing
   e. Among all the phases of software life cycle, which phase consumes the maximum effort?
      - Design
      - Maintenance
      - Testing
      - Coding
   f. In the classical waterfall model, during which phase is the Software Requirement Specification (SRS) document produced?
      - Design
      - Maintenance
      - Requirements analysis and specification
      - Coding
   g. Which phase is the last development phase in the classical waterfall software life cycle?
      - Design
      - Maintenance
      - Testing
      - Coding

h. Which development phase in classical waterfall life cycle immediately follows coding phase?
   - Design
   - Maintenance
   - Testing
   - Requirement analysis and specification

3. Identify the problem one would face, if he tries to develop a large software product without using software engineering principles.
4. Identify the two important techniques that software engineering uses to tackle the problem of exponential growth of problem complexity with its size.
5. State five symptoms of the present software crisis.
6. State four factors that have contributed to the making of the present software crisis.
7. Suggest at least two possible solutions to the present software crisis.
8. Identify at least four basic characteristics that differentiate a simple program from a software product.
9. Identify two important features of that a program must satisfy to be called as a structured program.
10. Explain exploratory program development style.
11. Show at least three important drawbacks of the exploratory programming style.
12. Identify at least two advantages of using high-level languages over assembly languages.
13. State at least two basic differences between control flow-oriented and data flow-oriented design techniques.
14. State at least five advantages of object-oriented design techniques.
15. State at least three differences between the exploratory style and modern styles of software development.
16. Explain the problems that might be faced by an organization if it does not follow any software life cycle model.
17. Differentiate between structured analysis and structured design.
18. Identify at least three activities undertaken in an object-oriented software design approach.
19. State why it is a good idea to test a module in isolation from other modules.
20. Identify why different modules making up a software product are almost never integrated in one shot.
21. Identify the necessity of integration and system testing.
22. Identify six different phases of a classical waterfall model. Mention the reasons for which classical waterfall model can be considered impractical and cannot be used in real projects.
23. Explain what a software prototype is. Identify three reasons for the necessity of developing a prototype during software development.
24. Explain the situations under which it is beneficial to develop a prototype during software development.
25. Identify the activities carried out during each phase of a spiral model. Discuss the advantages of using spiral model.