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

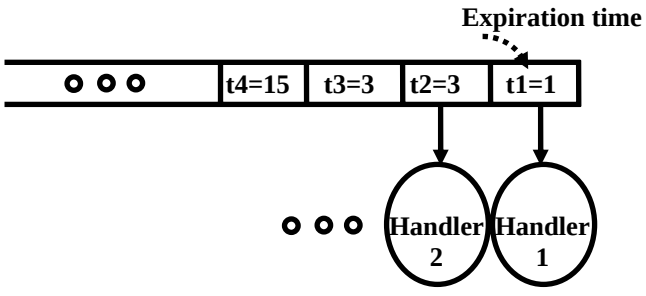

Fig. 31.1 Structure of a Timer Queue

[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.

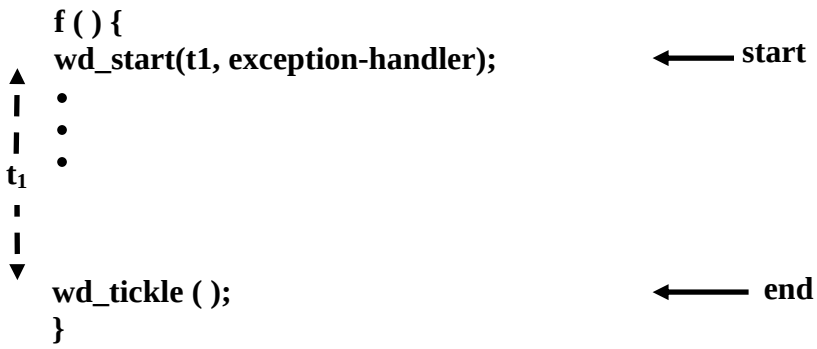

Fig. 31.2 Use of a Watchdog Timer

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.

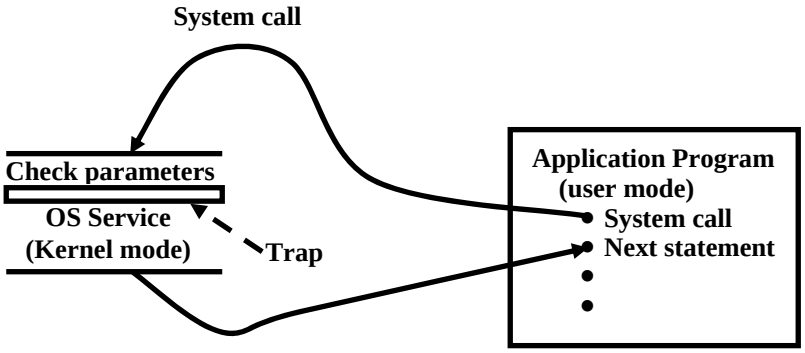

Fig. 31.3 Invocation of an Operating System Service through System Call

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.

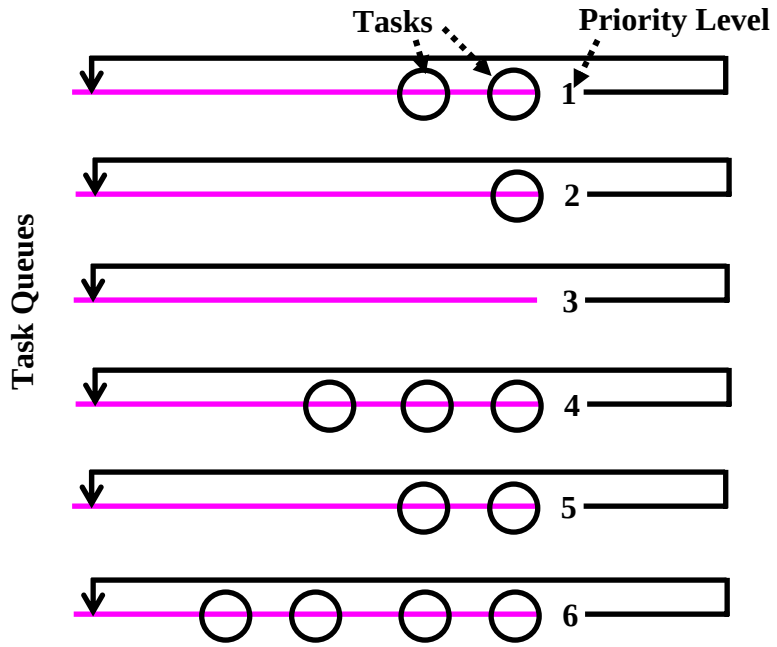

Fig. 31.4 Multi-Level Feedback Queues

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.

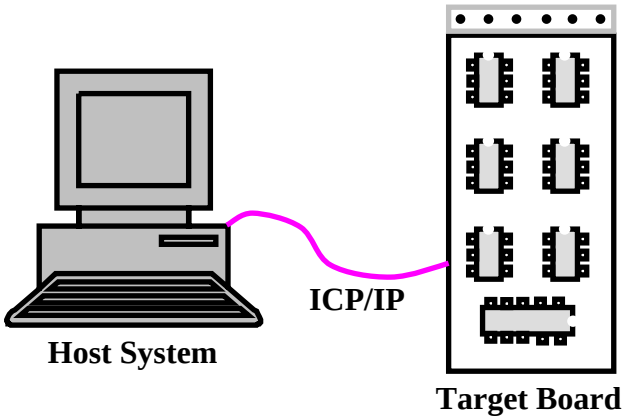

Fig. 31.5 Schematic Representation of a Host-Target System

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.

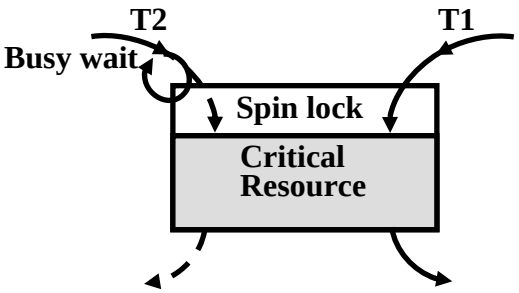

Fig. 31.6 Operation of a Spin Lock

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.

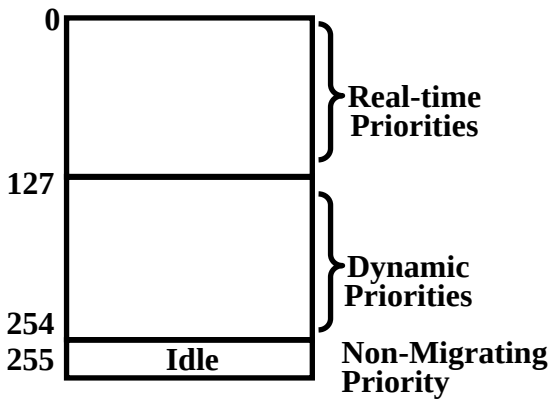

Fig. 31.7 Priority Changes in Self-host Unix Systems

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.

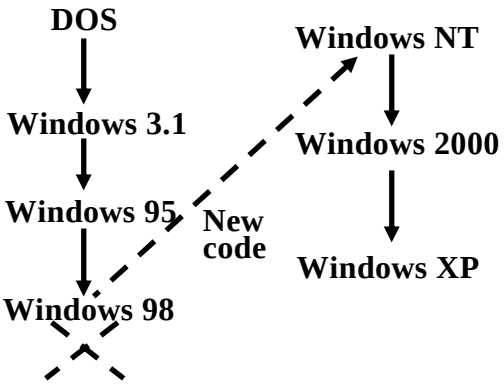

Fig. 31.8 Genealogy of Operating Systems from Microsoft's Stable

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.

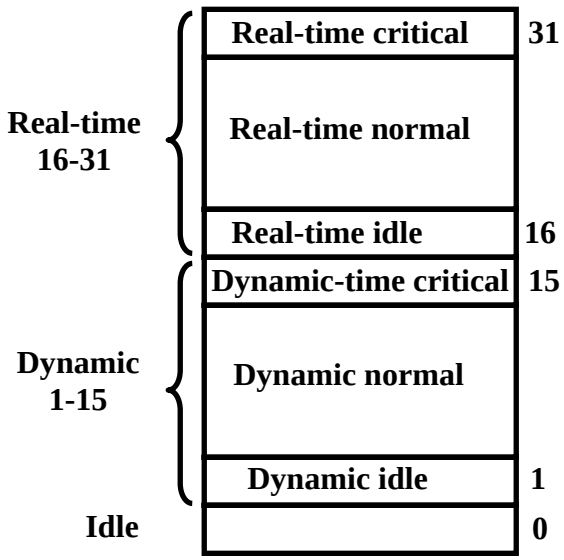

Fig. 31.9 Task Priorities in Windows NT

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.