Commercial Real-Time Operating Systems

In the last three Chapters 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 Chapter, we examine the important features that a real-time operating system is expected to support. Unless these features are adequately supported by an operating system, it becomes to satisfactorily implement certain categories of real-time applications on this operating system. We discuss to what extent these required features are supported by the various commercially available real-time operating systems. To gain a better insight, we also investigate the internals of the operating systems to examine the exact ways in which the required features are supported.

To appreciate some of the fundamental issues affecting the design and development of a satisfactory real-time operating system, we discuss the problems that would crop up if one attempts to use a general purpose operating system such as Unix or Windows for developing real-time applications. Many real-time operating systems are at present available commercially. We analyze some popular real-time operating systems, and investigate why these popular systems cannot be used across all applications. We also examine the POSIX standard for real-time operating systems and its implications.

This chapter is organized as follows. First, we discuss the important features that are usually required to be supported by a real-time operating system. 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. Subsequently, we discuss the issues that arise if we attempt to use a general purpose operating system such as Unix or Windows in a real-time applications. Next, we explain how some of the fundamental problems associated with the traditional operating systems (as far as real-time applications are concerned) are overcome in the the contemporary real-time operating systems. We then survey some of the important features of the different real-time operating systems that are being commercially used. Finally, we identify some of the important parameters based on which various real-time systems can be benchmarked.

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. The system clock is maintained by the operating system kernel based on the interrupts received from the hardware clock. Since hard real-time systems usually have timing constraints that are of the order of a few micro seconds, 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 fine resolution system clocks. In current technology, the resolution of hardware clocks is 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 only of the order of several milli seconds 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 in a computer 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 5.1.1). 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 that the response time of the `clock-gettime()` system call is not deterministic. In fact, every system call (or for that matter every function call) invocation has some associated jitter. 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). The jitter is caused partly on account of interrupts having higher priority than system calls. When an interrupt occurs, the processing of a system call is stalled. The problem gets aggravated in the following situation. 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. In commercially available operating systems, jitters associated with system calls can be several milliseconds. This jitter introduces an error in the time readings obtained by a program. 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 sufficiently fine resolution can be provided in the presence of jitter in function calls.

1.1 Clock Interrupt Processing

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

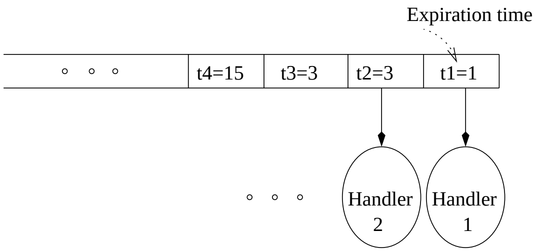


Figure 1: Structure of a Timer Queue

- **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. 5.1. 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 needs to be invoked when the timer expires. At each clock interrupt,

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

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. Recollect from a basic course on operating system that arriving tasks and tasks waiting for some event (such as a page fetch or a semaphore event) are queued in a wait queue. The tasks in the wait queue are checked to see if any task has become ready in the mean while. 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 for the task becomes zero and the task is not yet complete, then the task is preempted, and the scheduler is invoked to select another task to run.

1.2 Providing High Clock Resolution

We had pointed out in Sec. 5.1 that there are two main difficulties in providing a high resolution timer. First, the overhead associated with processing the clock interrupt becomes excessive as the time service resolution gets finer. 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 nano seconds. For such applications, an important question is: Is it at all possible to support time measurement with nano second 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. For example, 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 on and increments after each hardware clock interrupt. 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. For example, when an application running on a Pentium processor is ported to a different processor, the new processor may not have a high resolution counter, and certainly the memory address map and resolution would differ.

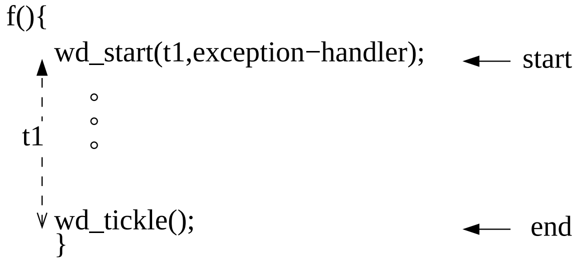
1.3 Timers

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, it expires periodically. This is implemented using the timer queue as follows. Each time the periodic timer expires, the corresponding handler routine is invoked, and the timer data structure is inserted back into the timer queue (see Fig. 5.1). 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 a popular example of one shot timers.

Watchdog timers are used extensively in real-time programs to detect if 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



```
f(){
  wd_start(t1,exception−handler);
     o
     o
     o
  wd_tickle();
  }
```

Figure 2: Use of a Watchdog Timer

illustrated in Fig. 5.2. In Fig. 5.2, 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 (t1 time units) from the starting of the task. If the function `f()` does not complete even after t_1 time units have elapsed, then the watchdog timer expires, indicating that the task deadline is 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.

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. This will also enable us to compare the features supported by different real-time operating systems for selecting an appropriate operating system for an application at hand. 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. Still finer resolutions 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 normally 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.* We discuss in section 5.3 that a traditional operating system dynamically changes the priority levels of tasks from programmer assigned values to maximize system throughput. Such priority levels that are changed dynamically by the operating system 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 *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 typically 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, but executes after ISR completes at a lower 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: We had already discussed in Chapter 3 that 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 discussed in Chapter 3. 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 real-time tasks of an 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 a page fault 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 all 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 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”.

Support for Asynchronous I/O: Asynchronous I/O means non-blocking I/O. Traditional read() or write() system calls perform synchronous I/O. If a process attempts to read or write using the normal, synchronous read() or write() system calls, then it needs to wait until the hardware has completed the physical I/O. It is then informed of the success or failure of the operation, and of course the required data in the case of a successful read. Thus in case of synchronous or blocking I/O execution of the process is blocked while it waits for the results of the system call.

However, if a process uses asynchronous *aio_read*() or *aio_write*() system calls (called *aioread*() and *aiowrite*() in some operating systems), then the system call will return immediately once the I/O request has been passed down to the hardware or queued in the operating system, typically before the physical I/O operation has even begun. The execution of the process is not blocked, because it does not need to wait for the results of the system call. Instead, it can continue executing and then receive the results of the I/O operation later, once they are available.

Additional Requirements for Embedded Real-Time Operating Systems: Embedded applications usually have constraints on cost, size, and power consumption. Embedded real-time operating systems are therefore often required to be capable of diskless operation. This is because disks are usually too bulky to use in embedded systems, increase power consumption, and also increase the cost of deployment. For this reason, of late flash memory is being increasingly used for this purpose.

Embedded operating systems usually reside on either flash memory or ROM. For certain applications which require faster response, it may be necessary to run the real-time operating system on a RAM. This would result in faster execution, since the access time of a RAM is lower than that of a ROM. Irrespective of whether ROM or RAM is used, all ICs are expensive. Therefore, for real-time operating systems designed 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 of rupees in profit.

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.

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

We elaborate these problems in the following two subsections.

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 can 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 (see Fig. 5.3). 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. 5.3.

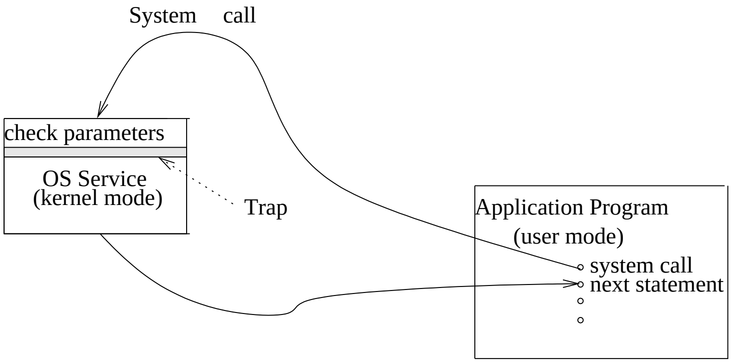


Figure 3: Invocation of An Operating System Service Through A System Call

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 unsatisfactory in general applications as well.

In Unix, 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 by the low priority process completes. For real-time applications, this causes a priority inversion. The longest system calls may take up to several hundreds of milli seconds 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. In Unix, when a kernel 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 of locks and resulted in lower average task preemption times. Though a non-preemptive kernel can result in worst-case task response time of up to a second, it was considered acceptable by the 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. 5.4.4 we investigate how modern real-time operating systems make the kernel preemptive without through use of kernel-level and spin locks.

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

3.2 Dynamic Priority Levels

In traditional Unix systems, real-time tasks can not be assigned static priority values. Soon after a programmer sets a priority value for a task, the operating system keeps on altering it during the course of execution of the task. 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 needs to dynamically change the priority values of tasks.

Unix uses round robin scheduling of tasks with multilevel feedback. In this scheme, the scheduler arranges tasks in multilevel queues as shown in Fig. 5.4. 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.

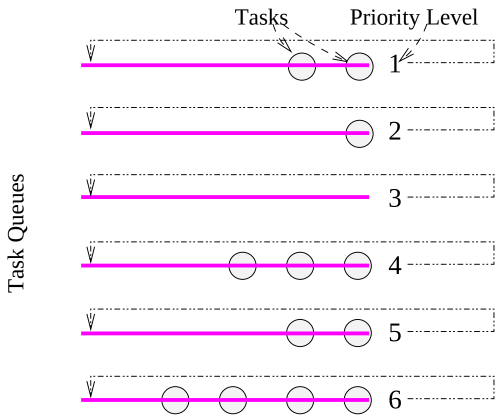


Figure 4: Multi-Level Feedback Queues

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 jth time slice using the following two expressions:

$$pri(T_i, j) = Base(T_i) + CPU(T_i, j) + nice(T_i) \qquad \ldots(5.1)$$

$$CPU(T_i, j) = \frac{U(T_i, j-1)}{2} + \frac{CPU(T_i, j-1)}{2} \qquad \ldots(5.2)$$

where $pri(T_i, j)$ is the priority of the task T_i at the end of its jth time slice; $U(T_i, j)$ is the utilization of the task T_i for its jth time slice, and $CPU(T_i, j)$ is the weighted history of CPU utilization of the task T_i at the end of its jth 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. 5.2 has been recursively defined. Unfolding the recursion, we get:

$$CPU(T_i, j) = \frac{U(T_i, j-1)}{2} + \frac{U(T_i, j-2)}{4} + \ldots \qquad \ldots(5.3)$$

It can be easily seen from Expr. 5.3 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 a task uses up CPU for the full duration of its allotted time slice (i.e. 100% CPU utilization), then $CPU(T_i, j)$ is computed (by Expr. 5.3) to be a high value — indicating a lowering of the priority of the task. On the other hand, if the task immediate blocked for I/O as soon as it started computing during its allotted time slice, then $CPU(T_i, j)$ would be computed as a low value, indicating an increase of the priority of the task. 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 jth time slice.

Now, substituting Expr 5.3 in Expr. 5.1, we get:

$$pri(T_i, j) = Base(T_i) + \frac{U(T_i, j-1)}{2} + \frac{U(T_i, j-2)}{4} + \ldots + nice(T_i) \qquad \ldots(5.4)$$

The purpose of the base priority term ($Base(T_i)$) in the priority computation expression (Expr. 5.4) is to divide all tasks into a set of fixed bands of priority levels. Once a task is assigned to a priority level, it is not possible for the task to move out from its assigned band to other priority bands due to dynamic priority recomputations. The values of $U(T_i, j)$ and $nice(T_i)$ components are deliberately restricted to be small enough to prevent a process from migrating from its assigned band. The bands have been designed to optimize I/O completion times, especially block I/O.

Dynamic recomputation of priorities was motivated from the following consideration. Unix designers observed that in any computer system, I/O transfer rate is primarily responsible for any slow response time. Processors are extremely fast compared to the transfer rates of I/O devices. Delay caused by I/O transfers therefore are the bottleneck in achieving faster task response times. To mitigate this problem, it is desirable to keep I/O channels as busy as possible. This can be achieved by assigning the I/O bound tasks high priorities.

As already mentioned, Unix has a set of priority bands to which different types of tasks are assigned. 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. But, when are block I/O required? To give an example of block I/O, consider the I/O that occurs while handling a page fault in a virtual memory system. Block I/O uses DMA-based transfer, and hence makes 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.

To keep the I/O channels busy, any task performing I/O should not be kept waiting very long for CPU. For this reason, as soon as a task blocks for I/O, its priority is increased by the priority recomputation rule given in Expr. 5.4. 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, such as Microsoft's Windows operating systems.

We can now state from the above observations that the overall effect of periodic recomputation of task priority values using Expr. 5.4 is as follows :

> In Unix, dynamic priority computations cause I/O intensive tasks to migrate to higher and higher priority levels, whereas CPU-intensive tasks are made to 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 recomputation 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 inappropriate, as it prevents tasks being scheduled at high priority levels, and also prevents scheduling under popular real-time task scheduling algorithms such as EDF and RMA.

3.3 Other Deficiencies of Unix

We have so far discussed two glaring shortcomings of Unix in handling the requirements of real-time applications: dynamic priority recomputations and non-preemptable kernel. 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 or 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 preallocates 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 milli seconds, which is too coarse for many hard real-time applications.

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.

4.1 Extensions To The Traditional Unix Kernel

A naive attempt made 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 fell wide short of the requirements of hard real-time applications.

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 a 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. 5.5.

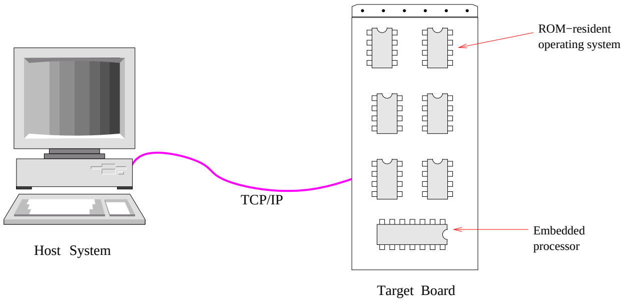


Figure 5: Schematic Representation of a Host-Target System

The main idea behind this approach is that the real-time operating system running on the target board needs to 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 would 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 is a Unix or Windows-based system supporting 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. 5.5). 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 Section 5.7. We would point out that the target 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 micro seconds compared to several hundreds of milli seconds for traditional Unix systems.

4.3 Preemption Point Approach

We have already pointed out that one of the major shortcomings of the traditional Unix V code arises from the fact 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 times of the order of a 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 structure is consistent. At such points, the kernel can safely be preempted to make way for any waiting higher priority real-time tasks to run 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 improves several folds compared to those for traditional operating systems without preemption points. This makes preemption point-based operating systems suitable for use in many categories of hard real-time applications, though it still falls short of the requirements of hard real-time applications requiring preemption latency of the order of a few micro seconds or less. Another advantage of the preemption point approach is that it involves only minor changes to be made to the kernel code. Many operating systems in fact have taken the preemption point approach in the past. Prominent commercial examples of this approach include HP-UX and Windows CE.

4.4 Self-Host Systems

Unlike the host-target approach where application development is carried out on a separate host 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 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 systems take a different approach. The real-time application is developed on the full-fledged operating system. Once the application runs satisfactorily on the host, it is fused on a ROM or flash memory on the target board along with a possibly stripped down version of the operating system.

Most of the self-host operating systems that are currently available, are based on micro-kernel architecture. Use of a microkernel architecture for a self-host operating system entails several advantages.

> In a micro-kernel 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.

The add-on modules can be easily excluded, whenever these are not required. As a result, it becomes very easy to configure the operating system, resulting in a small-sized system. Also, the micro kernel is lean and therefore becomes much more efficient compared to a monolithic one. Another difficulty with 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 usually very reliable.

We have already discussed in section 5.3 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. 5.3.2. 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 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 in ensuring the integrity of the kernel data structures, as the tasks running on other processors can still corrupt the kernel data structure.

It is necessary to use locks at appropriate places in the kernel code to overcome the problem. The fundamental issue surrounding locking is the need to provide synchronization in certain code segments in the kernel. These code segments, called critical sections. Without proper locking race conditions might develop. Just to exemplify the

problem, consider the case where there are two processes and each process needs to increment the value of a shared variable i. Suppose one process reads i, and then the other. They both increment it, then they both write i back to memory. If i was originally 2, it would now be 3, instead of 4!
It should be clear that in order to make the kernel preemptive, locks must be used at appropriate places in the kernel code.

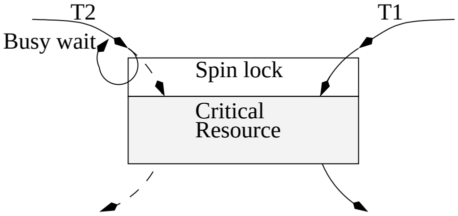


Figure 6: Operation of a Spin Lock

In fully preemptive Unix systems, normally two types of locks are used: kernel-level locks, and spin locks. A kernel-level lock is similar to a traditional lock. When a task waits for a kernel-level lock to be released by another task holding it, 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. Kernel-level locks are therefore inefficient when critical resources are required for short durations of times that are comparable to context switching times, i.e. of the order of a few milli seconds or less. In such situations, context switching overheads can increase the task response times unduly and are not acceptable. Let us elaborate this Consider that some task T_i requires the lock for carrying out very small processing (possibly a single arithmetic operation) on some critical resource that would require, say 1 msec of processing. Now, if a kernel level lock is used, another task s holding the lock at the time T_i request the lock, it would be blocked and a context switch would be incurred, also the cache contents, pages of the task etc. may be swapped. After a short time, say 1 msec, the lock becomes available. The, another context switch would be incurred for T_i to run assuming that it is the highest priority task at that time. Hence, the response time of T_i would be 3 msec — though if it had simply busy waited for the lock to be release, then the response time would have been 2 msec. In such a situation, a spin lock would be appropriate.

Let us now understand the operation of a spin lock. A spin lock has schematically been shown in Fig. 5.6. In Fig. 5.6 a critical resource is required by both 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. Suppose, task T_1 has acquired the spin lock guarding the resource, and in the meanwhile, task T_2 requests the resource. Since task T_1 has locked it, 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 a context switch. T_2 gets the resource as soon as T_1 relinquishes the resource.

In a multiprocessor system, spin locks are normally implemented using the cache coherency protocol used, such that each processor loops on a local copy of the lock variable. The exact details of implementation of a spin lock is beyond the scope of this book, and the interested reader is referred to [1]. Let us now discuss how a spin lock can be implemented on a uniprocessor system. On a uniprocessor system, when a critical resource is required for a very short time, mutual exclusion is easily accomplished by disabling interrupts. On Intel x86 systems, this is done with the "cli" instruction. Interrupts are re-enabled with "sti". Thus, spin locks on a uniprocessor get compiled in as calls to "cli" and "sti".

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. 5.7 schematically shows the three available priority levels. We now briefly discuss these three categories of priority levels.

Idle(Non-Migrating). This is the lowest priority level. The task that runs when there are no other tasks to run (the idle task), 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 (interactive) tasks. Dynamic recomputation 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. 5.7, tasks at the dynamic priority level operate at priorities higher than the idle priority, but at lower priority than the real-time priorities.

Real-Time. Real-time priorities are static priorities and are not recomputed during run time. Hard real-time tasks operate at these levels. Tasks having real-time priorities get precedence over tasks with dynamic priority levels (see Fig. 5.7).

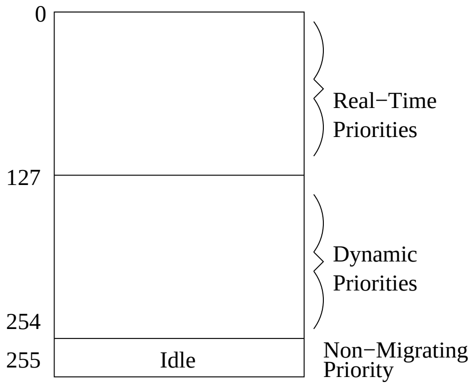


Figure 7: Priority Levels in Self-host Unix Systems

5 Windows As A Real-Time Operating System

Microsoft's Windows series of operating systems are extremely popular in desktop computers. Windows operating systems have evolved over the last twenty five years from the naive DOS (Disk Operating System). Microsoft developed DOS in the early Eighties. DOS was a very simple operating system that was single tasking and used a segmented memory management scheme. Microsoft kept on announcing new versions of DOS almost every year and each successive version supported new features. DOS evolved to the Windows series operating systems in the late Eighties. The main distinguishing feature of the Windows operating system from DOS was a graphical front-end. As several new versions of Windows kept on appearing through enhancements to the DOS code, the structure of the code degenerated to such an extent that it had become very difficult to debug and maintain the code. 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 and the DOS-based systems were scrapped. Fig. 5.8 shows the genealogy of the various operating systems from the Microsoft stable. As already mentioned, stability of ancestors of Windows NT was not satisfactory. Because stability of the operating system is a major concern in hard real-time applications, we restrict our discussions to only the Windows NT and its descendants and do not include the DOS line of products.

Computer systems based on Windows NT and its descendants are being extensively used in homes, offices, and industrial establishments. An organization owning Windows NT systems might be interested to use them for its real-time applications for 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

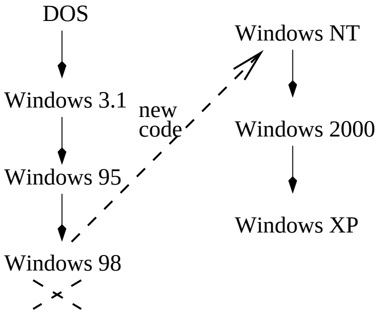


Figure 8: Genealogy of Operating Systems from Microsoft's Stable

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.

5.1 Important Features of Windows NT

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

Windows NT supports 32 priority levels (see Fig. 5.9). Each process belongs to one of the following priority classes: `idle, normal, high, real-time`. By default, the priority class at which a user task runs is `normal`. Both `normal` and `high` priority classes are variable type, in the sense that priorities of tasks in this class are recomputed periodically by the operating system. 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. NT uses priority-driven preemptive 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`.

5.2 Shortcomings of Windows NT

In spite of the impressive support that Windows provides for real-time programming (as discussed in subsection 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 problems are the most troublesome. We discuss these two problems in the following.

- **Interrupt Processing.** In Windows NT, the 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 very critical processing is performed in ISR and the bulk of the processing is done later at a lower priority in the form of 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 all DPCs at the same priority values. A DPC can not be preempted by another DPC but can be preempted by an interrupt. DPCs are executed in FIFO order at a priority lower than the hardware interrupt priority but higher than the priority of the scheduler/dispatcher. 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. As a result, 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. Therefore in presence of processes carrying out network or disk I/O, the effect of system-wide FIFO queues of DPCs may lead to unbounded response times even for real-time threads. This problem has been avoided in the Windows CE operating system (see Sec. 5.8.8) through the use of a priority inheritance mechanism.

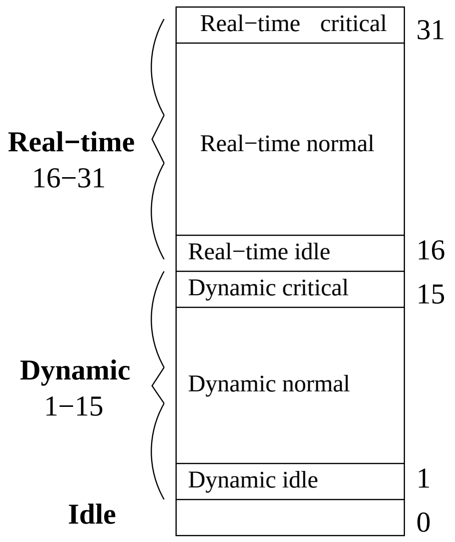


Figure 9: Task Priorities in Windows NT

- **Support for Resource Sharing Protocols.** We had discussed in Chapter 3 that unless appropriate resource sharing protocols supported by an operating system, tasks of a real-time application 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 resources among themselves. This is a major shortcoming of Windows NT when used in real-time applications. Since most real-time applications do require sharing critical resources among tasks, we outline below a few 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 by careful priority settings while acquiring and releasing locks. 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 can be restored. However, we know that this arrangement would lead to large inheritance-related inversions (See Sec. 3.4).

  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) only. The reason for this restriction is that Windows 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 to assign to tasks.

5.3 Windows NT versus Unix

In this section, we compare Windows NT and Unix with respect to their suitability for deployment in real-time applications.

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

Table 5.1 Windows NT versus Unix

A comparison of the extent to which some of the the basic features required for real-time programming are provided by Windows NT and Unix V is indicated in Table 5.1. It can be seen that Windows NT supports a much finer time granularity (1 mSec) compared to Unix (10 mSec). Windows NT has many of the features desired of a real-time operating system. However, the way it executes DPCs, together with its lack of protocol support for resource sharing among equal priority tasks makes it unsuitable for use in safety-critical hard real-time applications. With careful programming, Windows NT can successfully be used for applications that can tolerate occasional deadline misses, and have deadlines of the order of hundreds of milliseconds rather than a few 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.

Though Windows based systems are popular for desktop applications and provide most of the desired features required for real-time programming, Unix-based systems are overwhelmingly popular for real-time applications due to cost considerations and the ‘hacker mentality’ of the real-time programmers [3].

6 POSIX

POSIX stands for Portable Operating System Interface. The letter "X" has been suffixed to the abbreviation to make it sound Unix-like. Over the last decade, POSIX has become an important standard for operating systems, including real-time operating systems. The importance of POSIX can be gauzed from the fact that nowadays it is rare to come across a commercial operating system that is not POSIX-compliant. POSIX started as an open software initiative, but has now almost become a *de facto* standard for operating systems. Since POSIX has now become overwhelmingly popular, we discuss the POSIX standard for real-time operating systems. We start with a brief introduction to the 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.

6.1 Open Software

Before we discuss open software, let us discuss open systems, which has a much wider connotation. 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 copy righted, saving users from expensive intellectual property right (IPR) law suits. Open system advocates standard interfaces for similar products; so that users can easily integrate their application with the products supplied by any vendor. This leads to vendor-neutral solutions. The most important goals 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.

Open software holds out tremendous advantages to both the users as well as system developers. 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 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 exist.

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

6.2 Genesis of POSIX

Before we discuss the different features of the POSIX standard in the next subsection, let us understand the historical events that led to the development of POSIX. Unix was originally developed by AT&T Bell Labs in the early seventies. 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 recipients of Unix source code.

AT&T later got interested in computers. It 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) grant. UCB came up with its own version of Unix and named it as BSD (Berkeley Software Distribution). At this time, the commercial importance of Unix started to grow very 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 are some of the prominent examples. 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 Unix 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.

6.3 Overview of POSIX

POSIX is an off-shoot of the open software movement. A major concern of POSIX is the portability of applications across different variants of Unix operating systems. However, POSIX has been so widely accepted now that even non-Unix

> 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 source code-level compatibility, POSIX specifies the system calls that an operating system needs to support, the exact parameters of these system calls, and the semantics of the different system calls. Trying not to be unnecessarily restrictive, POSIX leaves the operating system vendors the freedom to implement the system calls as per their design. The standard does not specify whether the operating system kernel must be single threaded or multithreaded or at what priority level the kernel services are to be executed, or in what programming language it must be written.

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

- 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.

6.4 Real-Time POSIX Standard

POSIX.4 deals with real-time extensions to POSIX and is also popularly 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:** A POSIX-RT compliant operating system must provide support for real-time (static) priorities.
- **Performance requirements on system calls.** Worst case execution times required for most real-time operating system services have been specified by POSIX-RT.
- **Priority levels:** The number of priority levels supported should be at least 32.
- **Timers:** Periodic and one shot timers (also called watchdog 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 preallocate storage for files and should be able to store file blocks contiguously on the disk. This enables to have predictable delay in file access.
- **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 by a real-time program.
- **Multithreading support:** POSIX-RT mandates threading support by an operating system. Real-time threads are schedulable entities of a real-time application that can have individual timeliness constraints and may have collective timeliness constraints when belonging to a runnable set of threads.

7 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. A study of the features of the commercially available real-time operating systems can give us an idea about which real-time operating system to use in a specific real-time application.

Before we survey some of the popular commercial real-time operating systems, we need to mention that many of these operating systems come with a set of tools to facilitate the development of real-time applications. Besides the general programming tools such as editors, compilers, and debuggers, several advanced tools specifically designed to help real-time application development are included. The tools include memory analyzers, performance profilers, simulators, etc.

7.1 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 [3]. It is a host-target type of real-time operating system (see Sec. 5.4.2). PSOS is being used in several commercial embedded products. An example application of PSOS is in the base stations of cell phone systems.

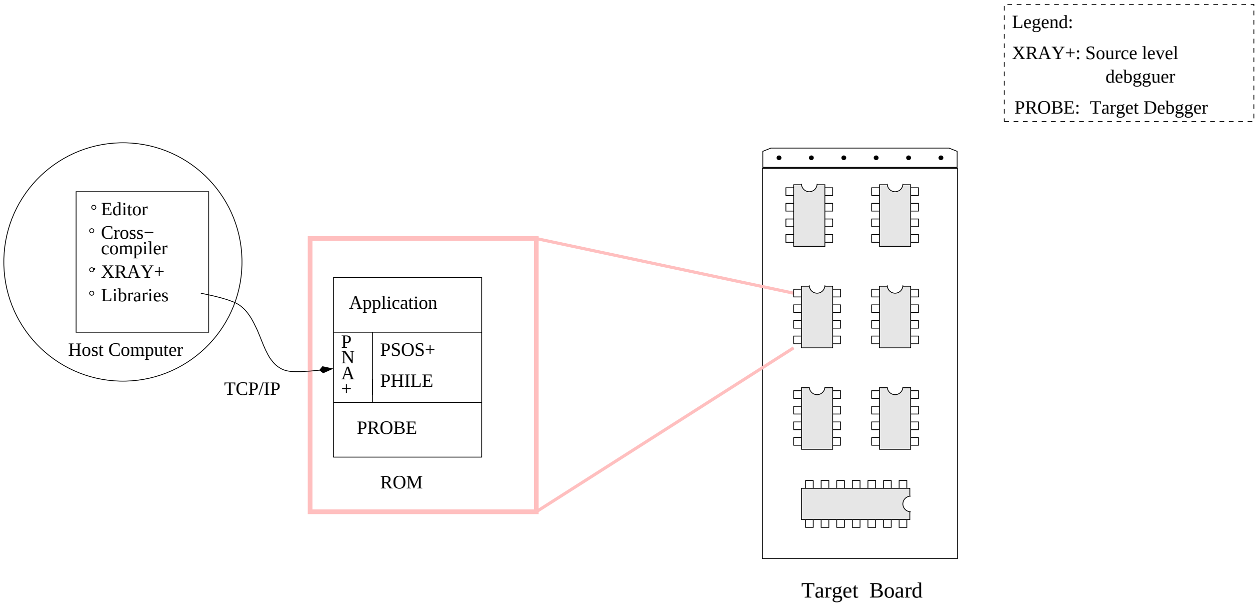


Figure 10: PSOS-based Development of Embedded Software

PSOS-based application development has schematically been shown in Fig. 5.10. 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. PSOS+ and other optional modules such as PNA+, PHILE, and PROBE are installed on a ROM on the target board. PNA+ is the network manager. It provides TCP/IP communication between the host and the target 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. XRAY+ invokes PROBE+ to provide a seamless debugging environment to the real-time application developer. 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+). During application development, the application is downloaded on to a RAM on the target. Once the application runs satisfactorily, it is fused on a ROM.

We now highlight some important features of PSOS. PSOS supports 32 priority levels which can be assigned to tasks. In the minimal configuration, the foot print of the target operating system is only 12KBytes. For sharing critical resources among real-time tasks, it supports priority inheritance and priority ceiling protocols. It supports segmented memory management and does not support virtual memory as it is intended to be used in small and moderate sized embedded applications. PSOS defines a *memory region* to be a physically contiguous block of memory. A memory region is created by the operating system in response to a call from an application. A programmer can allocate a task to a memory region.

In most modern operating systems, the control jumps to the kernel when an interrupt occurs. PSOS takes a different approach. 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.

7.2 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 is available in two multitasking kernels: VRTXsa and VRTXmc.

VRTXsa is used for large and medium-sized applications. It supports virtual memory. It has a POSIX-compliant library and supports priority inheritance. Its system calls complete deterministically in fixed time intervals and are 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 use in embedded applications such as computer-based toys, cell phones and other small hand-held devices.

7.3 VxWorks

VxWorks is a product from Wind River Systems. It is host-target type real-time operating system. The host can be either a Windows or a Unix machine. VxWorks conforms to POSIX-RT. 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 in the absence of the actual target board. 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, a hitch was that it repeatedly reset itself. Engineers on ground remotely using trace generation, logging, and debugging tools of VxWorks, determined that the cause was unbounded priority inversion. The unbounded priority inversion caused real-time tasks to miss their deadlines as a result, the exception handler reset the system each time. Although VxWorks supports priority inheritance, it was found out by using the remote debug tool to have been disabled by oversight in the configuration file. The problem was fixed by enabling it.

7.4 QNX

QNX is a product from QNX Software System Ltd (http://www.qnx.com). QNX is intended for use in mission-critical applications in the areas such as medical instrumentation, Internet routers, telemetric devices, process control applications, and air traffic control systems. QNX Neutrino offers POSIX-compliant APIs and is implemented using a microkernel architecture.

The microkernel architecture of QNX is shown in Fig. 5.11. Because of the fine grained scalability of the microkernel 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.

Neutrino and its “microGUI” — called Photon, are designed to operate extremely fast on a very small memory footprint, making their inclusion in portable devices possible. In fact, QNX and Neutrino already power Web appliances, set-top boxes, MP3 players, equipment used in the industrial and medical-equipment fields. QNX Neutrino

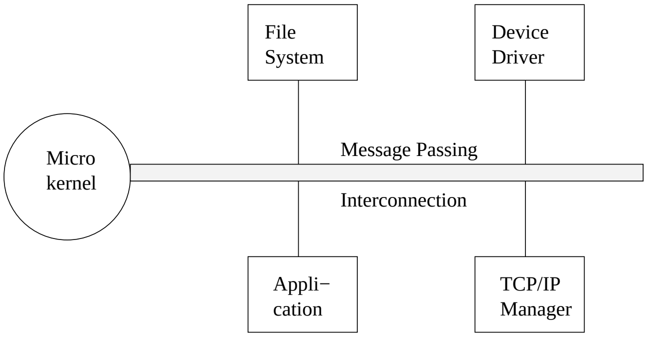


Figure 11: Microkernel Architecture of QNX

has been ported to a number of platforms and now runs on most modern CPUs that are used in the embedded market. This includes the Intel x86 family, MIPS, PowerPC, and the ARM family of processors. A version of QNX for non-commercial use can be downloaded for free from the company's web site (http://www.qnx.com).

7.5 μC/OS-II

μC/OS-II is available from Micrium Corporation (www.ucos-ii.com). This real-time operating system 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 to let the programmers have the option of using just a few of the offered services or select the entire range of services. This allows the programmers 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 is required to operate at a unique priority level, among the 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 scheme. 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 aircrafts and meets 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.

7.6 RT Linux

Linux is a free operating system. It is robust, feature rich, and efficient. However, Linux *per se* is a general purpose operating system. Several real-time implementations of Linux (RTLinux) are available. In this discussion, we consider some generic features of real-time Linux systems.

RT Linux is a self-host operating system (see Fig. 5.12). RT Linux runs along with a Linux system. The real-time kernel sits between the hardware and the Linux system. To the standard Linux kernel, the RT Linux layer appears to be the actual hardware. The RT Linux kernel intercepts all interrupts generated by the hardware. Hardware interrupts are not related to real-time activities are held and then passed to the Linux kernel as software interrupts when the RT Linux kernel is idle and the standard Linux kernel runs. Fig. 5.12 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 low priority background task of RT Linux.

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

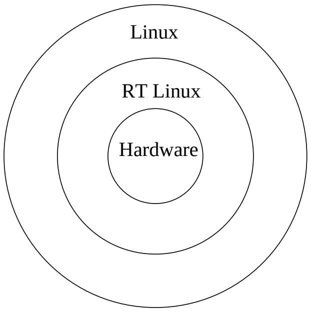


Figure 12: Schematic Overview of Operation of RT Linux

In the approach taken by RT Linux, there are effectively two independent kernels: real-time kernel and Linux kernel. This approach therefore 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 some of the important shortcomings of the dual-kernel approach.

- **Duplicated Coding Efforts:** Tasks running in the real-time kernel can not meaningfully use the Linux system services — file systems, networking, and so on. The reason for this is that 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. Thus, real-time application development might entail significant duplication of coding effort. For example, when a real-time task needs to use network communications, or for that matter file accesses, appropriate drivers must be written in the real-time kernel.
- **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. In stead, 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 (Application Programmer Interfaces). 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:** RTLinux kernels support only a limited subset of POSIX APIs. Therefore, application development can take more effort and time.

7.7 Lynx

Lynx is a self-host real-time operating system. It is available from www.lynuxworks.com. 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 for task 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 even 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.

7.8 Windows CE

Windows CE is a stripped down version of Windows operating system, 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 1msec for `sleep` and `wait` related APIs. The different functionalities of the kernel are broken down into small non-preemptive sections. As a result, 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 the priority inversion problem that is present in Windows NT. Normally, the kernel thread handling the page fault (i.e. DPC) runs at priority level higher than `NORMAL` (see Sec. 5.5.2). 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 that of the priority of the thread causing the page fault. This ensures that a thread is not blocked by any lower priority thread, even when it suffers a page fault.

8 Benchmarking Real-Time Systems

During design and platform evaluation stage, system developers often find it necessary to benchmark computer systems. Let us first examine how this is done for traditional computer systems. Subsequently, we shall examine this issue for real-time computers. To understand the issues involved, consider the following situation. Assume that the organization you work for entrusts you to select the "best" computer (from those available in the market) on price and performance considerations to be used to host the company's web site. It is possible that you can ask for bids from vendors and obtain their price and performance specifications. Now the question is how would you determine which among a set of computers quoted by different vendors would perform best for your application?

Of course, your evaluation would be very accurate if you actually got each of the machine for which you received quote and then actually ran your application on each of them and evaluated various performance parameters such as response time for queries under different load conditions. However, it would be infeasible to carry this out from cost, effort, and time considerations. One possibility is that you can use two traditional metrics used by vendors to indicate the performance of their computer systems: MIPS (Million Instructions Per Second) and FLOPS (Floating Point Operations Per Second). However, an evaluation based on these metrics can often be highly misleading. The situation is so disgusting that someone even proposed that a more suitable full form of MIPS should be "Misleading Information about Processor Speed". Let us examine why this is so. To determine the MIPS rating of a computer, a program is run on it and the run time is measured. The number of instructions in the program is divided by the run time to give the MIPS rating of the computer. However, it is not specified as to what type of program needs to run. Consequently, vendors take the liberty of running programs that generate only those (machine) instructions that have very short cycle times for their processor, resulting in artificially inflated MIPS ratings. As a result, if you make your buying decisions based on the quoted MIPS and FLOPS ratings, then you are most likely to be disappointed. It is very likely that when you actually get the computer you selected based on MIPS rating and run your application, your application would not run as fast as you had expected.

Everybody soon realized that MIPS and FLOPS ratings are misleading. Vendors therefore started to express performance ratings of their computer systems in terms of peak MIPS and peak FLOPS to indicate that the ratings have been obtained by using their fastest instructions. However, this did not ease the problem in any way.

To overcome this problem of MIPS and FLOPS ratings in performance evaluation, synthetic benchmarks were developed. Let us understand what are synthetic benchmarks. A large number of practical problems were analyzed to determine the statistical distribution of various instructions in an average program (e.g. arithmetic instructions 20%, I/O 10%, register transfer 10%, etc.). Using this information, a benchmark program is written that has the same distribution of the different instructions. Of course, the benchmark program need not compute any meaningful results. It is simply synthesized using instructions with required statistical distributions. It is therefore called a synthetic benchmark. It should be clear that the performance results obtained by running the synthetic benchmark program should be very closely related to what would be expected by running an average practical program. Some examples of synthetic benchmarks are *Whetstone, Linpack and Dhrystone.* Later, SPEC (Standard Performance Evaluation Corporation) was formed in the late eighties. SPEC is a non-profit association of computer manufacturers and academicians that develops and publicizes benchmark suites for specific applications. For example SPECWEB is the benchmark for web applications. The SPEC benchmark programs can be downloaded from www.spec.org.

8.1 Rhealstone Metric

For real-time systems *Rhealstone* metric [2] is popular for benchmarking. In the Rhealstone metric, six parameters of real-time systems are considered. These identified parameters indicate the parameters that are important in a typical real-time applications. We now briefly discuss these parameters in the following:

1. **Task Switching Time (t_{ts}).**

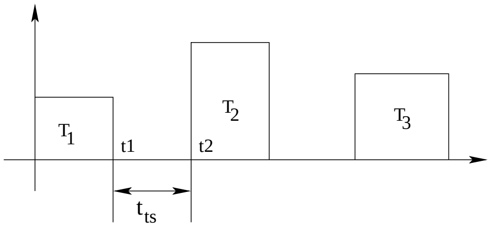


Figure 13: Task Switching Time Among Equal Priority Tasks

Task switching time is defined as the time it takes for one context switch among equal priority tasks. Consider the example shown in Fig. 5.13. Assume that T_1, T_2, T_3 are equal priority tasks and that round robin option among equal priority tasks is supported by the scheduler. In Fig. 5.13, t_{ts} is the time after which T_2 starts to execute after T_1 completes its time slice, blocks, or completes ($t_{ts} = t_2 - t_1$). Task switching time is determined by the efficiency of the kernel data structure.

2. **Task Preemption Time (t_{tp}).** Task preemption time is defined as the time it takes to start execution of a higher priority task (compared to the currently running task), after the condition enabling the task occurs. Task preemption time has been illustrated in the example shown in Fig. 14. Task preemption time consists of the following three components: task switching time t_{ts}, time to recognize the event enabling the higher priority, and the time to dispatch it. Clearly, task preemption time would be normally larger than task switching time.

3. **Interrupt Latency Time (t_{il}).** Interrupt latency time is defined as the time it takes to start the execution of

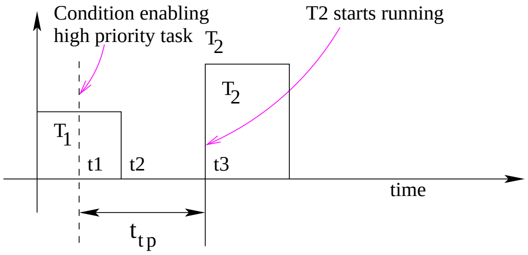


Figure 14: Task Preemption Time

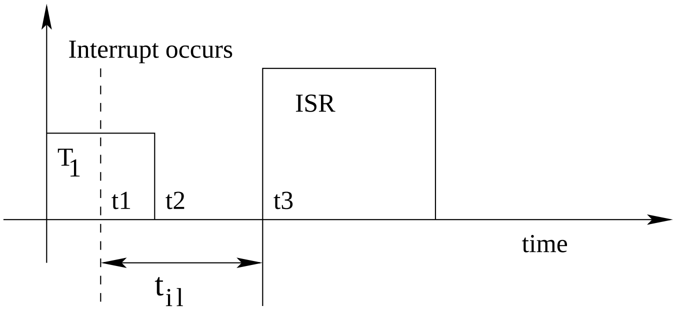


Figure 15: Interrupt Latency Time

the required ISR after an interrupt occurs. Interrupt latency time for an example has been shown in Fig. 15. Interrupt latency time consists of the following components: hardware delay in CPU recognizing the interrupt, time to complete the current instruction, time to save the context of the currently running task, and then to start the ISR.

4. **Semaphore Shuffling Time (t_{ss}).** Semaphore shuffling time is defined as the time that elapses between a

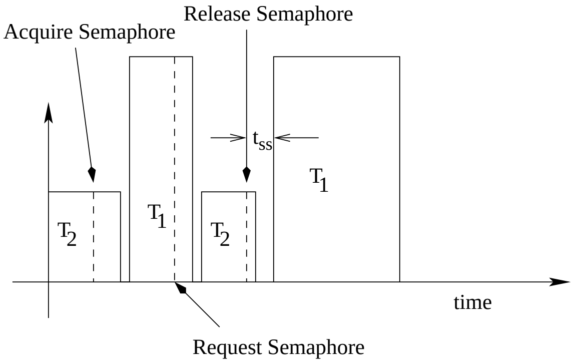


Figure 16: Semaphore Shuffling Time

lower priority task releasing a semaphore and a higher priority task to start running. Semaphore shuffling time has been illustrated in Fig. 5.16. In Fig. 5.16, at a certain time task T_2 holds the semaphore. The task T_1 requests for the semaphore and blocks. The interval between T_1 returning the semaphore and T_2 running is known as the semaphore shuffling time.

5. **Unbounded Priority Inversion Time (t_{up}).** As shown in figure 17, unbounded priority inversion time t_{up}

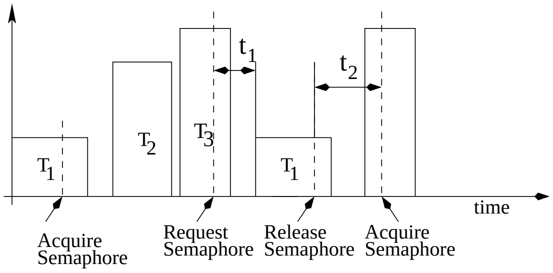


Figure 17: Unbounded Priority Inversion Time

is computed as the time it takes for the operating system to recognize priority inversion (t_1) and run the task holding the resource and start T_2 after T_1 completes (t_2). Symbolically, $t_{up} = t_1 + t_2$.

6. **Datagram Throughput Time (t_{dt}).** This parameter indicates the number of kilobytes of data that can be transfered between two tasks without using shared memory or pointers. This parameter measures the efficiency of the data structures handling message passing primitives.

The Rhealstone metric is computed as a weighted average of the identified parameters:

$$Rhealstone \quad Metric \quad = \quad a_1 * t_{ts} + a_2 * t_{tp} + a_3 * t_{il} + a_4 * t_{ss} + a_5 * t_{up} + a_6 * t_{dt}$$

Where, a1, ..., a6 are empirically determined constants.

Rhealstone benchmark's figure of merit characterizes the kernel-hardware combination of a system. This is therefore a comprehensive benchmark. However, Rhealstone has some serious drawbacks. First, the ability to meet deadlines of applications (task scheduling etc.) is not considered at all. Also, the six measurement categories are somewhat ad-hoc and there is no proper justification as why only these six measurements have been chosen.

8.2 Interrupt Processing Overhead

Let t_0 be the time to complete a certain task when there are no interrupts and let $t_{20,000}$ be the time to complete the same task when 20,000 interrupts occur per second. Then the overhead due to $20,000$ interrupts per second is given by:

$$I_{20,000} = \frac{t_{20,000} - t_0}{t_0}$$

Consider for example that $t_{20,000}$ is 1.6 sec and t_0 is 1 sec. Then, the interrupt processing overhead $I_{20,000}$ is 0.6 (or 60%) of the task execution time. The overhead considered is only for processing the immediate part of the interrupts and not the deferred parts.

8.3 Tridimensional Measure

Tridimensional measure considers three factors that affect the performance of a real-time system. These identified parameters are: Millions of Instructions Processed per Second (MIPS1), Millions of Interrupts Processed per Second (MIPS2), Number of I/O Operations Per Second (NIOPS). Once these three parameters have been determined, the tridimensional measure (TM) can be obtained by using the following expression:

$$TM \; = \; (MIPS1 \quad * \quad MIPS2 \quad * \quad NIOPS)^{\frac{1}{3}}$$

8.4 Determining Kernel Preemptability

Some times it becomes necessary to determine the extent to which an operating system kernel is preemptable by higher priority real-time tasks. Fig. 18 shows an experimental set up that can indicate kernel preemptability. In Fig. 18, the TRIANG module generates random triplets denoting three vertices of a triangle. There are n real-time tasks $RT_1, ...RT_n$ which read the triangle vertices and carry out certain processing and finally output the original triangle vertices to the analyzer module through message passing. TRIANG operates at higher priorities than real-time tasks. The timer first invokes TRIANG, and then invokes a system call such as process creation, and subsequently triggers a different real-time task ($RT_1, ...RT_n$) each time. The analyzer compares the triangle vertices received from the TRIANG and from the real-time task. If the system kernel is not preemptable, then even before the real-time task can read the triangle value, the next set of triangle values over writes the last generated values. The timer frequencies are varied and the analyzer results are observed. The timer frequency at which vertices get missed (as identified by the analyzer), indicates the worst case kernel preemption time.

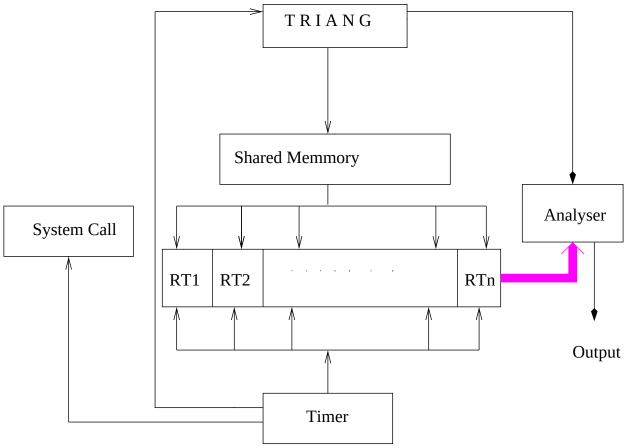


Figure 18: Setup for TRIANG

SUMMARY

- We first discussed the basic features that any real-time operating system is required to support. Some of the important features required of a real-time operating system, if it is to be used for hard real-time applications are: high resolution clock and timer services, static priority levels, low task preemption times (of the order of a few microseconds), low interrupt latency time, low jitter in memory access, and support for resource sharing among real-time tasks.

- We explained why traditional operating systems such as Unix and Windows are not suitable for hard real-time applications. We pointed out that the main problems that a programmer would face while developing a real-time application using Unix are dynamic priority levels and non-preemptable kernel. Though Windows NT sports many positive features for real-time programming, the way Windows NT handles interrupts might result in unbounded response times for real-time tasks.

- We surveyed the important features of some of the commercially used real-time operating systems. We discussed why free real-time operating systems such as RTLinux, μCOS, and Lynx are gaining ground.

- We highlighted the parameters based on which the performance of real-time systems can be benchmarked.

EXERCISES

1. State whether you consider the following statements to be **TRUE** or **FALSE**. Justify your answer in each case.

   (a) In real-time Linux (RTLinux), 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) Under the Unix operating system, computation intensive tasks dynamically gravitate to higher priorities.

   (e) When PCP (priority ceiling protocol) is implemented in Windows NT by an application programmer, any real-time priority levels can be assigned to tasks and ceiling values computed on these to implement PCP.

   (f) Normally, task switching time is larger than task preemption time.

   (g) Between segmented and virtual addressing schemes, the segmented addressing scheme would in general incur lower jitter in memory access compared to the virtual addressing scheme.

   (h) POSIX is an attempt by ANSI/IEEE to enable executable files to be portable across different Unix machines.

   (i) If FIFO scheduling among equal priority tasks is not supported by an operating system, then in an implementation of priority ceiling protocol (PCP) at the user-level, only half of the available priority levels can meaningfully be assigned to the tasks.

   (j) Suppose a real-time operating system does not support memory protection, then a procedure call and a system call are indistinguishable in that system.

   (k) Watch dog timers are typically used to start sensor and actuator processing tasks at regular intervals.

   (l) For 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.

   (m) For faster response time, an embedded real-time operating system needs to be run from a ROM rather than a RAM.

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 at all possible for an operating system to support nanosecond resolution clocks 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 why not? If your answer is “yes”, explain what additional features would you implement in the external layer of Unix System V kernel for supporting hard real-time applications?

5. As the developer of hard real-time applications, explain some of the features that you would require a real-time operating system (RTOS) to support.

6. Briefly indicate how Unix dynamically recomputes task priority values. Why is such recomputation of task priorities required? What are the implications of such priority recomputations on real-time application development?

7. What is the difference between synchronous I/O and asynchronous I/O? What are the implications of these two types of I/O for real-time applications.

8. Explain the pros and cons of supporting virtual memory in embedded real-time applications.

9. What do you understand by memory protection in operating system parlance. Compare the pros and cons of requiring an embedded real-time operating system (RTOS) to support memory protection?

10. What is the difference between block I/O and character I/O? For each type of I/O, give an example of a task that needs to use it. Which type of I/O is accorded higher priority by Unix? Why?

11. 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.

12. Suppose that you have to select a suitable computer for a process control application, out of several available computers. Write four important performance parameters which you would consider in benchmarking real-time computer systems. Define these parameters and explain why they are important in real-time applications.

13. Explain the time services that a real-time operating system (RTOS) is expected to support. Also, briefly highlight how such timer services are implemented in a real-time operating system.

14. What is an open system? What are its advantages compared to a closed system?

15. What is a watchdog timer? Explain the use of a watchdog timer using an example.

16. 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.

17. 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.

18. 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?

19. Windows NT does not provide any support to synchronize access of tasks to critical resources. Explain how resource sharing among tasks can be supported.

20. 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.

21. Explain the problems that you would encounter, if you try to develop and run a hard real-time application on the Windows NT operating system.

22. How is the integrity of the kernel data structure preserved in Unix V in the face of preemptions of the kernel by interrupts and higher priority real-time tasks? Why does this technique not work successfully in preserving the integrity of the kernel data structures in multiprocessor implementations? Very briefly explain the important approaches that have been adopted to overcome this problem of making Unix-based systems suitable for real-time application development.

23. Briefly explain why the traditional Unix kernel is not suitable to be used in a multiprocessor environment. Explain a spin lock and a kernel-level lock and discuss their use in realizing a preemptive kernel.

24. 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 in supporting real-time applications.

25. What is the difference between a microkernel-based real-time operating system and a dual kernel real-time operating system. Give an example of each. Compare the pros and cons of the two approaches to developing a real-time operating system.

26. What is a real-time file system? What additional features does a real-time file system support compared to traditional file systems? Why is use of a real-time file system essential in hard real-time applications?

27. What is the difference between a system call and a function call? What problems, if any, might arise if the system calls are indistinguishable from procedure calls?

28. Explain how a real-time operating system differs from a traditional operating system. Name a few real-time operating systems that are commercially available.

29. What is the difference between a self-host and a host-target based embedded operating system? Give at least one example of a commercial commercial operating system from each category. What problems would an real-time application developer might face while using RT-Linux for developing hard real-time applications?

30. What is an 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?

31. What are the important features does a real-time application developer expect from a real-time operating system? Analyze to what extent are these features provided by Windows NT and Unix V.

32. What do you understand by a host-target type of real-time operating system? Give two commercial examples of host-target type of real-time operating system. Explain the architecture of such an operating system and also explain how a real-time application can be developed on a host-target type of operating system.

33. Identify at least four important advantages of using VxWorks as the operating system for large hard real-time applications compared to using Unix V.3.

34. What is an open software? What are the pros and cons of using an open software product in program development, compared to using an equivalent proprietary product?

35. 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.

36. 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.

37. Explain why traditional (non-real-time) operating systems need to dynamically change the priority levels of tasks. What is the implication of this for real-time application development using such an operating system.

References

[1] Hennessy John L. and David A. Patterson. *Computer Architecture — A Quantitative Approach*. Morgan Kaufmann Publishers, 2003.

[2] Kar R.P. and Porter K. Rhealstone – a real-time benchmarking proposal. *Dr. Dobb's Journal*, February 1989.

[3] Brian Santo. Embedded battle royale. *IEEE Spectrum*, pages 36–42, December 2001.