scheduled interrupt handling routine. The execution of this routine is typically preemptable and should be scheduled at a suitable (software) priority in a priority-driven system. (As an example, in the real-time operating system LynxOS [Bunn], the priority of the kernel thread that executes a scheduled interrupt handling routine is the priority of the user thread that opened the interrupting device.) This is why Figure 12–2 shows that following immediate interrupt service, the scheduler executes and inserts the scheduled interrupt handling thread in the ready queue. The scheduler then lets the highest priority thread have the processor.

Since scheduled interrupt handling routines may modify kernel data, they are executed by kernel threads in operating systems that provide memory protection. LynxOS and SunOS are examples. When a device driver is created in LynxOS, a kernel thread comes into existence. At its creation, the thread has the lowest priority in the system. Whenever a user thread opens the device driver, the kernel thread inherits the current priority of the user thread. When the I/O operation of the user thread completes, the priority of the kernel thread returns to the level immediately before the priority inheritance. The LynxOS literature refers to this priority inheritance as priority tracking and points out that this mechanism gives us accountability of the delay incurred by threads due to interrupt handling. (Problems 6.24 and 6.30 are about how to analyze the schedulability of such a system.) In SunOS, the kernel thread, called the interrupt thread, is created at interrupt time and is terminated when interrupt handling completes. However unlike kernel threads in LynxOS, an interrupt thread executes immediately after creation. Without priority tracking, the execution of interrupt threads can lead to uncontrolled priority inversion.[6]

We can think of the scheduled interrupt handling step as an aperiodic or sporadic thread. When the immediate interrupt service routine completes, an aperiodic thread is released to execute the scheduled interrupt service routine or inserts the routine into the queue of a bandwidth-preserving server. The scheduler can use some of the bandwidth-preserving or slack-stealing algorithms described in Chapter 7 for this purpose. Unfortunately, existing operating systems not only do not provide such servers but also do not provide good hooks with which we can customize the scheduler for this purpose. It is expensive to implement most of these algorithms in the user level without help from the kernel. In Section 12.2.2, we will describe minor modifications of the scheduling mechanism that simplify the implementation of bandwidth-preserving servers both in the user level and within the kernel.

Hereafter, we will refer to the strategy of dividing interrupt handling into two steps as *split interrupting handling*. As we will see in Sections 12.6 and 12.7, most modern operating systems use this strategy in one form or the other.

12.2 TIME SERVICES AND SCHEDULING MECHANISMS

This and the next sections discuss the basic operating system functions that the previous section either did not mention or treated superficially. Specifically, this section discusses time services and scheduling mechanisms, leaving interprocess communication and synchronization, software interrupts, memory management, and I/O and networking to the next section.

[6] A way to reduce the effect of this incorrect prioritization is to have the interrupt thread sets its own priority to the priority of the thread causing the interrupt as soon as that thread is identified.

12.2.1 Time Services

Obviously, good time services are essential to real-time applications. To provide these services and support its own activities, every system has at least one clock device. We have been calling it the system clock. In a system that supports Real-Time POSIX, this clock is called CLOCK_REALTIME.

Clocks and Time. Again, a clock device contains a counter, a timer queue, and an interrupt handler. The counter is triggered to increment monotonically by a precise periodic sequence of pulses. At any time, the content of the counter gives a representation of the current time. We have been calling this counter a (hardware) clock and the triggering edge of each pulse a clock tick. The timer queue contains the pending expiration times of timers bound to the clock.

A system may have more than one clock device and uses them for different purposes. For example, a real-time monitor system may use a clock device to trigger the sampling and digitization of sensor readings and initialize the input of sensor data. A digital control system may use one or more clock devices to time the control-law computations of a rate group or commands to embedded plants.

Resolution. The *resolution* of a clock is the granularity of time provided by the clock. Today's technology makes it possible to have hardware clocks with resolution on an order of nanoseconds. However, the clock resolution available to applications is usually orders of magnitude coarser, on an order of hundreds of microseconds or milliseconds.

To see why there is such disparity, let us look at how an application finds what the current time is. Typically, the kernel maintains a software clock for each clock device it supports. It programs the clock device to raise interrupts periodically. Whenever it gets an interrupt from the clock device, it updates the software clock, and hence the current time according to the clock. Therefore, the resolution of the software clock is equal to the period of these interrupts. A thread gets the current time according to a clock by calling a get time function, such as the POSIX function *clock_gettime( )*, and providing as an argument the ID of the clock to be read. In response to the request, the operating system returns the current time according to the corresponding software clock. Therefore, the resolution of time seen by the calling thread is resolution of the software clock.

In Sections 6.8.5 and 12.1.2, we called the interrupts from the system clock that triggers the scheduler into action clock interrupts. We said that the typical period of clock interrupts is 10 milliseconds. If an operating system updates the system clock only at these interrupts, its clock resolution is 10 milliseconds. A 10-millisecond resolution is too coarse for many applications. For this reason, most modern operating systems support, or allow applications to request, a finer clock resolution. To support a finer resolution, the system clock device is set to interrupt at a higher frequency. At each of the higher-frequency interrupts, the kernel merely updates the software clock and checks the clock's timer queue for timer expirations. Typically, the period of these interrupts, hence the resolution of the software clock, ranges from hundreds of microseconds to tens of milliseconds. It is more convenient if the resolution divides the tick size, so the kernel does tick scheduling at one interrupt out of an integer x number of interrupts from the clock, where x is the ratio of tick size to clock resolution. For the sake of clarity, we will call the higher-frequency interrupts from the clock device *time-*

service interrupts and continue to call the one-out-of-x subsequence of interrupts at which the scheduler executes *clock interrupts*.

The resolution of each (software) clock is a design parameter of the operating system. The finer the resolution, the more frequent the time-service interrupts and the larger the amount of processor time the kernel spends in responding to these interrupts. This overhead places a limitation on how fine a resolution the system supports. Moreover, the response time of the get time function is not deterministic, and the variation in the response time introduces an error in the time the calling thread gets from the kernel. This error is far greater than a few nanoseconds! A software clock resolution finer than this error is not meaningful.

High Resolution. A way to provide a finer resolution than hundreds of microseconds and more accurate time to an application is to map a hardware clock into the address space of the application. Then, the application can read the clock directly. On a Pentium processor, a user thread can read the Pentium time stamp counter.[7] This counter starts at zero when the system is powered up and increments each processor cycle. At today's processor speed, this means that counter increments once a few nanoseconds. By computing from the cycle count provided by the counter, an application can get more precise and higher-resolution time than that provided by the operating system.

However, an operating system may not make the hardware clock readable for the sake of portability, and processors other than those in the Pentium family may not have a high-resolution time stamp counter. Using its own clock/timer device and device driver, an application can maintain its own high-resolution software clock. A higher overhead is a disadvantage of this scheme: The overhead incurred by an application in the maintenance of its own software clock is invariably larger than when the software clock is maintained by the kernel.

Timers and Timer Functions. Most operating systems (including all Real-Time POSIX compliant systems, Windows NT, and Solaris) allow a thread or process to have its own timers.[8] Specifically, by calling the create timer function, a thread (or process) can create a per thread (or per process) timer and, in a system containing multiple clocks, bind the timer to a specified clock. Associated with each timer is a data structure which the kernel creates in response to the create timer call. Among the information kept in the data structure is the expiration time of the timer. The data structure may also have a pointer to a handler; the handler is a routine that the calling thread wants to be execute when a timer event occurs. A thread destroys its own timer by calling the destroy timer function.

We say that a thread *sets* (or arms) a timer when it asks the kernel to give the timer a future expiration time. A timer is *canceled* (or disarmed) if before the occurrence of the timer event, it is set to expire at or before the current time. (In effect, the kernel does not act when it finds this expiration time.) Every operating system provides timer functions with

[7] Some operating systems also use this counter to improve clock resolution. At each time-service interrupt, the kernel reads and stores the time stamp. When servicing a *clock_gettime( )* call, the kernel reads the counter again. From the difference in the two counter readings, the kernel can compute the elapse of time since the last time-service interrupt. By adding this time to the clock reading, the kernel gets and returns a more accurate time.

[8] Operating systems, such as Linux, that do not support per process timers provide one or more systemwide timers. Processes cannot destroy these timers but can set them and use them for alarm purposes.

which a thread can set and cancel timers. Similarly, various forms of set-timer functions allow a thread to specify an action to take upon timer expiration. The types of action include calling a specified function, waking up a thread, and sending a message to a message queue.

The expiration time can be either *absolute* or *relative*. The former is a specific time instant. The latter is the length of the delay from the current time to the absolute expiration time. There are two kinds of timers: one-shot or periodic. When set, a *one-shot timer* expires once at the specified expiration time or when the specified delay has elapsed. In contrast, a thread sets a *periodic timer* by giving it a first expiration time and an interval between consecutive expiration times. Starting from the first expiration time, the periodic timer expires periodically until it is cancelled.

Asynchronous Timer Functions. As an example, we look at the set watchdog timer function *wdStart( )* provided by VxWorks [Wind]. A watchdog (or alarm) timer is a one-shot timer that supports relative time. After a timer is created, a thread can use it by calling the *wdStart( )* function and specifying as arguments the timer ID, the delay to the expiration time, the function to call at the expiration time, and an argument to be passed to the specified function. A thread can cancel a timer before its expiration by calling *wdCancel( )*.

A watchdog timer mechanism is useful for many purposes. Figure 12–4 gives an example: An implementation of a timing monitor. The timing monitor logs the deadlines missed by input tasks. Each of these tasks processes sporadic sensor readings collected by an input device. Whenever the input device presents a sensor reading by raising an interrupt, its input task must process the reading within the relative deadline of the task. The example assumes that the driver code for each device *device_k* consists of a short interrupt service routine *ISR_k* followed by a data processing routine *DPR_k*. When the *ISR_k* completes, it

- During system initialization: The application process creates sporadic thread *S_k* to execute the data processing routine *DPR_k* for each input device *device_k* and sets the *late* flag of the device to false.
- The sporadic thread *S_k*
  - After being created, creates watchdog timer $wdTimer\_k$ and then suspends itself.
  - When awaked and scheduled, executes *DPR_k*.
  - At the end of *DPR_k*, calls *wdTimercancel*($wdTimer\_k$) if *late* is false, and then enables interrupt from *device_k* and suspends itself.
- Input device *device_k*: When sensor data become available, raises an interrupt if interrupt is enabled.
- Immediate interrupt handling step:
  - After identifying the interrupting device *device_k*, disables interrupt from *device_k*.
  - Calls *wdTimerSet* (*wdTimer_k, relativeDeadline_k, timingMonitor, device_k*) so that the watchdog timer will expire at current time plus *relativeDeadline_k*.
  - Services *device_k*.
  - Sets *late* to false and wakes up sporadic thread *S_k*.
- Timing Monitor function *timingMonitor( )*: When called, sets *late* to true and increments by 1 the number of deadlines missed by the specified device.

FIGURE 12–4 Example illustrating the use of watchdog timers.

wakes up a sporadic thread S_k to execute *DPR_k*. The operating system schedules sporadic threads according to some suitable algorithm. During the immediate interrupt handling step, the *ISR_k* routine sets the watchdog timer. The watchdog function *wdTimerSet( )* in Figure 12–4 is similar to the VxWorks's *wdStart( )*. The first argument of the function identifies the watchdog timer. The specified delay is equal to the relative deadline *relativeDeadline_k* of the input task; the function to be called when the timer expires is the timing monitor, and the argument to be passed to the timing monitor is the ID *device_k* of the input device. After setting the watchdog timer, the interrupt service routine wakes up the sporadic thread S_k, which, when scheduled, executes the data processing routine *DPR_k*. If the thread completes before the timer expires, it cancels the timer before suspending itself again. On the other hand, if the timer expires before the thread completes, the timing monitor is called, which increments by one the number of sensor readings from the input device that are not processed in time.

The above watchdog timer function uses a function call as a means to notify the application of the occurrence of a timer event. The other mechanisms commonly used for this purpose are messages and signals. As an example, the *timer_arm( )* function in Real-Time Mach [StTo] takes as arguments the timer expiration time and a port. At the specified time, the kernel sends a notification message containing the current time to the specified port. Thus, such a timer function enables the calling thread to synchronize and communicate with the thread or threads that will receive the message at that port.

Signal is the notification mechanism used by Real-Time POSIX timer function *timer_settime( )*. When a thread calls the *timer_create( )* function to request the creation of a timer, it specifies the clock to which the timer is to be bound, as well as the type of signal to be delivered whenever the timer expires. If the type of signal is not specified and the clock to which the timer is bound is CLOCK_REALTIME, the system will deliver a SIGARLM (alarm clock expired) signal by default.

After creating a timer, a thread can set it by calling *timer_settime( )*. The parameters of this function include

1. the ID of the timer to be set;
2. a flag indicating whether the new expiration time is relative or absolute;
3. the new timer setting, which gives the delay to the expiration time or the absolute value of the expiration time, depending on value of the flag; and
4. the period between subsequent timer expiration times.[9]

Again, when the timer expires, the system delivers a signal of the specified type to the calling thread.

Earlier in Section 12.1, we mentioned that a periodic thread can be implemented at the user level. This can be done using a timer as shown in Figure 12–5(a). In this example, a thread named *thread_id* is created at configuration time to implement a periodic task. The intended behavior of the task is that it will be released for the first time at 10 time units after

[9]The timer can be cancelled before its expiration using the same function by setting the new expiration time argument zero. Similarly, if the period between consecutive timer expiration times is zero, the timer is to expire only once, and it will expire periodically if the period has some nonzero value.

```
timer_create(CLOCK_REALTIME, NULL, timer_id);
block SIGALRM and other signals;
instancesRemaining = 200;
timer_settime(timer_id, relative, 10, 100);
while (instancesRemaining > 0)
    sigwaitinfo(SIGALRM);
    statements in the program of the periodic tasks;
    instancesRemaining = instancesRemaining - 1;
endwhile;
timer_delete(timer_id);
thread_destroy(thread_id);
```

(a) Implementation 1

```
instancesRemaining = 200;
nextReleaseTime = clock + 10;
while (instancesRemaining > 0)
    NOW = clock;
    if (NOW < nextReleaseTime), do
        timer_sleep_until (nextReleaseTime);
        statements in the program of the periodic task;
        nextReleaseTime = nextReleaseTime +100;
    else
        statements in the program of the periodic task;
        nextReleaseTime = NOW +100;
    instancesRemaining = instancesRemaining −1;
endwhile;
thread_destroy(thread_id);
```

(b) Implementation 2

FIGURE 12–5 User level implementations of periodic tasks. (a) Implementation 1. (b) Implementation 2.

its creation and afterwards, periodically once every 100 time units. The task has only 200 jobs; so the thread will be deleted after it has executed 200 times. After the thread is created, it first creates a timer and specifies that the timer is bound to the system clock, indicated by the argument CLOCK_REALTIME. The NULL pointer argument indicates that the timer will deliver the default timer expiration signal SIGALRM whenever it expires. The *timer_create( )* function returns the ID (*timer_id*) of the timer. The thread then blocks SIGALRM signal, as well as other signals that it does not want to process, except when it synchronously waits for the signal. After it thus initializes itself, the thread sets the timer to expire periodically starting from 10 time units from the current time and then once every 100 time units. After

this, the thread calls *sigwaitinfo( )*[10] to wait for SIGALRM. This function returns immediately if a SIGALRM is waiting; otherwise, the thread is blocked until the signal arrives. When the function returns, the thread "is released" to execute the code of the periodic task. After it has been released and executed 200 times, the thread asks operating system to delete the timer and itself.

It is important to note that *such a periodic task may not behave as intended*. There are many reasons; we will discuss them at the end of this subsection.

Synchronous Timer Functions. The set-timer functions in the previous examples are asynchronous. After being set, the timer counts down while the calling thread continues to execute. Consequently, a thread can set multiple timers to alarm at different rates or times. In contrast, after calling a synchronous timer function, the calling thread is suspended.

As an example, we look at the *timer_sleep( )* function provided by Real-Time Mach. The function causes the calling thread to be suspended either until the specified absolute time or for the specified time interval. When the specified time is relative, the timer function is similar to the Real-Time POSIX *nanosleep(t)*; the parameter *t* specifies the length of the interval the calling thread sleeps.

We can implement the periodic task in Figure 12–5(a) using the *timer_sleep( )* function as well. After a thread is created, it executes the code in Figure 12–5(b). For clarity, we refer to the timer function as *timer_sleep_until( )* to indicate that the argument is absolute, and this timer function uses the system's timer. We assume here that the thread can get the current time by reading the clock directly, and the value of *clock* is the current time. When this assumption is not true, the thread must call a function [e.g., *clock_gettime( )*] to get the current time from the operating system.

Timer Resolution. We measure the quality of timers by their resolution and accuracy. The term resolution of a timer usually means the granularity of the absolute time or the time interval specified as an argument of a timer function. We call this granularity the *nominal timer resolution*. If the nominal timer resolution is x microseconds, then the operating system will not mistake two timer events set x microseconds apart as a single timer event. However, this does not mean that the granularity of time measured by the timer, as seen by application threads, is this fine.

Periodic Timer Interrupts. To illustrate, we recall that in most operating systems, the kernel checks for the occurrences of timer events and handles the events only at time-service interrupts and these interrupts occur periodically. Now let us suppose that the nominal timer resolution is 10 microseconds and the period of time-service interrupts is 5 milliseconds. A threads sets a timer to expire twice 10 microseconds apart and reads the current time at the occurrence of each timer event. Suppose that the first timer event occurs at 5 microseconds

[10] *sigwaitinfo( )* is a Real-Time POSIX function for synchronous signal waiting. When a signal arrives, the function does not call the signal handler. Rather, it returns the number of the signal to the thread. If a SIGARLM signal arrives while it is blocked and the thread is not waiting, the overrun count of the timer is incremented by one. By processing this count, the thread can determine the number of times the system has delivered SIGALRM. The example does not include the step to process the overrun counter.

before an interrupt and the second one at 5 microseconds after the interrupt. The kernel handles the first one at the interrupt and the second at the next time-service interrupt. The time values read by the thread are approximately 5 milliseconds apart.

We call the granularity of time measured by application threads using timers the *actual timer resolution*. In a system where the kernel checks for timer expirations periodically, this resolution is no finer than the period of time-service interrupts. We can get a finer actual resolution only by having the kernel check the timer queue more frequently.

One-Shot Timer Interrupts. Alternatively, some operating systems (e.g., QNX) program the clock device to raise an interrupt at each timer expiration time. In other words, the clock interrupts in the one-shot mode. As a part of timer interrupt service, the kernel finds the next timer expiration time and sets the clock to interrupt at that time. Thus, the kernel carries out the requested action as soon as a timer expires. With this implementation, the actual timer resolution is limited only by the amount of time the kernel takes to set and service the clock device and to process each timer event, since the kernel cannot respond to timer events more frequently than once per this amount of time. This time is in order of microseconds even on today's fast processors.

Since the clock device no longer interrupts periodically, some other mechanism is needed to maintain the system clock and to time the kernel's periodic activities. (Section 12.7.2 gives an example: UTIME [HSPN], a high-resolution time service on Linux. UTIME reads the Pentium time stamp counter at each timer interrupt and computes time based on the cycle counts.) This means that more work needs to be done upon each timer interrupt and the execution time of the the timer interrupt service routine is larger. (With UTIME, the execution time of timer interrupt service routine is several time larger than in standard Linux where timer expiration is checked periodically.) Moreover, each timer expiration causes an interrupt. As a consequence, the overhead of time services can be significant higher than when the occurrences of timer events are checked only periodically, especially when there are many timers and they expire frequently.

Timer Accuracy. By timer error, we mean the difference between the absolute time (or time interval) specified by the calling thread and the actual time at which the specified action starts. Timer error depends on three factors. The first is the frequency at which timer expirations are checked. This factor is the period of time-service interrupts in most operating systems since they check for timer expirations periodically.

The second source of error arises from the fact that timer events may not be acted upon by the kernel in time order. In some operating systems (e.g., Windows NT 4.0 and Linux), when more than one timer is found expired at a clock interrupt, the kernel takes care of the timer with the latest expiration time first and in decreasing time order. In other words, it services timer events in LIFO order. Therefore, if the order of occurrences of timer events is important, you will need to take care of this matter. For example, if two timer expiration times are within the same clock interrupt period, you need to give the timer that is supposed to trigger an earlier activity a later expiration time.

The time spent to process timer events is the third and the most unpredictable source of error in the absolute time or delay interval the calling thread gets from the kernel. Just

as lengthy interrupt service routines cause large variation in interrupt latency, lengthy timer-service routines increase timer error. By minimizing the amount of time spent processing each timer event, the error introduced by this factor can be kept negligibly small compared with the period of time-service interrupts.

Release-Time Jitters of Periodic Tasks. We conclude our discussion on timers by looking closely at the user-level implementations of a periodic task in Figure 12–5. In particular, we ask what can cause jitters in the release times of the periodic thread. Suppose that the kernel finishes creating the thread and places it in the ready queue at time t. The programmer's intention is for the thread to be released for the first time at time $t + 10$. We see that the actual first release time can be later than this time because (1) the thread may be preempted and not scheduled until a later time and (2) it takes some time to create a timer (and to get the current time) when the thread starts to execute. If our time unit is millisecond, the delay due to factor (2) is small and can be neglected, but the delay due to factor (1) is arbitrary. Therefore, the implementations are correct only if we accept that $t + 10$ is the earliest release time of the first instance, while the actual release time of this instance may be much later.

Similarly, 100 time units are the minimum length of time between the release times of instances of this task. The subsequent instances of the thread will be released periodically only if the while loop always completes within 100 time units. According to implementation 2, if the response time of an instance of the thread exceeds 100, the next instance is released as soon as the current instance completes. (Again, the next release can be delayed since the thread can be preempted between iterations of the loop.) In contrast, the timer in implementation 1 continues to signal every 100 time units. If a signal arrives while the thread is executing its program, the signal is blocked. The pseudocode in Figure 12–5(a) does not describe a correct implementation of a periodic task because it neglects this situation. In general, Real-Time POSIX signals that arrive while blocked are queued, but not SIGARLM signals. Instead, the number of times the SIGARLM signal has arrived while the signal is blocked is indicated by the timer overrun count. By examining this count, the thread can determine when the code of the periodic task should be executed again. How to do so is left for you as an exercise in Problem 12.6. In any case, the periodic task is not truly periodic. Moreover, because the thread can be preempted for an arbitrary amount of time, the interrelease time of consecutive instances can be arbitrarily large.

12.2.2 Scheduling Mechanisms

This section discusses several aspects regarding the implementation of algorithms for scheduling periodic tasks and aperiodic tasks. In particular, we call attention to those scheduling services that the kernel can easily provide which can significantly simplify the implementation of complex algorithms for scheduling aperiodic tasks in the user level.

Fixed-Priority Scheduling. All modern operating systems support fixed-priority scheduling. Many real-time operating systems provide 256 priority levels. As discussed in Section 6.8.4, a fixed-priority system with this many priority levels performs as well as an ideal system that has an infinite number of priority levels. (The loss in schedulable utilization

of a rate-monotonically scheduled system due to nondistinct priorities is negligible.) In contrast, a general-purpose operating system may provide fewer levels. For example, there are only 16 real-time priority levels in Windows NT.

A parameter of the create thread function is the priority of the new thread; the priority of each new thread is set at the *assigned priority*, that is, the priority chosen for the thread according to the algorithm used to schedule the application. (This is sometimes done in two steps: The thread is first created and then its priority is set.[11]) Once created, the assigned priority of the thread is kept in its TCB. In a system that supports priority inheritance or the ceiling-priority protocol, a thread may inherit a higher priority than its assigned priority. In Sections 8.4 and 8.6 where these protocols were described, we called the time varying priority which a thread acquires during its execution its *current priority*. The current priority also needs to be kept in the thread's TCB.

To support fixed-priority scheduling, the kernel maintains a ready queue for each priority level. Whenever a thread is ready to execute, the kernel places it in the ready queue of the thread's current priority. For this reason, the current priority of a thread is often called its dispatch priority.

Real-Time POSIX-compliant systems provide the applications with the choice between round-robin or FIFO policies for scheduling equal-priority threads. Having all equal-(current-) priority threads in one queue makes it convenient for the kernel to carry out either policy. We already described how to do this in Section 12.1.2.

Finding the highest priority ready thread amounts to finding the highest priority nonempty queue. The theoretical worst-case time complexity of this operation is $O(\Omega)$, where Ω is the number of priority levels supported by the operating system. In fact, the number of comparisons required to scan the queues is at most $\Omega/K + \log_2 K - 1$, where K is the word length of the CPU. (How this is done is left to you as an exercise in Problem 12.7.) Therefore, on a 32-bit CPU, the scheduler takes at most 12 comparisons to find the highest priority threads when there are 256 priority levels.

EDF Scheduling. Most existing operating systems supposely support dynamic priority. This claim typically means that the operating system provides a system call by which a thread can set and change its own priority or the priority of some other thread. This mechanism is adequate for mode-change and reconfiguration purposes but is too expensive to support dynamic scheduling policies such as the EDF algorithm. [We note that to change the priority of a thread that is ready to run, the thread must be removed from the queue for its current priority, the value of its priority (in TCB) changed, and then the thread inserted into the queue for its new priority.]

A better alternative is for the kernel to provide EDF scheduling capability. As it turns out, the kernel can support both fixed-priority and EDF scheduling using essentially the same queue structure, in particular, the queues for deadline-monotonic scheduling. In addition to the small modification of the queue structure, which we will describe below, some of the kernel functions need to be modified as follows.

[11] An example is a thread is created by a *fork( )* function. It inherits the priority of the parent thread in the parent process. If the system supports EDF scheduling, the absolute and relative deadlines of each thread should also be included in the TCB of the thread.

1. The create thread function specifies the relative deadline of the thread. (Earlier, we mentioned that if the operating system supports periodic threads, the relative deadline should be a parameter of each periodic thread. To support EDF scheduling, the operating system needs this parameter of every thread, not just periodic threads.)
2. Whenever a thread is released, its absolute deadline is calculated from its release time and relative deadline. This can the done by either a timer function or the scheduler.

In short, both the relative and absolute deadlines of each ready thread are known, and they are kept in the TCB.

Rather than maintaining a single EDF queue, the scheduler keeps a FIFO queue for threads of each relative deadline. The scheduler places each newly released thread at the end of the queue for threads which have the same relative deadline as the new thread, as if the threads were scheduled on the deadline-monotonic basis. Clearly, the threads in each queue are ordered among themselves according to their absolute deadlines. Therefore, to find the thread with the earliest absolute deadline, the scheduler only needs to search among the threads at the heads of all the FIFO queues.

To minimize the time for dequeueing the highest priority thread, the scheduler can keep the threads at the heads of the FIFO queues in a priority queue of length Ω', where Ω' is the number of distinct relative deadlines supported by the system. This queue structure is shown in Figure 12–6. The time to update the queue structure is the time required to insert a new thread into the priority queue. The time complexity of this operation is $O(\log \Omega')$, and it occurs only when a thread completes and when the scheduler inserts a new thread into an empty FIFO queue. The scheduler can dequeue the highest priority thread in $O(\log \Omega')$ time as well. The schedulable utilization of a large number of threads depends on Ω'. This dependency was discussed in Section 6.8.4.

Preemption Lock. Some kernel activities leave the system in inconsistent states if preempted. Consequently, it is necessary to make some portions of some system calls nonpreemptable. In a good operating system, system calls are preemptable whenever possible, and

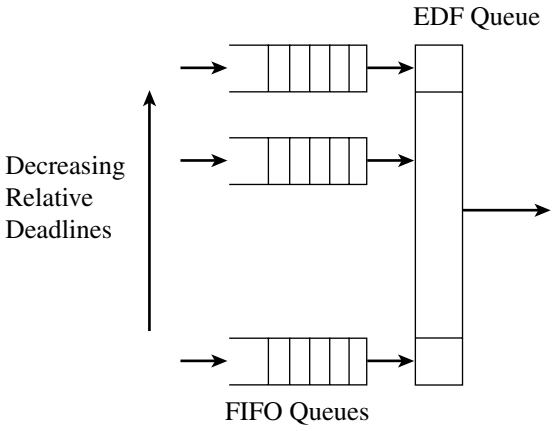

FIGURE 12–6 Queue structure for EDF scheduling.

each nonpreemptable section is implemented efficiently so its execution time, and hence the blocking time due to nonpreemptivity, is as small as possible.

Almost all real-time operating systems allow a thread to disable preemption. We use the name *preemption lock*, which is the name used by VxWorks [Wind] for this mechanism. [In VxWorks, a task can call *taskLock( )* to disable preemption; no other task can preempt it until it completes or executes the unlock routine *taskUnlock( )*.] You recall that the Nonpreemptable Critical Section (NPCS) protocol is a simple way to control access to shared resources (e.g., mutex objects and reader/writer locks). With preemption lock, this protocol is simple to implement at the user level. Each thread makes itself nonpreemptable immediately prior to entering a critical section and makes itself preemptable when it no longer holds any resource. Because the resource requested by the thread is always available and the thread has the highest priority just before it becomes nonpreemptable, the thread always acquires the lock successfully. Of course, a thread should never self-suspend when it holds any lock and is therefore nonpreemptable.

Preemption lock is often achieved by locking or disabling the scheduler. Alternatively, an operating system (e.g., pSOS+ [Moto]) may provide a task with a choice of running in preemptive or nonpreemptive mode. A task can disable preemption by setting a mode-control bit; task switching occurs only when a nonpreemptive running task blocks or reenables preemption. Since hardware interrupt service routines execute at priorities higher than the scheduler, as shown in Figure 12–3, interrupts are not disabled while some thread is preemption locked.

Aperiodic Thread Scheduling. None of the commercial real-time operating systems support bandwidth-preserving servers. The implementation of such a server as a user thread requires the ability to monitor the duration of the server's execution (i.e., the consumed server budget) at the user level. Timer functions of the kind described above are cumbersome, expensive, and inaccurate for this purpose. To replenish the budget of a sporadic server in a fixed-priority system, the thread that maintains the server (called the user-level scheduler below) needs to know when busy intervals of higher-priority threads begin and end. To get this information at the user level, the user-level scheduler needs the cooperation of those threads, and even with their cooperation, incurs large overhead. On the other hand, as we will see shortly, it is easy and inexpensive for the kernel to get and provide this information.

Monitoring Processor Time Consumption. Specifically, to maintain a server in the user level, we must monitor the server budget consumption at the user level. What we need for this purpose is an interval timer (a stop watch) that can be reset at the replenishment time of each server. The timer counts whenever the server executes. The timer signals the user-level scheduler when the count reaches the server budget. Such a timer is similar to the UNIX interval timer ITIMER_VIRTUAL[12] in the sense that it counts only at times when the server executes. Indeed, if ITIMER_VIRTUAL were more accurate, it could be used for this purpose.

[12]Interval timers ITIMER_VIRTUAL and ITIMER_PROF are used by performance monitoring and profiling tools. The former keeps track of the CPU time consumed by the thread itself. The latter gives the total CPU time consumed by the thread plus the time consumed by the kernel while working on behalf of the thread.

Unfortunately, ITIMER_VIRTUAL has two major shortcomings. The first is its resolution, and the second is its accuracy.

In most UNIX systems, the kernel updates ITIMER_VIRTUAL only at times when it services clock interrupts. As a consequence, the resolution of the timer is coarse and the consumed budget measured by the timer may be erroneous.

The coarse resolution is not a problem, since we can take into account the effect of (actual) timer resolution as we take care of the effect of tick scheduling in our choice of server size. If we use ITIMER_VIRTUAL to measure the consumed budget, the measured value has two kinds of errors. At each clock interrupt, the kernel increments ITIMER_VIRTUAL of a thread by a tick size if it finds the thread executing. If the kernel treats a server thread in this manner, the interval timer may give an overestimation of the consumed budget since the server may not have executed throughout the clock interrupt period. On the other hand, when the kernel finds a thread not executing at the time of the clock interrupt, it does not increment ITIMER_VIRTUAL, even when the thread has executed during part of the clock interrupt period. If the thread is the server, the interval timer may underestimate the consumed budget as a consequence. An overestimation is not a serious problem; the server may be less responsive because it may be suspended when in fact it should still be eligible for execution. In contrast, an underestimation of the consumed budget may cause the server to overrun and therefore does not work correctly.

We can easily modify the tick scheduling mechanism to monitor server budget consumption correctly, if not accurately. For this purpose, the kernel needs to keep the current budget of each server thread (and in the case of a SpSL server, the current budget chunks) in the TCB of the thread. The scheduler sets the time slice of a server to the current server budget whenever it schedules the server to run. The scheduler decrements the budget by the tick size during every clock interrupt period in which the server has executed.[13] Figure 12–7 shows a way to do so. As a consequence, the scheduler never underestimates the consumed budget. Whenever the scheduler preempts a server or suspends a server, it writes the server's current budget (i.e., the remaining time slice) back to the TCB of the server. When the budget becomes 0, the server is no longer eligible for execution, and the scheduler moves the server back to the suspended queue.

The kernel can keep track of server budget consumption much more accurately if it takes advantage of a high-resolution hardware clock (or Pentium time counter). The scheduler reads the clock and updates the current budget of each server thread at each context switch that involves the thread. Indeed, the kernel can more accurately monitor the processor time consumption of every thread in the same manner.

Tracking Busy Intervals. Similarly, in a fixed-priority system, it is simple for the kernel to monitor and detect the beginnings and ends of busy intervals of threads with priorities higher than that of a server (or any thread). In Chapter 7, we called this set of threads $\mathbf{T}_H$, server priority π_s, and a busy interval of $\mathbf{T}_H$ a level-(π_{s-1}) busy interval. You recall that the beginning of a busy interval of $\mathbf{T}_H$ is an instant prior to which no thread in the set is ready for

[13]The scheme described here is sufficient for SpSL or deferrable servers because the budget of a SpSL or a deferrable server is consumed only when the server executes. In contrast, according to the consumption rule of simple sporadic servers stated in Section 7.3.1, after the server begins to execute, its budget should also be decremented when lower-priority threads execute. A more complex monitoring scheme is required to support this rule.

Input information maintained by kernel:

- *currentBudget*: Set to server budget at replenishment time.
- *decremented*: FALSE if the budget has not been decremented in current clock interrupt period; TRUE if otherwise.
- server priority π_s.
- *level-π_iBusy*: TRUE if the current time is in a level-π_s busy interval; FALSE otherwise
- *BEGIN* and *END*: Beginning and end of the current level-π_s busy interval.

Monitoring consumed server budget:

- When scheduling the server,
  - set time slice to *currentBudget*;
  - set *decremented* to FALSE;
- When preempting or suspending the server in the midst of a clock interrupt period,
  - if *decremented* is FALSE,
    - decrement (remaining) time slice by tick size;
    - if time slice is 0, suspend server;
    - otherwise, set *decremented* to TRUE and *currentBudget* = remaining time slice;
- At a clock interrupt when the server is executing,
  - if *decremented* is TRUE, set *decremented* to FALSE;
  - Otherwise,
    - decrement time slice by tick size;
    - if time slice is 0, suspend server.

Tracking level-π_s busy interval:

- Upon clock interrupt,
  - if *level-π_sBusy* is TRUE,
    - if the queues for priorities π_s or higher are empty,
      - set *END* to current time and *level-π_sBusy* to FALSE;
      - service timer events;
      - if a thread in $\mathbf{T}_H$ is released, set *BEGIN* to current time and *level-π_sBusy* to TRUE;
    - otherwise (i.e., if the queues are not empty), service timer events;
  - otherwise (i.e., if *level-π_sBusy* is FALSE),
    - service timer events;
    - if a thread in $\mathbf{T}_H$ is released, set *BEGIN* to current time and *level-π_sBusy* to TRUE;
- When releasing a thread at times other than clock interrupts,
  - if *level-π_sBusy* is FALSE and the newly released thread is in $\mathbf{T}_H$, set *BEGIN* to current time and *level-π_sBusy* to TRUE;
- When a thread of priority π_s or higher completes,
  - if *level-π_sBusy* is TRUE and queues for priorities π_s and higher become empty, set *END* to current time and set *level-π_sBusy* to FALSE.

FIGURE 12–7 Monitoring budget consumption and tracking busy intervals by kernel.

execution and at which a thread in the set is released. The end of the busy interval is the first instant when all threads in $\mathbf{T}_H$ that were released before the instant have completed. Since a new busy interval may begin immediately after a busy interval ends, the processor may be continuously busy executing threads in $\mathbf{T}_H$ for the entire duration. This is why the boundaries of busy intervals cannot be detected by simply monitoring the idle/busy state of the thread set $\mathbf{T}_H$.

The kernel can check for the end of a busy interval at each clock interrupt as follows. When servicing a clock interrupt or releasing a thread upon the occurrence of a signal or interrupt, the kernel first checks whether the queues for priorities higher than π_s are empty. The current time is the end *END* of a busy interval if some of these queues were not empty the last time the kernel checked them but are all empty at the current time.

After the kernel checks the queues, it checks timer events if it is responding to a clock interrupt. The current time is the beginning *BEGIN* of a busy interval if the previous busy interval has ended and a thread in $\mathbf{T}_H$ is released at the time. Figure 12–7 also gives a pseudocode description of this scheme. *BEGIN* and *END* are information needed for determining the next replenishment time of a bandwidth-preserving server. (The replenishment rules of different servers are described in Sections 7.2 and 7.3.)

Hook for User-Level Implementation. In summary, the modified tick scheduling mechanism described above is only slightly more complicated than the existing one. The additional overhead is small. With this modification, it is relatively straightforward to implement bandwidth-preserving servers, as well as a slack stealer.

However, for modularity and customizability reasons, it is better for the operating system to provide good hooks so that bandwidth-preserving servers and slack stealers can be implemented efficiently in the user level. The minimum support the operating system should provide for this purpose is an ITIMER_VIRTUAL-like timer that never underestimates the processor time consumption of the specified thread. Better yet is for the operating system to also support server threads and provides a system call for a user thread to set the server thread budget.

Busy-interval tracking is essential. Again, this can be done with only a small modification to the standard scheduling mechanism. It would be most convenient if the operating system provides an API function which a thread can call if the thread wants to be notified when the current busy interval of the specified priority ends and when a new busy interval begins.

Static Configuration. As Chapter 9 pointed out, we want multiprocessor or distributed real-time applications to be statically configured so we can validate their end-to-end timing constraints. In a static system, computation tasks are partitioned into modules, each module is bound to a processor, and the tasks and threads on each processor are scheduled according to a uniprocessor scheduling algorithm. Over a network, each message stream is sent over a fixed route. Many real-time operating systems are uniprocessor, multitasking systems, so if we run our application on multiple networked machines on any of these operating systems, our application is naturally statically configured.

In contrast, modern general-purpose operating systems and some real-time operating systems (e.g., such as QNX [QNX]) are Symmetrical Multiprocessor (SMP) systems. Multi-

processor operating systems are designed to support dynamically configured applications. In a dynamic system, the dispatcher/scheduler puts all the ready threads in a common queue and schedules the highest priority thread on any available processor. This is what the scheduler in an SMP operating system does unless you tell it to do otherwise. The mechanism for this purpose is called *processor affinity* [Solo, QNX]. Among the scheduling information maintained in the TCB of each thread, there is an affinity mask. There is a bit for every processor in the system. By default, the affinity mask of a thread is set to all 1's, telling the scheduler that the thread can run on every processor. By using a set affinity mask system call, a thread can set its own affinity mask or that of another thread (or process) so there is only one "1" among all bits. This value of the mask tells the scheduler to schedule the thread only on the processor indicated by the "1" in the mask. In this way, a real-time application can bind its threads and processes to processors.

pSOS+m [Moto] is a multiprocessor kernel that supports the MPCP (Multiprocessor Priority-Ceiling) and end-to-end multiprocessor models in a natural way. We postpone this discussion until Section 12.6 when we describe this operating system.

Release Guard Mechanism. In Chapter 9 it was also pointed out that simply binding computation tasks to CPUs is not sufficient. If the tasks are scheduled on a fixed-priority basis, we should synchronize the releases of periodic tasks on different CPUs according to one of the nongreedy protocols discussed in Section 9.4.1. Similarly, sending message streams through fixed routes is not sufficient. Traffic shaping at the network interfaces is needed. By doing so, the worst-case response time of each end-to-end task can be computed by the sum of worst-case response times of the subtasks on individual CPUs and networks.

Among nongreedy protocols, the Release-Guard (RG) protocol is best, because it not only keeps the worst-case response time of every subtask small but also keeps the average response time of all subtasks small. Moreover, in any operating system that supports periodic threads, it is straightforward to integrate rule 1 of the protocol into the mechanism for releasing periodic threads. Specifically, the kernel maintains for each periodic thread a release guard, which the kernel sets to the current time plus the period of the thread whenever it releases the thread. The kernel releases the thread at that time only if it has received notification (in the form of a message, or an I/O complete notification, etc.) from the predecessor; otherwise, it waits until the notification is received. This rule is also simple to implement in the user level. (The implementations of periodic threads in Figure 12–5 can be modified straightforwardly to incorporate a release guard for the thread; the modification is left to you as an exercise.)

According to rule 2 of the protocol, the release guard of every thread is set to the current time at the end of each busy interval of the entire system. This rule is expensive to implement in the user level unless the kernel helps. If the kernel detects the end of each busy interval and provides a notification of some form, it is also simple to integrate rule 2 with the maintenance of periodic threads and bandwidth-preserving servers. At the end of each busy interval, which can be detected in the way shown in Figure 12–7, the kernel (or a user-level scheduler) can set the release guard of every periodic thread to the current time and release all those threads for which the notifications for the completion of their predecessors have already been received.

12.3 OTHER BASIC OPERATING SYSTEM FUNCTIONS

This section continues our discussion on operating system services that are essential for all but the simplest embedded applications. Specifically, it discusses real-time issues in communication and synchronization, software interrupt, memory management, I/O, and networking.

12.3.1 Communication and Synchronization

As they execute, threads *communicate* (i.e., they exchange control information and data). They *synchronize* in order to ensure that their exchanges occur at the right times and under the right conditions and that they do not get into each other's way. Shared memory, message queues, synchronization primitives (e.g., mutexes, conditional variables, and semaphores), and events and signals are commonly used mechanisms for these purposes.[14] Almost all operating systems provide a variety of them in one form or the other. (The only exceptions are single-threaded executives intended solely for small and deterministic embedded applications.)

This subsection discusses message queues and mutexes and reader/writer locks, leaving events and signals to the next subsection. We skip shared memory entirely. Shared memory provides a low-level, high-bandwidth and low-latency means of interprocess communication. It is commonly used for communication among not only processes that run on one processor but also processes that run on tightly coupled multiprocessors. (An example of the latter is radar signal processing. A shared memory between signal and data processors makes the large number of track records produced by signal processors available to the tracking process running on the data processor or processors.) We gloss over this scheme because we have little that is specific to real-time applications to add to what has already been said about it in the literature. (As an example, Gallmeister [Gall] has a concise and clear explanation on how to use shared memory in general and in systems that are compliant to POSOX real-time extensions in specific.) The little we have to add is the fact that real-time applications sometimes do not explicitly synchronize accesses to shared memory; rather, they rely on "synchronization by scheduling," that is, threads that access the shared memory are so scheduled as to make explicit synchronization unnecessary. Thus, the application developer transfers the burden of providing reliable access to shared memory from synchronization to scheduling and schedulability analysis. The cost is that many hard real-time requirements arise from this as a result and the system is brittle. Using semaphores and mutexes to synchronously access shared memory is the recommended alternative.

Message Queues. As its name tells us, a message queue provides a place where one or more threads can pass messages to some other thread or threads. Message queues provide a file-like interface; they are an easy-to-use means of many-to-many communication among threads. In particular, Real-Time POSIX message queue interface functions, such as *mq_send( )* and *mq_receive( )*, can be implemented as fast and efficient library functions. By

[14] We skip over pipes, UNIX FIFOs, and NT named pipes. These are efficient mechanisms for communication among equal-priority processes, but their lack of prioritization is an obvious shortcoming as a means of interprocess communication in general. Gallmeister [Gall] compares UNIX pipes, FIFOs, and message queues and Hart [Hart] compares UNIX FIFOs with NT named pipes.

making message queues location transparent, an operating system can make this mechanism as easy to use across networked machines as on a single machine.

As an example of how message queues are used, we consider a system service provider. Message queues provide a natural way of communication between it and its clients. The service provider creates a message queue, gives the message queue a name, and makes this name known to its clients. To request service, a client thread opens the message queue and places its Request-For-Service (RFS) message in the queue. The service provider may also use message queues as the means for returning the results it produces back to the clients. A client gets the result by opening the result queue and receiving the message in it.

Prioritization. You can see from the above example that message queues should be priority queues. The sending thread can specify the priority of its message in its send message call. (The parameters of the Real-Time POSIX send function *mq_send( )* are the name of the message queue, the location and length of the message, and the priority of the message.) The message will be dequeued before lower-priority messages. Thus, the service provider in our example receives the RFS messages in priority order.

Messages in Real-Time POSIX message queues have priorities in the range [0, MQ_MAX_PRIO], where the number MQ_MAX_PRIO of message priorities is at least 31. (In contrast, noncompliant operating systems typically support only two priority levels: normal and urgent. Normal messages are queued in FIFO order while an urgent message is placed at the head of the queue.) It makes sense for an operating system to offer equal numbers of message and thread priorities.[15] Some systems do. For the sake of simplicity, our subsequent discussion assumes equal numbers of threads and message priorities.

Message-Based Priority Inheritance. A message is not read until a receiving thread executes a receive [e.g., Real-Time POSIX *mg_receive( )*]. Therefore, giving a low priority to a thread that is to receive and act upon a high-priority message is a poor choice in general, unless a schedulability analysis can show that the receiving thread can nevertheless complete in time. (Section 6.8.6 gives a scheme: You can treat the sending and receiving threads as two job segments with different priorities.)

A way to ensure consistent prioritization is to provide message-based priority inheritance, as QNX [QNX] does. A QNX server process (i.e., a service provider) receives messages in priority order. It provides a work thread to service each request. Each work thread inherits the priority of the request message, which is the priority of the sender. Real-Time POSIX does not support message-based priority inheritance. A way suggested by Gallmeister [Gall] to emulate this mechanism is to give the service provider the highest priority while it waits for messages. When it receives a message, it lowers its priority to the message priority. Thus, the service provider tracks the priorities of the requests.

---

[15] A question here is whether 32 message priority levels are sufficient if the system provides a larger number of thread priorities. To answer this question, we suppose there are 256 thread priorities, and these priorities are mapped uniformly to the 32 message priorities: A thread of priority x sets its message priority to $\lfloor x/8 \rfloor$. (As we did in earlier chapters, a smaller integer represents a higher priority.) Its message may experience priority inversion only when threads of priorities $1 + 8\lfloor x/8 \rfloor, 2 + 8\lfloor x/8 \rfloor, \ldots, 8 + 8\lfloor x/8 \rfloor$ are sending messages via the same message queue at the same time.

Traditional service providers in microkernel systems are single threaded and service one request at a time. Since its execution is preemptable, uncontrolled priority inversion can still occur even with message-based priority inheritance. To control priority inversion in this case, we want the service provider to inherit the highest priority of all the threads requesting services at the time. In a system where message-queue send and receive functions are implemented as system calls, this priority inheritance can be done by having the kernel raise the priority of the service provider to the message priority (i.e., the sender's priority) whenever it places a message at the head of the service provider's request message queue if the priority of the service provider is lower at the time. The kernel adjusts the priority of the service provider as indicated by the priority of the message at the head of its (RFS) message queue each time the service provider executes a receive. A service provider is suspended when its message queue is empty. A suspended thread consumes no processor time; hence there is no harm leaving the service provider at its current priority. When the kernel puts a message in an empty queue, it sets the service provider's priority accordingly.

No Block and Notification. A useful feature is nonblocking. The Real-Time POSIX message-queue send function *mq_send( )* is nonblocking. As long as there is room in a message queue for its message, a thread can call the send function to put a message into the queue and continue to execute. However, when the queue is full, the *mq_send( )* may block. To ensure that the send call will not block when the message queue is full, we set the mode of the message queue to nonblocking (i.e., O_NONBLOCK). (The mode is an attribute of the message queue which can be set when the message queue is opened.) Similarly, by default, a thread is blocked if the message queue is empty when it calls *mq_receive( )*. We can make the receive call nonblocking in the same manner.

Notification means that a message queue notifies a process when the queue changes from being empty to nonempty. (A Real-Time POSIX message queue notifies only one process.) The service provider in the above example can arrange to be notified; thus it saves itself the trouble of having to poll its request message queue periodically after the queue becomes empty. Notification also enables a receiving process to respond quickly. As an example, suppose that a user-level bandwidth-preserving server uses a message queue as its ready queue. When an aperiodic thread is released, a message is placed in the message queue. The capability of the message queue to notify the server is essential.

Synchronization Mechanisms. Threads (and processes) synchronize using mutexes, reader/writer locks, conditional variables, and semaphores. Chapters 8 and 9 already discussed extensively protocols for controlling priority inversion that may occur when threads contend for these resources. This subsection describes a way to implement priority inheritance primitives for mutexes and reader/writer locks in a fixed-priority system. As you will see, the overhead of priority inheritance is rather high. Since the priority-ceiling protocol uses this mechanism, its overhead is also high (although not as high as simple priority inheritance since there is no transitive blocking). We will conclude the subsection by comparing priority inheritance protocol with the Ceiling-Priority Protocol (CPP). CPP is sometimes called a poor man's priority-ceiling protocol; you will see why it is so called.

Basic Primitives. In the remainder of this subsection, the term resource refers solely to either a mutex or reader/writer lock. We assume here that the operating system provides two functions: *inherit_pr( )* and *restore_pr( )*.[16] A resource manager within the operating system or at the user level can use them to raise and restore the priorities of threads, respectively, as threads acquire and release resources. Since each resource request or release may lead to the invocation of one of these functions, it is essential that they be implemented efficiently.

The function *inherit_pr(TH)* is called when a thread (named *TH* here) is denied a resource *R* and becomes blocked. The effect of this function is that all threads directly or indirectly blocking *TH* inherits *TH*'s priority. (We say that these threads inherit *TH*'s priority through resource *R*.) You may have noticed that this statement assumes that the current priorities of all these threads are lower than *TH*'s priority. This assumption is true when there is only one processor, there is no deadlock, and threads never self-suspend while holding any resource, so immediately before it becomes blocked, *TH* has the highest priority of all ready and blocked threads.

The function *restore_pr( )* is called when a resource is released. It has two parameters: The name of the resource that has just been released and the ID of the thread which releases the resource. The effect of the function is that the current priority of the specified thread is restored to a level π_r.

Inheritance Function. Figure 12–8 describes in pseudocode how the *inherit_pr( )* function works. Again, the function is called when a thread named *TH* is denied a resource *R* and becomes blocked. As the first step, the function looks up the thread holding *R* at the time. The owner TH_1 of *R* may itself be blocked waiting for another resource R_1, which may be held by yet another thread TH_2, and so on. In Chapter 8, we represented the blocking relationship by a chain in the wait-for graph, as shown in Figure 12–9. In this blocking chain, all the threads except the thread TH_i at the end is blocked. Figure 12–8 assumes that every blocking chain ends at a thread which is not blocked. If resource access is controlled according to the priority-ceiling protocol, a blocking chain may end at a resource node as well because a thread may be denied a resource even when the resource is free. The pseudocode description of *inherit_pr( )* does not take care of this case.

Tracking down threads on a blocking chain can be done easily if (1) in the date structure maintained for each resource that is in use, there is a pointer to the thread holding the resource and (2) in the TCB of each blocked thread, there is a pointer to the resource for which the thread waits. The pseudocode of *inherit_pr( )* assumes that these pointers are maintained and calls them owner and wait-for pointers, respectively. By following these pointers, *inherit_pr( )* finds each thread on the blocking chain, starting from the newly blocked thread. After finding a thread, the function changes the priority of the thread to that of the newly blocked thread. To ensure the consistency of scheduling and synchronization information, *inherit_pr( )* first locks the scheduler; it unlocks the scheduler when it completes.

Priority Restoration. A thread may hold multiple resources at the same time and may inherit different priorities as other threads become blocked when they contend for these resources. Historical information on how the thread inherited its current priority is needed to

[16]The implementation of these primitives is similar to that of the SunOS 5.0 priority inheritance mechanism [KhSZ]. The SunOS 5.0 names them *pi_willto( )* and *pi_waive( )*.

Information maintained by the kernel:

- A circular linked list, called Inheritance Log (IL), for every thread;
- For each resource R that is in use, a data structure consisting of
  - a priority queue for threads that will be blocked when they request R, and
  - an owner pointer to the thread *TH* holding R.

When called after a thread *TH*, whose current priority is π, is denied resource R, the function *inherit_pr(TH)*

1. locks scheduler;
2. adds a wait-for pointer to R in the thread's TCB;
3. starting from *TH* to the end of the blocking chain, finds the next thread, *nextThread*, on the chain as follows:
   (a) *thisThread* = *TH*;
   (b) follows *thisThread*'s wait-for pointer to the resource X *thisThread* is waiting for;
   (c) follows the owner pointer of X to find the ID, *nextThread*, of the thread holding X;
   (d) if a record on X is not in the IL of *nextThread*, inserts the record (resource name X, *nextThread*'s current priority) in the IL;
   (e) sets the current priority of *nextThread* to π;
   (f) if *nextThread* is not blocked, goes to step 4;
   otherwise, sets *thisThread* = *nextThread* and goes back to step 3(b);
4. unlocks scheduler.

When *restore_pr(R, TH)* is called, if the record (R, π) is in IL of *TH*,

1. locks scheduler;
2. computes the new priority π_r of the thread *TH* based on records in IL;
3. sets the priority of *TH* to π_r and removes (R, π) from IL;
4. unlocks scheduler.

FIGURE 12–8 Priority inheritance primitives.

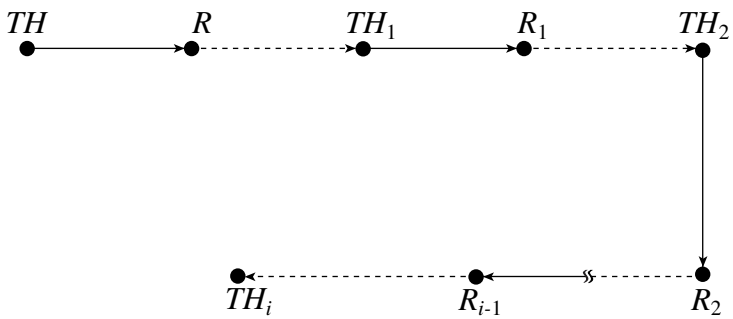

FIGURE 12–9 Blocking chain.

facilitate the determination of the thread's new current priority after it releases each resource. The pseudocode description of *inherit_pr( )* and *restore_pr( )* assumes that this information is maintained for each thread in its TCB as a linked list of records, one for each resource which the thread holds and through which the thread has inherited some priority. The list is called an Inheritance Log (IL).

When *restore_pr(TH, R)* is called, it searches the IL of the specified thread *TH* for the record on the specified resource R. The absence of a record on R indicates that the thread has not blocked any thread while it holds R. Hence, when the thread releases R, *restore_pr( )* does not change its priority. If a record on R exists, the function *restore_pr( )* computes the new thread priority, called π_r, based on the information provided by all records in the IL, restores the thread's priority to π_r, and then deletes the record on R. The computation of π_r is left for you as an exercise.

Ceiling-Priority Protocol. The overhead of the ceiling-priority protocol can be considerably lower than the priority inheritance protocol because each resource acquisition and release requires the change of the priority of at most the executing thread. CCP can be implemented easily by the system or at the user level in a fixed-priority system that supports FIFO within equal policy. (Problem 8.5 discussed how to implement this protocol if the system supports only round-robin within equal priority.) You recall that CCP requires prior knowledge of resource requirements of all threads. The resource manager generates from this knowledge the priority ceiling $\Pi(R)$ of every resource R. In addition to the current and assigned priorities of each thread, the thread's TCB also contains the names of all resources held by the thread at the current time.

Whenever a thread requests a lock on R, the resource manager locks the scheduler; looks up $\Pi(R)$; if the current priority of the requesting thread is lower than $\Pi(R)$, sets the thread's current priority to $\Pi(R)$; allocates R to the thread; and then unlocks the scheduler. Similarly, when a thread unlocks a resource R, the resource manager checks whether the thread's current priority is higher than $\Pi(R)$. The fact that the thread's current priority is higher than $\Pi(R)$ indicates that the thread still holds a resource with a priority ceiling higher than $\Pi(R)$. The thread's priority should be left unchanged in this case. On the other hand, if the thread's current priority is not higher than $\Pi(R)$, the priority may need to be lowered when R is released. In this case, the resource manager locks the scheduler, changes the current priority of the thread to the highest priority ceiling of all resources the thread still holds at the time, or the thread's assigned priority if the thread no longer holds any resource, and then unlocks the scheduler.

12.3.2 Event Notification and Software Interrupt

Event notification, exception handling, and software interrupts are essential for multitasking systems. Responsive mechanisms are needed to inform threads of the occurrences of timer events, the receipt of messages, the completion of asynchronous I/O operations, and so on. In UNIX systems, signal is the general mechanism for these purposes. Most of this subsection is devoted to the Real-Time POSIX signal as its features exemplify what are good and practical for real-time applications.

Signal and Similar Mechanisms. We saw earlier that in a UNIX system, interrupt handlers and the kernel use signals as a means to inform threads of the occurrences of ex-

ceptions (e.g., divide by zero and bad system call) or waited for events (e.g., the expiration of a timer and arrival of a message). A thread may signal another thread to synchronize and communicate. (For example, a predecessor thread may signal a successor thread when it completes.) A thread has a service function, called a signal handler. When the kernel delivers a signal to the thread, the signal handler executes. Thus, the signal mechanism provides asynchrony and immediacy, just as hardware interrupts do.

Non-UNIX systems typically provide these capabilities using more than one mechanism. As an example, in Windows NT [Solo, Hart], events and Asynchronous Procedure Calls (APCs) serve the same purposes as signals in UNIX systems. Events are set by threads for the purpose of notification and synchronization. They are synchronous. In other words, a thread must explicitly wait for an event for the event to have effect, and the thread is suspended while it waits. When a thread sets an event (object) to the signaled state, the thread(s) that is waiting on the event is awakened and scheduled. An NT event is an efficient and powerful notification and synchronization mechanism. Being synchronous, NT event delivery does not have the relatively high overhead of asynchronism. It is many-to-many in the sense that multiple threads can wait on one event and a thread can wait for multiple events. In contrast, each UNIX signal is targeted to an individual thread or process, and each signal is handled independently of other signals.

As its name implies, NT APCs are asynchronous.[17] The kernel and device drivers use kernel-mode APCs to do work in the address spaces of user threads (e.g., to copy data from kernel space to the space of the thread that initiated the I/O). Environment subsystems (i.e., the POSIX subsystem) uses APCs as software interrupts (e.g., to suspend a thread). Indeed, the POSIX subsystem on NT implements the delivery of POSIX signals to user processes using APCs. We will discuss this mechanism further in Section 12.7.1.

Similarly, the pSOSystem [Moto] provides events and asynchronous signals. The former are synchronous and point-to-point. An event is sent to a specified receiving task. The event has no effect on the receiving task if the task does not call the event receive function. In contrast, an asynchronous signal forces the receiving task to respond.

More on Signals. Internal to UNIX operating systems, signals used for different purposes are identified by numbers. In programs and in the literature, they are referred to by their symbolic names. The correspondence between the number and the symbolic name of each signal is given in the header file *<signal.h>*. Most POSIX signals (i.e., signal numbers) are used by the operating system; what they do is defined by the system. (We have already seen SIGALRM for signaling upon timer expiration. Other examples are SIGTERM for terminating a process, SIGFPE for floating-point exception, and SIGPIPE for writing a pipe with no readers.) There are also signal numbers which are to be used solely for application-defined purposes. A Real-Time POSIX compliant system provides at least eight application-defined signals. An application-defined signal (with one of the numbers reserved for applications) is defined by the function (i.e., the signal handler) which the application sets up for the operating system to call when a signal with that number arrives. [Specifically, the action to be taken upon

[17] More precisely, NT kernel-mode APCs are asynchronous; a kernel-mode APC interrupts a thread without the thread's cooperation. There are also user-mode APCs. A user-level APC queued for a thread is delivered to the thread only when the thread is in a wait state (e.g., by having previous executed a *SleepEx* to test whether it has a pending APC).

the occurrence of a signal is specified by the data structure *sigaction* on the signal. Among the information provided by this data structure is a pointer to the signal handler, a mask *sa_mask* and a flag field *sa_flag*. A process (thread) can ask the operating system to set these members by calling *sigaction( )* and providing these and other arguments of the function.]

By default, when a signal arrives, it is handled, meaning that the signal handler set up for the signal is executed. A process (thread) may ignore any signal, except those (e.g, SIGKILL) used by the operating system to stop and destroy an errant process. The operating system ignores a signal when the pointer to the associated signal handler has the system-defined constant SIG_IGN as its value.

Just as it is necessary to be able to disable interrupts, it is essential that signals can be blocked selectively from delivery. This is done via signal masks. Some signals may need to be blocked from delivery while a signal handler is executing. One specifies the signals blocked by the handler of a signal by putting their numbers in *sa_mask* in the data structure *sigaction* of that signal. The operating system does not deliver a signal thus blocked while the signal handler is executing. Each process (thread) also has a signal mask containing numbers of signals the process (thread) wants blocked. A signal thus blocked is delivered by the operating system but remains pending until the process (thread) unblocks the signal by removing its number from the signal mask.

Signal delivery is on per process basis, so if a thread ignores a signal, all threads in the process do also. On the other hand, each thread has its own signal mask. Therefore, threads in a process can choose to block or unblock signals independent of other threads. Some signals (e.g., SIGTERM for terminating a process) are intended for the entire process. Such a signal, if not ignored, is delivered to any thread that is in the process and does not block the signal. The signal is blocked only when all threads in the process blocks it.

Real-Time POSIX Signals. POSIX real-time extensions [IEEE98] change the POSIX signals to make the mechanism more responsive and reliable. To distinguish signals conforming to POSIX real-time extensions from POSIX and traditional UNIX signals, we call the former real-time signals. First, as we mentioned earlier, there are at least eight application-defined real-time signals versus only two provided by POSIX. These signals are numbered from SIGRTMIN to SIGRTMAX.

Second, real-time signals can be queued, while traditional UNIX signals cannot. Queueing is a per signal option; one chooses the queueing option for a real-time signal by setting bit SA_SIGINFO in the *sa_flags* field of the *sigaction* structure of the signal. If not queued, a signal that is delivered while being blocked may be lost. Hence, queueing ensures the reliable delivery of signals.

Third, a queued real-time signal can carry data. A traditional signal handler has only one parameter: the number of the signal. In contrast, the signal handler of a real-time signal whose SA_SIGINFO bit is set has as an additional parameter a pointer to a data structure that contains the data value to be passed to the signal handler. This capability increases the communication bandwidth of the signal mechanism. As an example, a server can use this mechanism to notify a client of the completion of a requested service and pass the result back to the client at the same time.

Fourth, queued signals are prioritized: The smaller the signal number, the higher the priority. They are delivered in priority order.

Fifth, POSIX real-time extensions provide a new and more responsive synchronous signal-wait function called *sigwaitinfo*. When a signal arrives for a process that is blocked after calling the POSIX synchronous signal-wait function *sigsuspend*, the signal handler executes before the process returns from the blocked state. In contrast, upon the arrival of the signal, the *sigwaitinfo* function does not call the signal handler; it merely unblocks the calling process. Figure 12–5(a) gives an example of how this function may be used.

Overhead and Timing Predictability. The down side of the signal mechanism is the slow speed and high overhead of signal delivery. Like a hardware interrupt, a signal also causes a trap out of the user mode, complicated operations by the operating system, and a return to user mode. The time taken by these activities is high in comparison with other communication and synchronization mechanisms.

Signal handler are executed ahead of threads of all priorities. (Typically, the kernel executes signal handlers prior to returning control to the user.) Hence, it is important that the execution times of signal handlers be kept small, just as it is important that execution times of interrupt service routines small.

12.3.3 Memory Management

Thus far, we have ignored memory and memory management. An underlying assumption throughout previous chapters is that all the real-time applications that may run together fit in memory. Indeed, a task is not admitted into the system if there is not enough memory to meet its peak memory space demand. A mode change cannot complete unless the operating system can find sufficient memory for the codes, data, and run-time stacks of the threads that execute in the new mode. We now discuss those aspects of memory management that call for attention. They are virtual memory mapping, paging, and memory protection. Whereas all general-purpose operating systems support virtual memory and memory protection, not all real-time operating systems do, and those that do typically provide the user with the choice of protection or no protection. Unlike nonreal-time applications, some real-time applications do not need these capabilities, and we do not get these capabilities without penalty.

Virtual Memory Mapping. We can divide real-time operating systems into three categories depending on whether they support virtual memory mapping (i.e., virtual contiguity) and paging (i.e., demand paging or swapping). Real-time operating systems designed primarily for embedded real-time applications such as data acquisition, signal processing, and monitoring,[18] may not support virtual memory mapping. The pSOSystem is an example [Moto]. Upon request, the system creates physically contiguous blocks of memory for the application. The application may request variable size segments from its memory block and define a memory partition consisting of physically contiguous, fixed-size buffers.

[18]The lack of virtual memory mapping is not a serious problem when the application has a small code and relatively small number of operating modes (and hence configurations). The application may be data intensive and its data come in fixed-size chunks. A radar signal processing application is an example. It needs fixed-size buffers to hold digitized radar returns from range bins, and the space required for its FFT code and run time stack is small by comparison. During a mode change, for example, from searching to tracking, it may request more buffers or free some buffers. The number of buffers required in each mode is usually known.

Memory fragmentation is a potential problem for a system that does not support virtual mapping. After allocating variable-size segments, large fractions of individual blocks may be unused. The available space may not be contiguous and contiguous areas in memory may not be big enough to meet the application's buffer space demand. The solution is to provide virtual memory mapping from physical addresses, which may not be contiguous, to a contiguous, linear virtual address space seen by the application.

The penalty of virtual address mapping is the address translation table, which must be maintained and hence contribute to the size of the operating system. Moreover, it complicates DMA-controlled I/O. When transferring data under DMA control, it is only necessary to set up the DMA controller once if the memory addresses to and from which the data are transferred are physically contiguous. On the other hand, when the addresses are not physically contiguous, the processor must set up the DMA controller multiple times, one for each physically contiguous block of addresses.

Memory Locking. A real-time operating system may support paging so that nonreal-time, memory demanding applications (e.g., editors, debuggers and performance profilers) needed during development can run together with target real-time applications. Such an operating system must provide applications with some means to control paging. Indeed, all operating systems, including general-purpose ones, offer some control, with different granularities.

An examples, Real-Time POSIX-compliant systems allow an application to pin down in memory all of its pages [i.e., *mlockall( )*] or a range of pages [i.e., *mlock( )* with the starting address of the address range and the length of the range as parameters]. So does Real-Time Mach. In some operating systems (e.g., Windows NT), the user may specify in the create thread system call that all pages belonging to the new thread are to be pinned down in memory. The LynxOS operating system controls paging according to the demand-paging priority. Memory pages of applications whose priorities are equal to or higher than this priority are pinned down in memory while memory pages of applications whose priorities are lower than the demand-paging priority may be paged out.

Memory Protection. As we stated in Section 12.1.2, many real-time operating systems do not provide protected address spaces to the operating system kernel and user processes. Argument for having only a single address space include simplicity and the light weight of system calls and interrupt handling. For small embedded applications, the overhead space of a few kilobytes per process is more serious. Critics points out a change in any module may require retesting the entire system. This can significantly increase the cost of developing all but the simplest embedded systems. For this reason, many real-time operating systems (e.g., QNX and LynxOS) support memory protection.

A good alterative is to provide the application with the choices in memory management such as the choices in virtual memory configuration offerred by VxWorks [Wind] and QNX [QNX]. In VxWorks, we can choose (by defining the configuration in *configAll.h*) to have only virtual address mapping, to have text segments and exception vector tables write protected, and to give each task a private virtual memory when the task requests for it.

12.3.4 I/O and Networking

Three modern features of file system and networking software are (1) multithreaded-server architecture, (2) early demultiplexing and (3) lightweight protocol stack. These features were developed to improve the performance of time-shared applications, high-performance applications and network appliances. As it turns out, they also benefit real-time applications with enhanced predictability.

Multithreaded Server Architecture. Servers in modern operating systems are typically multithreaded. In response to a request, such a server activates (or creates) a work thread to service the request. By properly prioritizing the work thread, we can minimize the duration of priority inversion and better account for the CPU time consumed by the server while serving the client.

As a example, we consider a client thread that does a read. Suppose that the file server's request queue is prioritized, the request message has the same priority as the client, and the work thread for each request inherits the priority of the request message. (Section 12.3.1 described a way to do this.) As a consequence, the length of time the client may be blocked is at most equal to the time the server takes to activate a work thread. I/O requests are sent to the disk controller in priority order. For the purpose of schedulability analysis, we can model the client thread as an end-to-end job consisting of a CPU subjob, which is followed by a disk-access subjob, and the disk-access job executes on the disk system (i.e., the controller and the disk) and is in turn followed by a CPU subjob. Both CPU subjobs have the same priority as the client thread, and their execution times include the lengths of time the work thread executes. Since the time taken by the server to activate a work thread is small, the possible blocking suffered by the client is small. In contrast, if the server were single-threaded, a client might be blocked for the entire duration when the server executes on behalf of another client, and the blocking time can be orders of magnitude larger.

Early Demultiplexing. Traditional protocol handlers are based on layered architectures. They can introduce a large blocking time. For example, when packets arrive over a TCP/IP connection, the protocol module in the network layer acknowledges the receipts of the packets, strips away their headers, reassembles them into IP datagrams, and hands off the datagram to the IP module after each datagram is reassembled. Similarly, the TCP and IP modules reassemble IP datagrams into messages, put the messages in order and then deliver the messages to the receiver. Because much of this work is done before the identity of the receiver becomes known, it cannot be correctly prioritized. Typically, the protocol modules execute at a higher priority than all user threads and block high-priority threads when they process packets of low-priority clients.

The duration of priority inversion can be reduced and controlled only by identifying the receiver of each message (i.e., demultiplexing incoming messages) as soon as possible. Once the receiver is known, the execution of the protocol modules can be at the priority of the receiver.

Traditionally, incoming messages are first moved to the buffer space of the protocol modules. (Figure 11-1 is based on this assumption.) They are then copied to the address space of the receiver. Early demultiplexing also makes it possible to eliminate the extra copying. Some communication mechanisms [e.g., Illinios FM (Fast Message] [PaLC]) for high-speed local networks take this approach. The fact that messages are copied directly between the net-

work interface card and the application is a major reason that these mechanisms can achieve an end-to-end (i.e., application-to-application) latency in the order of 10 to 20 microseconds.

Lightweight Protocol Stack. Protocol processing can introduce large overhead and long latency. This is especially true when the protocol stack has the client/server structure: Each higher-level protocol module uses the services provided by a protocol module at a layer below. This overhead can be reduced by combining the protocol modules of different layers into a single module and optimizing the module whenever possible. A challenge is how to provide lightweight, low overhead protocol processing and still retain the advantages of the layer architecture.

An example of operating systems designed to meet this challenge is the Scout operating system [MMOP, MoPe]. Scout is based on the Path abstraction. Specifically, Scout uses paths as a means to speed up the data delivery between the network and applications. Conceptually, a path can be thought of as a bidirectional virtual connection (channel) which cuts through the protocol layers in a host to connect a source and a sink of data to the network. From the implementation point of view, a path is an object that is obtained by glueing together protocol modules (called routers) to be used to process data between an application and network.

At configuration time, a path consisting multiple stages is created one stage at a time starting from one end of the path (e.g., the network layer router). A stage is created by invocating the function *pathCreate( )* on the router specified by one of the arguments of the function. (The other argument of the function specifies the attributes of the path.) As the result of the invocation of *pathCreate( )* and the subsequent invocation of *createStage* function, the router creates a stage of the path and identifies the next router, if any, on the path. Similarly, the next stage is created by the next router, and if the next router is to be followed by yet another router, that router identified. This process is repeated until the entire sequence of stages is created. The stages are then combined into a single path object and initialized. Whenever possible, Scout applies path transformation rules to optimize the path object.

At run time, each path is executed by a thread. Scout allows these threads to be scheduled according to multiple arbitrary scheduling policies and allocates a fraction of CPU time to threads scheduled according to each policy. As you will see shortly, this features complements ideally resource reserves that we will present in the next section.

Still, priority inversion is unavoidable because when a packet arrives, it may take some time to identify the path to which the packet belongs. Scout tries to minimize this time by giving each router a demultiplexing operation. A router asks the next router to refine its own decision only when it cannot uniquely decide to which path a packet belongs.

*12.4 PROCESSOR RESERVES AND RESOURCE KERNEL

One can easily argue that a real-time operating system should support as options admission control, resource reservation, and usage enforcement for applications that can afford the additional size and complexity of the option. By monitoring and controlling the workload, the operating system can guarantee the real-time performance of applications it admits into the system. This is the objective of the CPU capacity-reservation mechanism proposed by Mercer, *et al.* [MeST] to manage quality of services of multimedia applications. Rajkumar, *et al.* [RKMO] have since extended the CPU reserve abstraction to reserves of other resources and

used it as the basis of *resource kernels*. In addition to Real-Time Mach, NT/RK[19] also provides resource kernel primitives. Commercial operating systems do not support this option.

12.4.1 Resource Model and Reservation Types

A resource kernel presents to applications a uniform model of all time-multiplexed resources, for example, CPU, disk, network link. We been calling these resources processors and will continue to do so.

An application running on a resource kernel can ensure its real-time performance by reserving the resources it needs to achieve the performance. To make a reservation on a processor, it sends the kernel a request for a share of the processor. The kernel accepts the request only when it can provide the application with the requested amount on a timely basis. Once the kernel accepts the request, it sets aside the reserved amount for the application and guarantees the timely delivery of the amount for the duration of the reservation.

Reservation Specification. Specifically, each reservation is specified by parameters e, p, and D: The reservation is for e units (of CPU time, disk blocks, bits or packets, and so on) in every period of length p, and the kernel is to provide the e units of every period (i.e., every instance of the reservation) within a relative deadline D. Let ϕ denote the first time instant when the reservation for the processor is to be made and L denote the length of time for which the reservation remains. The application presents to the kernel the 5-tuple (ϕ, p, e, D, L) of parameters when requesting a reservation.

In addition to the parameters of individual reservations, the kernel has an overall parameter B for each type of processor. As a consequence of contention for nonpreemptable resources (such as buffers and mutexes), entities executing on the processor may block each other. B is the maximum allowed blocking time. (B is a design parameter. It is lower bounded by the maximum duration of time entities using the processor holding nonpreemptable resources. Hence, the larger B is, the less restriction is placed on applications using the processor, but the larger fraction of processor capacity becomes unavailable to reservations.) Every reservation request implicitly promises never to hold any nonpreemptable resources so long as to block higher-priority reservations on the processor for a duration longer than B, and the kernel monitors the usage of all resources so it can enforce this promise.

As an example, let us look at CPU reservation. From the resource kernel point of view, each CPU reservation (p, e, D) is analogous to a bandwidth-preserving server with period p and execution budget e whose budget must be consumed within D units of time after replenishment. Any number of threads can share a CPU reservation, just as any number of jobs may be executed by a server. In addition to the CPU, threads may use nonpreemptable resources. The kernel must use some means to control priority inversion. The specific protocol it uses is unimportant for the discussion here, provided that it keeps the duration of blocking bounded.

Maintenance and Admission Control. If all the threads sharing each CPU reservation are scheduled at the same priority (i.e., at the priority of the reservation), as it is suggested

[19] NT/RK is a middleware that runs on top of Windows NT. NT 5.0 allows processes to set execution and memory usage limits for a process or group of processes. This capability can be exploited to support CPU and memory resource reserves; it is described in Section 12.7.1.

in [RKMO], each CPU reservation behaves like a bandwidth-preserving server. The kernel can use any bandwidth-preserving server scheme that is compatible with the overall scheduling algorithm to maintain each reservation. The budget consumption and busy interval tracking mechanisms described in Figure 12–7 are useful for this purpose.

The simplest way is to replenish each CPU reservation periodically. After granting a reservation (p, e, D), the kernel sets the execution budget of the reservation to e and sets a timer to expire periodically with period p. Whenever the timer expires, it replenishes the budget (i.e., sets the budget in the reserve back to e). Whenever any thread using the reservation executes, the kernel decrements the budget. It suspends the execution of all threads sharing the reservation when the reservation no longer has any budget and allows their execution to resume after it replenishes the budget. You may have noticed that we have just described the budget consumption and replenishment rules of the deferrable server algorithm. (The algorithm was described in Section 7.2.) In general, threads sharing a reservation may be scheduled at different fixed priorities. This choice may make the acceptance test considerably more complicated, however. If budget replenishment is according to a sporadic server algorithm, budget replenishment also becomes complicated.

The connection-oriented approach is a natural way to maintain and guarantee a network reservation. A network reservation with parameters p, e, and D is similar to a connection on which the flow specification is (p, e, D). After accepting a network reservation request, the kernel establishes a connection and allocates the required bandwidth to the connection. By scheduling the message streams on the connection according to a rate-based (i.e., bandwidth-preserving) algorithm, the kernel can make sure that the message streams on the connection will receive the guaranteed network bandwidth regardless of the traffic on other connection.

In summary, the resource kernel for each type of processor schedules competing reservations according to an algorithm that provides isolation among reservations if such algorithms exist. Previous chapters give many such algorithms for CPU and network scheduling. The kernel can use the schedulability conditions described in those chapters as the basis of its acceptance test when deciding whether to accept and admit a reservation.

An exception is disk. Clearly, the kernel needs to use a real-time disk-scheduling algorithm that allows the maximum response time of each disk access request to be bounded from the above.[20] However, most real-time disk scheduling algorithms are not bandwidth-preserving. For this reason, the kernel must provide this capability separately from scheduling.

Types of Reservation. In addition to parameters p, e, and D, Real-Time Mach also allows an application to specify in its reservation request the type of reservation: hard, firm,

[20] Real-time disk scheduling algorithms typically combine EDF and SCAN strategies. According to the SCAN algorithm, the read/write head scans the disk, and the next request to service is the one closest to the head in the direction of scan. This strategy minimizes average seek and latency times but ignores timeliness. According to real-time scheduling algorithms such as EDF-SCAN and "just-in-time," when a request completes, the head moves to the request with the earliest deadline among all pending requests. A pure EDF scheme may yield poor average seek and latency. To improve this aspect of performance, requests on tracks between the just completed request and the request with the earliest deadline may also be serviced. Different algorithms make the decision on whether to service the in-between requests based on different criteria, for example, whether the request with the earliest deadline has slack, or whether its deadline is within a certain window of time.

or soft. Thus, it specifies the action it wants the kernel to take when the reservation runs out of budget.

The execution of all entities sharing a *hard reservation* are suspended when the reservation has exhausted its budget. In essence, a hard reservation does not use any spare processor capacity. A hard network reservation is rate controlled; its messages are not allowed to transmit above the reserved rate even when spare bandwidth is available. We make hard reservations for threads, messages, and so on, when completion-time jitters need to be kept small and early completions have no advantage.

In contrast, when a *firm reservation* (say a CPU reservation) exhausts its budget, the kernel schedules the threads using the reservation in the background of reservations that have budget and threads that have no reservation. When a *soft reservation* runs out of budget, the kernel lets the threads using the reservation execute along with threads that have no reservation and other reservations that no longer have budget; all of them are in the background of reservations that have budget. A firm or soft network reservation is rate allocated. We make firm and soft reservations when we want to keep the average response times small.

Rajkumar, *et al.* [RKMO] also suggested the division of reservations into *immediate reservations* and normal reservations. What we have discussed thus far are normal reservations. An immediate reservation has a higher priority than all normal reservations. We will return shortly to discuss the intended use of immediate reservations.

12.4.2 Application Program Interface and SSP Structure

The API functions Real-Time Mach provides to support resource reservation include *request( )* and *modify( )* for requesting and modifying reservations. There are also functions for binding threads to a reservation and specifying the type of a reservation. In addition, an application may create a port for a reservation and request the kernel to send a notification message to that port whenever the budget of the reservation is exhausted.

Mercer, *et al.* [MeST] pointed out that a System Service Provider (SSP) executes mostly in response to requests from clients. If each SSP were to reserve sufficient resources to ensure its responsiveness, most of its reserved resources would be left idle when there is no request for its service. A better alternative is to have the client pass the client's reservations needed by the SSP to do the work along with the request for service.

In a microkernel system adopting this approach, each SSP has only a small CPU reservation. Only its daemon thread executes using this reservation. When a client requests service from an SSP, it allows the SSP to use its own reservations. When the SSP receives a request, the daemon thread wakes up, executes using the SSP's CPU reservation, creates a work thread to execute on the client's behalf, and suspends itself. This work thread uses the client's CPU reservation, as well as reservations of other types of processors. In this way, the resources used by the SSP to service each request are charged to the requesting client. The SSP structure described in the next section is an extension, and we will provide more detail on this structure when we describe the extension.

To explain the use of the immediate reservation type, we consider the example given by [RKMO]: A stream of packets from the network is stored on the disk. The application clearly needs both network bandwidth and disk bandwidth; hence, it may have reservations for both types of processors. Moreover, CPU time is also needed for processing incoming packets and writing them to disk. If the application's CPU reservation were used to do this work, the

receive buffer might not be drained sufficiently fast to prevent overflow, since the application's CPU reservation may have a low priority and large relative deadline. The alternative proposed by Rajkumar, *et al.* is to have a separate (immediate) CPU reservation for processing incoming packets and writing the packets to disk and give the reservation the highest priority to ensure its immediate execution. However, immediate reservation introduces priority inversion. It is the same as executing a scheduled interrupt service routine immediately rather than properly prioritizing it. The execution times of threads using immediate reservations must be kept very small.

To conclude this section, we note that the above example illustrates an end-to-end task. Its subtasks execute on the network, on the CPU, and the disk. Providing guaranteed resource reservation for each processor type is a way to ensure that each subtask has a bounded response time and the response time of the end-to-end task is bounded by the sum of the maximum response times of its subtasks.

*12.5 OPEN SYSTEM ARCHITECTURE

For most of this book, we have assumed that all the hard real-time applications running on the same platform form a monolithic system. Their tasks are scheduled together according to the same algorithm. When we want to determine whether any application can meet its timing requirements, we carry out a global schedulability analysis based on the parameters for each combination of tasks from all applications that may run together. In essence, the system is closed, prohibiting the admission of real-time applications whose timing characteristics are not completely known.

12.5.1 Objectives and Alternatives

Ideally, a real-time operating system should create an open environment. Here, by an *open environment*, we mean specifically one in which the following development and configuration processes are feasible.

1. Independent Design Choice: The developer of a real-time application that is to run in an open environment can use a scheduling discipline best suited to the application to schedule the threads in the application and control their contention for resources used only by the application.
2. Independent Validation: To determine the schedulability of an application, its developer can assume that the application runs alone on a virtual processor that is the same as the target processor but has a speed that is only a fraction $0 < s < 1$ of the speed of the target processor. In other words, if the maximum execution time of a thread is e on the target processor, then the execution time used in the schedulability analysis is e/s. The minimum speed at which the application is schedulable is the *required capacity* of the application.[21]

[21] We note that the notion of the required capacity s here is essentially the same as that in the processor reserve model. As we will see shortly, it is not necessary for an application in the open system to specify the period p in real time over which ps units of processor time are to be allocated to the application. Instead, the application informs the operating system the shortest relative deadline of all of its threads.

3. Admission and Timing Guarantee: The open system always admits nonreal-time applications. Each real-time application starts in the nonreal-time mode. After initialization, a real-time application requests to execute in the real-time mode by sending an admission request to the operating system. In this request, the application informs the operating system of its required capacity, plus a few overall timing parameters of the application. (In the open environment described below, the parameters are the shortest relative deadline, the maximum execution time of all nonpreemptable sections, the presence or absence of sporadic threads and, if its periodic tasks have release-time jitters, an upper bound on release-time jitters.) The operating system subjects the requesting application to a simple but accurate acceptance test. If the application passes the test, the operating system switches the application to run in real-time mode. Once in real-time mode, the operating system guarantees the schedulability of the application, regardless of the behavior of the other applications in the system.

In short, independently developed and validated hard real-time applications can be configured at run time to run together with soft real-time and nonreal-time applications.

We recall that the fixed-time partitioning scheme discussed in Section 6.8.7 is a way to achieve the above goal and has been used to allow safety-critical applications to run together with nonsafety-critical applications. According to that scheme, we partition time into slots and confine the execution of each application in the time slots allocated to the application. We saw in Section 6.8.7 that if applications do not share any global resource (i.e., resources shared by tasks in different applications) and the time slots are assigned to applications on the TDM (Time-Division Multiplexing) basis, we can determine the schedulability of each application independently of other applications. The number of slots per round an application requires to be schedulable is then its required capacity. An application can be admitted into the system whenever the system has enough spare time slots per round to meet the required capacity of the application. The key parameter of the fixed-time partitioning scheme is the length of the time slots. Obviously, this length should be smaller than the shortest relative deadline of all applications. The shorter the time slots, the closer the system emulates a slower virtual processor for each application, but the higher the context-switch overhead. Hence, using the fixed-time partitioning scheme, we cannot accommodate stringent response time requirements without paying high overhead. Modern operating systems typically provide priority-driven scheduling. Some kind of middleware is needed to put the time-driven scheme on them.

The remainder of this section describes a uniprocessor open system architecture that uses the two-level priority-driven scheme described in Section 7.9. That scheme emulates infinitesmally fine-grain time slicing (often called fluid-flow processing sharing) and can accommodate stringent response time requirements. Deng, *et al.* [DLZS] implemented a prototype open system based on the architecture by extending the Windows NT operating systems. The prototype demonstrates that the architecture can easily be incorporated into any modern operating system to make the operating system open for real-time applications and still remain backward compatible for existing nonreal-time applications. By using a two-level scheme in a similar way to schedule messages to and from networks (e.g., Zhang, *et al.* [ZLDP] give a scheme for scheduling periodic messages over Myrinet), the architecture can be extended into a distributed one.

12.5.2 Two-Level Scheduler

According to the two-level priority-driven scheme described in Section 7.9, each real-time application (denoted by $\mathbf{T}_k$ for some $k \geq 1$) is executed by a bandwidth-preserving server S_k, which in essence is a constant utilization or total bandwidth server, and all the nonreal-time applications are executed by a total bandwidth server S_0. At the lower level, the *OS scheduler* maintains the servers and schedules all the ready servers. (A server is ready when it has ready threads to execute and budget to execute them.) At the higher level, the *server scheduler* of each server S_k schedules the ready threads in the application(s) executed by the server.

Scheduling Hierarchy. At each scheduling decision time, the OS scheduler schedules the server with the earliest deadline among all ready servers. When a server is scheduled, it executes the thread chosen by its own server scheduler according to the scheduling algorithm Σ_k used by the application(s) executed by the server. The block diagram in Figure 12–10 depicts this scheduling hierarchy. To highlight the fact that the scheduling disciplines used by the real-time applications may be different, the block diagram shows two different real-time scheduling disciplines. Nonreal-time applications are scheduled according to a time-shared scheduling discipline. The block diagram also suggests that we put all the schedulers in the

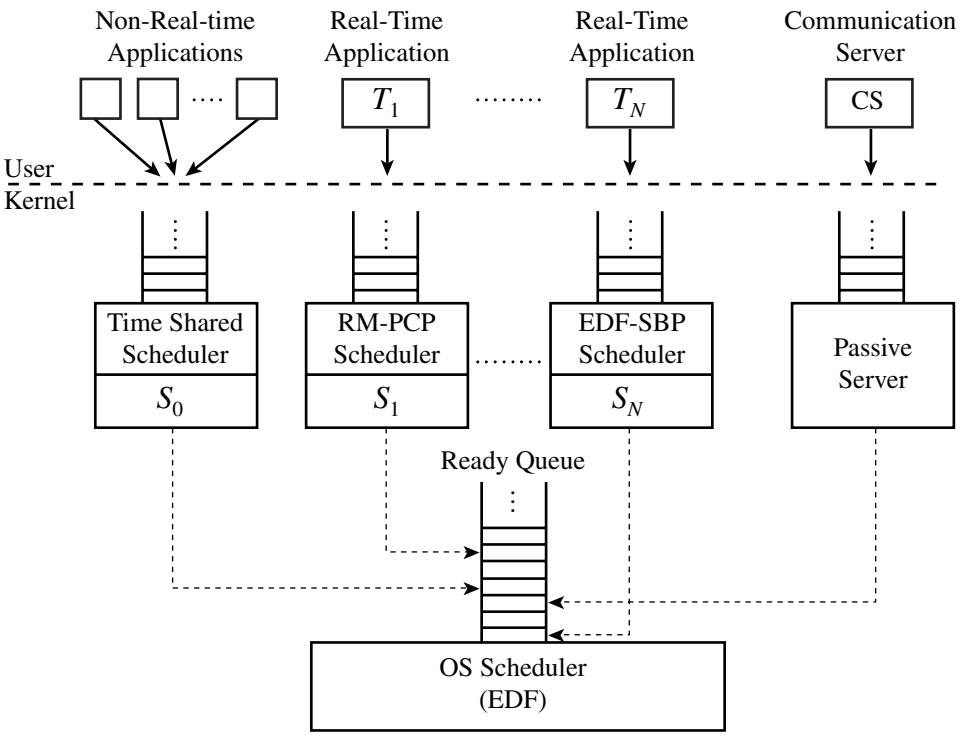

FIGURE 12–10 A two-level scheduler.

kernel; thus, only one context switch occurs at each scheduling decision time, just as in a one-level scheduling scheme.

An application can use any protocol to control access to its local resources. In this section, by a local resource, we mean one that is used by only one application. A resource shared by multiple applications is a global resource. In the open system, global resource contention is resolved according to the Nonpreemptable Critical Section (NPS) protocol or the ceiling-priority protocol. Consequently, a section of a thread may be nonpreemptable. In general, a thread or a section of a thread may be nonpreemptable for reasons other than global resource access control. There is no need for us to distinguish the reasons. Hereafter, we simply say that some applications have nonpreemptable sections, lumping applications that use global resources into this category.

Without loss of generality, we assume that the required capacity s_k of every real-time application $\mathbf{T}_k$ is less than 1. (Otherwise the application cannot share the processor with other applications.) The maximum execution time of every thread in each real-time application becomes known when the thread is released. One of the parameters a real-time application declares to the operating system when it requests to execute in the real-time mode is the maximum execution time of the nonpreemptable sections of all its threads. After admitting the application, the operating system never allows any thread in the application to be nonpreemptable for more than the declared amount of time. Similarly, the operating system ensures that no nonreal-time thread is nonpreemptable for longer than a certain length of time. (This length is a design parameter.) Consequently, at any time the maximum execution time of the nonpreemptable sections of all threads in the system is known.

In Section 7.9, we called an application that is scheduled according to a preemptive, priority-driven algorithm and contains threads whose release times are not fixed a *nonpredictable* application. It is so called because the OS scheduler needs to know the next event time of such an application at each server budget replenishment time but the server scheduler for the application cannot compute this time accurately. (At any time, the *next event time* of an application is the occurrence time of the earliest of all events in the applications that may cause a context switch.) Applications whose threads have fixed and known release times and applications that are scheduled nonpreemptively are *predictable*. An application of the former kind is predictable because its server scheduler can compute the next event time accurately. Similarly, an application that is scheduled in the clock-driven manner is predictable. A nonpreemptively scheduled application is predictable, even when it contains aperiodic and sporadic threads, because this computation is not necessary.

Operations of the OS scheduler. Figure 12–11(a) describes the operations of the OS scheduler and its interaction with the server schedulers. At initialization time, the OS scheduler creates a total bandwidth server S_0 to execute all the nonreal-time applications and a server for each service provider. The size of S_0 is $\tilde{u}_0$, and the total size of the servers for all system service providers is $\tilde{u}_p$. The total size $U_t = \tilde{u}_0 + \tilde{u}_p$ (< 1) of these servers is the processor utilization that is no longer available to real-time applications.

Scheduling NonReal-Time Applications. Again, the server scheduler of S_0 schedules all the nonreal-time threads according to a time-sharing scheduling algorithm. We let x denote

Initialization

- Create a total bandwidth server S_0 with size $u_0 < 1$ for nonreal-time applications and a passive server for each service provider in the system.
- Set U_t to the total size of S_0 and all passive servers.

Acceptance test and admission of a new real-time application $\mathbf{T}_k$:

- Upon request from the application, do the acceptance test as described in Figure 12–11(c).
- If $\mathbf{T}_k$ passes the acceptance test, create server S_k of size u_k, following the rules described in Figure 12–11(c).

Maintenance of each server S_k: Replenish the budget and set the deadline of server S_k according to Figure 12–11(b).

Interaction with server scheduler of each server S_k:

- When a thread in the application $\mathbf{T}_k$ becomes ready, invoke the server scheduler of S_k to insert the thread in S_k's ready queue.
- If $\mathbf{T}_k$ uses a preemptive scheduling algorithm, before replenishing the budget of S_k, invoke the server scheduler of S_k to update the next event time t_k of $\mathbf{T}_k$.
- When a thread in $\mathbf{T}_k$ enters its nonpreemptable section, mark the server S_k nonpreemptable until the thread exits the nonpreemptable section, and mark the server preemptable when the nonpreemptable section exceeds the declared maximum length.
- When a thread in $\mathbf{T}_k$ requests a global resource, grant the thread the resource and mark the server S_k nonpreemptable until the thread releases the global resource.

Scheduling of all servers: Schedule all ready servers on the EDF basis, except when a server is marked nonpreemptable; when a server is nonpreemptable, it has the highest priority among all servers.

Termination of a real-time application $\mathbf{T}_k$:

- Destroy server S_k.
- Decrease U_t by u_k.

(a) Operation of the OS scheduler

FIGURE 12–11 Acceptance test, admission, and server maintenance. (a) Operations of OS scheduler. (b) Server maintenance rules. (c) Acceptance test and choice of server size.

the length of the time slices used by the time-sharing scheduler. The OS scheduler sets the budget of S_0 to x when the first nonreal-time thread is ready to execute (say at time 0) and sets the deadline of the server at $x/\tilde{u}_0$. Hereafter, at each time t when the budget of S_0 is exhausted and there are threads ready to be executed by S_0, the OS scheduler sets the budget to x and the corresponding deadline to either $t + x/\tilde{u}_0$ or the current deadline of the server plus $x/\tilde{u}_0$ whichever is later. Moreover, whenever a busy interval ends, the OS server gives the server x units of budget and sets the server deadline at the current time plus $x/\tilde{u}_0$.

As long as the server S_0 is schedulable (meaning that its budget is always consumed before the current deadline), the open system offers nonreal-time applications a virtual slower processor with speed $\tilde{u}_0$. As you will see later, the acceptance test of the open system is such that S_0 is always schedulable. Hereafter, we will keep in mind the existence of server S_0 and

processor utilization $\tilde{u}_0$ committed to the server. Other than this, we will not be concerned with nonreal-time applications any further.

Passive Server. The server created by the OS scheduler for each service provider is a passive server. A *passive server* is a sporadic server. Its execution budget comes from two sources. The first is the server's processor time allocation; the server has a very small size. (This is analogous to processor reservation of a system service provider.) The OS scheduler treats the passive server as a total bandwidth server of its size and replenishes its budget according to the rules described in Section 7.4.3. The only "aperiodic task" executed by the server using this budget is the daemon thread of the service provider. The daemon thread is created during initialization (or when a service provider is admitted into the system). It is suspended immediately after it is created and wakes to run whenever the service provider receives a request for service. When the daemon thread runs, it creates a work thread to service the request and puts itself back to the suspended state again.

In addition, a passive server also gets execution budget from client applications of the SSP executed by the server. When a client application requests a service, it passes to the service provider its own budget and the associated deadline for consuming the budget.[22] The work thread created by the daemon to service the client's request executes with this budget and has this deadline. (Conceptually, we can think of the budget/deadline passed to the passive server as a CPU reservation. This reservation is not replenished periodically, however; rather it is under the control of the server scheduler of the client application.) We will return to provide more details on how the scheduler of a passive server works.

Admission, Termination, and Scheduler Interactions. To request admission, that is, to execute in the real-time mode, a new real-time application $\mathbf{T}_k$ provides in its request to the OS scheduler the information needed by the scheduler to do an acceptance test. If the application passes the test, the OS scheduler creates a server S_k of size $\tilde{u}_k$ to execute the new application. (The information the OS scheduler needs to do an acceptance test and the rules used by the scheduler to do the test and select the server type and size are described below.) Whenever a real-time application terminates, the OS scheduler destroys its server.

When a thread in a real-time application $\mathbf{T}_k$ becomes ready, the OS scheduler invokes the server scheduler of the server S_k to insert the thread into the server's ready queue. The server scheduler orders the ready threads according to the scheduling algorithm Σ_k of the real-time application $\mathbf{T}_k$.

12.5.3 Server Maintenance

Again, the OS scheduler maintains (i.e., replenishes its budget and sets its deadline) the server S_k for each real-time application $\mathbf{T}_k$ with the help of the scheduler of S_k. Figure 12–11(b) summarizes the rules the OS scheduler uses. In the figure, $\tilde{u}$ denotes the size of the server being maintained. For simplicity, we take as the time origin the time instant immediately after the server is created; this is the instant when the application executed by the server begins to execute in real-time mode.

[22] In the prototype in [DLZS], the budget and deadline are passed by the server scheduler of the client application to the scheduler of the passive server. In this way, the number of context switches is kept small.

Inputs: Server size= $\tilde{u}$, scheduling quantum= q, start time of the application= 0.

- If the application is cyclically scheduled with frame size f,
  - server type: Constant utilization
  - replenishment rule: At time 0, f, $2f$, ..., set budget to $\tilde{u}f$ and deadline to f, $2f$, $3f$, ..., respectively.
- If the application is nonpreemptively scheduled:
  - server type: Constant utilization.
  - replenishment rule: The constant utilization server algorithm.
- If the application is unpredictable,
  - server type: Total bandwidth server-like.
  - replenishment rule:
    - replenishment time t:

      (1) when the ready queue is empty and a thread is released, or

      (2) when the ready queue is nonempty,

      (i) t is the current server deadline d, or

      (ii) the budget is exhausted at t and the next event time is not the release time of a thread with priority higher than the thread at the head of the server ready queue.

    - budget replenished: $\tilde{u}(t_n + q - max(t, d))$, where t_n is a lower bound to the next event time and q is the scheduling quantum of the open system.
    - server deadline: $t_n + q$
- If the application is preemptively scheduled but predictable,
  - server type: Total bandwidth server
  - replenishment rule: Same as the rule for unpredictable applications except q is equal to 0 and t_n is the next event time.

(b) Server maintenance rules

FIGURE 12–11 (*continued*)

Predictable Applications. You recall that there are three types of predictable applications: cyclically scheduled applications, nonpreemptively scheduled applications, and priority-driven, preemptively scheduled applications that have known next event times. The server used to execute a cyclically scheduled application is a constant utilization server. For the purpose of server maintenance, the workload presented to the server appears to be a single thread. The thread is ready for execution at the beginning of each frame (i.e., at times 0, f, $2f$, and so on). The execution time of this thread is $\tilde{u}f$. Hence, the OS scheduler replenishes the server budget at the beginning of the frames. Each time, the OS scheduler gives the server $\tilde{u}f$ units of budget and sets the deadline of the server accordingly. When the application becomes idle, the OS reclaims the remaining budget. (This emulates the case where the virtual slow processor idles during parts of some frames.)

Similarly, each nonpreemptively scheduled real-time application is executed by a constant utilization server. The OS treats the application as a single aperiodic task and replenishes

the budget of the server according to the rules given in Section 7.4.2. If the server is schedulable, it emulates a slow processor of speed $\tilde{u}$ in the following sense for both types of applications. If we compare the schedule of the application in the open system with its schedule on a slow processor of speed $\tilde{u}$, (1) scheduling decisions are made at the same time and (2) every thread completes at the same time or sooner in the open system.

Each preemptively scheduled application is executed by a server that is similar to a total bandwidth server.[23] The rules for maintaining the server of a predictable preemptive application are motived by the example on priority inversion due to budget over replenishment that was described in Section 7.9.2. Whenever the OS scheduler gets ready to replenish the server budget, say at time t, it queries the server scheduler for the next event time t'. It gives the server $\tilde{u}(t' - t)$ units of budget; the new deadline of the server is t'. This way, if a new thread with a higher priority is released at the next event time t', the OS scheduler will be able to give a new chunk of budget to the server for the execution of the new thread.

Unpredictable Applications. When periodic tasks in a preemptively scheduled application have release-time jitters or sporadic tasks, it is impossible for the server scheduler to compute an accurate estimate of the next event time. In the extreme when nothing is known about the release times of some threads, the OS scheduler simply maintains the server in the same way as the server S_0 for nonreal-time applications. In other words, it replenishes the server budget periodically, q units of time apart, giving the server $\tilde{u}q$ units of budget each time. We call this design parameter of the open system the *scheduling quantum*. We will discuss shortly the effect of the scheduling quantum on scheduling overhead and processor utilization.

When it is possible for the server scheduler to provide the OS scheduler with a lower bound $\hat{t}$ to the next event time, the OS scheduler sets the server deadline at $\hat{t} + q$ and sets the server budget to $\tilde{u}(\hat{t} + q - t)$ where t is the current time. Since the actual next event time t' (i.e., the next context switch) is never more than q time units earlier than the current server deadline, should a higher-priority thread be released at t', it is delayed by at most q units of time. In other words, the duration of a priority inversion due to overreplenishment of server budget is never more than q units of time.

12.5.4 Sufficient Schedulability Condition and Acceptance Test

Again, the OS scheduler subjects the application to an acceptance test whenever an application $\mathbf{T}_k$ requests to execute in the real-time mode. The required capacity of the application is s_k, and in the open system, it is scheduled by its server scheduler according to algorithm Σ_k. Deng, *et al.* [DLZS] showed that the following conditions together constitute a sufficient condition for the application $\mathbf{T}_k$ to be schedulable when executed in the open system according to algorithm Σ_k.

[23] Unlike a total bandwidth server, the server of a preemptively scheduled application is not always work conserving. When the next event is the release of a thread with a higher priority than the thread currently at the head of the server queue, the OS server waits until the current server deadline to replenish the server budget, even if the executing thread completes before the deadline. As a consequence, every thread may be blocked once by a nonpreemptable section in another application. In contrast, in a one-level priority-driven system, a thread can be blocked only if it is released while some lower-priority thread is in a nonpreemptable section.

1. The size $\tilde{u}_k$ of the server S_k used to execute $\mathbf{T}_k$ is equal to s_k, if $\mathbf{T}_k$ is predictable, or $\tilde{u}_k = s_k\delta_{k,\min}/(\delta_{k,\min} - q)$ if $\mathbf{T}_k$ is unpredictable, where $\delta_{k,\min}$ is the shortest relative deadline of all threads in $\mathbf{T}_k$ whose release times or resource acquisition times are unknown.
2. The total size of all servers in the open system is no more than $1 - \max_{\text{all } j}(B_j/D_{\min,j})$ where the max function is over all real-time applications $\mathbf{T}_j$'s in the system, B_j is the maximum execution time of nonpreemptable sections of all applications other than $\mathbf{T}_j$, and $D_{\min,j}$ is the shortest relative deadline of all threads in $\mathbf{T}_j$.
3. In the interval $(t, d - q)$ from any budget replenishment time t of S_k to q time units before the corresponding server deadline d, where $q \geq 0$ and $d - q > t$, there would be no context switch among the threads in $\mathbf{T}_k$ if $\mathbf{T}_k$ were to execute alone on a slow processor of speed s_k.

Figure 12–11(c) summarizes the acceptance test that the OS scheduler subjects each application to and the choice of server size that the scheduler makes when admitting a new real-time application. The rationale behind them are conditions 1 and 2 stated above: The OS scheduler makes sure that these conditions are met. Similarly, the replenishment rules in Figure 12–11(b) are such that condition 3 is satisfied for all types of real-time applications.

Information provided by requesting application T:

- required capacity s and scheduling algorithm Σ;
- maximum execution time B of all nonpreemptable sections;
- existence of aperiodic/sporadic tasks, if any;
- the shortest relative deadline δ of all threads with release-time jitters;
- the shortest relative deadline $D_{\min}$ if the application is priority driven or the shortest length of time between consecutive timer events if the application is time driven.

Information maintained by the OS scheduler:

- the total size U_t of all servers in the system, and
- the above parameters provided by existing real-time application.

Step 1: Choose the size u of the server for the new application as follows:

- $u = s$ if $\mathbf{T}$ is predictable, or
- $u = s\delta/(\delta - q)$ if $\mathbf{T}$ is unpredictable.

Step 2: Acceptance test and admission:

- If $U_t + u + \max_{all\,j}(B_j/D_{\min,j}) > 1$, reject $\mathbf{T}$;
- Else, admit $\mathbf{T}$:
  - increase U_t by u,
  - create a server S of the chosen size for $\mathbf{T}$, and
  - set budget and deadline of S to 0.

(c) Acceptance test and choice of server size

FIGURE 12–11 (*continued*)

12.5.5 Scheduling Overhead and Processor Utilization

Deng *et al.* [DLZS] simulated systems with different workloads for the purpose of determining the scheduling overhead of the two-level priority-driven scheme. Specifically, they compare the numbers of context switches and queue insertion/deletion operations incurred in systems using the two-level scheme with the corresponding numbers in a closed system using a one-level priority-driven scheduler.

Their simulation results show that scheduling overhead of the open system depends critically on the scheduling quantum. This design parameter can be set to 0 when all real-time applications are predictable. In this case, the scheduling overhead of the open system is essentially the same as that of a closed system. (The number of context switches are the same. There are more queues to maintain but all the queues are shorter.) When some real-time applications are nonpredictable, we must choose a nonzero scheduling quantum. The scheduling overhead of the open system is significantly higher than that of the closed system only when the scheduling quantum is small.

We note that the smaller the minimum relative deadline, the smaller the scheduling quantum must be. Even when the minimum relative deadlines of all applications are large, we may still want to use a small scheduling quantum. The reason is that the size of the server for each nonpredictable application required to meet the schedulability condition 1 grows with the scheduling quantum. A larger server than the required capacity of the applications means a lower processor utilization for real-time applications. In short, in order to accommodate nonpredictable applications with large release-times jitters and small relative deadlines, some processor bandwidth (on the order of 30 percent) is not available for real-time applications. This bandwidth is either wasted as scheduling overhead, when the scheduling quantum is small, or set aside for the sake of compensating blocking time in nonpredictable applications, when the scheduling quantum is large. The latter is better because the extra processor bandwidth over the required capacity of the real-time application will be used by server S_0 for nonreal-time applications.

12.5.6 Service Provider Structure and Real-Time API Functions

In the open system, services such as network and file access are implemented as special-purpose user-level server applications. As stated earlier, the OS scheduler creates a passive server to execute each service provider. Each SSP has a daemon thread, which is created when the service provider is admitted into the system. Its responsibilities are (1) to create work threads in the address space of the service provider in response to requests for service from clients, and (2) to process incoming data to identify the receiving thread and to notify that thread. The daemon thread is suspended immediately after it is created. When the daemon thread wakes up to run, the passive server executes it using the server's own budget.

Scheduling of Work Threads When a client application $\mathbf{T}_k$ requests service from an SSP, it sends a budget and a deadline for consuming the budget along with its request to the service provider. The parameters of the work thread created to service the client include its budget and deadline, which are equal to the respective values given by the request. The scheduler of the passive server schedules its daemon thread and all the work threads on the EDF basis, and the OS scheduler schedules the passive server according to the deadline of the thread at the head of the server's ready queue.

The passive server also has a suspend queue. Whenever the budget of the passive server is exhausted but the executing thread remains incomplete, it is removed from the server's ready queue and inserted in the suspend queue. When the requesting application sends more budget, the scheduler of the passive server reactivates the work thread, setting its budget and deadline accordingly and moving it back to the ready queue. When the thread completes, any unused budget is returned to the requesting application.

API Functions for Invocation of System Services. A real-time client application requests service by sending the service provider one of two Real-Time Application Programming Interface (RTAPI) calls: *send_data( )* and *recv_data( )*. The former is called when a client application requests the service provider to accepts its data (e.g., to send a message or write a file.) When called at time t by an application $\mathbf{T}_k$, the function *send_data( )* first invokes the server scheduler of server S_k to transfer its budget and deadline to the passive server of the service provider. The call wakes up the daemon thread of the service provider. When this thread executes, it creates a work thread, as described in the previous paragraphs. In the meantime, the calling thread within the client application is blocked until the work thread completes.

A client application calls *recv_data( )* to request a service provider to process incoming data on its behalf. Similar to *send_data( )*, *recv_data( )* also causes a transfer of budget to the passive server of the system service provider. If the data to be received are not available at the time, the call effectively freezes the client application, since the server of the application has no budget to execute and the service provider cannot execute on behalf of the application as the data are not available. To circumvent this problem, the client application first calls *wait_data( )* to wait for the data. If the data are available, *wait_data( )* returns immediately. Otherwise, the thread calling *wait_data( )* is blocked. Unlike *send_data( )* and *recv_data( )*, *wait_data( )* does not transfer budget to service provider. Hence, when the calling thread is blocked, the server of the application can still execute other threads in the client application. When incoming data arrive, the daemon thread processes the data just to identify the receiver of the incoming data and, after the receiver is identified, wakes up the thread that called *wait_data( )* if the thread exists, and *wait_data( )* returns. The client application then calls *recv_data( )* to transfer to the service provider the budget needed by the SSP process and receive the incoming data.

In effect, a call of the *send_data( )* or *recv_data( )* function causes a transfer of budget from the server of the client application to the passive server of service provider. Because this transfer does not violate any of the schedulability conditions stated above, the client remains schedulable. The schedulability of other applications is not affected by the execution of the service provider.

Other Real-Time API Functions. In addition to the real-time API functions for communication between real-time applications and system service providers, the open system also needs to provide API functions for the creation, admission, and termination of real-time applications. As an example, the prototype system described in [DLZS] provides a function called *register_application( )*. A real-time application calls this function to specify for the operating system its required capacity, as well as its choice of real-time scheduling algorithm and resource-access protocol among the ones supported by the system.

The prototype supports periodic and aperiodic tasks. A real-time application calls a create periodic task function to create a periodic task of the specified phase, period, and relative

deadline, as well as informing the operating system of the resources the periodic task will require and the maximum length of time the task will hold these resources. Similarly, it calls a create aperiodic task function to create an aperiodic task that will execute in response to a specified list of events. After creating and initializing all its tasks, an application then calls *start_application( )* to request admission into the open system.

The prototype also provides API functions for use by SSPs in their interaction with client applications. Examples of these functions are *add_thread_budget( )*, which replenishes the budget of the specified thread in response to the *add_budget( )* request.

12.6 CAPABILITIES OF COMMERCIAL REAL-TIME OPERATING SYSTEMS

This section describes several real-time operating systems that run on common processors and have sizable user bases. They are LynxOS [Bunn], pSOSystem [Moto], QNX [QNX], VRTX [Ment] and VxWorks [Wind]. In many respects, these operating systems are similar. Below is a summary of their commonalities; with few exceptions, you can replace the noun "the operating system" in the summary by the name of any of them.

- *Conformance to Standards*: The operating system is compliant or partially compliant to the Real-Time POSIX API standard, so you have preemptive fixed-priority scheduling, standard synchronization primitives, and so on, but the operating system may support only threads or processes (i.e., not both) and may implement only a subset of the standard functions. The operating system also has its own set of API functions, which may differ significantly from the corresponding POSIX functions and, in the cases where they are similar, may have more features. Some of the operating systems also provide AT&T System V and BSD system call interfaces (e.g., LynxOS) and Willows Win32 Library (e.g., QNX). (You can find information on Willows Library from http://www.willows.com.)
- *Modularity and Scalability*: The kernel is small, and the operating system configurable. In particular, the operating system can be scaled down to fit with the application in ROM in small embedded systems. By adding optional components to the kernel, the operating system can be configured to provide I/O, file, and networking services.
- *Speed and Efficiency*: Most of these operating systems are microkernel systems. Unlike microkernel systems of old, which usually have higher run-time overhead than monolithic operating systems, these microkernel operating systems have low overhead. In some, sending a message to a system service provider incurs no context switch. Important timing figures such as context-switch time, interrupt latency, semaphore get/release latency, and so on, are typically small (i.e., one to a few microseconds).
- *System Calls*: Nonpreemptable portions of kernel functions necessary for mutual exclusion are highly optimized and made as short and deterministic as possible.
- *Split Interrupt Handling*: Similarly, the nonpreemptable portion of interrupt handling and the execution times of immediate interrupt handling routines are kept small. The bulk of the work done to handle each interrupt is scheduled and executed at an appropriate priority.

- *Scheduling*: All real-time operating systems offer at least 32 priority levels, which is the minimum number required to be Real-Time POSIX compliant; most offer 128 or 256 (or 255). They all offer the choice between FIFO or round-robin policies for scheduling equal-priority threads. They allow you to change thread priorities at run time, but none provides adequate support for EDF scheduling and good hook for user-level bandwidth-preserving servers and slack stealers.
- *Priority Inversion Control*: The operating system provides priority inheritance but may allow you to disable it to save the overhead of this mechanism if you choose to use a resource access-control scheme that does not need priority inheritance. The system may also provide ceiling-priority protocol.
- *Clock and Timer Resolution*: The operating system may provide a nominal timer resolution down to nanoseconds. However, even on the fastest processor today, the operating system takes about a microsecond to process a timer interrupt, so you should not expect to be able to release a thread periodically with a period down to microseconds or precision and granularity in time down to tens and hundreds of nanoseconds.
- *Memory Management*: The operating system may provide virtual-to-physical address mapping but does not do paging. The operating system may not offer memory protection. Even when it does, it provides the user with the choice among multiple levels of memory protection, ranging from no protection to private virtual memory.
- *Networking*: The operating system can be configured to support TCP/IP, streams, and so on.

The remainder of the section highlights some distinguishing features of the operating systems listed above, especially those features not mentioned in the above summary. The fact that we say that an operating system has a feature but do not mention the same feature in our discussion on another operating system does not imply that the other operating system does not have the same or similar features! The section refrains from giving you a table of features and performance data on the operating systems described here. Such information will most likely to be out-of-date by the time this book reaches you. The best source of up-to-date information on any operating system is the home page provided by the vendor. There you are also likely to find a good tutorial on basic operating system primitives in general.

12.6.1 LynxOS

LynxOS 3.0 moves from the monolithic architecture of earlier versions to a microkernel design. The microkernel is 28 kilobytes in size and provides the essential services in scheduling, interrupt dispatch, and synchronization. Other services are provided by kernel lightweight service modules, called Kernel Plug-Ins (KPIs). By adding KPIs to the microkernel, the system can be configured to support I/O and file systems, TCP/IP, streams, sockets, and so on, and functions as a multipurpose UNIX operating system as earlier versions do.

Unlike single-threaded system service providers in traditional microkernel systems, KPIs are multithreaded. Each KPI can create as many threads to execute its routines as needed. LynxOS says that there is no context switch when sending a message (e.g., RFS) to a KPI, and inter-KPI communication takes only a few instructions [Bunn].

LynxOS can be configured as a self-hosted system. In such a system, embedded applications are developed on the same system on which they are to be deployed and run.[24] This means that the operating system must also support tools such as compilers, debuggers, and performance profilers and allow them to run on the same platform as embedded applications. Protecting the operating system and critical applications from these possibly untrustworthy applications is essential. Moreover, their memory demands are large. For this reason, LynxOS not only provides memory protection through hardware Memory Management Units (MMUs) but also offers optional demand paging; we have already described this feature in Section 12.3.3.

In LynxOS, application threads (and processes) make I/O requests to the I/O system via system calls such as *open( ), read( ), write( ), select( )*, and *close( )* that resemble the corresponding UNIX system calls. Each I/O request is sent directly by the kernel to a device driver for the respective I/O device. LynxOS device drivers follow the split interrupt handling strategy. Each driver contains an interrupt handler and a kernel thread. The former carries out the first step of interrupt handling at an interrupt request priority. The latter is a thread that shares the same address space with the kernel but is separate from the kernel. If the interrupt handler does not complete the processing of an interrupt, it sets an asynchronous system trap to interrupt the kernel. When the kernel can respond to the (software) interrupt (i.e., when the kernel is in a preemptable state), it schedules an instance of the kernel thread at the priority of the thread which opened the interrupting device. When the kernel thread executes, it continues interrupt handling and reenables the interrupt when it completes. LynxOS calls this mechanism priority tracking. (LynxOS holds a patent for this scheme.) Section 12.1.1 gave more details.

12.6.2 pSOSystem

Again, pSOSystem is an object-oriented operating system. Like the other operating systems described in this section, it is modular. pSOS+ is a preemptive, multitasking kernel that runs on a single microprocessor, while pSOS+m is a distributed multiprocessor kernel. pSOS+m has the same API functions as pSOS+, as well as functions for interprocessor communication and synchronization. The most recent release offers a POSIX real-time extension-compliant layer. Additional optional components provide a TCP/IP protocol stack and target and host-based debugging tools.

The classes of pSOSystem objects include tasks, memory regions and partitions, message queues, and semaphores. Each object has a node of residence, which is the processor on which the system call that created the object was made. An object may be global or local. A global object (a task or a resource) can be accessed from any processor in the system, while a local object can be accessed only by tasks on its local processor. A remote node with respect to any object is a processor other than its node of residence. You may have notice that this is how the MPCP model views a static multiprocessor system. In Section 9.1.3, we called the node of residence of a task (object) its local processor and the node of a residence of a resource (e.g., a semaphore) its synchronization processor.

[24] The opposite is a cross-development environment, where the host environment is separated from the target embedded systems. Embedded applications are written and compiled in the host environment and then moved to the target system to run and to be debugged and tested.

The basic unit of execution is a task, which has its own virtual environment.[25] pSOS+ provides each task with the choice of either preemptive priority-driven or time-driven scheduling. The application developer can also choose to run user tasks in either user or supervisory mode. pSOSystem 2.5 offers, in addition to priority inheritance, priority-ceiling protocol.

In Section 12.1.2, we said that in most modern operating systems, the processor jumps to the kernel when an interrupt occurs. pSOSystem is an exception. Device drivers are outside the kernel and can be loaded and removed at run time. When an interrupt occurs, the processor jumps directly to the interrupt service routine pointed to by the vector table. The intention is not only to gain some speed, but also to give the application developer complete control over interrupt handling.

pSOSystem allocates to tasks memory regions. A memory region is a physically contiguous block of memory. Like all objects, it may be local (i.e., strictly in local memory) or global (i.e., in a systemwide accessible part of the memory). The operating system creates a memory region in response to a call from an application. A memory region may be divided into either fixed-size buffers or variable-size segments.

12.6.3 QNX/Neutrino

QNX is one of the few multiprocessor operating systems suitable for real-time applications that requires high-end, networked SMP machines with gigabytes of physical memory. Its microkernel provides essential thread and real-time services. Other operating system services are provided by optional components called resource managers. For example, in the recent version, called Neutrino, the microkernel supports only threads. A process manager is needed to support processes that are memory-protected from each other, and the process manager is optional. Because optional components can be excluded at run time, the operating system can be scaled down to a small size. (The QNX 4.x microkernel is 12 kilobytes in size.)

QNX is a message-passing operating system; messages are the basic means of interprocess communication among all threads. Section 12.3.1 already talked about its message-based priority inheritance feature. This is the basic mechanism for priority tracking: Messages are delivered in priority order and the service provider executes at the priority of the highest priority clients waiting for service.

QNX implements POSIX message queues outside the kernel but implements QNX's message passing within the kernel. QNX messages are sent and received over channels and connections. When a service provider thread wants to receive messages, it first creates a channel. The channel is named within the service provider by a small integer identifier. To request service from a service provider, a client thread first attaches to the service provider's channel. This attachment is called a connection to the channel. Within the client, this connection is mapped directly to a file descriptor. The client can then send its RFS messages over the connection, in other words, directly to the file descriptor.

---

[25] The most well-known application of pSoSystem is Iridium, the system of communication satellites. The version of pSOSystem used in Iridium takes 400 microseconds to context switch among tasks, while the application is required to run 125 thread repetitions in each 90-millisecond L-Band interval. A two-level scheme is used to keep the overhead of context switches among threads low. The operating system schedules a small number (two or three) of scheduler tasks. When a scheduler runs, it polls threads.

While POSIX messages are queued and the senders are not blocked, QNX's message send and receive functions are blocking. A thread that calls the QNX send function *MsgSendv( )* is blocked until the receiving thread executes a receive *MsgReceivev( )*, processes the message, and then executes a *MsgReply( )*. Similarly, a thread that executes a *MsgReceive( )* is blocked if there is no pending message waiting on the channel and becomes unblocked when another thread executes a *MsgSend( )*. The blocking and synchronous nature of message passing eliminates the need of data queueing and makes a separate call by the client to wait for response unnecessary. Each message is copied directly to the receiving thread's address space without intermediate buffering. Hence, the speed of message passing is limited only by the hardware speed. You recall that the thread used to process a message inherits the priority of the sender. Therefore, for the purpose of schedulability analysis, we can treat the client and the server as if they were a single thread with their combined execution time.

Neutrino also supports fixed-size, nonblocking messages. These messages are called pulses. QNX's own event notification mechanism, as well as some of the POSIX and real-time signals, use pulses. Like QNX messages, pulses are received over channels.

In addition to thread-level synchronization primitives, Neutrino provides atomic operations for adding and subtracting a value, setting and clearing a bit, and complementing a bit. Using these operations, we can minimize the need for disabling interrupts and preemptions for the purpose of ensuring atomicity.

12.6.4 VRTX

VRTX has two multitasking kernels: VRTXsa and VRTXmc. VRTXsa is designed for performance. It has a POSIX-compliant library, provides priority inheritance, and supports multiprocessing. Its system calls are deterministic and preemptable. In particular, VRTXsa 5.0 is compliant to POSIX real-time extensions. It is for medium and large real-time applications. VRTXmc, on the other hand, is optimized for power consumption and ROM and RAM sizes. It provides only basic functions and is intended for applications such as cellular phones and other small handheld devices. For such applications, the kernel typically requires only 4-8 kilobytes of ROM and 1 kilobyte of RAM.

Most noteworthy about VRTX is that it is the first commercial real-time operating system certified by the Federal Aviation Agency (FAA) for mission- and life-critical systems, such as avionics. (VRTX is compliant to the FAA RTCS/DO-178B level A standard. This is a standard for software on board aircraft, and level A is for software whose failure would cause or contribute to a catastrophic failure of the aircraft. The process of compliance certification involves 100 percent code coverage in testing.) VRTX is used for the avionics on board Boeing MD-11 aircraft.

Rather than providing users with a variety of optional components, VRTX provides hooks for extensibility. For example, the TCB of a new task created by *TCREATE( )* is extended to include application-specific information. An application can also add system calls.

Like the other operating systems described here, VRTX also has its own API functions. An example is mailboxes, which can be used to transfer one long word of data at a time (as opposed to queues for transferring multiple long words of data).

VRTX allocates memory in fixed-size blocks. For efficiency, free pool may be divided into noncontiguous partitions with different size blocks.

12.6.5 VxWorks

A man on the street may not know VxWorks by name but most likely knows one of its applications, as it was the operating system used on Mars Pathfinder, which NASA sent to Mars in 1997. You recall that after Pathfinder landed on Mars and started to respond to ground commands and take science and engineering data, its computer repeatedly reset itself. This frequent reset made worldwide news daily. Fortunately, JPL and VxWorks were able to pinpoint the problem. This was done with the aid of a trace generation and logging tool and extensive instrumentation of major services (such as pipe, message queues, and interrupt handling), which were used during debugging and testing and were left in the system. They fixed the problem on the ground and thus saved the day for Pathfinder. According to Glenn Reeves, the Mars Pathfinder flight software cognizant engineer [Reev],[26] the cause was the classical uncontrolled priority inversion problem. Although VxWorks provides priority inheritance, the mechanism was disabled. The prolonged blocking experienced by a high-priority task caused it to miss its deadline, and the computer was reset as a consequence. The fix was, of course, to enable priority inheritance.

Before moving on, we want make a couple of parenthetical remarks on lessons learned from this incident. First, we cannot rely on testing to determine whether tasks can complete in time and how often a task may be late. (The computer reset behavior was observed only once during the months of testing on the ground. It was deemed infrequent enough to not warrant concern before Pathfinder left the ground.) Second, it is good practice to leave the instrumentation code added for debugging and testing purposes in the deployed system. System and software engineering experts can give you numerous reasons for doing so, not just so that you can use it to debug the system after it is deployed.

Back to VxWorks, the Pathfinder incident highlights the fact that VxWorks allows us to disable major function such as memory protection and priority inheritance. You can specify what options are to be enabled and what options are to be disabled via global configuration parameters. In the case of Pathfinder, the global configuration variable is the one that specifies whether priority inheritance is enabled for mutual exclusion semaphores. Once this option is set, priority inheritance is enabled or disabled as specified for all mutual exclusion semaphores created afterwards.

---

[26] Just in case the Web page cited here is no longer available when you look for it, here is a brief description of the problem. Pathfinder used a single computer to control both the cruise and lander parts. The CPU connected via a VME bus with interfaces to the radio, the camera, and a MIL-STD-1553 bus. The 1553 bus in turn connected to the flight controller and monitoring devices, as well as an instrument, called ASI/MET, for meteorological science. The tasks of interest were the bus scheduler, the data distributor and the ASI/MET task.

The bus scheduler was the highest priority task in the system. It executed periodically to set up transactions on the 1553 bus. During each period, data acquisition and compression took place and the ASI/MET task executed. In the midst of each period, the data distributor was awakened to distribute data. Its deadline was the beginning of the next scheduling period when the scheduling task began to execute; the scheduling task reset the computer whenever it found the data distributor incomplete at the time.

The data distributor was the high-priority task involved in uncontrolled priority-inversion, and the ASI/MET was the low-priority task, while data acquisition and compression tasks has medium priorities. Specifically, the data distributor and ASI/MET shared a mutual exclusion semaphore, which is a binary semaphore that can have priority inheritance. However, the priority inheritance option was disabled at the time, so while the data distributor was blocked by ASI/MET, the medium priority tasks could execute, prolong the duration of priority inversion, and prevent the data distributor from completing on time. This was what happened in Pathfinder on Mars.

VxWorks is a monolithic system. It is not a UNIX system but provides most interface functions defined by POSIX real-time extensions. As we mentioned earlier, an operating system's own API function may have more features than the corresponding POSIX function. An example is the timeout feature of Wind semaphores, the operating system's own semaphores. A parameter of the take-semaphore *semTake( )* function is the amount time the calling task is willing to wait for the semaphore. The value of this parameter can be NO_WAIT, WAIT_FOREVER, or a positive timeout value. Indeed, many VxWorks API functions (e.g., message queues) have the timeout feature. Another feature of Wind semaphores is *deletion safety*, that is, the prevention of a task from being deleted while it is in a critical section. A task can prevent itself from being accidentally deleted while it is holding a semaphore by specifying SEM_DELETE_SAFE in its take-semaphore call.

A VxWorks feature that was mentioned earlier is preemption lock. The operating system also provides similar functions [i.e., *intLock( )* and *intUnlock( )*] which a task can call to enable and disable interrupts. These functions makes the user-level implementation of the NPCS protocol easy.

While VxWorks is not a multiprocessor operating system, it provides shared-memory multiprocessing support as an option. Specifically, VxMP provides three types of shared-memory objects: shared binary and counting semaphores, shared message queues, and shared-memory partitions. They can be used for synchronization and communication among tasks in static multiprocessor systems.

VxWorks virtual memory exemplifies the kind of memory management options typically found in modern real-time operating systems. In a system with an MMU, the basic memory management provides virtual-to-physical address mapping. It allows you add new mappings and make portions of the memory noncacheable. (An example where this capability is needed is when we add memory boards to enlarge the shared memory for interprocessor communication. We want the portions of memory that are accessed by multiple processors and DMA devices to be noncacheable, that is, disallow any CPU to copy these portions into its data cache. This way, you do not have to maintain cache consistency manually or turn off caching globally.) Full-level memory management, including protection of text segments and the exception vector table and interfaces to the MMU, is provided by VxVMI, which is an optional component. By default, all tasks use a common context at system initiation time. A task can create a private virtual memory for the sake of protection by calling VxVMI functions to create a new virtual memory and update the address translation table. Thus an application can control access to its data or code as needed.

Finally, while LynxOS is a self-host system, the Tornado environment for VxWorks is a cross-development environment. Tornado provides an extensive set of development and performance monitoring and analysis tools. These memory-demanding tools reside on a host machine, separate from embedded applications, and therefore do not create the necessity of memory protection and paging/swapping as in LynxOS. However, host resident tools cannot access symbol tables on the target machine. For the purpose of creating a system that can be dynamically configured at run time, you must use the target-resident shell for development and you can configure module loader and unloader, symbol tables, and some debugging tools into VxWorks.

12.7 PREDICTABILITY OF GENERAL-PURPOSE OPERATING SYSTEMS

Sometimes, we may choose to run real-time applications on a general-purpose operating system despite its shortcomings. Reasons for this choice include the timing requirements of our applications are not hard, the general-purpose operating system costs less, there is a large user base, and so on. This section assumes that the choice of using a general-purpose operating system has already been made. As application developers, we must be fully aware of the sources of unpredictability. By adopting an application software architecture that minimizes the use of problematic features of the operating system and a scheduling and resource access-control strategy that keeps the effect of priority inversion in check, we may achieve sufficient predictability despite the operating system.

Rather than talking about general-purpose operating systems in general, we focus here on the two most widely used ones, Windows NT [Solo, Hart] and Linux [CaDM]. Windows NT (NT for short) is the most popular, and there is a growing trend to use it for real-time computing. This section discusses in turn each of the shortcomings of NT from a real-time computing point of view and describes approaches to partially overcome the shortcomings. Even more than NT, Linux is unsuitable for real-time applications. However, unlike NT, which we must live with as it is, we are free to modify Linux. This section describes existing real-time extensions of Linux for hard and soft real-time applications.

12.7.1 Windows NT Operating System

Windows NT provides many features (e.g., threads, priority interrupt, and events) that are suitable for real-time applications. Its actual timer and clock resolutions are sufficiently fine for all but the most time-stringent applications. However, its large memory footprint rules out its use whenever size is a consideration. Even when its size is not a problem, its weak support for real-time scheduling and resource-access control and unpredictable interrupt handling and interprocess communication mechanisms can be problematic. It is essential for time-critical applications to avoid system services that can introduce unpredictable and prolonged blocking. By keeping processor utilization sufficiently low and providing priority-inversion control at the user level, these applications can get sufficient predictability out of NT.

Scheduling. The scheduling mechanisms provided by Windows NT are designed for good average performance of time-shared applications. We should not be surprised to find them less than ideal for real-time applications.

Limited Priority Levels. Windows NT provides only 32 thread priorities. The lowest 16 levels (i.e., 0–15) are for threads of time-shared applications. The operating system may boost the priority and adjust the time slice (i.e., scheduling quantum) of a thread running at one of these priority levels in order to speed up the progress of the thread. Levels 16–31 are real-time priorities. The operating system never adjusts the priorities of threads running at these levels.

The small number of system priority levels by itself is not a serious problem, even for applications that should ideally be scheduled at a large number of distinct priorities, if we

do not need to keep the processor utilization high. You recall that we can compensate for a small number of available priority levels by keeping the processor (sufficiently) lightly loaded. As an example, suppose that we assign priorities to threads according to the rate-monotonic algorithm and use the constant-ratio mapping scheme described in Section 6.8.4 to map the assigned priorities to system priority levels. Lehoczky *et al.* [LeSh] showed that if we keep the processor load from exceeding 1/2, we can map at least 4098 assigned thread priorities to the 16 available system priority levels.[27] (You can find the schedulable utilization of a rate-monotonic system as a function of mapping ratio in Section 6.8.4.)

Many kernel-mode system threads execute at real-time priority level 16. They should affect only real-time threads with priority 16. However, higher-priority real-time threads may delay system threads. This means that in the presence of real-time applications, memory managers, local and network file systems, and so on, may not work well.

Jobs, Job Scheduling Classes, and FIFO Policy.[28] Another problem with Windows NT 4.0 is that it does not support FIFO with equal priority. As we will see shortly, the lack of FIFO scheduling capability is more problematic than an insufficient number of priority levels. In this respect, NT 5.0 offers a significant improvement over NT 4.0, as it is possible to choose the FIFO option on NT 5.0.

To explain, we note that NT 5.0 introduces the concepts of jobs and scheduling classes. A job may contain multiple processes but each process can belong to only one job. In an SMP system, we can statically configure the system by setting a limit-affinity flag of each job and, thus, bind all the processes in the job to a processor. Other features of the new job objects include the ability to set limits of a job object that control all processes in the job. Examples are user-mode execution time and memory usage limits on a per job or per process basis. A process (or a job) is terminated when the system finds that the process has accumulated more user-mode execution time than its previously set limit. A job containing periodic threads can periodically reset its own execution time limits and thus turns its execution-time usage limit into an execution rate limit. NT 5.0 simply terminates an offending job (or process) when it exceeds any of its usage limits, rather than putting job in the background, so the per job execution time and memory limit feature cannot be used as it is to implement the CPU reservation scheme described in Section 12.4.

Like NT 4.0, NT 5.0 also offers different priority classes. Time-critical processes should be in the real-time priority class. As stated earlier, on NT 5.0, all 16 real-time priority levels are available to a user thread in the real-time priority class. In addition to priority classes, NT 5.0 offers nine job scheduling classes; class 9 is for real-time applications. Whether all processes in a job belong to the same job scheduling class depends on whether the scheduling class limit is enabled or disabled. When this flag to set to enabled, all processes in the job use the same scheduling class.

---

[27] A small complication is that in Windows NT 4.0, the API function *SetThreadPriority( )* allows a user thread to specify only 7 out of the 16 real-time priority levels. However, a kernel thread can set the priority of any real-time thread to any of the 16 levels. Hence, we need to provide the priority mapping function in a device driver. We set the priority of a thread by calling this function instead of *SetThreadPriority( )*. This kluge will not be necessary in NT 5.0 (also called Windows 2000) since all 16 real-time priority levels are available to user-level real-time threads.

[28] Here, the term job is overloaded: What NT 5.0 calls a job is not what we have meant by a job throughout the book.

A combination of parameters allow us to choose either the FIFO or round-robin policy (within equal priority) for a process, and hence all threads in it. The operating system schedules all processes according to the round-robin policy by default. However, a real-time process (i.e., a process of real-time priority class) in a job whose scheduling class has value 9 and scheduling class limit enabled is scheduled according to the FIFO policy. In other words, we choose the FIFO option for a process by putting the process in the real-time priority class, giving the scheduling class of the job containing the process a value 9, and enabling the scheduling class limits for the job.

Resource Access Control. NT does not support priority inheritance. There are two ways to control priority inversion in a uniprocessor system without the help of this mechanism. All of them are at the user level and do not completely control priority inversion.

User-Level NPCS. The simplest way to overcome the lack of priority inheritance is to use the NPCS protocol described in Section 8.3. You recall that according to this protocol, a thread executes nonpreemptively when it holds any resource (i.e., a lock, a mutex object, or a semaphore). In the case of reader/writer locks and mutexes, it is simple to implement an approximate version of this protocol as a library function. The function assumes that priority level 31 is reserved for exclusive use by threads in their nonpreemptable sections. When a thread calls the NPCS function to request a resource, the function stores the thread's current priority and sets the priority to 31 and then grants the resource to the thread. The function restores the thread's priority when the thread no longer holds any resource.

We say that this is an approximate implementation of the NPCS protocol because setting the thread priority to 31 is not the same as making the thread nonpreemptive. The user-level protocol cannot enforce the exclusive use of priority level 31 for the purpose of emulating nonpreemption. Threads that do not use this library may have priority 31. In NT 4.0, these threads are scheduled on the round-robin basis with the thread that is holding a lock. They can delay its progress. Moreover, this delay is theoretically unbounded.

In contrast, this user-level NPCS protocol is effective in a uniprocessor system running NT 5.0 when threads under the control of the protocol are scheduled on the FIFO basis. At the time when a thread's priority is raised to 31, no other thread with priority 31 is ready for execution. Since the thread is scheduled on the FIFO basis, after its priority is raised to 31, no other thread can preempt it until it releases all the resources it holds.

Ceiling Priority Protocol. An alternative is the Ceiling-Priority Protocol (CPP) described in Section 8.6. You recall that the CPP requires prior information on the resources required by each thread. Each resource has a priority ceiling, which was defined in Section 8.5 for single-unit resources and Section 8.9 for multiple-unit resources. According to the CPP, a thread holding any resource executes at the highest priority ceiling of all the resources it holds.

To implement this protocol on a uniprocessor running NT 4.0, we need to restrict real-time threads to have even priority levels (i.e., 16, 18, . . . , 30). (The reason for this restriction is again that NT 4.0 does not support FIFO among equal-priority policy.) If the highest priority

of all threads requiring a resource is $2k$, then the ceiling-priority of the resource is $2k + 1$. (The ceiling priority of a resource with multiple units can be defined analogously.) Like the NPCS protocol, the user-level CCP cannot prevent unbounded priority inversion if there are threads that have priorities higher than 16 and are not under its control

Again, we do not have this problem with NT 5.0. We can use all 16 real-time priorities. By scheduling all threads under the control of CCP according to the FIFO policy, the protocol keeps the duration of priority inversion among them bounded.

Deferred Procedure Calls. Interrupt handling is done in two steps in Windows NT. As in good real-time operating systems, each interrupt service routine executed in the immediate step is short. So interrupt latency in NT is comparable to the latencies of real-time operating systems.

Unpredictability arising from interrupt handling is introduced in the second step. NT device drivers are written such that an interrupt service routine queues a procedure, called a Deferred Procedure Call (DPC), to handle the second step of interrupt handling. DPCs are executed in FIFO order at a priority lower than all the hardware interrupt priorities but higher than the priority of the scheduler/dispatcher. Consequently, a higher-priority thread can be blocked by DPCs queued in response to interrupts caused by lower-priority threads. This blocking time can be significant since the execution times of some DPCs can be quite large (e.g., 1 millisecond or more). Worst yet, the blocking time is unbounded.

However, it is possible to do priority tracking within NT. We do so by using a kernel thread to execute the DPC function. In other words, rather than having the Internet Service Routine (ISR) part of a device driver queue a DPC, it wakes up a kernel mode thread to execute the DPC function. This is similar to how the second step of interrupt handling is done in LynxOS. Specifically, the initialization routine (i.e., DriverEntry routine), the DPC function, and ISR of a device driver should be as follows.

- The initialization routine creates a kernel mode thread and sets the priority of the thread at a level specified by the device driver. The thread blocks waiting to be signaled by the ISR and, when signaled, it will execute the function provided by the device driver.
- The function provided by the driver does the remaining part of interrupt handling when executed by the kernel thread.
- When the interrupt service routine runs, it wakes up the kernel thread after servicing the device.

A question that remains is, what priority should the device driver thread have? The correct priority of the thread may remain unknown until the thread has executed for a while. (For example, suppose that when an incoming message causes an interrupt, the network driver thread is to have the priority of the thread that will receive the message. The receiving thread is not identified until the message header is processed.) A way is to give the driver thread a high priority, say 30, initially. (We assume that priority 31 is used exclusively to emulate preemption lock.) When it is awaked and executes, it sets its own priority to the priority of the thread which caused the interrupt as soon as it identifies that thread. (This scheme was suggested by Gallmeister [Gall] as a way to emulate priority inheritance by message passing.)

Asynchronous Procedure Calls and LPC Mechanism. In Section 12.3.2 we described briefly NT events and Asynchronous Procedure Calls (APCs), which serve the same purpose as signals in UNIX systems. Events are synchronous. Like Real-Time POSIX signals, events are queued and hence will not be lost if not handled immediately. Unlike Real-Time POSIX signals, however, they are delivered in FIFO order and do not carry data. APCs complement events to provide asynchronous services to applications as well as the kernel. Each (kernel or user) thread has its own APC queue, which enqueues the APCs that will be executed by the thread. When called to do so, the kernel inserts an APC in the queue of the specified thread and requests a software interrupt at the APC priority level, which is lower than the priority of the dispatcher (i.e., the scheduler) but higher than the priorities of normal threads. When the thread runs, it executes the enqueued APC. Since a thread is at the APC priority level while it is executing an APC, the thread is nonpreemptable by other threads during this time and may block higher-priority threads. Therefore, it is important that the execution times of APCs be kept as small as possible.

Another source of unpredictability in Windows NT is Local Procedure Calls (LPCs). This is an example of incorrect prioritization. LPCs provide the interprocess communication mechanism by which environment subsystem Dynamic Link Libraries (DLLs) pass requests to subsystem service providers. It is also used by remote procedure calls between processes on the same machine, as well as by the WinLogin process and security reference monitor.

Specifically, the LPC mechanism provides three schemes to communicate data across address space boundaries. Short messages are sent over an LPC connection when the sending process or thread makes a LPC call which specifies the buffer containing the message. The message is copied into the kernel space and from there into the address space of the receiving process. The other schemes make use of shared memory sections and are for exchanges of long messages between the sender and receiver.

Since the LPC queue is FIFO ordered, priority inversion can occur when a thread sends a request over an LPC connection to a service provider. Furthermore, without priority tracking, a service provider thread which was created to service a request from a high-priority thread may execute at a nonreal-time or low real-time priority.

We can avoid this kind of priority inversion only by avoiding the use of the LPC mechanism. In NT 4.0, the Win32 API functions that use LPC to communicate with subsystem service providers are (1) console (text) window support, (2) process and thread creation and termination, (3) network drive letter mapping, and (4) creation of temporary files. This means that a time-critical application should not write to the console, create and delete threads, and create temporary files. A multimode application must create and initialize threads that may run in all modes at initialization time. This restriction not only further increases the memory demand of the system but also reduces the configurability of the system.

12.7.2 Real-Time Extensions of Linux Operating Systems

The adoption of Linux has increased steadily in recent years as the operating system becomes increasingly more stable, its performance improves, and more and more Linux applications become available. The remainder of this section describes two extensions that enable applications with firm or hard real-time requirements to run on Linux. As you will see shortly, these extensions fall far short of making Linux compliant to POSIX real-time extensions. More

seriously, applications written to run on these extensions are not portable to standard UNIX machines or other commercial real-time operating systems.

Important Features. Like NT, Linux also has many shortcomings when used for real-time applications. One of the most serious arises from the disabling of interrupts by Linux subsystems when they are in critical sections. While most device drivers disable interrupts for a few microseconds, the disk subsystem may disable interrupts for as long as a few hundred microseconds at a time. The predictability of the system is seriously damaged when clock interrupts can be blocked for such a long time. The solution to this problem is to rewrite all the offending drivers to make their nonpreemptable sections as short as possible, as they are in real-time operating systems. Neither extension described below attacked this problem head on; rather, one tries to live with it, while the other avoids it.

Scheduling. Linux provides individual processes with the choices among *SCHED_FIFO, SCHED_RR*, or *SCHED_OTHER* policies. Processes using the *SCHED_FIFO* and *SCHED_RR* policies are scheduled on a fixed-priority basis; these are real-time processes. Processes using *SCHED_OTHER* are scheduled on the time-sharing basis; they execute at priorities lower than real-time processes.

There are 100 priority levels altogether. You can determine the maximum and minimum priorities associated with a scheduling policy using the primitives *sched_get_priority_min( )* and *sched_get_priority_max( ).* You can also find the size of the time slices given to processes scheduled according to the round-robin policy using *sched_rr_get_interval( )*. Since you have the source, you can also change these parameters.

Clock and Timer Resolutions. Like NT, Linux updates the system clock and checks for timer expirations periodically, and the period is 10 milliseconds on most hardware platforms. [In Linux, each clock interrupt period is called a *jiffy*, and time is expressed in terms of (the number of) *jiffies*.] Consequently, the actual resolution of Linux timers is 10 milliseconds.

To improve the clock resolution on Pentium processors, the kernel reads and stores the time stamp counter at each clock interrupt. In response to a *gettimeofday* call, it reads the counter again and calculates from the difference in the two readings the number of microseconds that have elapsed from the time of the previous timer interrupt to the current time. In this way, it returns the current time to the caller in terms of *jiffies* and the number of microseconds into the current *jiffy*.

In addition to reading the time stamp counter at each clock interrupt, the timer interrupt service routine checks the timer queue to determine whether any timer has expired and, for each expired timer found, queues the timer function that is to be executed upon the expiration of that timer. The timer functions thus queued are executed just before the kernel returns control to the application. Timer error can be large and unpredictable because of the delay thus introduced by the kernel and possibly large execution times of the timer functions.

Threads. Until recently, Linux did not offer a thread library. Rather, it offers only the low-level system call *clone( )*. Using this system call, one can create a process that shares the address space of its parent process, as well as the other parts of the parent's context (e.g., open file descriptors, message managers, and signal handlers) as specified by the call.

Recently, X. Leroy developed LinuxThreads (http://pauillac.inria˜xleroy/linuxthreads/). Each thread provided by this thread library is a UNIX process that is created using the *clone( )* system call. These threads are scheduled by the kernel scheduler just like UNIX processes.

LinuxThread provides most of the POSIX thread extension API functions and conforms to the standard except for signal handling.[29] In particular, LinuxThreads uses signals SIGUSR1 and SIGUSR2 for its own purpose. As a consequence, these signals are no longer available to applications. Since Linux does not support POSIX real-time extensions, there is no signal for application-defined use! Moreover, signals are not queued and may not be delivered in priority order.

The major advantage of the one-thread-per process model of LinuxThreads is that it simplifies the implementation of the thread library and increases its robustness. A disadvantage is that context switches on mutex and condition operations must go through the kernel. Fortunately, context switches in the Linux kernel are efficient.

UTIME High-Resolution Time Service. UTIME [HSPN] is a high-resolution time service designed to provide microsecond clock and timer granularity in Linux. With UTIME, system calls that have time parameters, such as *select* and *poll*, can specify time down to microsecond granularity.

To provide microsecond resolution, UTIME makes use of both the hardware clock and the Pentium time stamp counter. Rather than having the clock device programmed to interrupt periodically, UTIME programs the clock to interrupt in one-shot mode. At any time, the next timer interrupt will occur at the earliest of all future timer expiration times. Since the kernel responds as soon as a timer expires, the actual timer resolution is only limited by the length of time the kernel takes to service a timer interrupt, which is a few microseconds with UTIME.

Since the clock device no longer interrupts periodically, UTIME uses the Pentium time stamp counter to maintain the software clock. When the system is booted, the clock is programmed to interrupt periodically. During initialization, UTIME reads and stores the time stamp counter periodically and calculates the length of a *jiffy* in term of the number of time stamp cycles in a *jiffy* and the number of time stamp cycles in a second. Having thus calibrated the time stamp counter with respect to the clock and obtained the numbers of cycles per *jiffy* and cycles per second, UTIME then reprograms the clock to run in one-shot mode.

Hereafter, whenever a timer interrupt occurs, the interrupt service routine first updates the software clock based on the time stamp counter readings obtained at the current time and at the previous timer interrupt. UTIME provides time in terms of *jiffies*, as well as *jiffies_u*, which give the number of microseconds that have elapsed since the beginning of the current *jiffy*. It then queues the timer functions that are to be executed at the current timer interrupt, finds the next timer expiration time from the timer queue, and sets the clock to interrupt at that time. Because of the extra work to update the time and set the timer, the execution time of the

[29] According to POSIX, an asynchronous signal, for example, one that is sent via *kill( )* or a tty interface, is intended for the entire process, not only a thread within the process. If the thread to which the signal is sent is blocking the signal, the signal is nevertheless handled immediately as long as some thread within the same process is not blocking the signal. In other words, any thread within the process that is not blocking the signal can handle the signal. This feature is not implemented in LinuxThreads. (It is not straightforward to implement this feature within LinuxThread because each thread is a process and has its own process ID.) When such a signal is sent to a Linux thread which is blocking the signal, the signal is queued and is handled only when the thread unblocks the signal.

timer interrupt service routine in Linux with UTIME is several times larger than in standard Linux.

If a jiffy has elapsed since the last timer interrupt, the kernel executes the standard Linux heartbeat maintenance code, including decrementing the remaining quantum of the running process. As in standard Linux, the execution of timer functions is postponed until the kernel completes all these chores and gets ready to return control to the application.

KURT, Kansas University Real-Time System. KURT [HSPN] makes use of UTIME and extends Linux for real-time applications. It is designed for firm real-time applications that can tolerate a few missed deadlines.

Real-Time Modes and Processes. KURT differentiates real-time processes from normal Linux processes. It has three operation modes: focused mode, normal mode, and mixed mode. Only real-time processes can run when the system is in focused mode. Applications with stringent real-time requirements should run in this mode.

All processes run as in standard Linux when the system is in normal mode. The system starts in this mode. In the mixed mode, nonreal-time processes run in the background of real-time processes.

Time-Driven Scheduling of Real-Time Processes. In the KURT model, a real-time application consists of periodic and nonperiodic processes. The invocation times (i.e., the release times) of all events (i.e., jobs or threads) in all these processes are known. Each process starts as a nonreal-time process. To run in one of the real-time modes (i.e., focused or mixed modes) as a real-time process, a process first registers itself with the system using a KURT system call for this purpose, while it declares its parameters and chooses a scheduling policy. (To return to be a nonreal-time process, it unregisters.) After all processes in the real-time application have registered, the system can then be switched to one of real-time modes.

Mode switch is done under the control of an executive process. There is at least one executive process. After all processes that are to run in a real-time mode have registered, the executive process computes a schedule of all events based on the invocation times of all events declared by real-time processes. Having thus prepared real-time processes to run, the executive process calls *switch_to_rt( )* to switch the system into one of the real-time modes: *SCHED_KURT_PROCS* for the focused mode and *SCHED_KURT_ALL* for the mixed mode. Once in a real-time mode, events in real-time processes are scheduled according to the precomputed schedule in the time-driven manner. (KURT says that they are explicitly scheduled.)

KURT Components. The KURT system consists of a core (i.e., a kernel) and real-time modules called RTMods. The core contains the KURT time-driven scheduler. It schedules all real-time events. Real-Time Modules are standard Linux kernel modules that run in the same address space as the Linux kernel. Each module provides functionality and is loadable. The only builtin real-time module is *Process RTMod*. This module provides the user processes with system calls for registering and unregistering KURT real-time processes, as well as a system call that suspends the calling process until the next time it is to be scheduled.

Source of Unpredictability. In Chapter 5, we assumed that the schedule table fits and resides in memory. In contrast, KURT allows large schedule files that fit only partially in

memory and have to be read into memory from time to time. Moreover, KURT makes no change in the disk device driver to minimize the length of time the driver may block timer interrupts and to correctly prioritize file system processing. Consequently, KURT does not provide sufficient predictability for hard real-time applications even when they run in the focus mode.

RT Linux. RT Linux [Bara] is designed to provide hard real-time performance in Linux. It assumes that the application system can be divided into two parts. The real-time part runs on the RT kernel, while the nonreal-time part runs on Linux. The parts communicate via FIFO buffers that are pinned down in memory in the kernel space. These FIFO buffers are called RT-FIFOs. They appear to Linux user processes as devices. Reads and writes to RT-FIFOs by real-time tasks are nonblocking and atomic.

RT Linux eliminates the problem of Linux kernel blocking clock interrupts by replacing hardware interrupts by software emulated interrupts. Rather than letting Linux interface interrupt control hardware directly, the RT kernel sits between the hardware and the Linux kernel. Thus, the RT kernel intersects and attends to all interrupts. If an interrupt is to cause a real-time task to run, the RT kernel preempts Linux if Linux is running at the time and lets the real-time task run. In this way, the RT kernel puts Linux kernel and user processes in the background of real-time tasks.

If an interrupt is intended for Linux, the RT kernel relays it to the Linux kernel. To emulate disabled interrupt for Linux, the RT kernel provides a flag. This flag is set when Linux enables interrupts and is reset when Linux disables interrupts. The RT kernel checks this flag and the interrupt mask whenever an interrupt occurs. It immediately relays the interrupt to the Linux kernel only if the flag is set. For as long as the flag is reset, RT kernel queues all pending interrupts to be handled by Linux and passes them to Linux when Linux enables interrupts again.

The real-time part of the application system is written as one or more loadable kernel modules. In essence, all real-time tasks run in the kernel space. Tasks in each module may have their scheduler; the current version provides a rate-monotonic scheduler and an EDF scheduler.

12.8 SUMMARY

This chapter gave an overview of several commercial and widely used operating systems. It also described operating system services that should be provided but are not by these systems.

12.8.1 Commercial Operating Systems

The real-time operating systems described in Section 12.6 provide either all or most of the Real-Time POSIX API functions. (It is no wonder many students in real-time systems classes at Illinois said “they are practically the same” after hearing fellow students’ presentations on these systems.) The systems typically offer their own API functions. Some of them are better than the standard functions either in functionality or in performance or both. Nevertheless, you may not want to use system-specific functions for the sake of portability.

Best Features. Section 12.6 emphasized implementation features that are important for real-time applications. Examples of best practices discussed in the section include priority tracking in interrupt handling (Section 12.6.1), support for the MPCP model (Section 12.6.2), message-based priority inheritance (Section 12.6.3), hooks for extensions in the user level (Section 12.6.4), shared-memory multiprocessing support (Section 12.6.5), and modularity and ability to disable unneeded mechanisms.

Performance Information. The chapter gave you no help in selecting an operating system for your applications. In particular, it presented no performance data. The best source of performance information, as well as distinguishing features, is the vendor's own home page. In addition to context-switch time, operating system vendors typically provide data on interrupt latency and dispatch latency of their own systems. Interrupt latency is the time the system takes to start the execution of an immediate interrupt routine. (It was defined more precisely in Section 12.1.2.) Dispatch latency is the length of time between the completion of an immediate interrupt service routine to the start of the thread released as a result of the interrupt. Both delays are measured in the absence of higher-priority interrupts, as they should be. The published data usually include both the vendor's system and close competitors', so by looking at such data provided by several vendors, you can get a reasonably good sense of how biased the data are.

Many other performance figures are also important. They include the amounts of time the system takes to block and deliver a signal, to grant semaphores and mutexes, and so on, as well as performance of file and networking systems. Such data are usually not published. If you ask, the vendor may give you some, but without similar data on competing systems for comparison. For this reason, you may need to measure candidate systems yourself, and you can find what and how to measure in [Gall]. (A point made by Gallmeister is that determinism is important for hard real-time applications. We have stressed here that determinism is not necessary; predictability is. The algorithms described in earlier chapters allow us to predict the real-time performance of the application system as long as the operating system allows us to bound the time and resources it spends and to prioritize our jobs.)

General-Purpose Operating Systems. Section 12.7 discussed some of the reasons that Windows NT and Linux are not ideal for real-time applications. The most recent version of NT, NT 5.0 (Windows 2000), provides user threads with 16 real-time priority levels and the FIFO with equal-priority policy. It can deliver reasonably predictable performance under the conditions discussed in Section 12.6.1.

Like deferred procedure calls in NT, Linux interrupt handlers may prevent the scheduler from carrying out its activities in a timely fashion. KURT and RTLinux are Linux extensions designed to allow real-time applications to run with Linux applications. Section 12.7.2 described these extensions. The major drawback of the extensions is their nonstandard API.

12.8.2 Desirable Operating System Primitives

Several services and mechanisms can significantly simplify the implementation of user-level schedulers and resource managers. Section 12.2.2 described ones that are simple to implement within any operating system, have lower overhead, and do not introduce backward-compatibility problems. They are not provided by existing real-time operating systems and, hopefully, will be provided by future versions.

Dynamic Priority Scheduling. EDF scheduling was considered less desirable than fixed-priority schemes because its implementation was thought to be more complex. More importantly, it is unstable under overloads. However, the total bandwidth and weighted fair-queueing algorithms can effectively keep individual tasks from interfering each other. They provide one of the best approaches to scheduling soft real-time tasks with hard real-time tasks. An operating system can easily support EDF scheduling (and therefore these bandwidth-preserving server algorithms) without high overhead in the same framework as fixed-priority scheduling.

Busy Interval Tracking. None of the existing operating systems provides busy interval tracking service. Without the operating system's help, it is impossible for a user thread to determine the beginning and end of busy intervals of the system, or threads with priorities above a specified level, without undue complexity and overhead. In contrast, it is simple for the kernel to track of busy intervals and export the information on their endpoints. This information is needed by many algorithms for scheduling aperiodic tasks, synchronizing end-to-end tasks, and so on. Figure 12–7 shows a way for the kernel to provide this service.

Correct Interval Timers. The interval timers provided by many operating systems may underestimate the amount of time a thread has executed. If such a timer is used to monitor the budget (or slack) consumption of a bandwidth-preserving server (or a slack stealer), the underestimation may cause periodic tasks to miss deadlines.

Support for Server Threads. The mechanism for round-robin scheduling can be naturally extended to support user-level bandwidth-preserving servers or slack stealers. Also needed is an API function for setting the time slice (scheduling quantum) of a thread.

*12.8.3 Resource Reservation and Open Environment

Section 12.4 presented the resource reservation and scheduling mechanisms of Real-Time Mach and NT/RK resource kernel. While some experimental operating systems support resource reservation, usage monitoring, and enforcement, most commercial operating systems still do not.

Most operating systems (even those that support multiple scheduling policies) schedule all applications according to the same scheduling algorithm at any given time. Whether each application can meet its timing requirements is determined by a global schedulability analysis based on parameters of every task in the system. Section 12.5 presented an architecture of an open system in which each application can be scheduled in a way best suited for the application and the schedulability of the application determined independently of other applications that may run with it on the same hardware platform.

EXERCISES

12.1 The following pseudocode shows a way to detect and handle overruns of a timed loop. During each iteration, k functions, names *function_1* through *function_k* are executed. Each iteration is never supposed to take more than MAX_TIME units of time. At the start, a watchdog timer is set using *wdTimerSet( )*. The arguments include the ID of the timer and the delay to the expiration