Processes and Operating Systems

- The process abstraction.
- Switching contexts between programs.
- Real-time operating systems (RTOSs).
- Interprocess communication.
- Task-level performance analysis and power consumption.
- A telephone answering machine design.

INTRODUCTION

Although simple applications can be programmed on a microprocessor by writing a single piece of code, many applications are sophisticated enough that writing one large program does not suffice. When multiple operations must be performed at widely varying times, a single program can easily become too complex and unwieldy. The result is spaghetti code that is too difficult to verify for either performance or functionality.

This chapter studies the two fundamental abstractions that allow us to build complex applications on microprocessors: the process and the operating system (OS). Together, these two abstractions let us switch the state of the processor between multiple tasks. The process cleanly defines the state of an executing program, while the OS provides the mechanism for switching execution between the processes.

These two mechanisms together let us build applications with more complex functionality and much greater flexibility to satisfy timing requirements. The need to satisfy complex timing requirements—events happening at very different rates, intermittent events, and so on—causes us to use processes and OSs to build embedded software. Satisfying complex timing tasks can introduce extremely complex control into programs. Using processes to compartmentalize functions and encapsulating in the OS the control required to switch between processes make it much easier to satisfy timing requirements with relatively clean control within the processes.
We are particularly interested in real-time operating systems (RTOSs), which are OSs that provide facilities for satisfying real-time requirements. A RTOS allocates resources using algorithms that take real time into account. General-purpose OSs, in contrast, generally allocate resources using other criteria like fairness. Trying to allocate the CPU equally to all processes without regard to time can easily cause processes to miss their deadlines.

In the next section, we will introduce the concepts of task and process. Section 6.2 looks at how the RTOS implements processes. Section 6.3 develops algorithms for scheduling those processes to meet real-time requirements. Section 6.4 introduces some basic concepts in interprocess communication. Section 6.5 considers the performance of RTOSs while Section 6.6 looks at power consumption. Section 6.7 walks through the design of a telephone answering machine.

6.1 MULTIPLE TASKS AND MULTIPLE PROCESSES
Most embedded systems require functionality and timing that is too complex to embody in a single program. We break the system into multiple tasks in order to manage when things happen. In this section we will develop the basic abstractions that will be manipulated by the RTOS to build multirate systems.

6.1.1 Tasks and Processes
Many (if not most) embedded computing systems do more than one thing—that is, the environment can cause mode changes that in turn cause the embedded system to behave quite differently. For example, when designing a telephone answering machine, we can define recording a phone call and operating the user’s control panel as distinct tasks, because they perform logically distinct operations and they must be performed at very different rates. These different tasks are part of the system’s functionality, but that application-level organization of functionality is often reflected in the structure of the program as well.

A process is a single execution of a program. If we run the same program two different times, we have created two different processes. Each process has its own state that includes not only its registers but all of its memory. In some OSs, the memory management unit is used to keep each process in a separate address space. In others, particularly lightweight RTOSs, the processes run in the same address space. Processes that share the same address space are often called threads.

In this book, we will use the terms tasks and processes somewhat interchangeably, as do many people in the field. To be more precise, task can be composed of several processes or threads; it is also true that a task is primarily an implementation concept and process more of an implementation concept.
To understand why the separation of an application into tasks may be reflected in the program structure, consider how we would build a stand-alone compression unit based on the compression algorithm we implemented in Section 3.7. As shown in Figure 6.1, this device is connected to serial ports on both ends. The input to the box is an uncompressed stream of bytes. The box emits a compressed string of bits on the output serial line, based on a predefined compression table. Such a box may be used, for example, to compress data being sent to a modem.

The program’s need to receive and send data at different rates—for example, the program may emit 2 bits for the first byte and then 7 bits for the second byte—will obviously find itself reflected in the structure of the code. It is easy to create irregular, ungainly code to solve this problem; a more elegant solution is to create a queue of output bits, with those bits being removed from the queue and sent to the serial port in 8-bit sets. But beyond the need to create a clean data structure that simplifies the control structure of the code, we must also ensure that we process the inputs and outputs at the proper rates. For example, if we spend too much time in packaging and emitting output characters, we may drop an input character. Solving timing problems is a more challenging problem.

**FIGURE 6.1**
An on-the-fly compression box.
The text compression box provides a simple example of rate control problems. A control panel on a machine provides an example of a different type of rate control problem, the **asynchronous input**. The control panel of the compression box may, for example, include a compression mode button that disables or enables compression, so that the input text is passed through unchanged when compression is disabled. We certainly do not know when the user will push the compression mode button—the button may be depressed asynchronously relative to the arrival of characters for compression.

We do know, however, that the button will be depressed at a much lower rate than characters will be received, since it is not physically possible for a person to repeatedly depress a button at even slow serial line rates. Keeping up with the input and output data while checking on the button can introduce some very complex control code into the program. Sampling the button’s state too slowly can cause the machine to miss a button depression entirely, but sampling it too frequently and duplicating a data value can cause the machine to incorrectly compress data. One solution is to introduce a counter into the main compression loop, so that a subroutine to check the input button is called once every \( n \) times the compression loop is executed. But this solution does not work when either the compression loop or the button-handling routine has highly variable execution times—if the execution time of either varies significantly, it will cause the other to execute later than expected, possibly causing data to be lost. We need to be able to keep track of these two different tasks separately, applying different timing requirements to each. This is the sort of control that processes allow.

The above two examples illustrate how requirements on timing and execution rate can create major problems in programming. When code is written to satisfy several different timing requirements at once, the control structures necessary to get any sort of solution become very complex very quickly. Worse, such complex control is usually quite difficult to verify for either functional or timing properties.

### 6.1.2 Multirate Systems

Implementing code that satisfies timing requirements is even more complex when multiple rates of computation must be handled. **Multirate** embedded computing systems are very common, including automobile engines, printers, and cell phones. In all these systems, certain operations must be executed periodically, and each operation is executed at its own rate. Application Example 6.1 describes why automobile engines require multirate control.

---

**Application Example 6.1**

**Automotive engine control**

The simplest automotive engine controllers, such as the ignition controller for a basic motorcycle engine, perform only one task—timing the firing of the spark plug, which takes the place
of a mechanical distributor. The spark plug must be fired at a certain point in the combustion cycle, but to obtain better performance, the phase relationship between the piston’s movement and the spark should change as a function of engine speed. Using a microcontroller that senses the engine crankshaft position allows the spark timing to vary with engine speed. Firing the spark plug is a periodic process (but note that the period depends on the engine’s operating speed).

The control algorithm for a modern automobile engine is much more complex, making the need for microprocessors that much greater. Automobile engines must meet strict requirements (mandated by law in the United States) on both emissions and fuel economy. On the other hand, the engines must still satisfy customers not only in terms of performance but also in terms of ease of starting in extreme cold and heat, low maintenance, and so on.

Automobile engine controllers use additional sensors, including the gas pedal position and an oxygen sensor used to control emissions. They also use a multimode control scheme. For example, one mode may be used for engine warm-up, another for cruise, and yet another for climbing steep hills, and so forth. The larger number of sensors and modes increases the number of discrete tasks that must be performed. The highest-rate task is still firing the spark plugs. The throttle setting must be sampled and acted upon regularly, although not as frequently as the crankshaft setting and the spark plugs. The oxygen sensor responds much more slowly than the throttle, so adjustments to the fuel/air mixture suggested by the oxygen sensor can be computed at a much lower rate.

The engine controller takes a variety of inputs that determine the state of the engine. It then controls two basic engine parameters: the spark plug firings and the fuel/air mixture. The engine control is computed periodically, but the periods of the different inputs and outputs range over several orders of magnitude of time. An early paper on automotive electronics by Marley [Mar78] described the rates at which engine inputs and outputs must be handled.
### Variable Time to move full range (ms) Update period (ms)

<table>
<thead>
<tr>
<th>Variable</th>
<th>Time to move full range (ms)</th>
<th>Update period (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Engine spark timing</td>
<td>300</td>
<td>2</td>
</tr>
<tr>
<td>Throttle</td>
<td>40</td>
<td>2</td>
</tr>
<tr>
<td>Airflow</td>
<td>30</td>
<td>4</td>
</tr>
<tr>
<td>Battery voltage</td>
<td>80</td>
<td>4</td>
</tr>
<tr>
<td>Fuel flow</td>
<td>250</td>
<td>10</td>
</tr>
<tr>
<td>Recycled exhaust gas</td>
<td>500</td>
<td>25</td>
</tr>
<tr>
<td>Set of status switches</td>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td>Air temperature</td>
<td>[seconds]</td>
<td>500</td>
</tr>
<tr>
<td>Barometric pressure</td>
<td>[seconds]</td>
<td>1000</td>
</tr>
<tr>
<td>Spark/dwell</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>Fuel adjustments</td>
<td>80</td>
<td>4</td>
</tr>
<tr>
<td>Carburetor adjustments</td>
<td>500</td>
<td>25</td>
</tr>
<tr>
<td>Mode actuators</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

#### 6.1.3 Timing Requirements on Processes

Processes can have several different types of timing requirements imposed on them by the application. The timing requirements on a set of processes strongly influence the type of scheduling that is appropriate. A scheduling policy must define the timing requirements that it uses to determine whether a schedule is valid. Before studying scheduling proper, we outline the types of process timing requirements that are useful in embedded system design.

Figure 6.2 illustrates different ways in which we can define two important requirements on processes: **release time** and **deadline**. The release time is the time at which the process becomes ready to execute; this is not necessarily the time at which it actually takes control of the CPU and starts to run. An aperiodic process is by definition initiated by an event, such as external data arriving or data computed by another process. The release time is generally measured from that event, although the system may want to make the process ready at some interval after the event itself. For a periodically executed process, there are two common possibilities. In simpler systems, the process may become ready at the beginning of the period. More sophisticated systems, such as those with data dependencies between processes, may set the release time at the arrival time of certain data, at a time after the start of the period.

A deadline specifies when a computation must be finished. The deadline for an aperiodic process is generally measured from the release time, since that is the only reasonable time reference. The deadline for a periodic process may in general occur at some time other than the end of the period. As seen in Section 6.3.1, some scheduling policies make the simplifying assumption that the deadline occurs at the end of the period.
6.1 Multiple Tasks and Multiple Processes

Rate requirements are also fairly common. A rate requirement specifies how quickly processes must be initiated. The period of a process is the time between successive executions. For example, the period of a digital filter is defined by the time interval between successive input samples. The process’s rate is the inverse of its period. In a multirate system, each process executes at its own distinct rate. The most common case for periodic processes is for the initiation interval to be equal to the period. However, pipelined execution of processes allows the initiation interval to be less than the period. Figure 6.3 illustrates process execution in a system with four CPUs. The various execution instances of program P1 have been subscripted to distinguish their initiation times. In this case, the initiation interval is equal to one-fourth of the period. It is possible for a process to have an initiation rate less than the period even in single-CPU systems. If the process execution time is significantly less than the period, it may be possible to initiate multiple copies of a program at slightly offset times.
What happens when a process misses a deadline? The practical effects of a timing violation depend on the application—the results can be catastrophic in an automotive control system, whereas a missed deadline in a multimedia system may cause an audio or video glitch. The system can be designed to take a variety of actions when a deadline is missed. Safety-critical systems may try to take compensatory measures such as approximating data or switching into a special safety mode. Systems for which safety is not as important may take simple measures to avoid propagating bad data, such as inserting silence in a phone line, or may completely ignore the failure.

Even if the modules are functionally correct, their timing improper behavior can introduce major execution errors. Application Example 6.2 describes a timing problem in space shuttle software that caused the delay of the first launch of the shuttle.

**Application Example 6.2**

*A space shuttle software error*

Garman [Gar81] describes a software problem that delayed the first launch of the U.S. space shuttle. No one was hurt and the launch proceeded after the computers were reset. However, this bug was serious and unanticipated.

The shuttle’s primary control system was known as the Primary Avionics Software System (PASS). It used four computers to monitor events, with the four machines voting to ensure fault tolerance. Four computers allowed one machine to fail while still leaving three operating machines to vote, such that a majority vote would still be possible to determine operating procedures. If at least two machines failed, control was to be turned over to a fifth computer called the Backup Flight Control System (BFS). The BFS used the same computer, requirements, programming language, and compiler, but it was developed by a different organization than the one that built the PASS to ensure that methodological errors did not cause simultaneous failure of both systems. The switchover from PASS to BFS was controlled by the astronauts.
During normal operation, the BFS would listen to the operation of the PASS computers so that it could keep track of the state of the shuttle. However, BFS would stop listening when it thought that PASS was compromising data fetching. This would prevent PASS failures from inadvertently destroying the state of the BFS. PASS used an asynchronous, priority-driven software architecture. If high-priority processes take too much time, the OS can skip or delay lower-priority processing. The BFS, in contrast, used a time-slot system that allocated a fixed amount of time to each process. Since the BFS monitored the PASS, it could get confused by temporary overloads on the primary system. As a result, the PASS was changed late in the design cycle to make its behavior more amenable to the backup system.

On the morning of the launch attempt, the BFS failed to synchronize itself with the primary system. It saw the events on the PASS system as inconsistent and therefore stopped listening to PASS behavior. It turned out that all PASS and BFS processing had been running late relative to telemetry data. This occurred because the system incorrectly calculated its start time.

After much analysis of system traces and software, it was determined that a few minor changes to the software had caused the problem. First, about 2 years before the incident, a subroutine used to initialize the data bus was modified. Since this routine was run prior to calculating the start time, it introduced an additional, unnoticed delay into that computation. About a year later, a constant was changed in an attempt to fix that problem. As a result of these changes, there was a 1 in 67 probability for a timing problem. When this occurred, almost all computations on the computers would occur a cycle late, leading to the observed failure. The problems were difficult to detect in testing since they required running through all the initialization code; many tests start with a known configuration to save the time required to run the setup code. The changes to the programs were also not obviously related to the final changes in timing.

The order of execution of processes may be constrained when the processes pass data between each other. Figure 6.4 shows a set of processes with data dependencies among them. Before a process can become ready, all the processes on which it depends must complete and send their data to it. The data dependencies define a partial ordering on process execution—P1 and P2 can execute in any order (or in interleaved fashion) but must both complete before P3, and P3 must complete before P4. All processes must finish before the end of the period. The data dependencies must form a directed acyclic graph (DAG)—a cycle in the data dependencies is difficult to interpret in a periodically executed system.

A set of processes with data dependencies is known as a task graph. Although the terminology for elements of a task graph varies from author to author, we will consider a component of the task graph (a set of nodes connected by data dependencies) as a task and the complete graph as the task set. Figure 6.4 also shows a second task with two processes. The two tasks (\{P1, P2, P3, P4\} and \{P5, P6\}) have no timing relationships between them.

Communication among processes that run at different rates cannot be represented by data dependencies because there is no one-to-one relationship between data coming out of the source process and going into the destination process.
Nevertheless, communication among processes of different rates is very common. Figure 6.5 illustrates the communication required among three elements of an MPEG audio/video decoder. Data come into the decoder in the system format, which multiplexes audio and video data. The system decoder process demultiplexes the audio and video data and distributes it to the appropriate processes. Multirate communication is necessarily one way—for example, the system process writes data to the video process, but a separate communication mechanism must be provided for communication from the video process back to the system process.

6.1.4 CPU Metrics

We also need some terminology to describe how the process actually executes. The initiation time is the time at which a process actually starts executing on the CPU. The completion time is the time at which the process finishes its work.

The most basic measure of work is the amount of CPU time expended by a process. The CPU time of process $i$ is called $C_i$. Note that the CPU time is not equal to the completion time minus initiation time; several other processes may interrupt execution. The total CPU time consumed by a set of processes is
We need a basic measure of the efficiency with which we use the CPU. The simplest and most direct measure is *utilization*:

\[
U = \frac{\text{CPU time for useful work}}{\text{total available CPU time}}.
\]  

(6.2)

Utilization is the ratio of the CPU time that is being used for useful computations to the total available CPU time. This ratio ranges between 0 and 1, with 1 meaning that all of the available CPU time is being used for system purposes. The utilization is often expressed as a percentage. If we measure the total execution time of all processes over an interval of time \( t \), then the CPU utilization is

\[
U = \frac{T}{t}.
\]  

(6.3)

### 6.1.5 Process State and Scheduling

The first job of the OS is to determine that process runs next. The work of choosing the order of running processes is known as scheduling.

The OS considers a process to be in one of three basic *scheduling states*: *waiting*, *ready*, or *executing*. There is at most one process executing on the CPU at any time. (If there is no useful work to be done, an idling process may be used to perform a null operation.) Any process that could execute is in the ready state; the OS chooses among the ready processes to select the next executing process. A process may not, however, always be ready to run. For instance, a process may be waiting for data from an I/O device or another process, or it may be set to run from a timer that has not yet expired. Such processes are in the waiting state. Figure 6.6 shows the possible transitions between states available to a process. A process goes into the waiting state when it needs data that it has not yet received or when it has finished all its work for the current period. A process goes into the ready state when it receives its required data and when it enters a new period. A process can go into the executing state only when it has all its data, is ready to run, and the scheduler selects the process as the next process to run.

### 6.1.6 Some Scheduling Policies

A *scheduling policy* defines how processes are selected for promotion from the ready state to the running state. Every multitasking OS implements some type of scheduling policy. Choosing the right scheduling policy not only ensures that the system will meet all its timing requirements, but it also has a profound influence on the CPU horsepower required to implement the system’s functionality.
Schedulability means whether there exists a schedule of execution for the processes in a system that satisfies all their timing requirements. In general, we must construct a schedule to show schedulability, but in some cases we can eliminate some sets of processes as unschedulable using some very simple tests. Utilization is one of the key metrics in evaluating a scheduling policy. Our most basic requirement is that CPU utilization be no more than 100% since we can’t use the CPU more than 100% of the time.

When we evaluate the utilization of the CPU, we generally do so over a finite period that covers all possible combinations of process executions. For periodic processes, the length of time that must be considered is the hyperperiod, which is the least-common multiple of the periods of all the processes. (The complete schedule for the least-common multiple of the periods is sometimes called the unrolled schedule.) If we evaluate the hyperperiod, we are sure to have considered all possible combinations of the periodic processes. The next example evaluates the utilization of a simple set of processes.

Example 6.1

Utilization of a set of processes
We are given three processes, their execution times, and their periods:

<table>
<thead>
<tr>
<th>Process</th>
<th>Period</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>$1.0 \times 10^{-3}$</td>
<td>$1.0 \times 10^{-4}$</td>
</tr>
<tr>
<td>P2</td>
<td>$1.0 \times 10^{-3}$</td>
<td>$2.0 \times 10^{-4}$</td>
</tr>
<tr>
<td>P3</td>
<td>$5.0 \times 10^{-3}$</td>
<td>$3.0 \times 10^{-4}$</td>
</tr>
</tbody>
</table>

The least common multiple of these periods is $5 \times 10^{-3}$ s.
In order to calculate the utilization, we have to figure out how many times each process is executed in one hyperperiod: P1 and P2 are each executed five times while P3 is executed once.

We can now determine the utilization over the hyperperiod:

\[
U = \frac{5.1 \times 10^{-4} + 5.2 \times 10^{-4} + 1.3 \times 10^{-4}}{5 \times 10^{-3}} = 0.36
\]

This is well below our maximum utilization of 1.0.

We will see that some types of timing requirements for a set of processes imply that we cannot utilize 100% of the CPU’s execution time on useful work, even ignoring context switching overhead. However, some scheduling policies can deliver higher CPU utilizations than others, even for the same timing requirements. The best policy depends on the required timing characteristics of the processes being scheduled.

One very simple scheduling policy is known as cyclostatic scheduling or sometimes as Time Division Multiple Access scheduling. As illustrated in Figure 6.7, a cyclostatic schedule is divided into equal-sized time slots over an interval equal to the length of the hyperperiod \( H \). Processes always run in the same time slot. Two factors affect utilization: the number of time slots used and the fraction of each time slot that is used for useful work. Depending on the deadlines for some of the processes, we may need to leave some time slots empty. And since the time slots are of equal size, some short processes may have time left over in their time slot. We can use utilization as a schedulability measure: the total CPU time of all the processes must be less than the hyperperiod.

Another scheduling policy that is slightly more sophisticated is round robin. As illustrated in Figure 6.8, round robin uses the same hyperperiod as does cyclostatic. It also evaluates the processes in order. But unlike cyclostatic scheduling, if a process
does not have any useful work to do, the round-robin scheduler moves on to the next process in order to fill the time slot with useful work. In this example, all three processes execute during the first hyperperiod, but during the second one, \( P_1 \) has no useful work and is skipped. The processes are always evaluated in the same order. The last time slot in the hyperperiod is left empty; if we have occasional, non-periodic tasks without deadlines, we can execute them in these empty time slots. Round-robin scheduling is often used in hardware such as buses because it is very simple to implement but it provides some amount of flexibility.

In addition to utilization, we must also consider **scheduling overhead**—the execution time required to choose the next execution process, which is incurred in addition to any context switching overhead. In general, the more sophisticated the scheduling policy, the more CPU time it takes during system operation to implement it. Moreover, we generally achieve higher theoretical CPU utilization by applying more complex scheduling policies with higher overheads. The final decision on a scheduling policy must take into account both theoretical utilization and practical scheduling overhead.

### 6.1.7 Running Periodic Processes

We need to find a programming technique that allows us to run periodic processes, ideally at different rates. For the moment, let’s think of a process as a subroutine; we will call them \( p1() \), \( p2() \), etc. for simplicity. Our goal is to run these subroutines at rates determined by the system designer.

Here is a very simple program that runs our process subroutines repeatedly:

```c
while (TRUE) {
    p1();
    p2();
}
```

This program has several problems. First, it does not control the rate at which the processes execute—the loop runs as quickly as possible, starting a new iteration as soon as the previous iteration has finished. Second, all the processes run at the same rate.

Before worrying about multiple rates, let’s first make the processes run at a controlled rate. One could imagine controlling the execution rate by carefully designing the code—by determining the execution time of the instructions executed during an iteration, we could pad the loop with useless operations (NOPs) to make the execution time of an iteration equal to the desired period. Although some video games were designed this way in the 1970s, this technique should be avoided. Modern processors make it hard to accurately determine execution time, as we saw in Chapter 5. Conditionals anywhere in the program make it even harder to be sure that the loop consumes the same amount of execution time on every iteration. Furthermore, if any part of the program is changed, the entire timing scheme must be re-evaluated.
A timer is a much more reliable way to control execution of the loop. We would probably use the timer to generate periodic interrupts. Let’s assume for the moment that the `pall()` function is called by the timer’s interrupt handler. Then this code will execute each process once after a timer interrupt:

```c
void pall() {
    p1();
    p2();
}
```

But what happens when a process runs too long? The timer’s interrupt will cause the CPU’s interrupt system to mask its interrupts, so the interrupt will not occur until after the `pall()` routine returns. As a result, the next iteration will start late. This is a serious problem, but we will have to wait for further refinements before we can fix it.

Our next problem is to execute different processes at different rates. If we have several timers, we can set each timer to a different rate. We could then use a function to collect all the processes that run at that rate:

```c
void pA() {
    /* processes that run at rate A*/
    p1();
    p3();
}
void pB() {
    /* processes that run at rate B */
    p2();
    p4();
    p5();
}
```

This works, but it does require multiple timers, and we may not have enough timers to support all the rates required by a system.

An alternative is to use counters to divide the counter rate. If, for example, process `p2()` must run at 1/3 the rate of `p1()`, then we can use this code:

```c
static int p2count = 0; /* use this to remember count across timer interrupts */
void pall() {
    p1();
    if (p2count >= 2) { /* execute p2() and reset count */
        p2();
        p2count = 0;
    } else p2count++; /* just update count in this case */
}
```
This solution allows us to execute processes at rates that are simple multiples of each other. However, when the rates aren't related by a simple ratio, the counting process becomes more complex and more likely to contain bugs.

We have developed somewhat more reliable code, but this programming style is still limited in capability and prone to bugs. To improve both the capabilities and reliability of our systems, we need to invent the RTOS.

6.2 PREEMPTIVE REAL-TIME OPERATING SYSTEMS

A RTOS executes processes based upon timing constraints provided by the system designer. The most reliable way to meet timing constraints accurately is to build a preemptive OS and to use priorities to control what process runs at any given time. We will use these two concepts to build up a basic RTOS. We will use as our example OS FreeRTOS.org [Bar07]. This operating system runs on many different platforms.

6.2.1 Preemption

Preemption is an alternative to the C function call as a way to control execution. To be able to take full advantage of the timer, we must change our notion of a process as something more than a function call. We must, in fact, break the assumptions of our high-level programming language. We will create new routines that allow us to jump from one subroutine to another at any point in the program. That, together with the timer, will allow us to move between functions whenever necessary based upon the system's timing constraints.

We want to share the CPU across two processes. The kernel is the part of the OS that determines what process is running. The kernel is activated periodically by the timer. The length of the timer period is known as the time quantum because it is the smallest increment in which we can control CPU activity. The kernel determines what process will run next and causes that process to run. On the next timer interrupt, the kernel may pick the same process or another process to run.

Note that this use of the timer is very different from our use of the timer in the last section. Before, we used the timer to control loop iterations, with one loop
iteration including the execution of several complete processes. Here, the time quantum is in general smaller than the execution time of any of the processes.

How do we switch between processes before the process is done? We cannot rely on C-level mechanisms to do so. We can, however, use assembly language to switch between processes. The timer interrupt causes control to change from the currently executing process to the kernel; assembly language can be used to save and restore registers. We can similarly use assembly language to restore registers not from the process that was interrupted by the timer but to use registers from any process we want. The set of registers that define a process are known as its context and switching from one process’s register set to another is known as context switching. The data structure that holds the state of the process is known as the process control block.

### 6.2.2 Priorities

How does the kernel determine what process will run next? We want a mechanism that executes quickly so that we don’t spend all our time in the kernel and starve out the processes that do the useful work. If we assign each task a numerical priority, then the kernel can simply look at the processes and their priorities, see which ones actually want to execute (some may be waiting for data or for some event), and select the highest priority process that is ready to run. This mechanism is both flexible and fast. The priority is a non-negative integer value. The exact value of the priority is not as important as the relative priority of different processes. In this book, we will generally use priority 1 as the highest priority, but it is equally reasonable to use 1 or 0 as the lowest priority value (as FreeRTOS.org does).

Example 6.2 shows how priorities can be used to schedule processes.

### Example 6.2

**Priority-driven scheduling**

For this example, we will adopt the following simple rules:

- Each process has a fixed priority that does not vary during the course of execution. (More sophisticated scheduling schemes do, in fact, change the priorities of processes to control what happens next.)

- The ready process with the highest priority (with 1 as the highest priority of all) is selected for execution.
A process continues execution until it completes or it is preempted by a higher-priority process.

Let's define a simple system with three processes as seen below.

<table>
<thead>
<tr>
<th>Process</th>
<th>Priority</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>P2</td>
<td>2</td>
<td>30</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>20</td>
</tr>
</tbody>
</table>

In addition to describing the properties of the processes in general, we need to know the environmental setup. We assume that P2 is ready to run when the system is started, P1 is released at time 15, and P3 is released at time 18.

Once we know the process properties and the environment, we can use the priorities to determine which process is running throughout the complete execution of the system.

When the system begins execution, P2 is the only ready process, so it is selected for execution. At time 15, P1 becomes ready; it preempts P2 and begins execution since it has a higher priority. Since P1 is the highest-priority process in the system, it is guaranteed to execute until it finishes. P3’s data arrive at time 18, but it cannot preempt P1. Even when P1 finishes, P3 is not allowed to run. P2 is still ready and has higher priority than P3. Only after both P1 and P2 finish can P3 execute.

### 6.2.3 Processes and Context

The best way to understand processes and context is to dive into an RTOS implementation. We will use the FreeRTOS.org kernel as an example; in particular, we will use version 4.7.0 for the ARM7 AT91 platform. A process is known in FreeRTOS.org as a task. Task priorities in FreeRTOS.org are ranked opposite to the convention we use in the rest of the book: higher numbers denote higher priorities and the priority 0 task is the idle task.
To understand the basics of a context switch, let’s assume that the set of tasks is in steady state: Everything has been initialized, the OS is running, and we are ready for a timer interrupt. Figure 6.9 shows a sequence diagram for a context switch in freeRTOS.org. This diagram shows the application tasks, the hardware timer, and all the functions in the kernel that are involved in the context switch:

- `vPreemptiveTick()` is called when the timer ticks.
- `portSAVE_CONTEXT()` swaps out the current task context.
- `vTaskSwitchContext()` chooses a new task.
- `portRESTORE_CONTEXT()` swaps in the new context.

Here is the code for `vPreemptiveTick()` in the file `portISR.c`:

```c
void vPreemptiveTick( void )
{
    /* Save the context of the interrupted task. */
    portSAVE_CONTEXT();

    /* WARNING - Do not use local (stack) variables here. Use globals if you must! */
    static volatile unsigned portLONG ulDummy;

    /* Clear tick timer interrupt indication. */
    ulDummy = portTIMER_REG_BASE_PTR->TC_SR;
    /* Increment the RTOS tick count, then look for the highest priority task that is ready to run. */
    vTaskIncrementTick();
    vTaskSwitchContext();
```
/* Acknowledge the interrupt at AIC level... */
AT91C_BASE_AIC->AIC_EOICR = portCLEAR_AIC_INTERRUPT;

/* Restore the context of the new task. */
portRESTORE_CONTEXT();
}

vPreemptiveTick() has been declared as a naked function; this means that it
does not use the normal procedure entry and exit code that is generated by the
compiler. Because the function is *naked*, the registers for the process that was
interrupted are still available; vPreemptiveTick() doesn’t have to go to the proce-
dure call stack to get their values. This is particularly handy since the procedure
mechanism would save only part of the process state, making the state-saving code
a little more complex.

The first thing that this routine must do is save the context of the task that
was interrupted. To do this, it uses the routine portSAVE_CONTEXT(), which saves
all the context of the stack. It then performs some housekeeping, such as incre-
menting the tick count. The tick count is the internal timer that is used to determine
deadlines. After the tick is incremented, some tasks may have become ready as they
passed their deadlines.

Next, the OS determines which task to run next using the routine
vTaskSwitchContext(). After some more housekeeping, it uses port
RESTORE_CONTEXT() to restore the context of the task that was selected by
vTaskSwitchContext(). The action of portRESTORE_CONTEXT() causes control
to transfer to that task without using the standard C return mechanism.

The code for portSAVE_CONTEXT(), in the file portmacro.h, is defined as a
macro and not as a C function. It is structured in this way so that it doesn’t dis-
turb the register values that need to be saved. Because it is a macro, it has to be
written in a hard-to-read way—all code must be on the same line or end-of-line
continuations (back slashes) must be used. Here is the code in more readable form,
with the end-of-line continuations removed and the assembly language that is the
heart of this routine temporarily removed:

```c
#define portSAVE_CONTEXT()
{
  extern volatile void * volatile pxCurrentTCB;
  extern volatile unsigned portLONG ulCriticalNesting;

  /* Push R0 as we are going to use the register. */
  asm volatile( /* assembly language code here */ );
  ( void ) ulCriticalNesting;
  ( void ) pxCurrentTCB;
}
```

The asm statement allows assembly language code to be introduced in-line into
the C program. The keyword volatile tells the compiler that the assembly language
may change register values, which means that many compiler optimizations cannot be performed across the assembly language code. The code uses ulCriticalNesting and pxCurrentTCB simply to avoid compiler warnings about unused variables—the variables are actually used in the assembly code, but the compiler cannot see that.

The asm statement requires that the assembly language be entered as strings, one string per line, which makes the code hard to read. The fact that the code is included in a #define makes it even harder to read. Here is a cleaned-up version of the assembly language code from the asm volatile() statement:

```c
STMDB SP!, {R0}
/* Set R0 to point to the task stack pointer. */
STMDB SP, {SP}^  
NOP
SUB SP, SP, #4  
LDMIA SP!, {R0}
/* Push the return address onto the stack. */
STMDB R0!, {LR}
/* Now we have saved LR we can use it instead of R0. */
MOV LR, R0
/* Pop R0 so we can save it onto the system mode stack. */
LDMIA SP!, {R0}
/* Push all the system mode registers onto the task stack. */
STMDB LR, {R0-LR}^  
NOP
SUB LR, LR, #60 /*
Push the SPSR onto the task stack. */
MRS R0, SPSR
STMDB LR!, {R0}
LDR R0, =ulCriticalNesting
LDR R0, [R0]
STMDB LR!, {R0}
/*Store the new top of stack for the task. */
LDR R0, =pxCurrentTCB
LDR R0, [R0]
STR LR, [R0]
```

Here is the code for vTaskSwitchContext(), which is defined in the file tasks.c:

```c
void vTaskSwitchContext( void )
{
    if( uxSchedulerSuspended != ( unsigned portBASE_TYPE ) pdFALSE )
```
{ /* The scheduler is currently suspended - do not allow a context switch. */
xMissedYield = pdTRUE;
return;
}

/* Find the highest priority queue that contains ready tasks. */
while( listLIST_IS_EMPTY(&( pxReadyTasksLists[uxTopReadyPriority ])) )
{
    --uxTopReadyPriority;
}

/* listGET_OWNER_OF_NEXT_ENTRY walks through the list, so the tasks of the same priority get an equal share of the processor time. */
listGET_OWNER_OF_NEXT_ENTRY( pxCurrentTCB,
&((pxReadyTasksLists[uxTopReadyPriority ])));
vWriteTraceToBuffer();

This function is relatively straightforward—it walks down the list of tasks to identify the highest-priority task. This function is designed to deterministically choose the next task to run as long as the selected task is of equal or higher priority to the interrupted task; the list of tasks that is checked is determined by the variable uxTopReadyPriority. Each list contains the set of processes with the same priority; once the proper priority has selected by determining the value of uxTopReadyPriority, the system rotates through processes of equal priority by walking down their list.

The portRESTORE_CONTEXT() routine is also defined in portmacro.h and is implemented as a macro with embedded assembly language. Here is the macro with the line continuations and assembly language code removed:

#define portRESTORE_CONTEXT()
{
extern volatile void * volatile pxCurrentTCB;
extern volatile unsigned portLONG ulCriticalNesting;
    /* Set the LR to the task stack. */
    asm volatile (/* assembly language code here */);
Here is the assembly language code for portRESTORE_CONTEXT:

```assembly
    ( void ) ulCriticalNesting;
    ( void ) pxCurrentTCB;
}

LDR R0, =pxCurrentTCB
LDR R0, [R0]
LDR LR, [R0]
    /* The critical nesting depth is the first item on the
       stack. */
    /* Load it into the ulCriticalNesting variable. */
LDR R0, =ulCriticalNesting
LDMFD LR!, {R1}
STR R1, [R0]
    /* Get the SPSR from the stack. */
LDMFD LR!, {R0}
MSR SPSR, R0
    /* Restore all system mode registers for the task. */
LDMFD LR, {R0-R14}
NOP
    /* Restore the return address. */
LDR LR, [LR, #+60]
    /* And return - correcting the offset in the LR to obtain
       the */
    /* correct address. */
SUBS PC, LR, #4
```

### 6.2.4 Processes and Object-Oriented Design

We need to design systems with processes as components. In this section, we survey the ways we can describe processes in UML and how to use processes as components in object-oriented design.

UML often refers to processes as **active objects**, that is, objects that have independent threads of control. The class that defines an active object is known as an **active class**. Figure 6.10 shows an example of a UML active class. It has all the normal characteristics of a class, including a name, attributes, and operations. It also provides a set of signals that can be used to communicate with the process. A signal is an object that is passed between processes for asynchronous communication. We describe signals in more detail in Section 6.2.4.

We can mix active objects and normal objects when describing a system. Figure 6.11 shows a simple collaboration diagram in which an object is used as an interface between two processes: \( p1 \) uses the \( w \) object to manipulate its data before the data is sent to the **master** process.
6.3 PRIORITY-BASED SCHEDULING

Now that we have a priority-based context switching mechanism, we have to determine an algorithm by which to assign priorities to processes. After assigning priorities, the OS takes care of the rest by choosing the highest-priority ready process. There are two major ways to assign priorities: static priorities that do not change during execution and dynamic priorities that do change. We will look at examples of each in this section.

6.3.1 Rate-Monotonic Scheduling

Rate-monotonic scheduling (RMS), introduced by Liu and Layland [Liu73], was one of the first scheduling policies developed for real-time systems and is still very widely used. RMS is a static scheduling policy. It turns out that these fixed priorities are sufficient to efficiently schedule the processes in many situations.

The theory underlying RMS is known as rate-monotonic analysis (RMA). This theory, as summarized below, uses a relatively simple model of the system.

- All processes run periodically on a single CPU.
- Context switching time is ignored.
There are no data dependencies between processes.

The execution time for a process is constant.

All deadlines are at the ends of their periods.

The highest-priority ready process is always selected for execution.

The major result of RMA is that a relatively simple scheduling policy is optimal under certain conditions. Priorities are assigned by rank order of period, with the process with the shortest period being assigned the highest priority. This fixed-priority scheduling policy is the optimum assignment of static priorities to processes, in that it provides the highest CPU utilization while ensuring that all processes meet their deadlines.

Example 6.3 illustrates RMS.

### Example 6.3

**Rate-monotonic scheduling**

Here is a simple set of processes and their characteristics.

<table>
<thead>
<tr>
<th>Process</th>
<th>Execution time</th>
<th>Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>P2</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>12</td>
</tr>
</tbody>
</table>

Applying the principles of RMA, we give P1 the highest priority, P2 the middle priority, and P3 the lowest priority. To understand all the interactions between the periods, we need to construct a time line equal in length to hyperperiod, which is 12 in this case.

All three periods start at time zero. P1’s data arrive first. Since P1 is the highest-priority process, it can start to execute immediately. After one time unit, P1 finishes and goes out of the ready state until the start of its next period. At time 1, P2 starts executing as the
highest-priority ready process. At time 3, P2 finishes and P3 starts executing. P1’s next iteration starts at time 4, at which point it interrupts P3. P3 gets one more time unit of execution between the second iterations of P1 and P2, but P3 does not get to finish until after the third iteration of P1.

Consider the following different set of execution times for these processes, keeping the same deadlines.

<table>
<thead>
<tr>
<th>Process</th>
<th>Execution time</th>
<th>Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>12</td>
</tr>
</tbody>
</table>

In this case, we can show that there is no feasible assignment of priorities that guarantees scheduling. Even though each process alone has an execution time significantly less than its period, combinations of processes can require more than 100% of the available CPU cycles. For example, during one 12 time-unit interval, we must execute P1 three times, requiring 6 units of CPU time; P2 twice, costing 6 units of CPU time; and P3 one time, requiring 3 units of CPU time. The total of 6 + 6 + 3 = 15 units of CPU time is more than the 12 time units available, clearly exceeding the available CPU capacity.

Liu and Layland [Liu73] proved that the RMA priority assignment is optimal using critical-instant analysis. We define the response time of a process as the time at which the process finishes. The critical instant for a process is defined as the instant during execution at which the task has the largest response time. It is easy to prove that the critical instant for any process $P$, under the RMA model, occurs when it is ready and all higher-priority processes are also ready—if we change any higher-priority process to waiting, then $P$’s response time can only go down.

We can use critical-instant analysis to determine whether there is any feasible schedule for the system. In the case of the second set of execution times in Example 6.3, there was no feasible schedule. Critical-instant analysis also implies that priorities should be assigned in order of periods. Let the periods and computation times of two processes $P_1$ and $P_2$ be $\tau_1$, $\tau_2$ and $T_1$, $T_2$, with $\tau_1 < \tau_2$. We can generalize the result of Example 6.3 to show the total CPU requirements for the two processes in two cases. In the first case, let $P_1$ have the higher priority. In the worst case we then execute $P_2$ once during its period and as many iterations of $P_1$ as fit in the same interval. Since there are $\lceil \tau_2/\tau_1 \rceil$ iterations of $P_1$ during a single period of $P_2$, the required constraint on CPU time, ignoring context switching overhead, is

$$\left\lceil \frac{\tau_2}{\tau_1} \right\rceil T_1 + T_2 \leq \tau_2.$$ 

(6.4)
If, on the other hand, we give higher priority to \( P_2 \), then critical-instant analysis tells us that we must execute all of \( P_2 \) and all of \( P_1 \) in one of \( P_1 \)'s periods in the worst case:

\[
T_1 + T_2 \leq \tau_1. \tag{6.5}
\]

There are cases where the first relationship can be satisfied and the second cannot, but there are no cases where the second relationship can be satisfied and the first cannot. We can inductively show that the process with the shorter period should always be given higher priority for process sets of arbitrary size. It is also possible to prove that RMS always provides a feasible schedule if such a schedule exists.

The bad news is that, although RMS is the optimal static-priority schedule, it does not always allow the system to use 100% of the available CPU cycles. In the RMS framework, the total CPU utilization for a set of \( n \) tasks is

\[
U = \frac{\sum_{i=1}^{n} T_i}{\sum_{i=1}^{n} \tau_i}. \tag{6.6}
\]

The fraction \( T_i/\tau_i \) is the fraction of time that the CPU spends executing task \( i \). It is possible to show that for a set of two tasks under RMS scheduling, the CPU utilization \( U \) will be no greater than \( 2(2^{1/2} - 1) \approx 0.83 \). In other words, the CPU will be idle at least 17% of the time. This idle time is due to the fact that priorities are assigned statically; we see in the next section that more aggressive scheduling policies can improve CPU utilization. When there are \( m \) tasks with fixed priorities, the maximum processor utilization is

\[
U = m(2^{1/m} - 1). \tag{6.7}
\]

As \( m \) approaches infinity, the least upper bound to CPU utilization is \( \ln 2 = 0.69 \)—the CPU will be idle 31% of the time. This does not mean that we can never use 100% of the CPU. If the periods of the tasks are arranged properly, then we can schedule tasks to make use of 100% of the CPU. But the least upper bound of 69% tells us that RMS can in some cases deliver utilizations significantly below 100%.

The implementation of RMS is very simple. Figure 6.12 shows C code for an RMS scheduler run at the OS's timer interrupt. The code merely scans through the list of processes in priority order and selects the highest-priority ready process to run. Because the priorities are static, the processes can be sorted by priority in advance before the system starts executing. As a result, this scheduler has an asymptotic complexity of \( O(n) \), where \( n \) is the number of processes in the system. (This code assumes that processes are not created dynamically. If dynamic process creation is required, the array can be replaced by a linked list of processes, but the asymptotic complexity remains the same.) The RMS scheduler has both low asymptotic complexity and low actual execution time, which helps minimize the discrepancies between the zero-context-switch assumption of RMA and the actual execution of an RMS system.
/* processes[] is an array of process activation records,
   stored in order of priority, with processes[0] being
   the highest-priority process */
Activation_record processes[NPROCESSES];

void RMA(int current) { /* current = currently executing
   process */
   int i;
   /* turn off current process (may be turned back on) */
   processes[current].state = READY_STATE;
   /* find process to start executing */
   for (i = 0; i < NPROCESSES; i++)
     if (processes[i].state == READY_STATE) {
       /* make this the running process */
       processes[i].state = EXECUTING_STATE;
       break;
     }
}

FIGURE 6.12
C code for rate-monotonic scheduling.

6.3.2 Earliest-Deadline-First Scheduling

Earliest deadline first (EDF) is another well-known scheduling policy that was
also studied by Liu and Layland [Liu73]. It is a dynamic priority scheme—it changes
process priorities during execution based on initiation times. As a result, it can
achieve higher CPU utilizations than RMS.

The EDF policy is also very simple: It assigns priorities in order of deadline. The
highest-priority process is the one whose deadline is nearest in time, and the lowest-
priority process is the one whose deadline is farthest away. Clearly, priorities must
be recalculated at every completion of a process. However, the final step of the OS
during the scheduling procedure is the same as for RMS—the highest-priority ready
process is chosen for execution.

Example 6.4 illustrates EDF scheduling in practice.

Example 6.4

Earliest-deadline-first scheduling

Consider the following processes:

<table>
<thead>
<tr>
<th>Process</th>
<th>Execution time</th>
<th>Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

The hyperperiod is 60. In order to be able to see the entire period, we write it as a table:
<table>
<thead>
<tr>
<th>Time</th>
<th>Running process</th>
<th>Deadlines</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>P1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>P2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>P3</td>
<td>P1</td>
</tr>
<tr>
<td>3</td>
<td>P3</td>
<td>P2</td>
</tr>
<tr>
<td>4</td>
<td>P1</td>
<td>P3</td>
</tr>
<tr>
<td>5</td>
<td>P2</td>
<td>P1</td>
</tr>
<tr>
<td>6</td>
<td>P1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>P3</td>
<td>P2</td>
</tr>
<tr>
<td>8</td>
<td>P3</td>
<td>P1</td>
</tr>
<tr>
<td>9</td>
<td>P1</td>
<td>P3</td>
</tr>
<tr>
<td>10</td>
<td>P2</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>P3</td>
<td>P1, P2</td>
</tr>
<tr>
<td>12</td>
<td>P1</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>P3</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>P2</td>
<td>P1, P3</td>
</tr>
<tr>
<td>15</td>
<td>P1</td>
<td>P2</td>
</tr>
<tr>
<td>16</td>
<td>P2</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>P3</td>
<td>P1</td>
</tr>
<tr>
<td>18</td>
<td>P1</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>P3</td>
<td>P2, P3</td>
</tr>
<tr>
<td>20</td>
<td>P2</td>
<td>P1</td>
</tr>
<tr>
<td>21</td>
<td>P1</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>P3</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>P3</td>
<td>P1, P2</td>
</tr>
<tr>
<td>24</td>
<td>P1</td>
<td>P3</td>
</tr>
<tr>
<td>25</td>
<td>P2</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>P3</td>
<td>P1</td>
</tr>
<tr>
<td>27</td>
<td>P1</td>
<td>P2</td>
</tr>
<tr>
<td>28</td>
<td>P3</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>P2</td>
<td>P1, P3</td>
</tr>
<tr>
<td>30</td>
<td>idle</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>P1</td>
<td>P2</td>
</tr>
<tr>
<td>32</td>
<td>P3</td>
<td>P1</td>
</tr>
<tr>
<td>33</td>
<td>P3</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>P1</td>
<td>P3</td>
</tr>
<tr>
<td>35</td>
<td>P2</td>
<td>P1, P2</td>
</tr>
<tr>
<td>36</td>
<td>P1</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>P2</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>P3</td>
<td>P1</td>
</tr>
<tr>
<td>39</td>
<td>P3</td>
<td>P2, P3</td>
</tr>
<tr>
<td>40</td>
<td>P1</td>
<td></td>
</tr>
</tbody>
</table>

(Continued)
Liu and Layland showed that EDF can achieve 100% utilization. A feasible schedule exists if the CPU utilization (calculated in the same way as for RMA) is \( \leq 1 \). They also showed that when an EDF system is overloaded and misses a deadline, it will run at 100% capacity for a time before the deadline is missed.

The implementation of EDF is more complex than the RMS code. Figure 6.13 outlines one way to implement EDF. The major problem is keeping the processes sorted by time to deadline—since the times to deadlines for the processes change during execution, we cannot presort the processes into an array, as we could for RMS. To avoid resorting the entire set of records at every change, we can build a binary tree to keep the sorted records and incrementally update the sort. At the end of each period, we can move the record to its new place in the sorted list by deleting it from the tree and then adding it back to the tree using standard tree manipulation techniques. We must update process priorities by traversing them in sorted order, so the incremental sorting routines must also update the linked list pointers that let us traverse the records in deadline order. (The linked list lets us avoid traversing the tree to go from one node to another, which would require more time.) After putting in the effort to building the sorted list of records, selecting the next executing process is done in a manner similar to that of RMS. However, the dynamic sorting adds complexity to the entire scheduling process. Each update of the sorted list

<table>
<thead>
<tr>
<th>Time</th>
<th>Running process</th>
<th>Deadlines</th>
</tr>
</thead>
<tbody>
<tr>
<td>41</td>
<td>P2</td>
<td>P1</td>
</tr>
<tr>
<td>42</td>
<td>P1</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>P3</td>
<td>P2</td>
</tr>
<tr>
<td>44</td>
<td>P3</td>
<td>P1, P3</td>
</tr>
<tr>
<td>45</td>
<td>P1</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>P2</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>P3</td>
<td>P1, P2</td>
</tr>
<tr>
<td>48</td>
<td>P3</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>P1</td>
<td>P3</td>
</tr>
<tr>
<td>50</td>
<td>P2</td>
<td>P1</td>
</tr>
<tr>
<td>51</td>
<td>P1</td>
<td>P2</td>
</tr>
<tr>
<td>52</td>
<td>P3</td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>P3</td>
<td>P1</td>
</tr>
<tr>
<td>54</td>
<td>P2</td>
<td>P3</td>
</tr>
<tr>
<td>55</td>
<td>P1</td>
<td>P2</td>
</tr>
<tr>
<td>56</td>
<td>P2</td>
<td>P1</td>
</tr>
<tr>
<td>57</td>
<td>P1</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>P3</td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>P3</td>
<td>P1, P2, P3</td>
</tr>
</tbody>
</table>

There is one time slot left at \( t = 30 \), giving a CPU utilization of \( \frac{59}{60} \).
**Priority-Based Scheduling**

**Deadline_tree**

**Activation_record**

**Data structure**

/* linked list, sorted by deadline */
Activation_record *processes;
/* data structure for sorting processes */
Deadline_tree *deadlines;

void expired_deadline(Activation_record *expired){
    remove(expired); /* remove from the deadline-sorted list */
    add(expired,expired->deadline); /* add at new deadline */
}

void EDF(int current) { /* current = currently executing process */
    int i;
    /* turn off current process (may be turned back on) */
    processes->state = READY_STATE;
    /* find process to start executing */
    for (alink = processes; alink != NULL; alink = alink->next_deadline)
        if (processes->state == READY_STATE) {
            /* make this the running process */
            processes->state == EXECUTING_STATE;
            break;
        }
}

**FIGURE 6.13**

C code for earliest-deadline-first scheduling.

requires $O(\log n)$ steps. The EDF code is also significantly more complex than the RMS code.

### 6.3.3 RMS vs. EDF

Which scheduling policy is better: RMS or EDF? That depends on your criteria. EDF can extract higher utilization out of the CPU, but it may be difficult to diagnose the possibility of an imminent overload. Because the scheduler does take some overhead to make scheduling decisions, a factor that is ignored in the schedulability analysis of both EDF and RMS, running a scheduler at very high utilizations is somewhat problematic. RMS achieves lower CPU utilization but is easier to ensure that all deadlines
will be satisfied. In some applications, it may be acceptable for some processes to occasionally miss deadlines. For example, a set-top box for video decoding is not a safety-critical application, and the occasional display artifacts caused by missing deadlines may be acceptable in some markets.

What if your set of processes is unschedulable and you need to guarantee that they complete their deadlines? There are several possible ways to solve this problem:

- **Get a faster CPU.** That will reduce execution times without changing the periods, giving you lower utilization. This will require you to redesign the hardware, but this is often feasible because you are rarely using the fastest CPU available.

- **Redesign the processes to take less execution time.** This requires knowledge of the code and may or may not be possible.

- **Rewrite the specification to change the deadlines.** This is unlikely to be feasible, but may be in a few cases where some of the deadlines were initially made tighter than necessary.

### 6.3.4 A Closer Look at Our Modeling Assumptions

Our analyses of RMS and EDF have made some strong assumptions. These assumptions have made the analyses much more tractable, but the predictions of analysis may not hold up in practice. Since a misprediction may cause a system to miss a critical deadline, it is important to at least understand the consequences of these assumptions.

In all of the above discussions, we have assumed that each process is totally self-contained. However, that is not always the case—for instance, a process may need a system resource, such as an I/O device or the bus, to complete its work. Scheduling the processes without considering the resources those processes require can cause **priority inversion**, in which a low-priority process blocks execution of a higher-priority process by keeping hold of its resource. Example 6.5 illustrates priority inversion.

#### Example 6.5

**Priority inversion**

Consider a system with two processes: the higher-priority P1 and the lower-priority P2. Each uses the microprocessor bus to communicate to peripherals. When P2 executes, it requests the bus from the operating system and receives it. If P1 becomes ready while P2 is using the bus, the OS will preempt P2 for P1, leaving P2 with control of the bus. When P1 requests the bus, it will be denied the bus, since P2 already owns it. Unless P1 has a way to take the bus from P2, the two processes may deadlock.

The most common method for dealing with priority inversion is to promote the priority of any process when it requests a resource from the OS. The priority of the process temporarily becomes higher than that of any other process that may use
the resource. This ensures that the process will continue executing once it has the resource so that it can finish its work with the resource, return it to the OS, and allow other processes to use it. Once the process is finished with the resource, its priority is demoted to its normal value. Several methods have been developed to manage the priority swapping process [Liu00].

Rate-monotonic scheduling assumes that there are no data dependencies between processes. Example 6.6 shows that knowledge of data dependencies can help use the CPU more efficiently.

Example 6.6

Data dependencies and scheduling

Data dependencies imply that certain combinations of processes can never occur. Consider the simple example [Yen98] below.

We know that P1 and P2 cannot execute at the same time, since P1 must finish before P2 can begin. Furthermore, we also know that because P3 has a higher priority, it will not preempt both P1 and P2 in a single iteration. If P3 preempts P1, then P3 will complete before P2 begins; if P3 preempts P2, then it will not interfere with P1 in that iteration. Because we know that some combinations of processes cannot be ready at the same time, we know that our worst-case CPU requirements are less than would be required if all processes could be ready simultaneously.

6.4 INTERPROCESS COMMUNICATION MECHANISMS

Processes often need to communicate with each other. Interprocess communication mechanisms are provided by the operating system as part of the process abstraction.
In general, a process can send a communication in one of two ways: blocking or nonblocking. After sending a blocking communication, the process goes into the waiting state until it receives a response. Nonblocking communication allows the process to continue execution after sending the communication. Both types of communication are useful.

There are two major styles of interprocess communication: shared memory and message passing. The two are logically equivalent—given one, you can build an interface that implements the other. However, some programs may be easier to write using one rather than the other. In addition, the hardware platform may make one easier to implement or more efficient than the other.

6.4.1 Shared Memory Communication

Figure 6.14 illustrates how shared memory communication works in a bus-based system. Two components, such as a CPU and an I/O device, communicate through a shared memory location. The software on the CPU has been designed to know the address of the shared location; the shared location has also been loaded into the proper register of the I/O device. If, as in the figure, the CPU wants to send data to the device, it writes to the shared location. The I/O device then reads the data from that location. The read and write operations are standard and can be encapsulated in a procedural interface.

Example 6.7 describes the use of shared memory as a practical communication mechanism.

**Example 6.7**

*Elastic buffers as shared memory*

The text compressor of Application Example 3.4 provides a good example of a shared memory. As shown below, the text compressor uses the CPU to compress incoming text, which is then sent on a serial line by a UART.
The input data arrive at a constant rate and are easy to manage. But because the output data are consumed at a variable rate, these data require an elastic buffer. The CPU and output UART share a memory area—the CPU writes compressed characters into the buffer and the UART removes them as necessary to fill the serial line. Because the number of bits in the buffer changes constantly, the compression and transmission processes need additional size information. In this case, coordination is simple—the CPU writes at one end of the buffer and the UART reads at the other end. The only challenge is to make sure that the UART does not overrun the buffer.

As an application of shared memory, let us consider the situation of Figure 6.14 in which the CPU and the I/O device want to communicate through a shared memory block. There must be a flag that tells the CPU when the data from the I/O device is ready. The flag, an additional shared data location, has a value of 0 when the data are not ready and 1 when the data are ready. The CPU, for example, would write the data, and then set the flag location to 1. If the flag is used only by the CPU, then the flag can be implemented using a standard memory write operation. If the same flag is used for bidirectional signaling between the CPU and the I/O device, care must be taken. Consider the following scenario:

1. CPU reads the flag location and sees that it is 0.
2. I/O device reads the flag location and sees that it is 0.
3. CPU sets the flag location to 1 and writes data to the shared location.
4. I/O device erroneously sets the flag to 1 and overwrites the data left by the CPU.

The above scenario is caused by a critical timing race between the two programs. To avoid such problems, the microprocessor bus must support an atomic **test-and-set** operation, which is available on a number of microprocessors. The test-and-set operation first reads a location and then sets it to a specified value. It returns the result of the test. If the location was already set, then the additional set has no effect but the test-and-set instruction returns a false result. If the location was not set, the
instruction returns true and the location is in fact set. The bus supports this as an atomic operation that cannot be interrupted. Programming Example 6.1 describes a test-and-set operation in more detail.

A test-and-set can be used to implement a semaphore, which is a language-level synchronization construct. For the moment, let’s assume that the system provides one semaphore that is used to guard access to a block of protected memory. Any process that wants to access the memory must use the semaphore to ensure that no other process is actively using it. As shown below, the semaphore names by tradition are P() to gain access to the protected memory and V() to release it.

```c
/* some nonprotected operations here */
P(); /* wait for semaphore */
/* do protected work here */
V(); /* release semaphore */
```

The P() operation uses a test-and-set to repeatedly test a location that holds a lock on the memory block. The P() operation does not exit until the lock is available; once it is available, the test-and-set automatically sets the lock. Once past the P() operation, the process can work on the protected memory block. The V() operation resets the lock, allowing other processes access to the region by using the P() function.

---

**Programming Example 6.1**

**Test-and-set operation**

The SWP (swap) instruction is used in the ARM to implement atomic test-and-set:

```
SWP Rd,Rm,Rn
```

The SWP instruction takes three operands—the memory location pointed to by \( Rn \) is loaded and saved into \( Rd \), and the value of \( Rm \) is then written into the location pointed to by \( Rn \). When \( Rd \) and \( Rn \) are the same register, the instruction swaps the register’s value and the value stored at the address pointed to by \( Rd/Rn \). For example, consider this code sequence:

```
ADR r0, SEMAPHORE ; get semaphore address
LDR r1, #1
GETFLAG SWP r1,r1,[r0] ; test-and-set the flag
BNZ GETFLAG ; no flag yet, try again
HASFLAG ...
```

The program first loads the constant 1 into \( r1 \) and the address of the semaphore FLAG1 into register \( r2 \), then reads the semaphore into \( r0 \) and writes the 1 value into the semaphore. The code then tests whether the semaphore fetched from memory is zero; if it was, the semaphore was not busy and we can enter the critical region that begins with the HASFLAG label. If the flag was nonzero, we loop back to try to get the flag once again.
6.4.2 Message Passing

Message passing communication complements the shared memory model. As shown in Figure 6.15, each communicating entity has its own message send/receive unit. The message is not stored on the communications link, but rather at the senders/receivers at the end points. In contrast, shared memory communication can be seen as a memory block used as a communication device, in which all the data are stored in the communication link/memory.

Applications in which units operate relatively autonomously are natural candidates for message passing communication. For example, a home control system has one microcontroller per household device—lamp, thermostat, faucet, appliance, and so on. The devices must communicate relatively infrequently; furthermore, their physical separation is large enough that we would not naturally think of them as sharing a central pool of memory. Passing communication packets among the devices is a natural way to describe coordination between these devices. Message passing is the natural implementation of communication in many 8-bit microcontrollers that do not normally operate with external memory.

6.4.3 Signals

Another form of interprocess communication commonly used in Unix is the signal. A signal is simple because it does not pass data beyond the existence of the signal itself. A signal is analogous to an interrupt, but it is entirely a software creation. A signal is generated by a process and transmitted to another process by the operating system.

A UML signal is actually a generalization of the Unix signal. While a Unix signal carries no parameters other than a condition code, a UML signal is an object. As such, it can carry parameters as object attributes. Figure 6.16 shows the use of a signal in UML. The sigbehavior() behavior of the class is responsible for throwing the signal, as indicated by <<send>>. The signal object is indicated by the <<signal>> stereotype.
6.5 EVALUATING OPERATING SYSTEM PERFORMANCE

The scheduling policy does not tell us all that we would like to know about the performance of a real system running processes. Our analysis of scheduling policies makes some simplifying assumptions:

- We have assumed that context switches require zero time. Although it is often reasonable to neglect context switch time when it is much smaller than the process execution time, context switching can add significant delay in some cases.

- We have assumed that we know the execution time of the processes. In fact, we learned in Section 5.6 that program time is not a single number, but can be bounded by worst-case and best-case execution times.

- We probably determined worst-case or best-case times for the processes in isolation. But, in fact, they interact with each other in the cache. Cache conflicts among processes can drastically degrade process execution time.

The zero-time context switch assumption used in the analysis of RMS is not correct—we must execute instructions to save and restore context, and we must execute additional instructions to implement the scheduling policy. On the other hand, context switching can be implemented efficiently—context switching need not kill performance. The effects of nonzero context switching time must be carefully analyzed in the context of a particular implementation to be sure that the predictions of an ideal scheduling policy are sufficiently accurate.

Example 6.8 shows that context switching can, in fact, cause a system to miss a deadline.

Example 6.8

*Scheduling and context switching overhead*

Appearing below is a set of processes and their characteristics.
First, let us try to find a schedule assuming that context switching time is zero. Following is a feasible schedule for a sequence of data arrivals that meets all the deadlines:

<table>
<thead>
<tr>
<th>Process</th>
<th>Execution time</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td>10</td>
</tr>
</tbody>
</table>

Now let us assume that the total time to initiate a process, including context switching and scheduling policy evaluation, is one time unit. It is easy to see that there is no feasible schedule for the above release time sequence, since we require a total of $2T_{P1} + T_{P2} = 2 \times (1 + 3) + (1 + 3) = 11$ time units to execute one period of P2 and two periods of P1.

In Example 6.8, overhead was a large fraction of the process execution time and of the periods. In most real-time operating systems, a context switch requires only a few hundred instructions, with only slightly more overhead for a simple real-time scheduler like RMS. When the overhead time is very small relative to the task periods, then the zero-time context switch assumption is often a reasonable approximation. Problems are most likely to manifest themselves in the highest-rate processes, which are often the most critical in any case. Completely checking that all deadlines will be met with nonzero context switching time requires checking all possible schedules for processes and including the context switch time at each preemption or process initiation. However, assuming an average number of context switches per process and computing CPU utilization can provide at least an estimate of how close the system is to CPU capacity.

Another important assumption we have made thus far is that process execution time is constant. As seen in Section 5.6, this is definitely not the case—both data-dependent behavior and caching effects can cause large variations in run times. If we can determine worst-case execution time, then shorter run times for a process simply mean unused CPU time. If we cannot accurately bound WCET, then we will be left with a very conservative estimate of execution time that will leave even more CPU time unused.
We also assumed that processes don’t interact, but the cache causes the execution of one program to influence the execution time of other programs. The techniques for bounding the cache-based performance of a single program do not work when multiple programs are in the same cache. Many real-time systems have been designed based on the assumption that there is no cache present, even though one actually exists. This grossly conservative assumption is made because the system architects lack tools that permit them to analyze the effect of caching. Since they do not know where caching will cause problems, they are forced to retreat to the simplifying assumption that there is no cache. The result is extremely overdesigned hardware, which has much more computational power than is necessary. However, just as experience tells us that a well-designed cache provides significant performance benefits for a single program, a properly sized cache can allow a microprocessor to run a set of processes much more quickly. By analyzing the effects of the cache, we can make much better use of the available hardware.

Li and Wolf [Li99] developed a model for estimating the performance of multiple processes sharing a cache. In the model, some processes can be given reservations in the cache, such that only a particular process can inhabit a reserved section of the cache; other processes are left to share the cache. We generally want to use cache partitions only for performance-critical processes since cache reservations are wasteful of limited cache space. Performance is estimated by constructing a schedule, taking into account not just execution time of the processes but also the state of the cache. Each process in the shared section of the cache is modeled by a binary variable: 1 if present in the cache and 0 if not. Each process is also characterized by three total execution times: assuming no caching, with typical caching, and with all code always resident in the cache. The always-resident time is unrealistically optimistic, but it can be used to find a lower bound on the required schedule time. During construction of the schedule, we can look at the current cache state to see whether the no-cache or typical-caching execution time should be used at this point in the schedule. We can also update the cache state if the cache is needed for another process. Although this model is simple, it provides much more realistic performance estimates than assuming the cache either is nonexistent or is perfect. Example 6.9 shows how cache management can improve CPU utilization.

**Example 6.9**

*Effects of scheduling on the cache*

Consider a system containing the following three processes:

<table>
<thead>
<tr>
<th>Process</th>
<th>Worst-case CPU time</th>
<th>Average-case CPU time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>P2</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>P3</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>
Each process uses half the cache, so only two processes can be in the cache at the same time.

Appearing below is a first schedule that uses a least-recently-used cache replacement policy on a process-by-process basis.

In the first iteration, we must fill up the cache, but even in subsequent iterations, competition among all three processes ensures that a process is never in the cache when it starts to execute. As a result, we must always use the worst-case execution time.

Another schedule in which we have reserved half the cache for P1 is shown below. This leaves P2 and P3 to fight over the other half of the cache.

In this case, P2 and P3 still compete, but P1 is always ready. After the first iteration, we can use the average-case execution time for P1, which gives us some spare CPU time that could be used for additional operations.

---

6.6 POWER MANAGEMENT AND OPTIMIZATION FOR PROCESSES

We learned in Section 3.6 about the features that CPUs provide to manage power consumption. The RTOS and system architecture can use static and dynamic power management mechanisms to help manage the system’s power consumption. A **power management policy** [Ben00] is a strategy for determining when to
perform certain power management operations. A power management policy in general examines the state of the system to determine when to take actions. However, the overall strategy embodied in the policy should be designed based on the characteristics of the static and dynamic power management mechanisms.

Going into a low-power mode takes time; generally, the more that is shut off, the longer the delay incurred during restart. Because power-down and power-up are not free, modes should be changed carefully. Determining when to switch into and out of a power-up mode requires an analysis of the overall system activity.

- Avoiding a power-down mode can cost unnecessary power.
- Powering down too soon can cause severe performance penalties.

Re-entering run mode typically costs a considerable amount of time. A straightforward method is to power up the system when a request is received. This works as long as the delay in handling the request is acceptable. A more sophisticated technique is **predictive shutdown**. The goal is to predict when the next request will be made and to start the system just before that time, saving the requestor the start-up time. In general, predictive shutdown techniques are probabilistic—they make guesses about activity patterns based on a probabilistic model of expected behavior. Because they rely on statistics, they may not always correctly guess the time of the next activity. This can cause two types of problems:

- The requestor may have to wait for an activity period. In the worst case, the requestor may not make a deadline due to the delay incurred by system start-up.
- The system may restart itself when no activity is imminent. As a result, the system will waste power.

Clearly, the choice of a good probabilistic model of service requests is important. The policy mechanism should also not be too complex, since the power it consumes to make decisions is part of the total system power budget.

Several predictive techniques are possible. A very simple technique is to use fixed times. For instance, if the system does not receive inputs during an interval of length $T_{on}$, it shuts down; a powered-down system waits for a period $T_{off}$ before returning to the power-on mode. The choice of $T_{off}$ and $T_{on}$ must be determined by experimentation. Srivastava and Eustace [Sri94] found one useful rule for graphics terminals. They plotted the observed idle time ($T_{off}$) of a graphics terminal versus the immediately preceding active time ($T_{on}$). The result was an L-shaped distribution as illustrated in Figure 6.17. In this distribution, the idle period after a long active period is usually very short, and the length of the idle period after a short active period is uniformly distributed. Based on this distribution, they proposed a shut down threshold that depended on the length of the last active period—they shut
down when the active period length was below a threshold, putting the system in the vertical portion of the $L$ distribution.

The *Advanced Configuration and Power Interface (ACPI)* is an open industry standard for power management services. It is designed to be compatible with a wide variety of OSs. It was targeted initially to PCs. The role of ACPI in the system is illustrated in Figure 6.18. ACPI provides some basic power management facilities and abstracts the hardware layer, the OS has its own power management module that determines the policy, and the OS then uses ACPI to send the required controls to the hardware and to observe the hardware’s state as input to the power manager.

ACPI supports the following five basic global power states:

- G3, the mechanical off state, in which the system consumes no power.
- G2, the soft off state, which requires a full OS reboot to restore the machine to working condition. This state has four substates:
  - S1, a low wake-up latency state with no loss of system context;
  - S2, a low wake-up latency state with a loss of CPU and system cache state;
  - S3, a low wake-up latency state in which all system state except for main memory is lost; and
  - S4, the lowest-power sleeping state, in which all devices are turned off.
- G1, the sleeping state, in which the system appears to be off and the time required to return to working condition is inversely proportional to power consumption.
### FIGURE 6.18
The advanced configuration and power interface and its relationship to a complete system.

- G0, the working state, in which the system is fully usable.
- The legacy state, in which the system does not comply with ACPI.

The power manager typically includes an observer, which receives messages through the ACPI interface that describe the system behavior. It also includes a decision module that determines power management actions based on those observations.

#### Design Example 6.7 TELEPHONE ANSWERING MACHINE

In this section we design a digital telephone answering machine. The system will store messages in digital form rather than on an analog tape. To make life more interesting, we use a simple algorithm to compress the voice data so that we can make more efficient use of the limited amount of available memory.

#### 6.7.1 Theory of Operation and Requirements

In addition to studying the compression algorithm, we also need to learn a little about the operation of telephone systems.

The compression scheme we will use is known as **adaptive differential pulse code modulation (ADPCM)**. Despite the long name, the technique is relatively simple but can yield $2 \times$ compression ratios on voice data.
The ADPCM coding scheme is illustrated in Figure 6.19. Unlike traditional sampling, in which each sample shows the magnitude of the signal at a particular time, ADPCM encodes changes in the signal. The samples are expressed in a coding alphabet, whose values are in a relative range that spans both negative and positive values. In this case, the value range is \{-3, -2, -1, 1, 2, 3\}. Each sample is used to predict the value of the signal at the current instant from the previous value. At each point in time, the sample is chosen such that the error between the predicted value and the actual signal value is minimized.

An ADPCM compression system, including an encoder and decoder, is shown in Figure 6.20. The encoder is more complex, but both the encoder and decoder use an integrator to reconstruct the waveform from the samples. The integrator simply computes a running sum of the history of the samples; because the samples are differential, integration reconstructs the original signal. The encoder compares the incoming waveform to the predicted waveform (the waveform that will be generated in the decoder). The quantizer encodes this difference as the best predictor of the next waveform value. The inverse quantizer allows us to map bit-level symbols onto real numerical values; for example, the eight possible codes in a 3-bit code can be mapped onto floating-point numbers. The decoder simply uses an inverse quantizer and integrator to turn the differential samples into the waveform.

The answering machine will ultimately be connected to a telephone subscriber line (although for testing purposes we will construct a simulated line). At the other end of the subscriber line is the central office. All information is carried on the phone line in analog form over a pair of wires. In addition to analog/digital and digital/analog converters to send and receive voice data, we need to sense two other characteristics of the line.
- **Ringing**: The central office sends a ringing signal to the telephone when a call is waiting. The ringing signal is in fact a 90 V RMS sinusoid, but we can use analog circuitry to produce 0 for no ringing and 1 for ringing.

- **Off-book**: The telephone industry term for answering a call is going off-book; the technical term for hanging up is going on-book. (This creates some initial confusion since off-book means the telephone is active and on-book means it is not in use, but the terminology starts to make sense after a few uses.) Our interface will send a digital signal to take the phone line off-hook, which will cause analog circuitry to make the necessary connection so that voice data can be sent and received during the call.

We can now write the requirements for the answering machine. We will assume that the interface is not to the actual phone line but to some circuitry that provides voice samples, off-hook commands, and so on. Such circuitry will let us test our system with a telephone line simulator and then build the analog circuitry necessary to connect to a real phone line. We will use the term **outgoing message (OGM)** to refer to the message recorded by the owner of the machine and played at the start of every phone call.
Design Example: Telephone Answering Machine

<table>
<thead>
<tr>
<th>Name</th>
<th>Digital telephone answering machine</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>Telephone answering machine with digital memory, using speech compression.</td>
</tr>
</tbody>
</table>
| **Inputs** | *Telephone:* voice samples, ring indicator.  
*User interface:* microphone, play messages button, record OGM button. |
| **Outputs** | *Telephone:* voice samples, on-hook/off-hook command.  
*User interface:* speaker, # messages indicator, message light. |
| **Functions** | *Default mode:* When machine receives ring indicator, it signals off-hook, plays the OGM, and then records the incoming message. Maximum recording length for incoming message is 30 s, at which time the machine hangs up. If the machine runs out of memory, the OGM is played and the machine then hangs up without recording.  
*Playback mode:* When the play button is depressed, the machine plays all messages. If the play button is depressed again within five seconds, the messages are played again. Messages are erased after playback.  
*OGM editing mode:* When the user hits the record OGM button, the machine records an OGM of up to 10 s. When the user holds down the record OGM button and hits the play button, the OGM is played back. |
| **Performance** | Should be able to record about 30 min of total voice, including incoming and OGMs. Voice data are sampled at the standard telephone rate of 8 kHz. |
| **Manufacturing cost** | Consumer product range: approximately $50. |
| **Power** | Powered by AC through a standard power supply. |
| **Physical size and weight** | Comparable in size and weight to a desk telephone. |

We have made a few arbitrary decisions about the user interface in these requirements. The amount of voice data that can be saved by the machine should in fact be determined by two factors: the price per unit of DRAM at the time at which the device goes into manufacturing (since the cost will almost certainly drop from the start of design to manufacture) and the projected retail price at which the machine must sell. The protocol when the memory is full is also arbitrary—it would make at least as much sense to throw out old messages and replace them with new ones, and ideally the user could select which protocol to use. Extra features such as an indicator showing the number of messages or a save messages feature would also be nice to have in a real consumer product.
6.7.2 Specification

Figure 6.21 shows the class diagram for the answering machine. In addition to the classes that perform the major functions, we also use classes to describe the incoming and OGMs. As seen below, these classes are related.

The definitions of the physical interface classes are shown in Figure 6.22. The buttons and lights simply provide attributes for their input and output values. The phone line, microphone, and speaker are given behaviors that let us sample their current values.

The message classes are defined in Figure 6.23. Since incoming and OGM types share many characteristics, we derive both from a more fundamental message type.

The major operational classes—Controls, Record, and Playback—are defined in Figure 6.24. The Controls class provides an operate() behavior that oversees the user-level operations. The Record and Playback classes provide behaviors that handle writing and reading sample sequences.

The state diagram for the Controls activate behavior is shown in Figure 6.25. Most of the user activities are relatively straightforward. The most complex is answering an incoming call. As with the software modem of Section 5.11, we want to be sure that a single depression of a button causes the required action to be taken exactly once; this requires edge detection on the button signal.

State diagrams for record-msg and playback-msg are shown in Figure 6.26. We have parameterized the specification for record-msg so that it can be used either from the phone line or from the microphone. This requires parameterizing the source itself and the termination condition.

FIGURE 6.21
Class diagram for the answering machine.
6.7 Design Example: Telephone Answering Machine

**FIGURE 6.22**
Physical class interfaces for the answering machine.

**FIGURE 6.23**
The message classes for the answering machine.

**FIGURE 6.24**
Operational classes for the answering machine.
6.7.3 System Architecture

The machine consists of two major subsystems from the user’s point of view: the user interface and the telephone interface. The user and telephone interfaces both appear internally as I/O devices on the CPU bus with the main memory serving as the storage for the messages.

The software splits into the following seven major pieces:

- The **front panel module** handles the buttons and lights.
- The **speaker module** handles sending data to the user’s speaker.
- The **telephone line module** handles off-hook detection and on-hook commands.
- The **telephone input and output modules** handle receiving samples from and sending samples to the telephone line.
The *compression module* compresses data and stores it in memory.

The *decompression module* uncompresses data and sends it to the speaker module.

We can determine the execution model for these modules based on the rates at which they must work and the ways in which they communicate.

- The front panel and telephone line modules must regularly test the buttons and phone line, but this can be done at a fairly low rate. As seen below, they can therefore run as polled processes in the software’s main loop.

```c
while (TRUE) {
    check_phone_line();
    run_front_panel();
}
```

- The speaker and phone input and output modules must run at higher, regular rates and are natural candidates for interrupt processing. These modules don’t run all the time and so can be disabled by the front panel and telephone line modules when they are not needed.
- The compression and decompression modules run at the same rate as the speaker and telephone I/O modules, but they are not directly connected to devices. We will therefore call them as subroutines to the interrupt modules.

One subtlety is that we must construct a very simple file system for messages, since we have a variable number of messages of variable lengths. Since messages vary in length, we must record the length of each one. In this simple specification, because we always play back the messages in the order in which they were recorded, we don’t have to keep a full-fledged directory. If we allowed users to selectively delete messages and save others, we would have to build some sort of directory structure for the messages.

The hardware architecture is straightforward and illustrated in Figure 6.27. The speaker and telephone I/O devices appear as standard A/D and D/A converters. The telephone line appears as a one-bit input device (ring detect) and a one-bit output device (off-hook/on-hook). The compressed data are kept in main memory.

### 6.7.4 Component Design and Testing

Performance analysis is important in this case because we want to ensure that we don’t spend so much time compressing that we miss voice samples. In a real consumer product, we would carefully design the code so that we could use the slowest, cheapest possible CPU that would still perform the required processing in the available time between samples. In this case, we will choose the microprocessor in advance for simplicity and simply ensure that all the deadlines are met.

An important class of problems that should be adequately tested is memory overflow. The system can run out of memory at any time, not just between messages. The modules should be tested to ensure that they do reasonable things when all the available memory is used up.

![FIGURE 6.27](image)

The hardware structure of the answering machine.
6.7.5 System Integration and Testing

We can test partial integrations of the software on our host platform. Final testing with real voice data must wait until the application is moved to the target platform.

Testing your system by connecting it directly to the phone line is not a very good idea. In the United States, the Federal Communications Commission regulates equipment connected to phone lines. Beyond legal problems, a bad circuit can damage the phone line and incur the wrath of your service provider. The required analog circuitry also requires some amount of tuning, and you need a second telephone line to generate phone calls for tests. You can build a telephone line simulator to test the hardware independently of a real telephone line. The phone line simulator consists of A/D and D/A converters plus a speaker and microphone for voice data, an LED for off-hook/on-hook indication, and a button for ring generation. The telephone line interface can easily be adapted to connect to these components, and for purposes of testing the answering machine the simulator behaves identically to the real phone line.

SUMMARY

The process abstraction is forced on us by the need to satisfy complex timing requirements, particularly for multirate systems. Writing a single program that simultaneously satisfies deadlines at multiple rates is too difficult because the control structure of the program becomes unintelligible. The process encapsulates the state of a computation, allowing us to easily switch among different computations.

The operating system encapsulates the complex control to coordinate the process. The scheme used to determine the transfer of control among processes is known as a scheduling policy. A good scheduling policy is useful across many different applications while also providing efficient utilization of the available CPU cycles.

It is difficult, however, to achieve 100% utilization of the CPU for complex applications. Because of variations in data arrivals and computation times, reserving some cycles to meet worst-case conditions is often necessary. Some scheduling policies achieve higher utilizations than others, but often at the cost of unpredictability—they may not guarantee that all deadlines are met. Knowledge of the characteristics of an application can be used to increase CPU utilization while also complying with deadlines.

What We Learned

- A process is a single thread of execution.
- Pre-emption is the act of changing the CPU's execution from one process to another.
- A scheduling policy is a set of rules that determines the process to run.
Rate-monotonic scheduling (RMS) is a simple but powerful scheduling policy.

Interprocess communication mechanisms allow data to be passed reliably between processes.

Scheduling analysis often ignores certain real-world effects. Cache interactions between processes are the most important effects to consider when designing a system.

**FURTHER READING**

Gallmeister [Gal95] provides a thorough and very readable introduction to POSIX in general and its real-time aspects in particular. Liu and Layland [Liu73] introduce rate-monotonic scheduling; this paper became the foundation for real-time systems analysis and design. The book by Liu [Liu00] provides a detailed analysis of real-time scheduling. Benini et al. [Ben00] provide a good survey of system-level power management techniques. Falik and Intrater [Fal92] describe a custom chip designed to perform answering machine operations.

**QUESTIONS**

Q6-1 Identify activities that operate at different rates in
   a. a PDA;
   b. a laser printer; and
   c. an airplane.

Q6-2 Name an embedded system that requires both periodic and aperiodic computation.

Q6-3 An audio system processes samples at a rate of 44.1 kHz. At what rate could we sample the system’s front panel to both simplify analysis of the system schedule and provide adequate response to the user’s front panel requests?

Q6-4 Draw a UML class diagram for a process in an operating system. The process class should include the necessary attributes and behaviors required of a typical process.

Q6-5 What factors provide a lower bound on the period at which the system timer interrupts for preemptive context switching?

Q6-6 What factors provide an upper bound on the period at which the system timer interrupts for preemptive context switching?
Q6-7 You are given these periodic tasks:

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>5 ms</td>
<td>2 ms</td>
</tr>
<tr>
<td>P2</td>
<td>10 ms</td>
<td>3 ms</td>
</tr>
<tr>
<td>P3</td>
<td>10 ms</td>
<td>3 ms</td>
</tr>
<tr>
<td>P4</td>
<td>15 ms</td>
<td>6 ms</td>
</tr>
</tbody>
</table>

Compute the utilization of this set of tasks.

Q6-8 You are given these periodic tasks:

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>5 ms</td>
<td>1 ms</td>
</tr>
<tr>
<td>P2</td>
<td>10 ms</td>
<td>2 ms</td>
</tr>
<tr>
<td>P3</td>
<td>10 ms</td>
<td>2 ms</td>
</tr>
<tr>
<td>P4</td>
<td>15 ms</td>
<td>3 ms</td>
</tr>
</tbody>
</table>

a. Show a cyclostatic schedule for the tasks.
b. Compute the CPU utilization for the system.

Q6-9 For the task set of question Q6-8, show a round robin schedule assuming that P1 does not execute during its first period and P3 does not execute during its second period.

Q6-10 What is the distinction between the ready and waiting states of process scheduling?

Q6-11 Provide examples of

a. blocking interprocess communication, and
b. nonblocking interprocess communication.

Q6-12 Assuming that you have a routine called swap(int *a, int *b) that atomically swaps the values of the memory locations pointed to a and b, write C code for:

a. P(); and
b. V().

Q6-13 Draw UML sequence diagrams of two versions of P(): one that incorrectly uses a nonatomic operation to test and set the semaphore location and another that correctly uses an atomic test-and-set.
Q6-14 For the following periodic processes, what is the shortest interval we must examine to see all combinations of deadlines?

a.  

<table>
<thead>
<tr>
<th>Process</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>3</td>
</tr>
<tr>
<td>P2</td>
<td>5</td>
</tr>
<tr>
<td>P3</td>
<td>15</td>
</tr>
</tbody>
</table>

b.  

<table>
<thead>
<tr>
<th>Process</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
</tr>
<tr>
<td>P3</td>
<td>6</td>
</tr>
<tr>
<td>P4</td>
<td>10</td>
</tr>
</tbody>
</table>

c.  

<table>
<thead>
<tr>
<th>Process</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>3</td>
</tr>
<tr>
<td>P2</td>
<td>4</td>
</tr>
<tr>
<td>P3</td>
<td>5</td>
</tr>
<tr>
<td>P4</td>
<td>6</td>
</tr>
<tr>
<td>P5</td>
<td>10</td>
</tr>
</tbody>
</table>

Q6-15 Consider the following system of periodic processes executing on a single CPU:

<table>
<thead>
<tr>
<th>Process</th>
<th>CPU time</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>4</td>
<td>200</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>40</td>
</tr>
<tr>
<td>P4</td>
<td>6</td>
<td>50</td>
</tr>
</tbody>
</table>

Can we add another instance of P1 to the system and still meet all the deadlines using RMS?

Q6-16 Given the following set of periodic processes running on a single CPU, what is the maximum execution time for P5 for which all the processes will be schedulable using RMS?
<table>
<thead>
<tr>
<th>Process</th>
<th>CPU time</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>P2</td>
<td>18</td>
<td>100</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>P4</td>
<td>5</td>
<td>50</td>
</tr>
<tr>
<td>P5</td>
<td>x</td>
<td>25</td>
</tr>
</tbody>
</table>

Q6-17 A set of periodic processes is scheduled using RMS. For the process execution times and periods shown below, show the state of the processes at the critical instant for each of these processes.

a. P1  
b. P2  
c. P3

<table>
<thead>
<tr>
<th>Process</th>
<th>CPU time</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>P2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>P3</td>
<td>1</td>
<td>20</td>
</tr>
</tbody>
</table>

Q6-18 For the given periodic process execution times and periods, show how much CPU time of higher-priority processes will be required during one period of each of the following processes:

a. P1  
b. P2  
c. P3  
d. P4

<table>
<thead>
<tr>
<th>Process</th>
<th>CPU time</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>P2</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>25</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>50</td>
</tr>
</tbody>
</table>

Q6-19 For the periodic processes shown below:

a. Schedule the processes using an RMS policy.  
b. Schedule the processes using an EDF policy.

In each case, compute the schedule for the hyperperiod of the processes. Time starts at $t = 0$. 
Q6-20 For the periodic processes shown below:

a. Schedule the processes using an RMS policy.
b. Schedule the processes using an EDF policy.

In each case, compute the schedule for an interval equal to the hyperperiod of the processes. Time starts at $t = 0$.

<table>
<thead>
<tr>
<th>Process</th>
<th>CPU time</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>P3</td>
<td>1</td>
<td>12</td>
</tr>
</tbody>
</table>

Q6-21 For the given set of periodic processes, all of which share the same deadline of 12:

a. Schedule the processes for the given arrival times using standard rate-monotonic scheduling (no data dependencies).
b. Schedule the processes taking advantage of the data dependencies. By how much is the CPU utilization reduced?

![Diagram of process dependencies]

<table>
<thead>
<tr>
<th>Process</th>
<th>CPU time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
</tr>
</tbody>
</table>
Q6-22 For the periodic processes given below, find a valid schedule
   a. using standard RMS, and
   b. adding one unit of overhead for each context switch.

<table>
<thead>
<tr>
<th>Process</th>
<th>CPU time</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>2</td>
<td>30</td>
</tr>
<tr>
<td>P2</td>
<td>4</td>
<td>40</td>
</tr>
<tr>
<td>P3</td>
<td>7</td>
<td>120</td>
</tr>
<tr>
<td>P4</td>
<td>5</td>
<td>60</td>
</tr>
<tr>
<td>P5</td>
<td>1</td>
<td>15</td>
</tr>
</tbody>
</table>

Q6-23 For the periodic processes and deadlines given below:
   a. Schedule the processes using RMS.
   b. Schedule using EDF and compare the number of context switches required for EDF and RMS.

<table>
<thead>
<tr>
<th>Process</th>
<th>CPU time</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>P4</td>
<td>9</td>
<td>50</td>
</tr>
<tr>
<td>P5</td>
<td>7</td>
<td>100</td>
</tr>
</tbody>
</table>

Q6-24 In each circumstance below, would shared memory or message passing communication be better? Explain.
   b. A digital video decoder and a process that overlays user menus on the display.
   c. A software modem process and a printing process in a fax machine.

Q6-25 If you wanted to reduce the cache conflicts between the most computationally intensive parts of two processes, what are two ways that you could control the locations of the processes’ cache footprints?

Q6-26 Draw a state diagram for the predictive shutdown mechanism of a cell phone. The cell phone wakes itself up once every five minutes for 0.01 second to listen for its address. It goes back to sleep if it does not hear its address or after it has received its message.

Q6-27 How would you use the ADPCM method to encode an unvarying (DC) signal with the coding alphabet \{-3, -2, -1, 1, 2, 3\}?
LAB EXERCISES

L6-1 Using your favorite operating system, write code to spawn a process that
writes “Hello, world” to the screen or flashes an LED, depending on your
available output devices.

L6-2 Build a small serial port device that lights LEDs based on the last character
written to the serial port. Create a process that will light LEDs based on
keyboard input.

L6-3 Write a driver for an I/O device.

L6-4 Write context switch code for your favorite CPU.

L6-5 Measure context switching overhead on an operating system.

L6-6 Using a CPU that runs an operating system that uses RMS, try to get the CPU
utilization up to 100%. Vary the data arrival times to test the robustness of the
system.

L6-7 Using a CPU that runs an operating system that uses EDF, try to get the CPU
utilization as close to 100% as possible without failing. Try a variety of data
arrival times to determine how sensitive your process set is to environmental
variations.