CHAPTER

# Processes and Operating Systems

# 6

## CHAPTER POINTS

- Process abstraction.
- Switching contexts between programs.
- Real-time operating systems (RTOSs).
- Interprocess communication.
- Task-level performance analysis and power consumption.
- Design example: engine control unit.

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

Together these two mechanisms allow us to build applications with more complex functionality and much greater flexibility to satisfy timing requirements. The need to satisfy complex timing requirements—events happening at distinctive rates, intermittent events, and so on—causes us to use processes and operating systems to build embedded software. Satisfying complex timing tasks can introduce extremely complex control into programs. Using processes to compartmentalize functions and encapsulating in the operating system, the control required to switch between processes makes it much easier to satisfy timing requirements with relatively clean control within the processes.

**Computers as Components.** https://doi.org/10.1016/B978-0-323-85128-2.00006-2

We are particularly interested in the **real-time operating system** (**RTOS**), an operating system that provide facilities for satisfying real-time requirements. An RTOS allocates resources using algorithms that take real time into account. General-purpose operating systems, in contrast, generally allocate resources using other criteria, such as fairness. Trying to allocate the CPU equally to all processes without regard to time can easily cause processes to miss deadlines.

In the next section, we introduce the concept of a process. Section 6.2 motivates our need for multiple processes. Section 6.3 looks at how the RTOS implements processes. Section 6.4 develops algorithms for scheduling those processes to meet real-time requirements. Section 6.6 introduces some basic concepts of interprocess communication. Section 6.7 considers the performance of real-time operating systems. Section 6.8 surveys various RTOSs. We study an engine control unit as a design example in Section 6.9.

## 6.2 Multiple tasks and processes

We studied the design of the programs in Chapter 5. An RTOS allows us to run several programs concurrently. The RTOS helps build more complex systems using several programs that run concurrently. Processes and tasks are the building blocks of multitasking systems, much as C functions and code modules are the building blocks of programs.

**Tasks**

Many (if not most) embedded computing systems do more than one thing—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 that must be performed at distinctive rates.

The term **task** is used in several different ways in real-time computing. We use it to mean a set of real-time programs that can communicate. Fig. 6.1 shows a task that consists of three subtasks; arrows in the **task graph** show data dependencies. P3 cannot start before it receives the results of P1 and P2, and P1 and P2 can be executed in parallel.

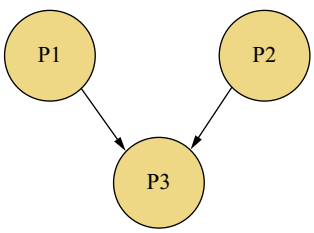



**FIGURE 6.1**

A task made up of three subtasks.

A **process** is a single execution of a program. If we run the same program at two different times, we have created two different processes. Each process has its own state that includes not only its registers but also all its memory. In some operating systems, 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**.

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 text compression unit based on the compression algorithm implemented in Section 3.8. As shown in Fig. 6.2, 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 the data being sent to a modem.

**Variable data rates**

There is a need to receive and send data at different rates. For example, the program may emit two bits for the first byte and then seven bits for the second byte. This routine will obviously find itself reflected in the structure of the code. It is possible to create irregular, ungainly code to solve this problem, but a more elegant solution is to create a queue of output bits, with those bits being removed from the

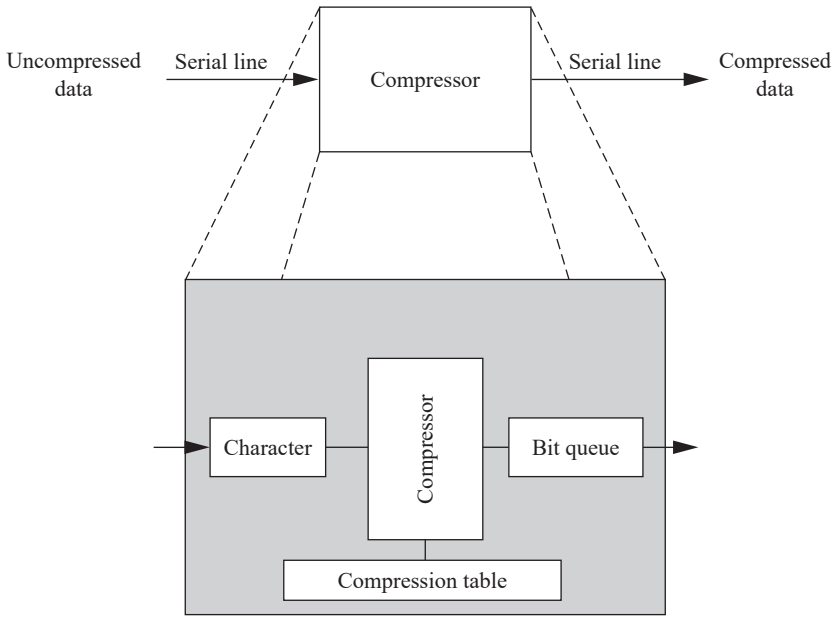



**FIGURE 6.2**

The text compression engine as a multirate system.

queue and sent to the serial port in eight-bit sets. However, 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 packaging and emitting output characters, we may drop an input character. Solving timing problems is a challenging problem.

**Asynchronous input**

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 don't know when the user will push the button for compression mode, but 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 the characters will be received because 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 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 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. However, this solution doesn't 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 must be able to keep track of these two tasks separately by applying different timing requirements to each. This is the sort of control that processes allow.

**Events**

An asynchronous input is an example of an **event**. We consider the timing analysis of events in Section 6.5.8.

**Timing and programming**

These two examples illustrate how requirements for 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 quickly become very complex. Worse, such complex control is usually quite difficult to verify for either functional or timing properties.

## 6.3 Multirate systems

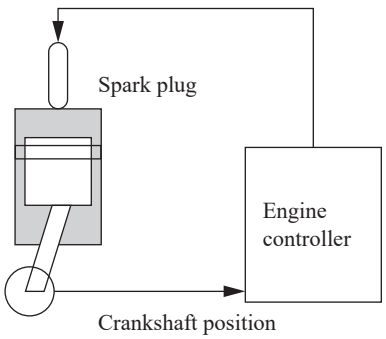
Implementing code that satisfies timing requirements is even more complex when multiple rates of computation must be handled. **Multirate** embedded computing systems, including automobile engines, printers, and cell phones, are very common.

In 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 much greater. Automobile engines must meet strict requirements, as mandated by law in the United States, for both emissions and fuel economy. On the other hand, engines must 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 a gas-pedal position and an oxygen sensor, which is used to control emissions. They also used a multimode control scheme. For example, one mode may be used for engine warm-up, another for cruise, another for climbing steep hills, and so forth. A larger number of sensors and modes increases the number of discrete tasks that must be performed. The highest-rate task is firing the spark plugs. The throttle setting must be sampled and acted upon regularly, although not as frequently as the crankshaft setting and spark plugs. The oxygen sensor responds much more slowly than does 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: spark-plug firings and the fuel/air mixture. 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) |
|---|---|---|
| Engine spark timing | 300 | 2 |
| Throttle | 40 | 2 |
| Air flow | 30 | 4 |
| Cranking battery voltage | 80 | 4 |
| Fuel flow | 250 | 10 |
| Percentage recycled exhaust gas | 500 | 25 |
| Set of status switches | 100 | 50 |
| Air temperature | Seconds | 500 |
| Barometric pressure | Seconds | 1,000 |
| Spark/dwell | 10 | 1 |
| Fuel adjustments | 80 | 4 |
| Carburetor adjustments | 500 | 25 |
| Mode actuators | 100 | 100 |

The fastest rate that the engine controller must handle is 2 ms, and the slowest rate is 4 s, a range of three orders of magnitude.

### 6.3.1 Timing requirements for processes

Processes can have several different types of timing requirements imposed on them by the application. The timing requirements of 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 in more detail, we outline the types of process timing requirements that are useful in embedded system design.

Fig. 6.3 illustrates different ways in which we can define two important requirements for processes: **initiation time** and **deadline.** The initiation time is the time at which the process goes from the waiting state to the ready state. An aperiodic process is, by definition, initiated by an event, such as the arrival of external data or data computed by another process. The initiation 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 be ready at the beginning of the period.

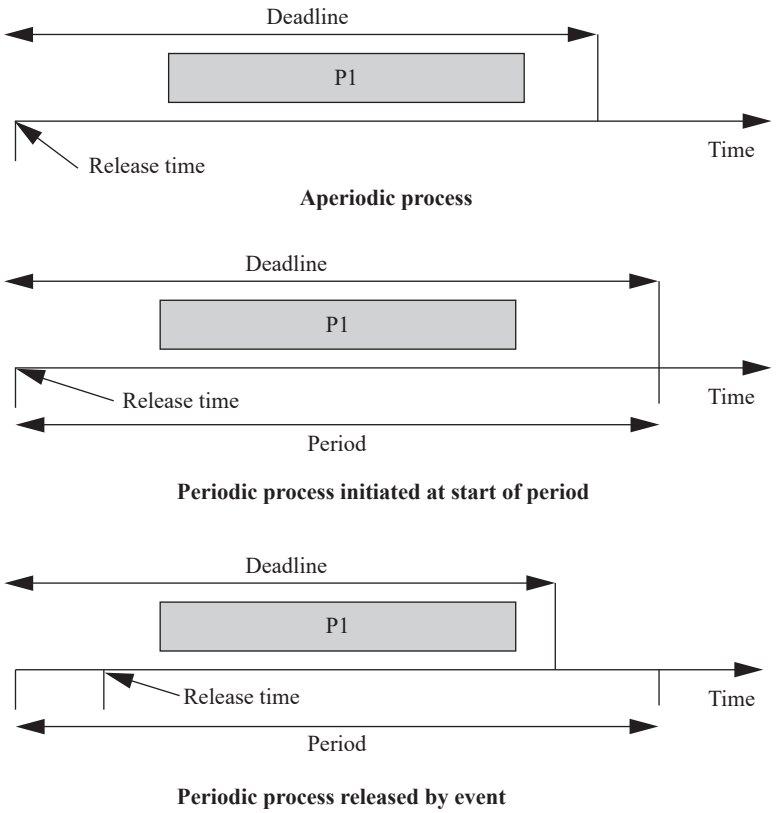



**FIGURE 6.3**

Example definitions of initiation times and deadlines.

More sophisticated systems may set the initiation 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 initiation time because 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 we will see in Section 6.5, some scheduling policies simplify the assumption that the deadline occurs at the end of the period.

Rate requirements are also 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 requirement for periodic processes is for the **initiation interval** to be equal to the period. However, the pipelined execution of processes allows the

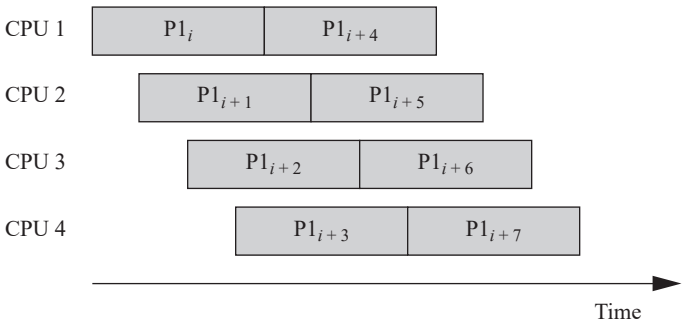



**FIGURE 6.4**

A sequence of processes with a high initiation rate.

initiation interval to be less than the period. Fig. 6.4 illustrates the 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.

**Hyperperiod**

When we consider a set of processes, we often talk about the **hyperperiod** of the set of processes—the least-common multiple (LCM) of the periods of the processes. We will see later that the hyperperiod tells us the length of the time interval over which we must analyze the schedule.

**Response time**

Although a period is a specification of the expected behavior of a task, we also want to talk about its actual behavior. We define the **response time** of a process as the time at which the process finishes. If the schedule meets its requirements, the response time will be before the end of the process's period. The response time depends only in part on its **computation time**: how long it takes to execute. In a multitasking system, the process may be interrupted to allow other processes to run. The response time accounts for all the CPU time allocated to other processes.

**Jitter**

We may also be concerned with the **jitter** of a task, which is the allowable variation in its completion. Jitter can be important to a variety of applications: the playback of multimedia data to avoid audio gaps or jerky images and the control of machines to ensure that the control signal is applied at the right time.

**Missing a deadline**

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 telephone system may cause a temporary silence on the line. 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 they may completely ignore the failure.

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

### Application Example 6.2: A Space Shuttle Software Error

Garman [Gar81] described a software problem that delayed the first launch of the US 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 is 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 the 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 they could keep track of the state of the shuttle. However, BFS would stop listening when it thought PASS was compromising data fetching. This would prevent PASS failures from inadvertently destroying the state of the BFS. PASS uses an asynchronous, priority-driven software architecture. If high-priority processes take too much time, the operating system 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. Because the BFS monitored the PASS, it could be 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 in 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 two years before the incident, a subroutine used to initialize the data bus was modified. Because 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 because 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 timing constraints between processes may be constrained when the processes pass data among each other. Fig. 6.5 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 both must complete before P3, and P3 must complete before P4. All processes must be finished before the end of the period. The data dependencies must form

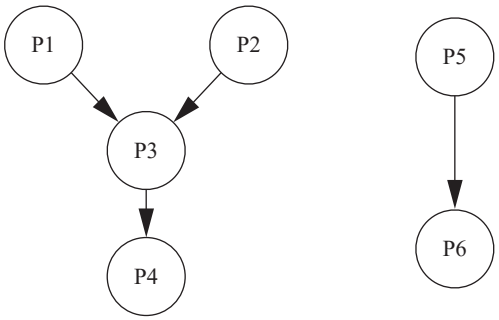



**FIGURE 6.5**

Data dependencies among processes.

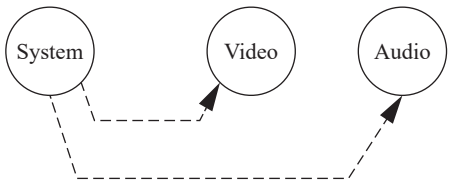



**FIGURE 6.6**

Communication among processes at different rates.

a directed acyclic graph; a cycle in the data dependencies is difficult to interpret in a periodically executed system.

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. Fig. 6.6 illustrates the communication required among the 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 them 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.

### 6.3.2 CPU usage metrics

In addition to the application characteristics, we must have a basic measure of the efficiency with which we use the CPU. The simplest and most direct measure is **utilization**:

$$U = \frac{\text{CPU time for useful work}}{\text{total available CPU time}} \qquad \text{(Eq. 6.1)}$$

available CPU time. This ratio ranges between zero and one, with one meaning that all the available CPU time is being used for system purposes. Utilization is often expressed as a percentage, and it is typically calculated over the hyperperiod of the task set.

### 6.3.3 Process state and scheduling

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

The operating system considers a process to be in one of three basic **scheduling states**: **waiting**, **ready**, or **executing**. At most, there is 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 null operations. Any process that could be executed is in the ready state; the operating system 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 a waiting state. Fig. 6.7 shows the possible transitions between the 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 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 and is ready to run, and the scheduler selects the process as the next process to run.

A **scheduling policy** defines how processes are selected for promotion from the ready state to the running state. Every multitasking operating system 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.

Scheduling policies vary widely in terms of the generality of the timing requirements they can handle and the efficiency with which they use the CPU. Utilization

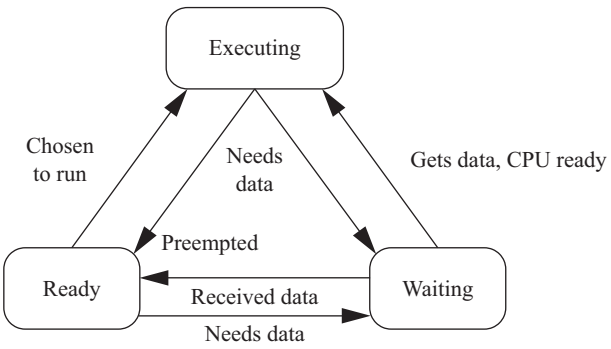



**FIGURE 6.7**

Scheduling states of a process.

is one of the key metrics in evaluating a scheduling policy. 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 utilization than others, even for the same timing requirements. The best policy depends on the required timing characteristics of the scheduled processes.

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. Generally, 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 regarding a scheduling policy must account for both theoretical utilization and practical scheduling overhead.

### 6.3.4 Running periodic processes

We must 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 call them `p1()`, `p2()`, etc., for simplicity. Our goal is to run these subroutines at rates determined by the system designer.

**First step: while loop**

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

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

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

**A timed loop**

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 "no operations" to make the execution time of an iteration equal the desired period. Although some video games were designed this way in the 1970s, this technique should be avoided. Modern processors make it difficult to accurately determine execution time, and even simple processors have nontrivial timing, as we saw in Chapter 3. Conditionals anywhere in the program make it even harder to be sure that the loop consumes the same amount of execution time for every iteration. Furthermore, if any part of the program is changed, the entire timing scheme must be reevaluated.

A timer is a much more reliable way to control the 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:

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

What happens when a process runs too long? The timer's interrupt will cause the CPU's interrupt system to mask its interrupts (at least on a reasonable processor), so the interrupt won't 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.

**Multiple timers**

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

```
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 requires multiple timers, and we may not have enough timers to support all rates required by a system.

**Timer plus counters**

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:

```
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 likelier to contain bugs.

The next example illustrates an approach to cooperative multitasking in PIC16F.

### Programming Example 6.1: Cooperative Multitasking in PIC16F887

We can establish a time base using Timer 0. The period of the timer is set to the period for the execution of all tasks. The flag `T0IE` enables interrupts for Timer 0. When the timer finishes, it causes an interrupt, and `T0IF` is set. The interrupt handler for the timer tells us when the timer has ticked using the global variable timer_flag:

```
void interrupt timer_handler() {
      if (T0IE && T0IF) { /* timer 0 interrupt */
              timer_flag=1; /* tell main that the next time period has
                  started */
              T0IF=0; /* clear timer 0 interrupt flag */
}
```

The main program first initializes the timer and interrupt system, including setting the desired period for the timer. It then uses a while loop to run the tasks at the start of each period:

```
main() {
       init(); /* initialize system, timer, etc. */
       while (1) { /* do forever */
              if (timer_flag) { /* now do the tasks */
                       task1();
                       task2();
                       task3();
                       timer_flag=0 ; /* reset timer flag */
              }
       }
}
```

Why not just put the tasks into the timer handler? We want to ensure that an iteration of the tasks is completed. If the tasks were executed in `timer_handler()` but ran past the period, the timer would interrupt again and stop execution of the previous iteration. The interrupted task may be left in an inconsistent state.

## 6.4 Preemptive Real-Time Operating Systems

A preemptive RTOS solves the fundamental problems of a cooperative multitasking system. It executes processes based on timing requirements provided by the system designer. The most reliable way to meet timing requirements accurately is to build a **preemptive** operating system and use **priorities** to control what process runs at any given time. We use these two concepts to build a basic RTOS. We will use FreeRTOS

[Bar07] as our example operating system. This operating system runs on many different platforms.

### 6.4.1 Two basic concepts

To make our operating system work, we must introduce two basic concepts simultaneously. First, we introduce preemption as an alternative to the C-function call to control execution. Second, we introduce priority-based scheduling as a way for the programmer to control the order in which the processes run. We will explain these ideas one at a time as general concepts, and then, go on in the next sections to see how they are implemented in FreeRTOS.org.

**Preemption**

To take full advantage of the timer, we must change our notion of a process. 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. This, together with the timer, will allow us to move between functions whenever necessary based upon the system's timing constraints.

Fig. 6.8 shows an example of the preemptive execution of an operating system. We want to share the CPU across the two processes. The **kernel** is the part of the operating system that determines what process is running. The timer periodically activates the kernel. 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 distinctive from our use of the timer in the previous section. Previously, we used the timer to control loop iterations, with one loop iteration including the execution of several complete processes. Here, the time quantum is generally smaller than the execution time of any of the processes.

**Process 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

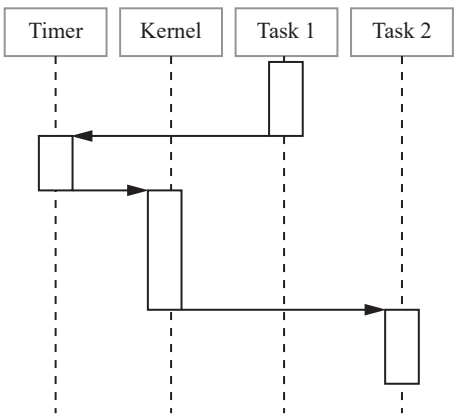



**FIGURE 6.8**

Sequence diagram for preemptive execution.

the processes that do 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 flexible and fast.

### 6.4.2 Processes and context

**Context switching mechanism**

How does the kernel switch between processes before the process is completed? We can't rely on C-level mechanisms to do so; a process is not a C-function or a subroutine. However, we can 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, and to use registers from any process we want. The set of registers that defines a process is 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 **record.**

**RTOS implementation**

The best way to understand processes and context is to dive into an RTOS implementation. We use the FreeRTOS.org kernel as an example; in particular, we use version 7.0.1 for the ARM7 AVR32 platform.

A process is known in FreeRTOS.org as a *task*.

Let's start with the simplest case, namely, steady state: everything has been initialized, the operating system is running, and we are ready for a timer interrupt. Fig. 6.9 shows a sequence diagram in FreeRTOS.org. This diagram shows the application tasks, the hardware timer, and all the functions in the kernel involved in the context switch:

- `vPreemptiveTick()` is called when the timer ticks.
- `SIG_OUTPUT_COMPARE1A` responds to the timer interrupt and uses `portSAVE_CONTEXT()` to swap out the current task context.

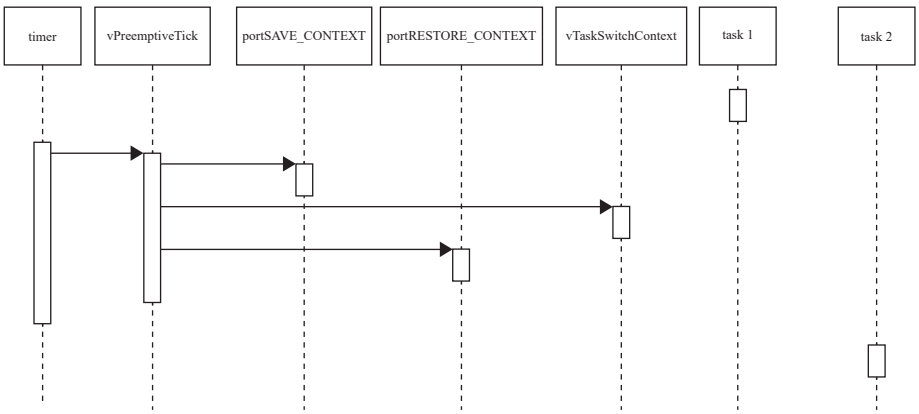



**FIGURE 6.9**

Sequence diagram for a FreeRTOS.org context switch.

- `vTaskIncrementTick()` updates the time and `vTaskSwitchContext` chooses a new task.
- `portRESTORE_CONTEXT()` swaps in the new context.

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

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

    /* Increment the tick count - this may wake a task. */
    vTaskIncrementTick();

    /* Find the highest priority task that is ready to run. */
    vTaskSwitchContext();

    /* End the interrupt in the AIC. */
    AT91C_BASE_AIC->AIC_EOICR = AT91C_BASE_PITC->PITC_PIVR;;

    portRESTORE_CONTEXT();
}
```

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()`. `vTaskIncrementTick()`; then, it performs housekeeping, such as incrementing the tick count. It then determines which task to run next using the routine `vTaskSwitchContext()`. After some more housekeeping, it uses `portRESTORE_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 function and not as a C-function. Let's look at the assembly code that is actually executed.

```
    push  r0
    in    r0, __SREG__
    cli
    push  r0
    push  r1
    clr   r1
    push  r2
; continue pushing all the registers
    push  r31
    lds   r26,  pxCurrentTCB
    lds   r27,  pxCurrentTCB + 1
    in    r0,   __SP_L__
    st    x+,   r0
    in    r0,   __SP_H__
    st    x+,   r0
```

The context includes the 32 general-purpose registers, PC, status registers, and stack pointers SPH and SPL. Register `r0` is saved first because it is used to save the status register. Compilers assume that `r1` is set to zero, so the context switch does

so after saving the old value of `r1`. Most of the routine simply consists of pushing the registers; we have commented out some of those register pushes for clarity. Next, the kernel stores the stack pointer.

Here is the code for `vTaskSwitchContext()`, which is defined in the file tasks.c (minus some preprocessor directives that include some optional code):

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

          taskFIRST_CHECK_FOR_STACK_OVERFLOW();
          taskSECOND_CHECK_FOR_STACK_OVERFLOW();

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

          traceTASK_SWITCHED_IN();
          vWriteTraceToBuffer();
     }
}
```

This function is relatively straightforward; it walks down the list of tasks to identify the highest-priority task.

As with `portSAVE_CONTEXT()`, the `portRESTORE_CONTEXT()` routine is also defined in portmacro.h and is implemented as a macro with embedded assembly language. Here is the underlying assembly code:

```
lds   r26,  pxCurrentTCB
lds   r27,  pxCurrentTCB + 1
ld    r28,  x+
```

```
        out   __SP_L__, r28
        ld    r29, x+
        out   __SP_H__, r29
        pop   r31
    ; pop the registers
        pop   r1
        pop   r0
        out   __SREG__, r0
        pop   r0
```

This code first loads the address for the new task's stack pointer, then gets the stack pointer register values, and finally restores the general-purpose and status registers.

### 6.4.3 Processes and object-oriented design

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

**UML active objects**

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**. Fig. 6.10 shows an example of an active UML 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 the signals in more detail in Section 6.6.

We can mix active objects and normal objects when describing a system. Fig. 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 are sent to the *master* process.

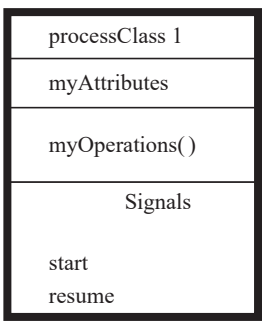



**FIGURE 6.10**

An active class in UML.

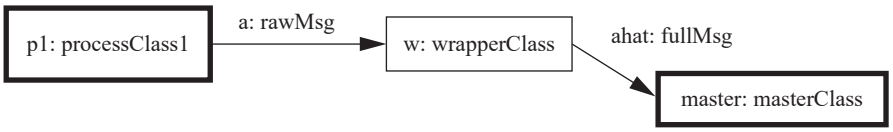



**FIGURE 6.11**

A collaboration diagram with active and normal objects.

## 6.5 Priority-based scheduling

The operating system's fundamental job is to allocate resources in the computing system to programs that request them. Naturally, the CPU is the scarcest resource, so scheduling the CPU is the operating system's most important job. In this section, we consider the structure of operating systems, how they schedule processes to meet performance requirements, shared resources, and other problems in scheduling, scheduling for low power, and the assumptions underlying our scheduling algorithms.

**Round-robin scheduling**

A common scheduling algorithm in general-purpose operating systems is **round-robin**. All processes are kept on a list and scheduled, one after the other. This is generally combined with preemption so that one process does not consume all the CPU time. Round-robin scheduling provides a form of fairness in which all processes get a chance to execute. However, it does not guarantee the completion time of any task; as the number of processes increases, the response time of all processes increases. Real-time systems, in contrast, require their notion of fairness to include timeliness and the satisfaction of deadlines.

**Process priorities**

A common way to choose the next executing process in an RTOS is based on process priorities. Each process is assigned a priority, an integer-valued number. The next process to be chosen for execution is the process in the set of ready processes that has the highest-valued priority. Example 6.1 shows how priorities can be used to schedule processes.

### Example 6.1: Priority-driven Scheduling

For this example, we adopt the following simple rules:

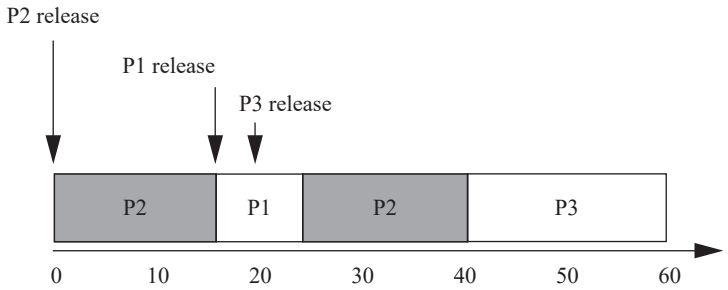
- Each process has a fixed priority that does not vary during the course of execution. More sophisticated scheduling schemes do, in fact, change the priorities of processes to control what happens next.
- The ready process with the highest priority (with one as the highest priority of all) is selected for execution.
- A process continues until it is completed or is preempted by a higher-priority process.

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

| Process | Priority | Execution time |
|---|---|---|
| P1 | 1 | 10 |
| P2 | 2 | 30 |
| P3 | 3 | 20 |

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's data arrive at time 15, and P3's data arrive 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 because it has a higher priority. Because 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 a higher priority than P3. Only after both P1 and P2 finish can P3 execute.

### 6.5.1 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. We say that RMS is a **static scheduling policy** because it assigns fixed priorities to processes. It turns out that these fixed priorities are sufficient to efficiently schedule 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 end of their periods.
- The highest-priority ready process is always selected for execution.

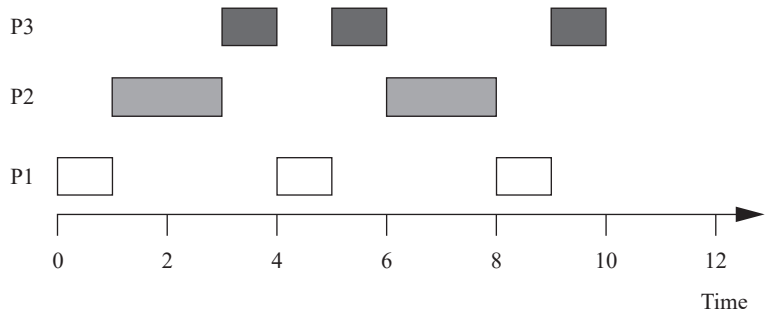
The major result of RMA is that a relatively simple scheduling policy is optimal. Priorities are assigned by rank order of period, with the process with the shortest period being assigned the highest priority. This fixed-priority scheduling policy is the optimum assignment of static priorities to processes, in that it provides the highest CPU utilization while ensuring that all processes meet their deadlines. Example 6.2 illustrates RMS.

## Example 6.2: Rate-monotonic Scheduling

Here is a simple set of processes and their characteristics.

| Process | Execution time | Period |
|---|---|---|
| P1 | 1 | 4 |
| P2 | 2 | 6 |
| P3 | 3 | 12 |

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 must construct a timeline equal in length to the least-common multiple of the process periods (12 in this case). The complete schedule for the least-common multiple of the periods is called the **unrolled schedule**.




All three periods start at time zero. P1's data arrive first. Because 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 the 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 doesn't get to finish until after the third iteration of P1.

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

| Process | Execution time | Period |
|---|---|---|
| P1 | 2 | 4 |
| P2 | 3 | 6 |
| P3 | 3 | 12 |

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 six units of CPU time; P2 two times, costing six units of CPU time; and P3 one time, requiring three 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. The **critical instant** for a process is defined as the instant during execution at which the task has the largest response time; the **critical interval** is the complete interval for 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 a feasible schedule for the system. In the case of the second set of execution times, 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, P1 and P2, be $\tau_1$, $\tau_2$ and $T_1$, $T_2$, respectively, with $\tau_1 < \tau_2$. We can generalize the result of Example 6.2 to show the total CPU requirements for the two processes in two cases. In the first case, let P1 have higher priority. In the worst case, we then execute P2 once during its period and as many iterations of P1 that can fit in the same interval. Because there are $\lfloor \tau_2/\tau_1 \rfloor$ iterations of P1 during a single period of P2, the required constraint on CPU time, ignoring context switching overhead, is

$$\left\lfloor \frac{\tau_2}{\tau_1} \right\rfloor T_1 + T_2 \leq \tau_2 \qquad \text{(Eq. 6.2)}$$

If, on the other hand, we give higher priority to P2, then critical-instant analysis tells us that we must execute all of P2 and all of P1 in one of P1's periods in the worst case:

$$T_1 + T_2 \leq \tau_1 \qquad \text{(Eq. 6.3)}$$

There are cases in which the first relationship can be satisfied, and the second cannot, but there are no cases in which the second relationship can be satisfied and the first cannot. We can inductively show that a process with a 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.

**CPU utilization**

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

$$\mathrm{U} = \sum_{i=1}^{n} \frac{T_1}{\tau_1} \qquad \text{(Eq. 6.4)}$$

It is possible to show that, for a set of two tasks under RMS scheduling, the CPU utilization, *U*, has a least upper bound of $2(2^{1/2} 1) \cong 0.83$. In other words, the CPU will be idle at least 17% of the time. This idle time is because priorities are assigned

statically; we see in the next section that more aggressive scheduling policies can improve the CPU utilization. When there are $m$ tasks and the ratio between any two periods is less than two, the maximum processor utilization is

$$U = m\left(2^{1/m} - 1\right) \qquad \text{(Eq. 6.5)}$$

As $m$ approaches infinity, the CPU utilization (with the factor-of-two restriction on the relationship between periods) asymptotically approaches $\ln 2 = 0.69$; the CPU will be idle 31% of the time. We can use processor utilization $U$ as an easy measure of the feasibility of an RMS scheduling problem.

Consider an example of an RMS schedule for a system in which P1 has a period of 4 and an execution time of 2 and P2 has a period of 7 and an execution time of 1; these tasks satisfy the factor-of-two restriction on relative periods. The hyperperiod of the processes is 28, so the CPU utilization of this set of processes is $[(2 \times 7) + (1 \times 4)]/28 = 0.64$, which is less than our bound of $\ln 2$.

**Implementation**

The implementation of RMS is easy. Fig. 6.12 shows the C code for an RMS scheduler run at the operating system'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 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

```
/* 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.

complexity and low actual execution time, which help minimize the discrepancies between the zero-context switch assumption of RMA and the actual execution of an RMS system.

### 6.5.2 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 utilization than RMS.

The EDF policy is also easy, but in contrast to RMS, EDF updates the priorities of processes at every time quantum. In EDF, priorities are assigned 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. Once EDF has recalculated priorities, the scheduling procedure is the same as that for RMS; the highest-priority ready process is chosen for execution.

Example 6.3 illustrates EDF scheduling in practice.

---

**Example 6.3: Earliest-deadline-first Scheduling**

Consider the following processes:

| Process | Execution time | Period |
|---|---|---|
| P1 | 1 | 3 |
| P2 | 1 | 4 |
| P3 | 2 | 5 |

The least-common multiple of the periods is 60, and the utilization is $1/3 + 1/4 + 1/5 = 0.9833333$. This utilization is too high for RMS, but it can be handled with the EDF schedule. Here is the schedule:

| Time | Running process | Deadlines |
|---|---|---|
| 0 | P1 | |
| 1 | P2 | |
| 2 | P3 | P1 |
| 3 | P3 | P2 |
| 4 | P1 | P3 |
| 5 | P2 | P1 |
| 6 | P1 | |
| 7 | P3 | P2 |

*Continued*

*—Continued*

| Time | Running process | Deadlines |
|---|---|---|
| 8 | P3 | P1 |
| 9 | P1 | P3 |
| 10 | P2 | |
| 11 | P3 | P1, P2 |
| 12 | P3 | |
| 13 | P1 | |
| 14 | P2 | P1, P3 |
| 15 | P1 | P2 |
| 16 | P2 | |
| 17 | P3 | P1 |
| 18 | P3 | |
| 19 | P1 | P2, P3 |
| 20 | P2 | P1 |
| 21 | P1 | |
| 22 | P3 | |
| 23 | P3 | P1, P2 |
| 24 | P1 | P3 |
| 25 | P2 | |
| 26 | P3 | P1 |
| 27 | P3 | P2 |
| 28 | P1 | |
| 29 | P2 | P1, P3 |
| 30 | P1 | |
| 31 | P3 | P2 |
| 32 | P3 | P1 |
| 33 | P1 | |
| 34 | P2 | P3 |
| 35 | P3 | P1, P2 |
| 36 | P1 | |
| 37 | P2 | |
| 38 | P3 | P1 |
| 39 | P1 | P2, P3 |

—*Continued*

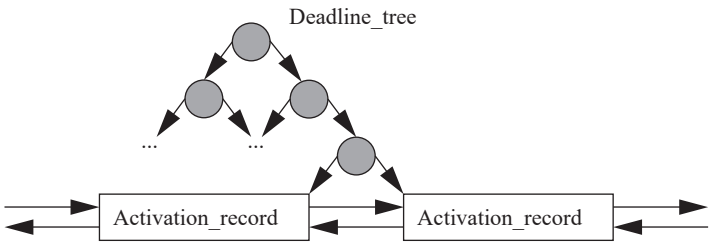
| Time | Running process | Deadlines |
|---|---|---|
| 40 | P2 | |
| 41 | P3 | P1 |
| 42 | P1 | |
| 43 | P3 | P2 |
| 44 | P3 | P1, P3 |
| 45 | P1 | |
| 46 | P2 | |
| 47 | P3 | P1, P2 |
| 48 | P3 | |
| 49 | P1 | P3 |
| 50 | P2 | P1 |
| 51 | P1 | P2 |
| 52 | P3 | |
| 53 | P3 | P1 |
| 54 | P2 | P3 |
| 55 | P1 | P2 |
| 56 | P2 | P1 |
| 57 | P3 | |
| 58 | P3 | |
| 59 | Idle | P1, P2, P3 |

There is one time slot left at the end of this unrolled schedule, which is consistent with our earlier calculation that the CPU utilization is 59/60.

Liu and Layland showed that EDF can achieve 100% utilization. A feasible schedule exists if the CPU utilization (calculated in the same way as that for RMA) is less than or equal to one.

**Implementation**

The implementation of EDF is more complex than RMS code. Fig. 6.13 outlines one way to implement EDF. The major problem is keeping the processes sorted by time to deadline. Because 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 re-sorting 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




**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;
        }
   }
```

**Code**

**FIGURE 6.13**

C code for earliest-deadline-first scheduling.

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 build the sorted list of records, selecting the next executing process is done in a manner like that of RMS. However, dynamic sorting adds complexity to the entire scheduling process. Each update of the sorted list requires $O(n \log n)$ steps. The EDF code is also significantly more complex than the RMS code.

### 6.5.3 RMS vs. EDF

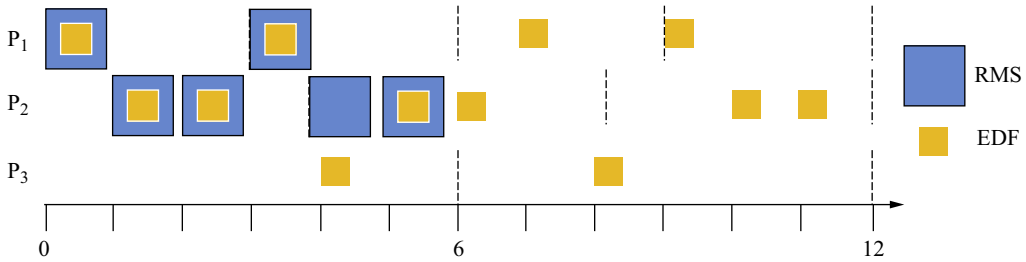
EDF can schedule some task sets that RMS cannot, as shown in the following example.

**Example 6.4 RMS vs. EDF Scheduling**

Consider this example:

| | C | T |
|---|---|---|
| P1 | 1 | 3 |
| P2 | 2 | 4 |
| P3 | 1 | 6 |

The hyperperiod of this task set is 12. Here is an attempt to construct RMS and EDF schedules for these tasks:




EDF successfully schedules tasks with a utilization of 100%. In contrast, RMS misses P3's deadline at Time 6.

### 6.5.4 Shared resources, mutexes, and semaphores

A process may need to do more than read and write values to and from memory. For example, it may need to communicate with an I/O device. It may also use shared memory locations to communicate with other processes. When dealing with **shared resources,** special care must be taken.

**Race condition**

Consider the case in which an I/O device has a flag that must be tested and modified by a process. Problems can arise when other processes also want to access the device. If combinations of events from the two tasks operate on the device in the wrong order, we may create a **critical timing race** or **race condition** that causes erroneous operation. Consider the situation illustrated in Fig. 6.14:

1. Task 1 reads the flag location and sees that it is 0.
2. Task 2 reads the flag location and sees that it is 0.

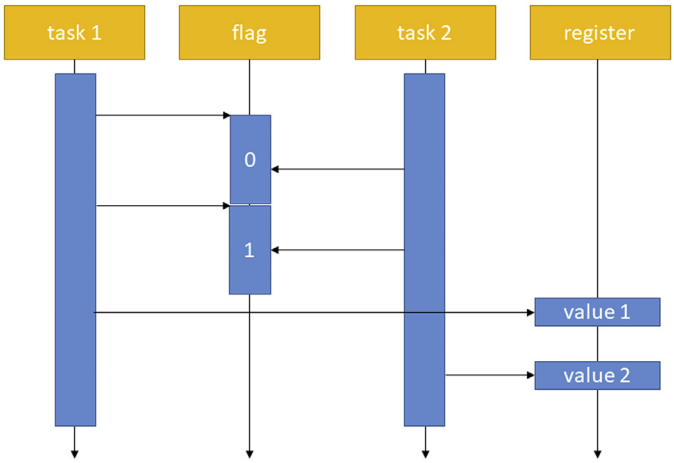



**FIGURE 6.14**

A race condition.

3. Task 1 sets the flag location to 1 and writes the data to the I/O device's data register.
4. Task 2 also sets the flag to one and writes its own data to the device data register, overwriting the data from Task 1.

In this case, both devices thought they were able to write to the device, but Task 1's write was never completed because it was overridden by Task 2.

**Critical sections**

To prevent this type of problem, we need to control the order in which some operations occur. For example, we need to be sure that a task finishes an I/O operation before allowing another task to start its own operation on that I/O device. We do so by enclosing sensitive sections of code in a critical section that executes without interruption.

**Mutex**

We use a **mutex** (for mutual exclusion) to protect a critical section, as illustrated in Fig. 6.15. The mutex is called before each task enters its critical section. In this example, Task 1 calls the mutex first and receives it; the mutex changes state to record that it has been reserved. By the time Task 2 asks for the mutex, Task 1 has already reserved it, so Task 2 waits until Task 1 releases the mutex. At that point, Task 2 proceeds with its critical section, and then, releases the mutex when it is done. Many operating systems allow several mutexes with different names or identifiers to be created to allow multiple critical sections to be managed.

**Semaphores**

A **semaphore** is useful when several copies of a resource are available. The term "semaphore" was derived from railroads, in which a shared section of the track is guarded by signal flags that use semaphores to signal when it is safe to enter the track.

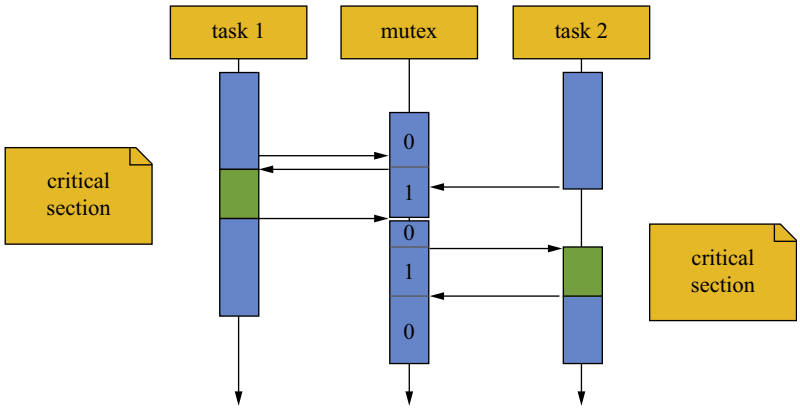



**FIGURE 6.15**

Example application of a mutex to protect critical sections.

A semaphore has two properties: a name allows us to create more than one semaphore in the system, and a count keeps track of the number of units of that type of resource currently in use. The semaphore names are, by tradition, `P()`and `V()`. The `P()` operation is used to gain access. If a unit of the resource is available, that resource's count is incremented. If the count equals the maximum number of units available, `P()` waits until one is released using the `V()` operator, which decrements the count.

**Test-and-set**

To implement mutexes and semaphores, the microprocessor bus must support an **atomic read/write** operation, which is available on some microprocessors. These types of instructions first read a location, and then, set it to a specified value, returning the results of the test. If the location was already set, then the additional set has no effect, but the 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 6.2 describes the advanced reduced instruction set architecture machine (ARM) atomic read/write operation in more detail.

## Programming Example 6.2 Compare-and-swap operations

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

```
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*.

```
        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 ...
```

For example, the code sequence first loads the constant, 1, into `r1` and the address of the semaphore `FLAG1` into register `r2`. It 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 is nonzero, we loop back to try to get the flag once again.

The test-and-set allows us to implement semaphores. 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 is passed, the process can work on the protected memory block. The `V()` operation resets the lock, allowing other processes to access the region by using the `P()` function.

**Critical sections and timing**

Critical sections pose some problems for real-time systems. Because the interrupt system is shut off during the critical section, the timer cannot interrupt, and other processes cannot start to execute. The kernel may also have its own critical sections that keep interrupts from being serviced and other processes from executing.

### 6.5.5 Priority inversion

Shared resources cause a new and subtle scheduling problem: a low-priority process blocks the execution of a higher-priority process by keeping hold of its resource, a phenomenon known as **priority inversion**. Example 6.5 illustrates the problem.

#### Example 6.5 Priority Inversion

A system with three processes: P1 has the highest priority, P3 has the lowest priority, and P2 has a priority between those of P1 and P3. P1 and P3 both use the same shared resource. Processes become ready in this order:

- P3 becomes ready and enters its critical region, reserving the shared resource.
- P2 becomes ready and preempts P3.
- P1 becomes ready. It will preempt P2 and start to run, but only until it reaches its critical section for the shared resource. At that point, it will stop executing.

For P1 to continue, P2 must finish, allowing P3 to resume and finish its critical section. Only when P3 is finished with its critical section can P1 resume.

**Priority inheritance**

The most common method for dealing with priority inversion is **priority inheritance**, which promotes the priority of any process when it requests a resource from the operating system. 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 operating system, and allow other processes to use it. Once the process is finished with the resource, its priority is demoted to its normal value.

### 6.5.6 Scheduling for low power

We can adapt RMS and EDF to consider power consumption. Although the race-to-dark case is more challenging, we have well-understood methods for using **dynamic voltage and frequency scaling** (**DVFS**) in conjunction with priority-based real-time scheduling [Qua07].

Using DVFS with EDF is relatively straightforward. The critical interval determines the worst case that must be handled. We first set the clock speed to meet the performance requirements in the critical interval. We then select the second-most critical interval and set the clock speed, continuing until the entire hyperperiod has been covered.

Using DVFS with RMS is, unfortunately, NP-complete. However, heuristics can be used to compute a good schedule that meets deadlines and reduces power consumption.

### 6.5.7 A closer look at our modeling assumptions

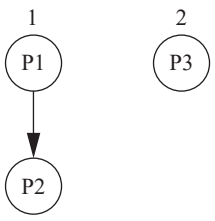
Our analyses of RMS and EDF made some strong assumptions. These assumptions have made the analyses much more tractable, but the predictions of the analysis may not hold up in practice. Because a misprediction may cause a system to miss a critical deadline, it is important to understand the consequences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

RMS assumes that there are no data dependencies between processes. Example 6.6 shows that knowledge of data dependencies can help to use the CPU more efficiently.

---

**Example 6.6: Data Dependencies and Scheduling**

Data dependencies imply that certain combinations of processes can never occur. Consider this simple example [Mal96]:




**Task graph**

| Task | Deadline |
|---|---|
| 1 | 10 |
| 2 | 8 |

**Task rates**

| Process | CPU time |
|---|---|
| P1 | 2 |
| P2 | 1 |
| P3 | 4 |

**Execution times**

We know that P1 and P2 cannot execute at the same time because P1 must finish before P2 can begin. Furthermore, we 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.

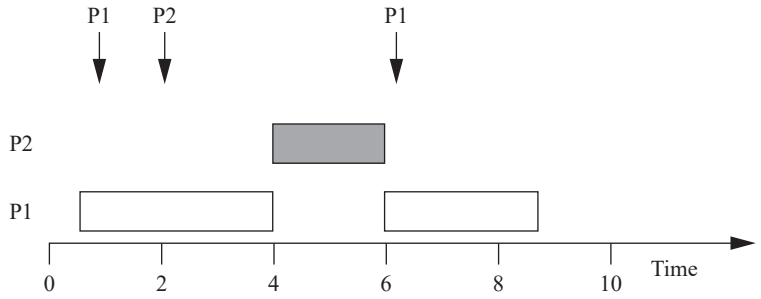
One important simplification we have made is that contexts can be switched in zero time. On the one hand, this is clearly wrong. We must execute instructions to save and restore context, and we must execute additional instructions to implement the scheduling policy. On the other hand, context switching 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 6.7 shows that context switching can, in fact, cause a system to miss a deadline.

### Example 6.7: Scheduling and Context Switching Overhead

Appearing below is a set of processes and their characteristics.

| Process | Execution time | Deadline |
|---|---|---|
| P1 | 3 | 5 |
| P2 | 3 | 10 |

First, let us try to find a schedule assuming that context switching time is zero. The following is a feasible schedule for a sequence of data arrivals that meets all the deadlines:




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 data arrival sequence because we require a total of $2TP_1 + TP_2 = 2 \times (1 + 3) + (1 + 3) = 12$ time units to execute one period of P2 and two periods of P1.

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. These small overhead times are not likely to cause serious scheduling problems. 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.

Rhodes and Wolf [Rho97] developed a computer-assisted design algorithm for implementing processes that compute exact schedules to accurately predict the effects of context switching. Their algorithm selects interrupt-driven and polled implementations for processes on a microprocessor. Polled processes introduce less overhead, but they do not respond to events as quickly as interrupt-driven processes. Furthermore, because adding interrupt levels to a microprocessor usually incurs some cost in added logic, we do not want to use interrupt-driven processes when they are unnecessary. Their algorithm computes precise schedules for the processes, including the overhead for polling or interrupts, as appropriate, and then, uses heuristics to select implementations for the processes. The major heuristic starts with all processes implemented as polled mode, and then, changes processes that miss deadlines to use interrupts. Some iterative improvement steps try various combinations of interrupt-driven processes to eliminate deadline violations. These heuristics minimize the number of processes implemented with interrupts.

Another important assumption we have made is that the process execution time is constant. As seen in Section 5.6, this is not the case; both data-dependent behavior and caching effects can cause large variations in run times. The techniques for bounding the cache-based performance of a single program do not work when multiple programs are in the same cache. The state of the cache depends on the product of the states of all programs executing in the cache, making the state space of the multiple-process system exponentially larger than that of a single program. We discuss this problem in more detail in Section 6.7.

### 6.5.8 Events and sporadic tasks

An event, such as a nonperiodic input, requires processing by a **sporadic task**. That task will, in general, be active, along with a mix of periodic and other sporadic tasks. The first question to ask about the scheduling of a sporadic task is its the response time [Aud93].

An example is shown in Fig. 6.16. The sporadic task in this example is $\tau_3$, which in this case has the lower priority than either $\tau_1$ or $\tau_2$. The deadlines of the three tasks are $D_1,\ D_2,\ D_3$. $\tau_3$ is released while $\tau_2$ is running, causing interference. After $\tau_3$ starts to execute, $\tau_1$ interferes with its execution. The total response time of $\tau_3$ is

$$R_3 = C_3 + B_3 + I_3 \qquad \text{(Eq. 6.6)}$$

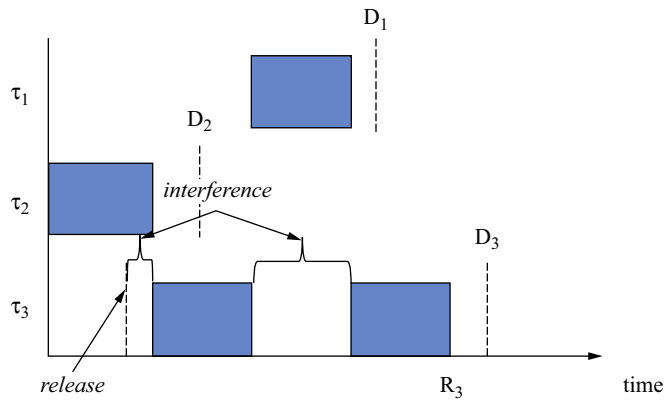



**FIGURE 6.16**

Response time of a sporadic task.

where $R_3$ is the response time of $\tau_3$, $C_3$ is its computation time, $B_3$ is the time that $\tau_3$is blocked by tasks owing to priority inheritance, and $I_3$ is its total interference time. While $C_3$ is known, $B_3$ and $I_3$ depend on the schedule of the higher-priority tasks. We can analyze the components of the response time given an interval during which the event can occur and the corresponding deadline for finishing the computation associated with that deadline. This analysis does not provide a worst case for any possible event timing.

As we saw with RMA, the calculation of interference time for a task $\tau_i$ depends on how many times it executes during the interval of interest. If we let $hp(i)$ represent the tasks with higher priority than $\tau_i$, then

$$I_i = \sum_{j \epsilon hp(i)} \left\lceil \frac{R_j}{T_j} \right\rceil C_j \qquad \text{(Eq. 6.7)}$$

where $T_j$ is the execution time of $\tau_j$. This gives a response time of

$$R_i = C_i + \sum_{j \epsilon hp(i)} \left\lceil \frac{R_j}{T_j} \right\rceil C_j \qquad \text{(Eq. 6.8)}$$

The blocking time, $B_i$, can be determined from the length of the longest critical section of a lower-priority task that may run under priority inheritance during the interval. The response time formula can be extended to consider jitter in the task's release time.

## 6.6 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.

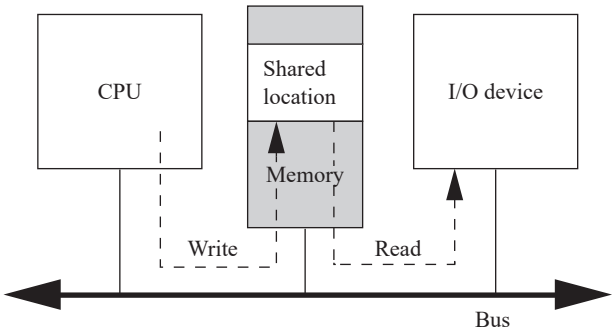



**FIGURE 6.17**

Shared memory communication implemented on a bus.

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. Moreover, the hardware platform may make it easier to implement or more efficient than the other.

### 6.6.1 Shared memory communication

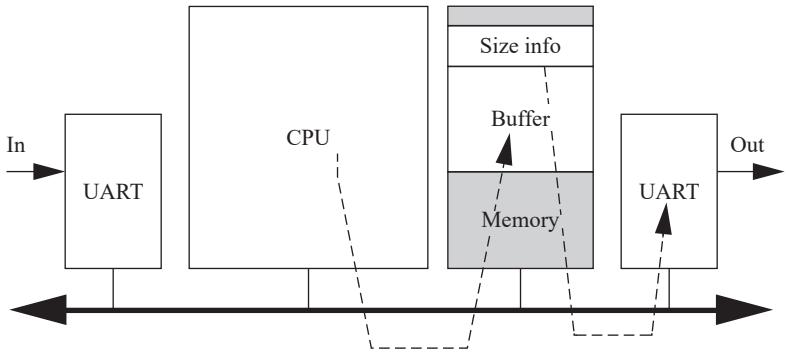
Fig. 6.17 illustrates how shared memory communication works in a bus-based system. Two components, a CPU and an I/O device, communicate through a shared memory location. The software on the CPU has been designed to know the address of the shared location; the shared location has also been loaded into the proper register of the I/O device. If, as in the figure, the CPU wants to send data to the device, it writes them 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 6.8 describes the use of shared memory as a practical communication mechanism.

---

**Example 6.8: 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 Universal Asynchronous Receiver/Transmitter (UART).




The input data arrive at a constant rate and are easy to manage. However, 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 require 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 ensure that the UART does not overrun the buffer.

### 6.6.2 Message passing

Message-passing communication complements the shared memory model. As shown in Fig. 6.18, 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 endpoints. 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.

**Messages**

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 (e.g., lamp, thermostat, faucet, and appliance). The devices must communicate infrequently; furthermore, their physical

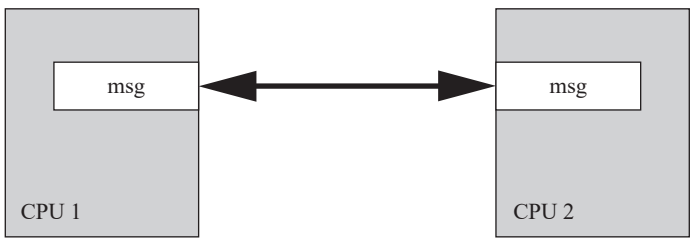



**FIGURE 6.18**

Message-passing communication.

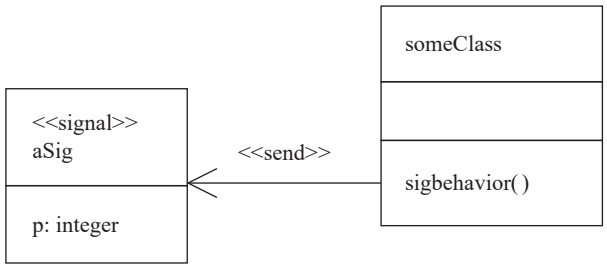



**FIGURE 6.19**

Use of a UML signal.

separation is large enough that we would not naturally think of them as sharing a central pool of memory. Passing communication packets among 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.

**Queues**

A **queue** is a common mechanism for keeping track of messages. The queue uses first in–first out (FIFO) discipline and holds records that represent messages. The FreeRTOS.org system provides a set of queue functions. It allows queues to be created and deleted so that the system may have as many queues as necessary. A queue is described by the data type xQueueHandle and created using xQueueCreate:

```
xQueueHandle q1;
q1 = xQueueCreate(MAX_SIZE,sizeof(msg_record)); /* maximum number of
records in queue, size of each record */
if (q1 == 0) /* error */
...
```

The queue is created using the vQueueDelete() function.

A message is put into the queue using xQueueSend() and received using xQueueReceive():

```
xQueueSend(q1,(void *)msg,(portTickType)0); /* queue, message to
send, final parameter controls timeout */
if (xQueueReceive(q2,&(in_msg),0); /* queue, message received,
timeout */
```

The final parameter in these functions determines how long the queue waits to finish. In the case of a send, the queue may have to wait for something to leave to make room. In the case of the receive, the queue may have to wait for the data to arrive.

### 6.6.3 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. Fig. 6.16 shows the use of a signal in UML. The *sigbehavior()* behavior of the class handles throwing the signal, as indicated by <<*send*>>. The signal object is indicated by the <<*signal*>> stereotype.

### 6.6.4 Mailboxes

A mailbox is a simple mechanism for asynchronous communication. Some architectures define mailbox registers. These mailboxes have a fixed number of bits and can be used for small messages. We can also implement a mailbox using semaphores with the mailbox messages held in the main memory. A very simple version of a mailbox, one that holds only one message at a time, illustrates some important principles of interprocess communication.

For the mailbox to be most useful, we want it to contain two items: the message itself and a mail-ready flag. The flag is true when a message has been put into the mailbox and cleared when the message is removed. This assumes that each message is destined for exactly one recipient. Here is a simple function to put a message into the mailbox, assuming that the system supports only one mailbox used for all messages:

```
void post(message *msg) {
          P(mailbox.sem); /* wait for the mailbox */
          copy(mailbox.data,msg); /* copy the data into the mailbox */
          mailbox.flag = TRUE; /* set the flag to indicate a message
            is ready */
          V(mailbox.sem); /* release the mailbox */
}
```

Here is a function to read from the mailbox:

```
boolean pickup(message *msg) {
        boolean pickup = FALSE; /* local copy of the ready flag */
        P(mailbox.sem); /* wait for the mailbox */
        pickup = mailbox.flag; /* get the flag */
        mailbox.flag = FALSE; /* remember that this message was
           received */
        copy(msg, mailbox.data); /* copy the data into the caller's
           buffer */
        V(mailbox.sem); /* release the flag---can't get the mail if
           we keep the mailbox */
        return(pickup); /* return the flag value */
}
```

Why do we need to use semaphores to protect the read operation? If we don't, a pickup could receive the first part of one message and the second part of another. The semaphores in `pickup()` ensure that a `post()` cannot interleave between the memory reads of the pickup operation.

## 6.7 Evaluating operating system performance

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

- We 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 delays in some cases.
- We have largely ignored interrupts. The latency from when an interrupt is requested to when the device's service is complete is a critical parameter of real-time performance.
- We assumed that we know the execution time of the processes. In fact, we learned in Section 5.7 that program time is not a single number but can be bounded by worst-case and best-case execution times.
- We probably determined the worst-case or best-case times for the processes in isolation. However, in fact, they interact with each other in the cache. Cache conflicts among processes can drastically degrade the process execution time.

We must examine the validity of all these assumptions.

**Context switching time**

Context switching time depends on several factors:

- the amount of CPU context that must be saved; and
- scheduler execution time.

The execution time of the scheduler can, of course, be affected by coding practices. However, the choice of scheduling policy also affects the time required by the schedule to determine the next process to run. Scheduling complexity can be classified as a function of the number of tasks to be scheduled. For example, round-robin scheduling, although it doesn't guarantee deadline satisfaction, is a constant-time algorithm whose execution time is independent of the number of processes. Round-robin is often referred to as an $O(1)$ scheduling algorithm because its execution time is constant independent of the number of tasks. EDF scheduling, in contrast, requires sorting deadlines, which is an $O(n \log n)$ activity.

**Interrupt latency**

**Interrupt latency** for an RTOS is the duration of time from the assertion of a device interrupt to the completion of the device's requested operation. In contrast, when we discussed CPU interrupt latency, we were concerned only with the time the hardware took to start the execution of the interrupt handler. Interrupt latency is critical because data may be lost when an interrupt is not serviced in a timely fashion.

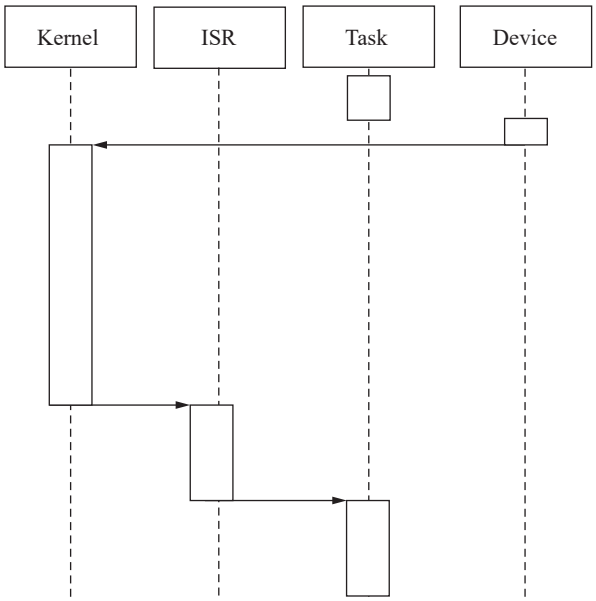



**FIGURE 6.20**

Sequence diagram for RTOS interrupt latency.

Fig. 6.20 shows a sequence diagram of RTOS interrupt latency. A task is interrupted by a device. The interrupt goes to the kernel, which may need to finish a protected operation. Once the kernel can process the interrupt, it calls the interrupt service routine (ISR), which performs the required operations on the device. Once the ISR is completed, the task can resume execution.

Several factors in both hardware and software affect interrupt latency:

- the processor interrupt latency;
- the execution time of the interrupt handler; and
- delays due to RTOS scheduling.

The processor interrupt latency was chosen when the hardware platform was selected; this is often not the dominant factor in overall latency. The execution time of the handler depends on the device operation required, assuming that the interrupt handler code is not poorly designed. This leaves RTOS scheduling delays, which can be the dominant component of RTOS interrupt latency, particularly if the operating system is not designed for low interrupt latency.

**Critical sections and interrupt latency**

The RTOS can delay the execution of an interrupt handler in two ways. First, critical sections in the kernel prevent the RTOS from taking interrupts. A critical section may not be interrupted, so the semaphore code must turn off interrupts. Some operating systems have extensive critical sections that disable interrupt handling for extensive periods. Linux is an example of this phenomenon. Linux was not originally

designed for real-time operation, and interrupt latency was not a major concern. Longer critical sections can improve performance for some types of workloads because they reduce the number of context switches. However, long critical sections cause major problems for interrupts.

Fig. 6.21 shows the effect of critical sections on interrupt latency. If a device interrupts during a critical section, that critical section must finish before the kernel can handle the interrupt. The longer the critical section, the greater the potential delay. Critical sections are an important source of scheduling jitter because a device may interrupt at different points in the execution of processes and hit critical sections at different points.

**Interrupt priorities and interrupt latency**

Second, a higher-priority interrupt may delay a lower-priority interrupt. A hardware interrupt handler runs as part of the kernel, not as a user thread. The priorities for interrupts are determined by hardware, not by the RTOS. Furthermore, any interrupt handler preempts all user threads because interrupts are part of the CPU's fundamental operation. We can reduce the effects of hardware preemption by dividing interrupt handling into two pieces of code. First, a very simple piece of code, usually called an **interrupt service handler** (**ISH**), performs the minimal operations required to respond to the device. The rest of the required processing, which may include updating user buffers or other more complex operations, is performed by a user-mode thread known as an **interrupt service routine** (**ISR**). Because the ISR runs as a thread, the RTOS can use its standard policies to ensure that all the tasks in the system receive their required resources.

**RTOS performance evaluation tools**

Some RTOSs provide simulators or other tools that allow you to view the operation of the processes in the system. These tools will show not only abstract events, such as

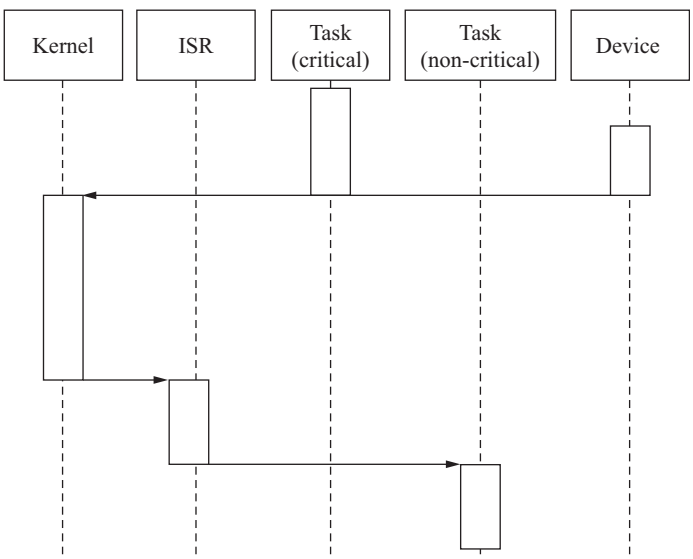



**FIGURE 6.21**

Interrupt latency during a critical section.

processes, but also context switching time, interrupt response time, and other overheads. This sort of view can be helpful in both functional and performance debugging.

**Caches and RTOS performance**

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 system architects lack tools that permit them to analyze the effect of caching. Because 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.

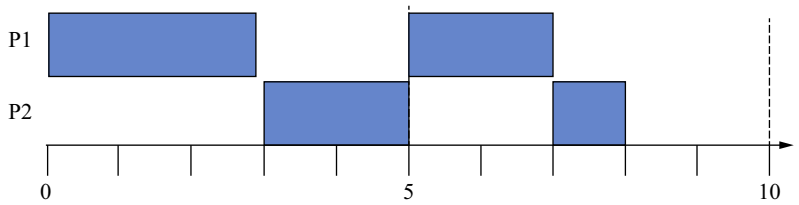
Li and Wolf [Li99] developed a model for estimating the performance of multiple processes that share a cache. In the model, some processes can be given reservations in the cache, such that only a particular process can inhabit a reserved section of the cache; other processes are left to share the cache. We generally want to use cache partitions only for performance-critical processes because cache reservations are wasteful of limited cache space. Performance is estimated by constructing a schedule that considers not just the 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 residing 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 the 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 that 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:

| Process | Worst-case CPU time | Average-case CPU time | Period |
|---|---|---|---|
| P1 | 3 | 2 | 5 |
| P2 | 2 | 1 | 5 |

Each process runs slower when it is not resident in the cache, for example, on its first execution. It runs faster when it is cache-resident. If we can arrange the memory addresses of the processes so that they do not interfere in the cache, then their execution looks like this:




The first execution of each process runs at the worst-case execution time. In their second executions, each process is in the cache, and so runs in less time, leaving extra time at the end of the period.

## 6.8 POSIX real-time operating systems

In this section, we look at the Portable Operating System Interface (POSIX) standard for Unix-style operating systems and its support for real-time systems.

**Posix**

POSIX is a version of the Unix operating system created by the IEEE Computer Society. POSIX-compliant operating systems are source-code compatible. An application can be compiled and run without modification on a new POSIX platform, assuming that the application uses only POSIX-standard functions. Although Unix was not originally designed as an RTOS, POSIX has been extended to support real-time requirements. Many RTOSs are POSIX-compliant, and it serves as a good model for basic RTOS techniques. The POSIX standard has many options, and particular implementations do not have to support all options. The existence of features is determined by C preprocessor variables; for example, the `FOO` option is available if the `_POSIX_FOO` preprocessor variable is defined. All these options are defined in the system, including file `unistd.h`.

The POSIX standard covers a huge expanse of operating systems and applications. The features that are useful for one application may not be applicable to another. Because features generally incur costs in memory size and performance, the appropriate choice of features is important for resource-limited devices. POSIX supports two mechanisms that are important to real-time systems [Gal92]: threads and real-time scheduling.

**POSIX threads**

POSIX supports multiple processes, which is the way that many larger systems operate. However, these processes incur significant memory management overhead. For smaller real-time applications, POSIX provides a thread mechanism [Ope18] under the `pthread.h` include file. An application can use a single process to execute several concurrent threads that share the same memory space.

**Real-time scheduling in POSIX**

POSIX supports two mechanisms for real-time scheduling: one for processes and another for threads [Har03]. Both processes and threads can use any of three different scheduling policies: `SCHED_FIFO`, `SCHED_RR`, and `SCHED_OTHER`. Both `SCHED_FIFO` and

SCHED_RR provide fixed-priority, preemptive scheduling. Unfortunately, the name of SCHED_FIFO is misleading. It is a strict priority-based scheduling scheme in which a process runs until it is preempted or terminated. The term FIFO simply refers to the fact that, within a priority, processes run in first-come, first-served order.

---

Processes and threads may use different schedulers, causing a phenomenon known as **contention scope**. A global scheduling scope ignores process scheduling directives and schedules entirely at the thread level; a mixed scheduling scope first schedules processes and global threads, then local threads.POSIX mutexes

---

POSIX supports mutexes in the pthread.h include file [Ope18]. A mutex is created using pthread_mutex_init(), which returns a pointer to a struct of type pthread_mutex_t. The mutex can be locked using pthread_mutex_lock(), which will block if the mutex is already locked. The function pthread_mutex_trylock() will return immediately if the mutex is already locked. The function pthread_mutex_unlock() is used to unlock the mutex.

**Linux**

The Linux operating system has become increasingly popular as a platform for embedded computing. Linux is a POSIX-compliant operating system that is available as an open source. However, Linux was not originally designed for real-time operations [Yag08, Hal11]. Some versions of Linux may exhibit long interrupt latencies, primarily owing to large critical sections in the kernel that delay interrupt processing. Two methods have been proposed to improve interrupt latency. A **dual-kernel** approach uses a specialized kernel, the **co-kernel**, for real-time processes and a standard kernel for non-real-time processes. All interrupts must go through the co-kernel to ensure that real-time operations are predictable. The other method is a kernel patch that provides priority inheritance to reduce the latency of many kernel operations. These features are enabled using the PREEMPT_RT mode.

## 6.9 Design example: engine control unit

In this section, we design a simple engine control unit (ECU). This unit controls the operation of a fuel-injected engine based on several measurements taken from the running engine.

### 6.9.1 Theory of operation and requirements

We design a basic engine controller for a simple fuel-injected engine [Toy]. As shown in Fig. 6.22, the throttle is the command input. The engine measures throttle, revolutions per minute (RPM), air volume intake, and other variables. The engine controller computes the injector pulse width and spark. This design doesn't compute all the outputs required by a real engine; we only concentrate on a few essentials. We also ignore the different modes of engine operation: warm-up, idle, cruise, and so on. Multimode

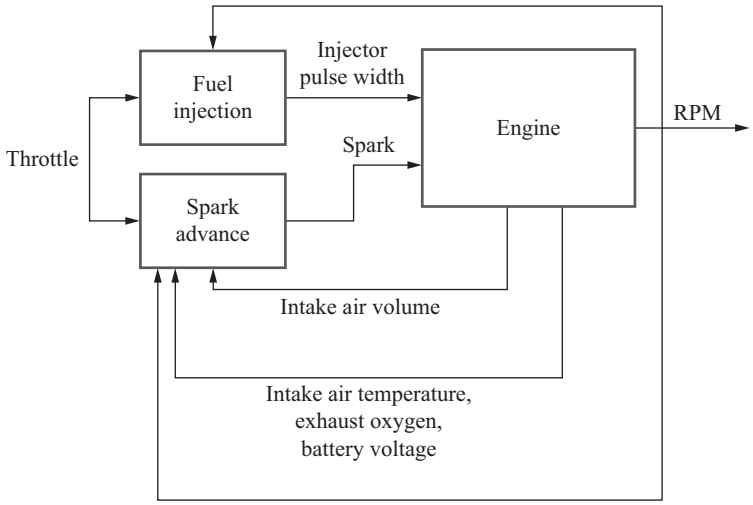



**FIGURE 6.22**

Engine block diagram.

control is one of the principal advantages of engine control units, but we will concentrate here on a single mode to illustrate basic concepts in multirate control.

Our requirements chart for the ECU is shown in Fig. 6.23.

## 6.9.2 Specification

As we saw in Application Example 6.1, the engine controller must deal with processes that happen at different rates. Fig. 6.24 shows the update periods for the different signals.

| Name | ECU |
|---|---|
| Purpose | Engine controller for fuel-injected engine |
| Inputs | Throttle, RPM, intake air volume, intake manifold pressure |
| Outputs | Injector pulse width, spark advance angle |
| Functions | Compute injector pulse width and spark advance angle as a function of throttle, RPM, intake air volume, intake manifold pressure |
| Performance | Injector pulse updated at 2-ms period, spark advance angle updated at 1-ms period |
| Manufacturing cost | Approximately $50 |
| Power | Powered by engine generator |
| Physical size and weight | Approx 4 in × 4 in, less than 1 pound. |

**FIGURE 6.23**

Requirements for the engine controller.

| Signal | Variable name | In/out | Update period (ms) |
|---|---|---|---|
| Throttle | T | input | 2 |
| RPM | NE | input | 2 |
| Intake air volume | VS | input | 25 |
| Injector pulse width | PW | output | 2 |
| Spark advance angle | S | output | 1 |
| Intake air temperature | THA | input | 500 |
| Exhaust oxygen | OX | input | 25 |
| Battery voltage | +B | input | 4 |

**FIGURE 6.24**

Periods for data in the engine controller.

We will use $\Delta NE$ and $\Delta T$ to represent the change in RPM and throttle position, respectively. Our controller computes two output signals: injector pulse width PW and spark advance angle, S [Toy]. It first computes the initial values for these variables:

$$PW = \frac{2.5}{2NE} \times VS \times \frac{1}{10 - K_1 \Delta T} \quad \text{(Eq. 6.9)}$$

$$S = k_2 \times \Delta NE - k_3 VS \quad \text{(Eq. 6.10)}$$

The controller then applies corrections to these initial values:

- As the intake air temperature (THA) increases during engine warm-up, the controller reduces the injection duration.
- As the throttle opens, the controller temporarily increases the injection frequency.
- The controller adjusts the duration up or down based upon readings from the exhaust oxygen sensor (OX).
- The injection duration increases as the battery voltage (+B) drops.

### 6.9.3 System architecture

Fig. 6.25 shows the class diagram for the engine controller. The two major processes, PW and advance angle, compute the control parameters for the spark plugs and injectors.

The control parameters rely on changes in some of the input signals. We use the physical sensor classes to compute these values. Each change must be updated at the variable's sampling rate. The update process is simplified by performing it in a task that runs at the required update rate. Fig. 6.26 shows the state diagram for throttle sensing, which saves both the current value and the change in value of the throttle. We can use a similar control flow to compute changes to the other variables.

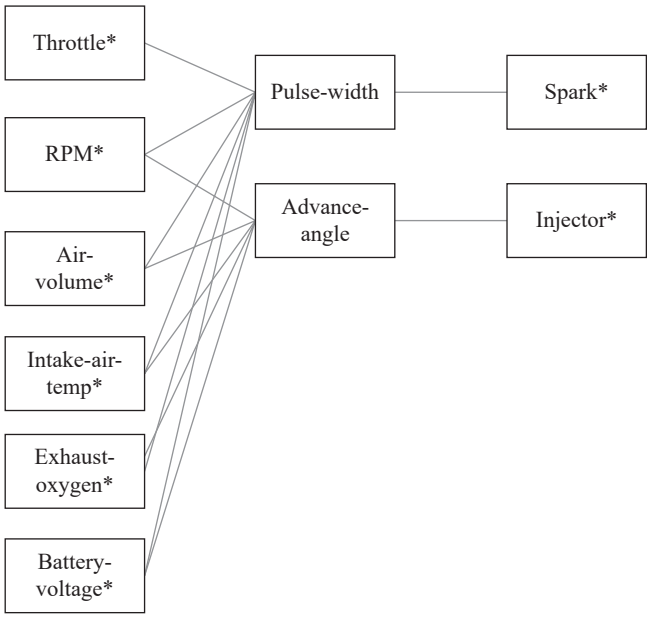



**FIGURE 6.25**

Class diagram for the engine controller.

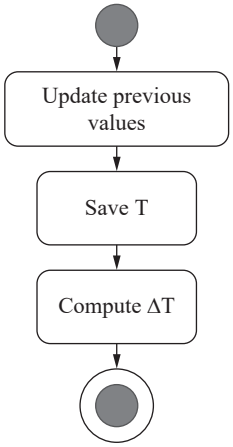



**FIGURE 6.26**

State diagram for throttle position sensing.

Fig. 6.27 shows the state diagram for injector pulse width, and Fig. 6.28 shows the state diagram for spark advance angle. In each case, the value is computed in two stages: first, an initial value, followed by a correction.

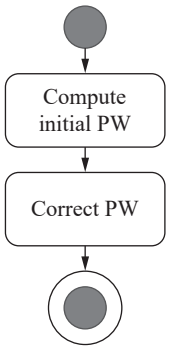



**FIGURE 6.27**

State diagram for injector pulse width.

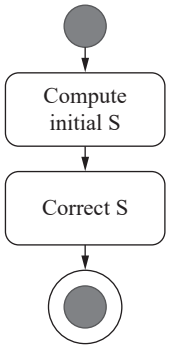



**FIGURE 6.28**

State diagram for spark advance angle.

The pulse width and advance angle processes do not, however, generate the waveforms to drive the spark and injector waveforms. These waveforms must be carefully timed to the engine's current state. Each spark plug and injector must fire at exactly the right time in the engine cycle, accounting for the engine's current speed, as well as the control parameters.

Some engine controller platforms provide hardware units that generate high-rate, changing waveforms. One example is MPC5602 D [Fre11]. The main processor is a PowerPC processor. The enhanced modular I/O subsystem (eMIOS) provides 28 input and output channels controlled by timers. Each channel can perform a variety of functions. The output pulse width and frequency modulation buffered mode automatically generate a waveform whose period and duty cycle can be varied by writing registers in the eMIOS. The details of the waveform timing are then handled by the output channel hardware.

Because these objects must be updated at different rates, their execution will be controlled by an RTOS. Depending on the RTOS latency, we can separate the I/O functions into interrupt service handlers and threads.

### 6.9.4 Component design and testing

The various tasks must be coded to satisfy the requirements of RTOS processes. Variables maintained across task execution, such as the change-of-state variables, must be allocated and saved in appropriate memory locations. The RTOS initialization phase is used to set up the task periods.

Because some of the output variables depend on changes in state, these tasks should be tested with multiple input variable sequences to ensure that both the basic and adjustment calculations are performed correctly.

The Society of Automotive Engineers (SAE) has several standards for automotive software: J2632 for coding practices for C code, J2516 for software development lifecycle, J2640 for software design requirements, and J2734 for software verification and validation.

### 6.9.5 System integration and testing

Engines generate huge amounts of electrical noise that can cripple digital electronics. They also operate over vast temperature ranges: hot during engine operation, and potentially very cold before the engine is started. Any testing performed on an actual engine must be conducted using an engine controller that has been designed to withstand the harsh environment of the engine compartment.

## 6.10 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 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.

However, it is difficult 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 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.
- Preemption 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 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] introduced RMS; their paper became the foundation for real-time systems analysis and design. The book by Liu [Liu00] provides a detailed analysis of real-time scheduling.

## Questions

**Q6-1** Identify activities that operate at different rates in

**a.** a DVD player
**b.** a laser printer
**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 the 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 for a typical process.

**Q6-5** Draw a task graph in which P1 and P2 each process separate inputs, and then, pass their results onto P3 for further processing.

**Q6-6** Compute the utilization for these task sets:

**a.** P1: period = 1 s, execution time = 10 ms; P2: period = 100 ms, execution time = 10 ms.

**b.** P1: period = 100 ms; execution time = 25 ms; P2: period = 80 ms; execution time = 15 ms; P3: period = 40 ms; execution time = 5 ms.

**c.** P1: period = 10 ms; execution time = 1 ms; P2: period = 1 ms; execution time = 0.2 ms; P3: period = 0.2 ms; execution time = 0.05 ms.

**Q6-7** What factors provide a lower bound on the period at which the system timer interrupts for preemptive context switching?

**Q6-8** What factors provide an upper bound on the period at which the system timer interrupts for preemptive context switching?

**Q6-9** What is the distinction between the ready and waiting states of process scheduling?

**Q6-10** A set of processes changes state, as shown over the interval [0,1 ms]. P1 has the highest priority, and P3 has the lowest priority. Draw a UML sequence diagram showing the state of all the processes during this interval.

| t | Process states |
|---|---|
| 0 | P1 = waiting, P2 = waiting, P3 = executing |
| 0.1 | P1 = ready |
| 0.15 | P2 = ready |
| 0.2 | P1 = waiting |
| 0.3 | P1 = ready, P3 = ready |
| 0.4 | P1 = waiting |
| 0.5 | P2 = waiting |
| 0.6 | P3 = waiting |
| 0.8 | P2 = ready, P3 = ready |
| 0.9 | P2 = waiting |

**Q6-11** Provide examples of

**a.** blocking interprocess communication

**b.** nonblocking interprocess communication

**Q6-12** For the following periodic processes, what is the shortest interval we must examine to see all combinations of deadlines?

**a.**

| Process | Deadline |
|---|---|
| P1 | 2 |
| P2 | 5 |
| P3 | 10 |

**b.**

| Process | Deadline |
|---|---|
| P1 | 2 |
| P2 | 4 |
| P3 | 5 |
| P4 | 10 |

**c.**

| Process | Deadline |
|---|---|
| P1 | 3 |
| P2 | 4 |
| P3 | 5 |
| P4 | 6 |
| P5 | 10 |

**Q6-13** Consider the following system of periodic processes executing on a single CPU:

| Process | Execution time | Deadline |
|---|---|---|
| P1 | 4 | 200 |
| P2 | 1 | 10 |
| P3 | 2 | 40 |
| P4 | 6 | 50 |

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

**Q6-14** Given the following set of periodic processes running on a single CPU (P1 has highest priority), what is the maximum execution time *x* of P3 for which all the processes will be schedulable using EDF?

| Process | Execution time | Deadline |
|---|---|---|
| P1 | 1 | 10 |
| P2 | 3 | 25 |
| P3 | x | 50 |
| P4 | 10 | 100 |

**Q6-15** A set of periodic processes is scheduled using RMS; P1 has the highest priority. For the process execution times and periods shown below, show the state of the processes at the critical instant for each process.

**a.** P1
**b.** P2
**c.** P3

| Process | Time | Deadline |
|---|---|---|
| P1 | 1 | 4 |
| P2 | 1 | 5 |
| P3 | 1 | 10 |

**Q6-16** For the given periodic process execution times and periods (P1 has the highest priority), 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

| Process | Time | Deadline |
|---|---|---|
| P1 | 1 | 5 |
| P2 | 2 | 10 |
| P3 | 2 | 25 |
| P5 | 5 | 50 |

**Q6-17** 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 least-common multiple of the periods of the processes. P1 has the highest priority, and time starts at t = 0.

| Process | Time | Deadline |
|---|---|---|
| P1 | 1 | 3 |
| P2 | 1 | 4 |
| P3 | 1 | 12 |

**Q6-18** 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 least-common multiple of the periods of the processes. P1 has the highest priority and time starts at t = 0.

| Process | Time | Deadline |
|---|---|---|
| P1 | 1 | 3 |
| P2 | 1 | 4 |
| P3 | 2 | 6 |

**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 an interval equal to the least-common multiple of the periods of the processes. P1 has the highest priority and time starts at t = 0.

| Process | Time | Deadline |
|---|---|---|
| P1 | 1 | 2 |
| P2 | 1 | 3 |
| P3 | 2 | 10 |

**Q6-20** 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 RMS (no data dependencies).

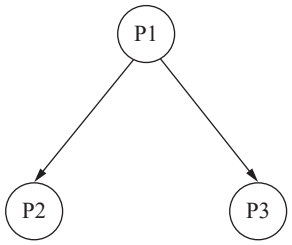
**b.** Schedule the processes, taking advantage of the data dependencies. By how much is the CPU utilization reduced?

| Process | Execution time |
|---|---|
| P1 | 2 |
| P2 | 1 |
| P3 | 2 |

**Q6-21** 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.




| Process | Time | Deadline |
|---|---|---|
| P1 | 2 | 30 |
| P2 | 5 | 40 |
| P3 | 7 | 120 |
| P4 | 5 | 60 |
| P5 | 1 | 15 |

**Q6-22** 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.

| Process | Time | Deadline |
|---|---|---|
| P1 | 1 | 5 |
| P2 | 1 | 10 |
| P3 | 2 | 20 |
| P4 | 10 | 50 |
| P5 | 7 | 100 |

**Q6-23** 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?

**Q6-24** A system has two processes P1 and P2, with P1 having higher priority. They share an I/O device ADC. If P2 acquires the ADC from the RTOS and P1 becomes ready, how does the RTOS schedule the processes using priority inheritance?

**Q6-25** Explain the roles of IRSs and interrupt service handlers in interrupt handling.

**Q6-26** Briefly explain the dual-kernel approach to RTOS design.

## Lab exercises

**L6-1** Using your favorite operating system, write code to spawn a process that writes "Hello, world!" to the screen or flashes a light-emitting diode (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.

**L6-8** Measure the effect of cache conflicts on real-time execution time. First, set up your system to measure the execution time of your real-time process. Next, add a background process to the system. One version of the background process should do nothing; another should do some work that will invalidate as many of the cache entries as possible.