11 Scheduling

Contents

Chapter 10 has explained multitasking, where multiple imperative tasks execute concurrently, either interleaved on a single processor or in parallel on multiple processors. When there are fewer processors than tasks (the usual case), or when tasks must be performed at a particular time, a **scheduler** must intervene. A scheduler makes the decision about what to do next at certain points in time, such as the time when a processor becomes available.

Real-time systems are collections of tasks where in addition to any ordering constraints imposed by precedences between the tasks, there are also timing constraints. These constraints relate the execution of a task to **real time**, which is physical time in the environment of the computer executing the task. Typically, tasks have deadlines, which are values of physical time by which the task must be completed. More generally, real-time programs can have all manner of **timing constraint**s, not just deadlines. For example, a task may be required to be executed no earlier than a particular time; or it may be required to be executed no more than a given amount of time after another task is executed; or it may be required to execute periodically with some specified period. Tasks may be dependent on one another, and may cooperatively form an application. Or they may be unrelated except that they share processor resources. All of these situations require a scheduling strategy.

11.1 Basics of Scheduling

In this section, we discuss the range of possibilities for scheduling, the properties of tasks that a scheduler uses to guide the process, and the implementation of schedulers in an operating system or microkernel.

11.1.1 Scheduling Decisions

A scheduler decides what task to execute next when faced with a choice in the execution of a concurrent program or set of programs. In general, a scheduler may have more than one processor available to it (for example in a multicore system). A **multiprocessor scheduler** needs to decide not only which task to execute next, but also on which processor to execute it. The choice of processor is called **processor assignment**.

A **scheduling decision** is a decision to execute a task, and it has the following three parts:

- **assignment**: which processor should execute the task;
- **ordering**: in what order each processor should execute its tasks; and
- **timing**: the time at which each task executes.

Each of these three decisions may be made at **design time**, before the program begins executing, or at **run time**, during the execution of the program.

Depending on when the decisions are made, we can distinguish a few different types of schedulers (Lee and Ha, 1989). A **fully-static scheduler** makes all three decisions at design time. The result of scheduling is a precise specification for each processor of what to do when. A fully-static scheduler typically does not need semaphores or locks. It can use timing instead to enforce mutual exclusion and precedence constraints. However, fully-static schedulers are difficult to realize with most modern microprocessors because the time it takes to execute a task is difficult to predict precisely, and because tasks will typically have data-dependent execution times (see Chapter 15).

A **static order scheduler** performs the task assignment and ordering at design time, but defers until run time the decision of when in physical time to execute a task. That decision may be affected, for example, by whether a mutual exclusion lock can be acquired, or whether precedence constraints have been satisfied. In static order scheduling, each processor is given its marching orders before the program begins executing, and it simply executes those orders as quickly as it can. It does not, for example, change the order of tasks based on the state of a semaphore or a lock. A task itself, however, may block on a semaphore or lock, in which case it blocks the entire sequence of tasks on that processor. A static order scheduler is often called an **off-line scheduler**.

A **static assignment scheduler** performs the assignment at design time and everything else at run time. Each processor is given a set of tasks to execute, and a **run-time scheduler** decides during execution what task to execute next.

A **fully-dynamic scheduler** performs all decisions at run time. When a processor becomes available (e.g., it finishes executing a task, or a task blocks acquiring a mutex), the scheduler makes a decision at that point about what task to execute next on that processor. Both static assignment and fully-dynamic schedulers are often called **on-line scheduler**s.

There are, of course, other scheduler possibilities. For example, the assignment of a task may be done once for a task, at run time just prior to the first execution of the task. For subsequent runs of the same task, the same assignment is used. Some combinations do not make much sense. For example, it does not make sense to determine the time of execution of a task at design time and the order at run time.

A **preemptive** scheduler may make a scheduling decision during the execution of a task, assigning a new task to the same processor. That is, a task may be in the middle of executing when the scheduler decides to stop that execution and begin execution of another task. The interruption of the first task is called **preemption**. A scheduler that always lets tasks run to completion before assigning another task to execute on the same processor is called a **non-preemptive** scheduler.

In preemptive scheduling, a task may be preempted if it attempts to acquire a mutual exclusion lock and the lock is not available. When this occurs, the task is said to be **blocked** on the lock. When another task releases the lock, the blocked task may resume. Moreover, a task may be preempted when it releases a lock. This can occur for example if there is a higher priority task that is blocked on the lock. We will assume in this chapter well-structured programs, where any task that acquires a lock eventually releases it.

11.1.2 Task Models

For a scheduler to make its decisions, it needs some information about the structure of the program. A typical assumption is that the scheduler is given a finite set T of tasks. Each task may be assumed to be finite (it terminates in finite time), or not. A typical operating system scheduler does not assume that tasks terminate, but real-time schedulers often do. A scheduler may make many more assumptions about tasks, a few of which we discuss in this section. The set of assumptions is called the **task model** of the scheduler.

Some schedulers assume that all tasks to be executed are known before scheduling begins, and some support **arrival of tasks**, meaning tasks become known to the scheduler as other tasks are being executed. Some schedulers support scenarios where each task $\tau \in T$ executes repeatedly, possibly forever, and possibly periodically. A task could also be **sporadic**, which means that it repeats, and its timing is irregular, but that there is a lower bound on the time between task executions. In situations where a task $\tau \in T$ executes repeatedly, we need to make a distinction between the task τ and the **task execution**s $\tau_1, \tau_2, \cdots$. If each task executes exactly once, then no such distinction is necessary.

Task executions may have **precedence constraints**, a requirement that one execution precedes another. If execution i must precede j, we can write $i < j$. Here, i and j may be distinct executions of the same task, or executions of different tasks.

A task execution i may have some **preconditions** to start or resume execution. These are conditions that must be satisfied before the task can execute. When the preconditions

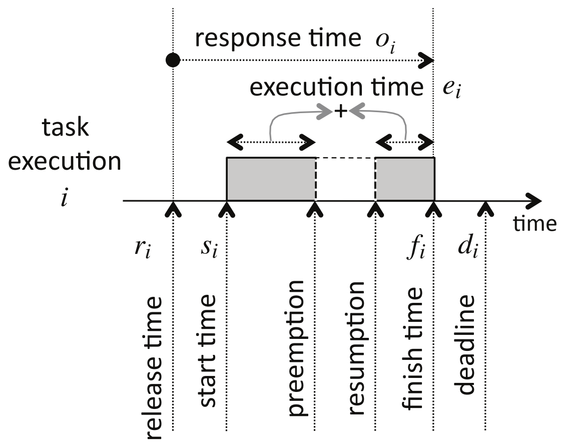


Figure 11.1: Summary of times associated with a task execution.

are satisfied, the task execution is said to be **enabled**. Precedences, for example, specify preconditions to start a task execution. Availability of a lock may be a precondition for resumption of a task.

We next define a few terms that are summarized in Figure 11.1.

For a task execution i, we define the **release time** r_i (also called the **arrival time**) to be the earliest time at which a task is enabled. We define the **start time** s_i to be the time at which the execution actually starts. Obviously, we require that

$$s_i \geq r_i \, .$$

We define the **finish time** f_i to be the time at which the task completes execution. Hence,

$$f_i \geq s_i \, .$$

The **response time** o_i is given by

$$o_i = f_i - r_i \, .$$

The response time, therefore, is the time that elapses between when the task is first enabled and when it completes execution.

The **execution time** e_i of τ_i is defined to be the total time that the task is actually executing. It does not include any time that the task may be blocked or preempted. Many scheduling strategies assume (often unrealistically) that the execution time of a task is known and fixed. If the execution time is variable, it is common to assume (often unrealistically) that the worst-case execution time (WCET) is known. Determining execution times of software can be quite challenging, as discussed in Chapter 15.

The **deadline** d_i is the time by which a task must be completed. Sometimes, a deadline is a real physical constraint imposed by the application, where missing the deadline is considered an error. Such a deadline is called a **hard deadline**. Scheduling with hard deadlines is called **hard real-time scheduling**.

Often, a deadline reflects a design decision that need not be enforced strictly. It is better to meet the deadline, but missing the deadline is not an error. Generally it is better to not miss the deadline by much. This case is called **soft real-time scheduling**.

A scheduler may use **priority** rather than (or in addition to) a deadline. A priority-based scheduler assumes each task is assigned a number called a priority, and the scheduler will always choose to execute the task with the highest priority (which is often represented by the lowest priority number). A **fixed priority** is a priority that remains constant over all executions of a task. A **dynamic priority** is allowed to change for during execution.

A **preemptive priority-based scheduler** is a scheduler that supports arrivals of tasks and at all times is executing the enabled task with the highest priority. A **non-preemptive priority-based scheduler** is a scheduler that uses priorities to determine which task to execute next after the current task execution completes, but never interrupts a task during execution to schedule another task.

11.1.3 Comparing Schedulers

The choice of scheduling strategy is governed by considerations that depend on the goals of the application. A rather simple goal is that all task executions meet their deadlines, $f_i \leq d_i$. A schedule that accomplishes this is called a **feasible schedule**. A scheduler that yields a feasible schedule for any task set (that conforms to its task model) for which there is a feasible schedule is said to be **optimal with respect to feasibility**.

A criterion that might be used to compare scheduling algorithms is the achievable processor **utilization**. The utilization is the percentage of time that the processor spends executing tasks (vs. being idle). This metric is most useful for tasks that execute periodically. A scheduling algorithm that delivers a feasible schedule whenever processor utilization is less than or equal to 100% is obviously optimal with respect to feasibility. It only fails to deliver a feasible schedule in circumstances where *all* scheduling algorithms will fail to deliver a feasible schedule.

Another criterion that might be used to compare schedulers is the maximum **lateness**, defined for a set of task executions T as

$$L_{\max} = \max_{i \in T}(f_i - d_i) \ .$$

For a feasible schedule, this number is zero or negative. But maximum lateness can also be used to compare infeasible schedules. For soft real-time problems, it may be tolerable for this number to be positive, as long as it does not get too large.

A third criterion that might be used for a finite set T of task executions is the **total completion time** or **makespan**, defined by

$$M = \max_{i \in T} f_i - \min_{i \in T} r_i \ .$$

If the goal of scheduling is to minimize the makespan, this is really more of a performance goal rather than a real-time requirement.

11.1.4 Implementation of a Scheduler

A scheduler may be part of a compiler or code generator (for scheduling decisions made at design time), part of an operating system or microkernel (for scheduling decisions made at run time), or both (if some scheduling decisions are made at design time and some at run time).

A run-time scheduler will typically implement tasks as threads (or as processes, but the distinction is not important here). Sometimes, the scheduler assumes these threads complete in finite time, and sometimes it makes no such assumption. In either case, the scheduler is a procedure that gets invoked at certain times. For very simple, non-preemptive schedulers, the scheduling procedure may be invoked each time a task completes. For preemptive schedulers, the scheduling procedure is invoked when any of several things occur:

- A timer interrupt occurs, for example at a jiffy interval.
- An I/O interrupt occurs.
- An operating system service is invoked.
- A task attempts to acquire a mutex.

- A task tests a semaphore.

For interrupts, the scheduling procedure is called by the interrupt service routine (ISR). In the other cases, the scheduling procedure is called by the operating system procedure that provides the service. In both cases, the stack contains the information required to resume execution. However, the scheduler may choose not to simply resume execution. I.e., it may choose not to immediately return from the interrupt or service procedure. It may choose instead to preempt whatever task is currently running and begin or resume another task.

To accomplish this preemption, the scheduler needs to record the fact that the task is preempted (and, perhaps, why it is preempted), so that it can later resume this task. It can then adjust the stack pointer to refer to the state of the task to be started or resumed. At that point, a return is executed, but instead of resuming execution with the task that was preempted, execution will resume for another task.

Implementing a preemptive scheduler can be quite challenging. It requires very careful control of concurrency. For example, interrupts may need to be disabled for significant parts of the process to avoid ending up with a corrupted stack. This is why scheduling is one of the most central functions of an operating system kernel or microkernel. The quality of the implementation strongly affects system reliability and stability.

11.2 Rate Monotonic Scheduling

Consider a scenario with $T = \{\tau_1, \tau_2, \cdots, \tau_n\}$ of n tasks, where the tasks must execute periodically. Specifically, we assume that each task τ_i must execute to completion exactly once in each time interval p_i. We refer to p_i as the **period** of the task. Thus, the deadline for the j-th execution of τ_i is $r_{i,1} + jp_i$, where $r_{i,1}$ is the release time of the first execution.

Liu and Layland (1973) showed that a simple preemptive scheduling strategy called **rate monotonic** (**RM**) scheduling is optimal with respect to feasibility among fixed priority uniprocessor schedulers for the above task model. This scheduling strategy gives higher priority to a task with a smaller period.

The simplest form of the problem has just two tasks, $T = \{\tau_1, \tau_2\}$ with execution times e_1 and e_2 and periods p_1 and p_2, as depicted in Figure 11.2. In the figure, the execution time e_2 of task τ_2 is longer than the period p_1 of task τ_1. Thus, if these two tasks are to execute on the same processor, then it is clear that a non-preemptive scheduler will not

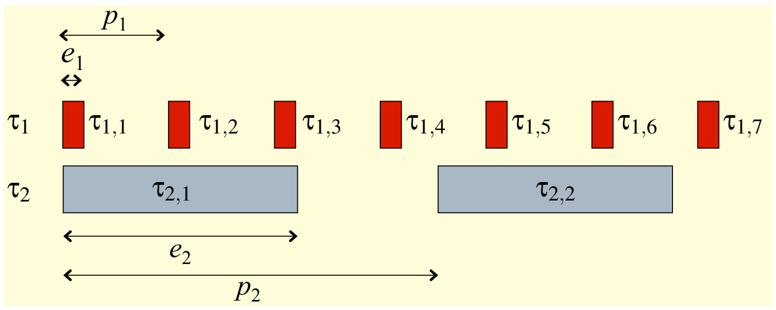


Figure 11.2: Two periodic tasks $T = \{\tau_1, \tau_2\}$ with execution times e_1 and e_2 and periods p_1 and p_2.

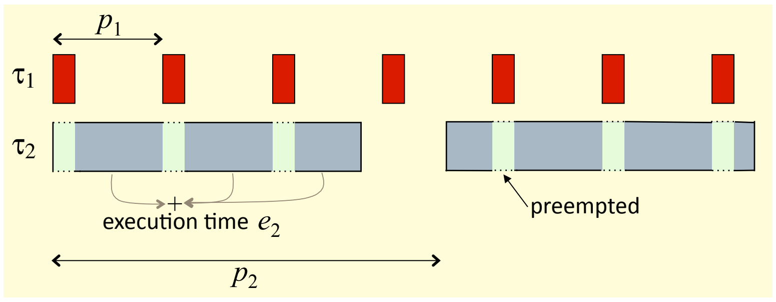


Figure 11.3: Two periodic tasks $T = \{\tau_1, \tau_2\}$ with a preemptive schedule that gives higher priority to τ_1.

yield a feasible schedule. If task τ_2 must execute to completion without interruption, then task τ_1 will miss some deadlines.

A preemptive schedule that follows the rate monotonic principle is shown in Figure 11.3. In that figure, task τ_1 is given higher priority, because its period is smaller. So it executes at the beginning of each period interval, regardless of whether τ_2 is executing. If τ_2 is executing, then τ_1 preempts it. The figure assumes that the time it takes to perform the preemption, called the **context switch time**, is negligible.[1] This schedule is feasible, whereas if τ_2 had been given higher priority, then the schedule would not be feasible.

For the two task case, it is easy to show that among all preemptive fixed priority schedulers, RM is optimal with respect to feasibility, under the assumed task model with negligible context switch time. This is easy to show because there are only two fixed priority schedules for this simple case, the RM schedule, which gives higher priority to task τ_1, and the non-RM schedule, which gives higher priority to task τ_2. To show optimality, we simply need to show that if the non-RM schedule is feasible, then so is the RM schedule.

[1]The assumption that context switch time is negligible is problematic in practice. On processors with caches, a context switch often causes substantial cache-related delays. In addition, the operating system overhead for context switching can be substantial.

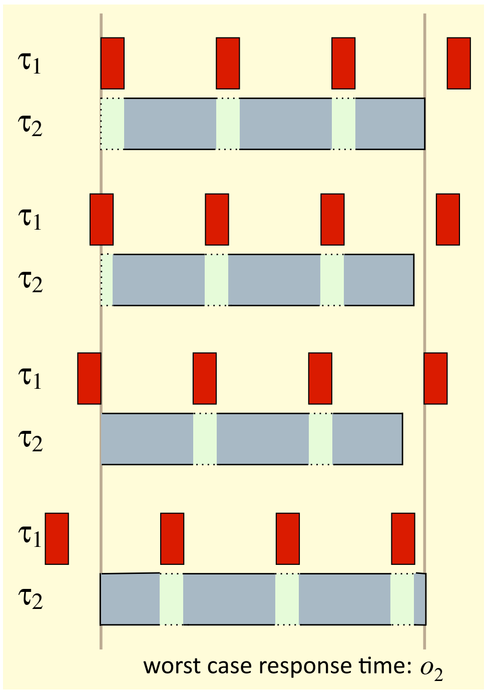


Figure 11.4: Response time o_2 of task τ_2 is worst when its cycle starts at the same time that the cycle of τ_1 starts.

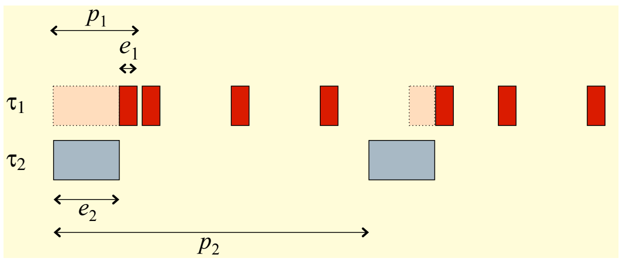


Figure 11.5: The non-RM schedule gives higher priority to τ_2. It is feasible if and only if $e_1 + e_2 \leq p_1$ for this scenario.

Before we can do this, we need to consider the possible alignments of task executions that can affect feasibility. As shown in Figure 11.4, the response time of the lower priority task is worst when its starting phase matches that of higher priority tasks. That is, the worst-case scenario occurs when all tasks start their cycles at the same time. Hence, we only need to consider this scenario.

Under this worst-case scenario, where release times align, the non-RM schedule is feasible if and only if

$$e_1 + e_2 \leq p_1 \ . \tag{11.1}$$

This scenario is illustrated in Figure 11.5. Since task τ_1 is preempted by τ_2, for τ_1 to not miss its deadline, we require that $e_2 \leq p_1 - e_1$, so that τ_2 leaves enough time for τ_1 to execute before its deadline.

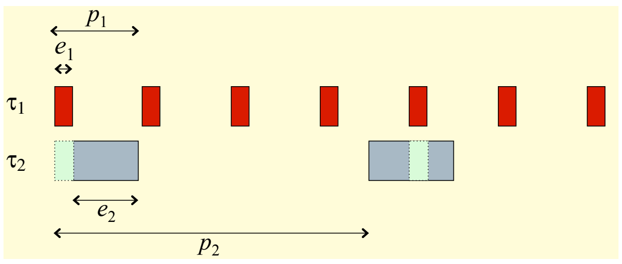


Figure 11.6: The RM schedule gives higher priority to τ_1. For the RM schedule to be feasible, it is sufficient, but not necessary, for $e_1 + e_2 \leq p_1$.

To show that RM is optimal with respect to feasibility, all we need to do is show that if the non-RM schedule is feasible, then the RM schedule is also feasible. Examining Figure 11.6, it is clear that if equation (11.1) is satisfied, then the RM schedule is feasible. Since these are the only two fixed priority schedules, the RM schedule is optimal with respect to feasibility. The same proof technique can be generalized to an arbitrary number of tasks, yielding the following theorem (Liu and Layland, 1973):

Theorem 11.1. *Given a preemptive, fixed priority scheduler and a finite set of repeating tasks $T = \{\tau_1, \tau_2, \cdots, \tau_n\}$ with associated periods $p_1, p_2, \cdots, p_n$ and no precedence constraints, if any priority assignment yields a feasible schedule, then the rate monotonic priority assignment yields a feasible schedule.*

RM schedules are easily implemented with a timer interrupt with a time interval equal to the greatest common divisor of the periods of the tasks. They can also be implemented with multiple timer interrupts.

It turns out that RM schedulers cannot always achieve 100% utilization. In particular, RM schedulers are constrained to have fixed priority. This constraint results in situations where a task set that yields a feasible schedule has less than 100% utilization and yet cannot tolerate any increase in execution times or decrease in periods. This means that there are idle processor cycles that cannot be used without causing deadlines to be missed. An example is studied in Exercise 3.

Fortunately, Liu and Layland (1973) show that this effect is bounded. First note that the utilization of n independent tasks with execution times e_i and periods p_i can be written

$$\mu = \sum_{i=1}^{n} \frac{e_i}{p_i}.$$

If $\mu = 1$, then the processor is busy 100% of the time. So clearly, if $\mu > 1$ for any task set, then that task set has no feasible schedule. Liu and Layland (1973) show that if μ is less than or equal to a **utilization bound** given by

$$\mu \leq n(2^{1/n} - 1), \tag{11.2}$$

then the RM schedule is feasible.

To understand this (rather remarkable) result, consider a few cases. First, if $n = 1$ (there is only one task), then $n(2^{1/n} - 1) = 1$, so the result tells us that if utilization is 100% or less, then the RM schedule is feasible. This is obvious, of course, because with only one task, $\mu = e_1/p_1$, and clearly the deadline can only be met if $e_1 \leq p_1$.

If $n = 2$, then $n(2^{1/n} - 1) \approx 0.828$. Thus, if a task set with two tasks does not attempt to use more than 82.8% of the available processor time, then the RM schedule will meet all deadlines.

As n gets large, the utilization bound approaches $\ln(2) \approx 0.693$. That is

$$\lim_{n\to\infty} n(2^{1/n} - 1) = \ln(2) \approx 0.693.$$

This means that if a task set with any number of tasks does not attempt to use more than 69.3% of the available processor time, then the RM schedule will meet all deadlines.

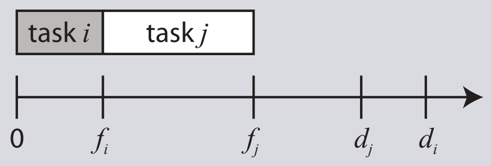
In the next section, we relax the fixed-priority constraint and show that dynamic priority schedulers can do better than fixed priority schedulers, in the sense that they can achieve higher utilization. The cost is a somewhat more complicated implementation.

11.3 Earliest Deadline First

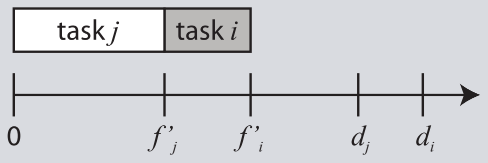
Given a finite set of non-repeating tasks with deadlines and no precedence constraints, a simple scheduling algorithm is **earliest due date** (**EDD**), also known as **Jackson's algorithm** (Jackson, 1955). The EDD strategy simply executes the tasks in the same order as their deadlines, with the one with the earliest deadline going first. If two tasks have the same deadline, then their relative order does not matter.

Theorem 11.2. *Given a finite set of non-repeating tasks $T = \{\tau_1, \tau_2, \cdots, \tau_n\}$ with associated deadlines $d_1, d_2, \cdots, d_n$ and no precedence constraints, an EDD schedule is optimal in the sense that it minimizes the maximum lateness, compared to all other possible orderings of the tasks.*

Proof. This theorem is easy to prove with a simple **interchange argument**. Consider an arbitrary schedule that is not EDD. In such a schedule, because it is not EDD, there must be two tasks τ_i and τ_j where τ_i immediately precedes τ_j, but $d_j < d_i$. This is depicted here:



Since the tasks are independent (there are no precedence constraints), reversing the order of these two tasks yields another valid schedule, depicted here:



We can show that the new schedule has a maximum lateness no greater than that of the original schedule. If we repeat the above interchange until there are no more tasks eligible for such an interchange, then we have constructed the EDD schedule. Since this schedule has a maximum lateness no greater than that of the original schedule, the EDD schedule has the minimum maximum lateness of all schedules.

To show that the second schedule has a maximum lateness no greater than that of the first schedule, first note that if the maximum lateness is determined by some task other than τ_i or τ_j, then the two schedules have the same maximum lateness, and we are done. Otherwise, it must be that the maximum lateness of the first schedule is

$$L_{\max} = \max(f_i - d_i, f_j - d_j) = f_j - d_j,$$

where the latter equality is obvious from the picture and follows from the facts that $f_i \leq f_j$ and $d_j < d_i$.

The maximum lateness of the second schedule is given by

$$L'_{\max} = \max(f'_i - d_i, f'_j - d_j)\ .$$

Consider two cases:

Case 1: $L'_{\max} = f'_i - d_i$. In this case, since $f'_i = f_j$, we have

$$L'_{\max} = f_j - d_i \leq f_j - d_j\ ,$$

where the latter inequality follows because $d_j < d_i$. Hence, $L'_{\max} \leq L_{\max}$.

Case 2: $L'_{\max} = f'_j - d_j$. In this case, since $f'_j \leq f_j$, we have

$$L'_{\max} \leq f_j - d_j\ ,$$

and again $L'_{\max} \leq L_{\max}$.

In both cases, the second schedule has a maximum lateness no greater than that of the first schedule. QED.

$\square$

EDD is also optimal with respect to feasibility, because it minimizes the maximum lateness. However, EDD does not support arrival of tasks, and hence also does not support periodic or repeated execution of tasks. Fortunately, EDD is easily extended to support these, yielding what is known as **earliest deadline first** (**EDF**) or **Horn's algorithm** (Horn, 1974).

Theorem 11.3. *Given a set of n independent tasks $T = \{\tau_1, \tau_2, \cdots, \tau_n\}$ with associated deadlines $d_1, d_2, \cdots, d_n$ and arbitrary arrival times, any algorithm that at any instant executes the task with the earliest deadline among all arrived tasks is optimal with respect to minimizing the maximum lateness.*

The proof of this uses a similar interchange argument. Moreover, the result is easily extended to support an unbounded number of arrivals. We leave it as an exercise.

Note that EDF is a dynamic priority scheduling algorithm. If a task is repeatedly executed, it may be assigned a different priority on each execution. This can make it more complex to implement. Typically, for periodic tasks, the deadline used is the end of the period of the task, though it is certainly possible to use other deadlines for tasks.

Although EDF is more expensive to implement than RM, in practice its performance is generally superior (Buttazzo, 2005b). First, RM is optimal with respect to feasibility only among fixed priority schedulers, whereas EDF is optimal w.r.t. feasibility among dynamic priority schedulers. In addition, EDF also minimizes the maximum lateness. Also, in practice, EDF results in fewer preemptions (see Exercise 2), which means less overhead for context switching. This often compensates for the greater complexity in the implementation. In addition, unlike RM, any EDF schedule with less than 100% utilization can tolerate increases in execution times and/or reductions in periods and still be feasible.

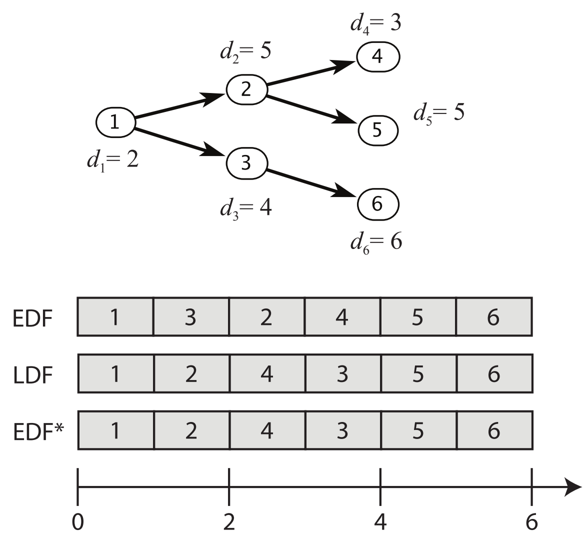


Figure 11.7: An example of a precedence graph for six tasks and the schedule under three scheduling policies. Execution times for all tasks are one time unit.

11.3.1 EDF with Precedences

Theorem 11.2 shows that EDF is optimal (it minimizes maximum lateness) for a task set without precedences. What if there are precedences? Given a finite set of tasks, precedences between them can be represented by a **precedence graph**.

Example 11.1: Consider six tasks $T = \{1, \cdots, 6\}$, each with execution time $e_i = 1$, with precedences as shown in Figure 11.7. The diagram means that task 1 must execute before either 2 or 3 can execute, that 2 must execute before either 4 or 5, and that 3 must execute before 6. The deadline for each task is shown in the figure. The schedule labeled EDF is the EDF schedule. This schedule is not feasible. Task 4 misses its deadline. However, there is a feasible schedule. The schedule labeled LDF meets all deadlines.

The previous example shows that EDF is not optimal if there are precedences. In 1973, Lawler (1973) gave a simple algorithm that is optimal with precedences, in the sense that it minimizes the maximum lateness. The strategy is very simple. Given a fixed, finite set of tasks with deadlines, Lawler's strategy constructs the schedule backwards, choosing first the *last* task to execute. The last task to execute is the one on which no other task depends that has the latest deadline. The algorithm proceeds to construct the schedule backwards, each time choosing from among the tasks whose dependents have already been scheduled the one with the latest deadline. For the previous example, the resulting schedule, labeled LDF in Figure 11.7, is feasible. Lawler's algorithm is called **latest deadline first** (**LDF**).

LDF is optimal in the sense that it minimizes the maximum lateness, and hence it is also optimal with respect to feasibility. However, it does not support arrival of tasks. Fortunately, there is a simple modification of EDF, proposed by Chetto et al. (1990). **EDF*** (EDF with precedences), supports arrivals and minimizes the maximal lateness. In this modification, we adjust the deadlines of all the tasks. Suppose the set of all tasks is T. For a task execution $i \in T$, let $D(i) \subset T$ be the set of task executions that immediately depend on i in the precedence graph. For all executions $i \in T$, we define a modified deadline

$$d'_i = \min(d_i, \min_{j \in D(i)} (d'_j - e_j)) .$$

EDF* is then just like EDF except that it uses these modified deadlines.

Example 11.2: In Figure 11.7, we see that the EDF* schedule is the same as the LDF schedule. The modified deadlines are as follows:

$$d'_1 = 1, \quad d'_2 = 2, \quad d'_3 = 4, \quad d'_4 = 3, \quad d'_5 = 5, \quad d'_6 = 6 .$$

The key is that the deadline of task 2 has changed from 5 to 2, reflecting the fact that its successors have early deadlines. This causes EDF* to schedule task 2 before task 3, which results in a feasible schedule.

EDF* can be thought of as a technique for rationalizing deadlines. Instead of accepting arbitrary deadlines as given, this algorithm ensures that the deadlines take into account deadlines of successor tasks. In the example, it makes little sense for task 2 to have a later deadline, 5, than its successors. So EDF* corrects this anomaly before applying EDF.

11.4 Scheduling and Mutual Exclusion

Although the algorithms given so far are conceptually simple, the effects they have in practice are far from simple and often surprise system designers. This is particularly true when tasks share resources and use mutual exclusion to guard access to those resources.

11.4.1 Priority Inversion

In principle, a **priority-based preemptive scheduler** is executing at all times the high-priority enabled task. However, when using mutual exclusion, it is possible for a task to become blocked during execution. If the scheduling algorithm does not account for this possibility, serious problems can occur.

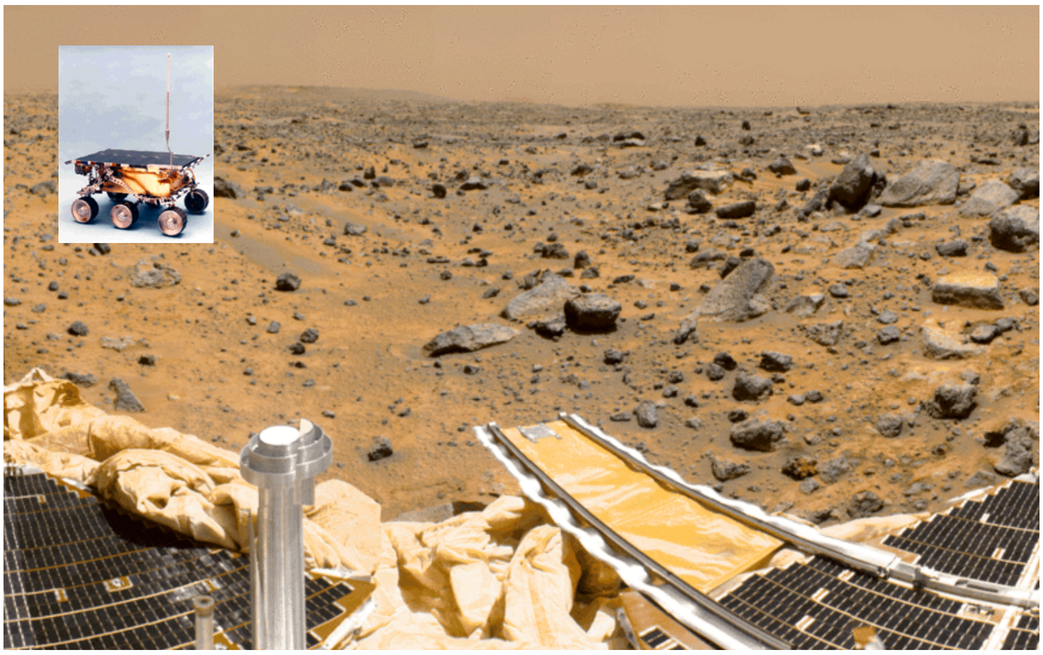


Figure 11.8: The Mars Pathfinder and a view of the surface of Mars from the camera of the lander (image from the Wikipedia Commons).

Example 11.3: The Mars Pathfinder, shown in Figure 11.8, landed on Mars on July 4th, 1997. A few days into the mission, the Pathfinder began sporadically missing deadlines, causing total system resets, each with loss of data. Engineers on the ground diagnosed the problem as priority inversion, where a low priority meteorological task was holding a lock and blocking a high-priority task, while medium priority tasks executed. (**Source**: What Really Happened on Mars? Mike Jones, RISKS-19.49 on the comp.programming.threads newsgroup, Dec. 07, 1997, and What Really Happened on Mars? Glenn Reeves, Mars Pathfinder Flight Software Cognizant Engineer, email message, Dec. 15, 1997.)

Priority inversion is a scheduling anomaly where a high-priority task is blocked while unrelated lower-priority tasks are executing. The phenomenon is illustrated in Figure 11.9. In the figure, task 3, a low priority task, acquires a lock at time 1. At time 2, it is preempted by task 1, a high-priority task, which then at time 3 blocks trying to acquire the same lock. Before task 3 reaches the point where it releases the lock, however, it gets preempted by an unrelated task 2, which has medium priority. Task 2 can run for an unbounded amount

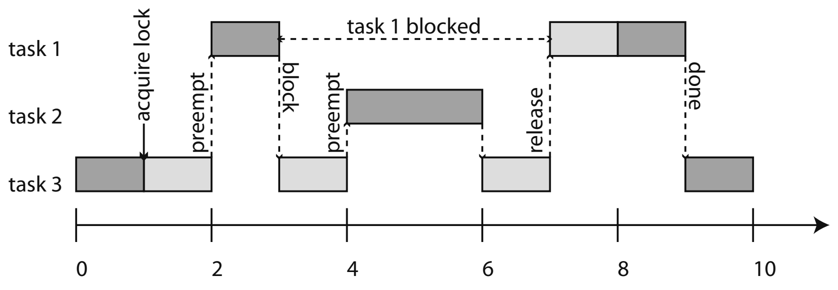


Figure 11.9: Illustration of priority inversion. Task 1 has highest priority, task 3 lowest. Task 3 acquires a lock on a shared object, entering a critical section. It gets preempted by task 1, which then tries to acquire the lock and blocks. Task 2 preempts task 3 at time 4, keeping the higher priority task 1 blocked for an unbounded amount of time. In effect, the priorities of tasks 1 and 2 get inverted, since task 2 can keep task 1 waiting arbitrarily long.

of time, and effectively prevents the higher-priority task 1 from executing. This is almost certainly not desirable.

11.4.2 Priority Inheritance Protocol

In 1990, Sha et al. (1990) gave a solution to the priority inversion problem called **priority inheritance**. In their solution, when a task blocks attempting to acquire a lock, then the task that holds the lock inherits the priority of the blocked task. Thus, the task that holds the lock cannot be preempted by a task with lower priority than the one attempting to acquire the lock.

Example 11.4: Figure 11.10 illustrates priority inheritance. In the figure, when task 1 blocks trying to acquire the lock held by task 3, task 3 resumes executing, but now with the higher priority of task 1. Thus, when task 2 becomes enabled at time 4, it does not preempt task 3. Instead, task 3 runs until it releases the lock at time 5. At that time, task 3 reverts to its original (low) priority, and task 1 resumes executing. Only when task 1 completes is task 2 able to execute.

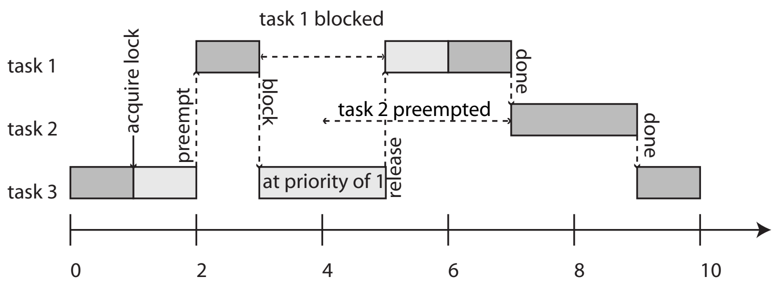


Figure 11.10: Illustration of the priority inheritance protocol. Task 1 has highest priority, task 3 lowest. Task 3 acquires a lock on a shared object, entering a critical section. It gets preempted by task 1, which then tries to acquire the lock and blocks. Task 3 inherits the priority of task 1, preventing preemption by task 2.

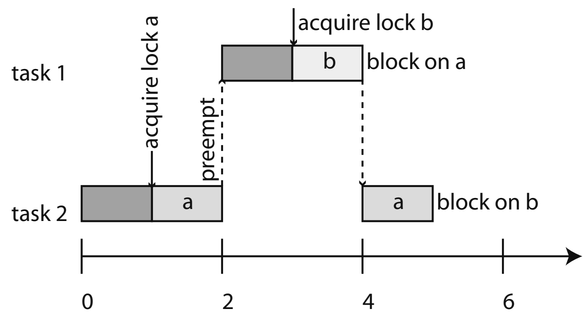


Figure 11.11: Illustration of deadlock. The lower priority task starts first and acquires lock a, then gets preempted by the higher priority task, which acquires lock b and then blocks trying to acquire lock a. The lower priority task then blocks trying to acquire lock b, and no further progress is possible.

11.4.3 Priority Ceiling Protocol

Priorities can interact with mutual exclusion locks in even more interesting ways. In particular, in 1990, Sha et al. (1990) showed that priorities can be used to prevent certain kinds of deadlocks.

Example 11.5: Figure 11.11 illustrates a scenario in which two tasks deadlock. In the figure, task 1 has higher priority. At time 1, task 2 acquires lock a. At time 2, task 1 preempts task 2, and at time 3, acquires lock b. While holding lock b, it attempts to acquire lock a. Since a is held by task 2, it blocks. At time 4, task 2 resumes executing. At time 5, it attempts to acquire lock b, which is held by task 1. Deadlock!

The deadlock in the previous example can be prevented by a clever technique called the **priority ceiling** protocol (Sha et al., 1990). In this protocol, every lock or semaphore is assigned a priority ceiling equal to the priority of the highest-priority task that can lock it. A task τ can acquire a lock a only if the task's priority is strictly higher than the priority ceilings of all locks currently held by other tasks. Intuitively, if we prevent task τ from

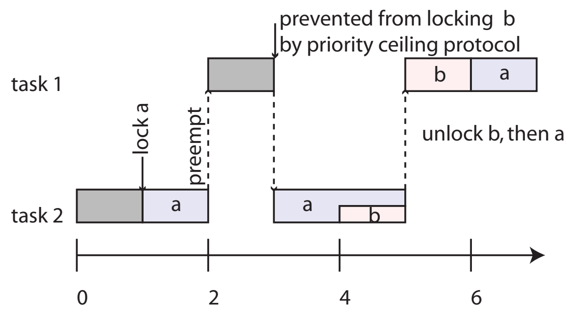


Figure 11.12: Illustration of the priority ceiling protocol. In this version, locks a and b have priority ceilings equal to the priority of task 1. At time 3, task 1 attempts to lock b, but it cannot because task 2 currently holds lock a, which has priority ceiling equal to the priority of task 1.

acquiring lock *a*, then we ensure that task τ will not hold lock *a* while later trying to acquire other locks held by other tasks. This prevents certain deadlocks from occurring.

Example 11.6: The priority ceiling protocol prevents the deadlock of Example 11.5, as shown in Figure 11.12. In the figure, when task 1 attempts to acquire lock *b* at time 3, it is prevented from doing so. At that time, lock *a* is currently held by another task (task 2). The priority ceiling assigned to lock *a* is equal to the priority of task 1, since task 1 is the highest priority task that can acquire lock *a*. Since the priority of task 1 is not *strictly higher* than this priority ceiling, task 1 is not permitted to acquire lock *b*. Instead, task 1 becomes blocked, allowing task 2 to run to completion. At time 4, task 2 acquires lock *b* unimpeded, and at time 5, it releases both locks. Once it has released both locks, task 1, which has higher priority, is no longer blocked, so it resumes executing, preempting task 2.

Of course, implementing the priority ceiling protocol requires being able to determine in advance which tasks acquire which locks. A simple conservative strategy is to examine the source code for each task and inventory the locks that are acquired in the code. This is conservative because a particular program may or may not execute any particular line of code, so just because a lock is mentioned in the code does not necessarily mean that the task will attempt to acquire the lock.

11.5 Multiprocessor Scheduling

Scheduling tasks on a single processor is hard enough. Scheduling them on multiple processors is even harder. Consider the problem of scheduling a fixed finite set of tasks with precedences on a finite number of processors with the goal of minimizing the makespan. This problem is known to be NP-hard. Nonetheless, effective and efficient scheduling strategies exist. One of the simplest is known as the **Hu level scheduling** algorithm. It assigns a priority to each task τ based on the **level**, which is the greatest sum of execution times of tasks on a path in the precedence graph from τ to another task with no dependents. Tasks with larger levels have higher priority than tasks with smaller levels.

Example 11.7: For the precedence graph in Figure 11.7, task 1 has level 3, tasks 2 and 3 have level 2, and tasks 4, 5, and 6 have level 1. Hence, a Hu level scheduler will give task 1 highest priority, tasks 2 and 3 medium priority, and tasks 4, 5, and 6 lowest priority.

Hu level scheduling is one of a family of **critical path** methods because it emphasizes the path through the precedence graph with the greatest total execution time. Although it is not optimal, it is known to closely approximate the optimal solution for most graphs (Kohler, 1975; Adam et al., 1974).

Once priorities are assigned to tasks, a **list scheduler** sorts the tasks by priorities and assigns them to processors in the order of the sorted list as processors become available.

Example 11.8: A two-processor schedule constructed with the Hu level scheduling algorithm for the precedence graph shown in Figure 11.7 is given in Figure 11.13. The makespan is 4.

11.5.1 Scheduling Anomalies

Among the worst pitfalls in embedded systems design are **scheduling anomalies**, where unexpected or counterintuitive behaviors emerge due to small changes in the operating conditions of a system. We have already illustrated two such anomalies, priority inversion and deadlock. There are many others. The possible extent of the problems that can arise are well illustrated by the so-called **Richard's anomalies** (Graham, 1969). These show that multiprocessor schedules are **non-montonic**, meaning that improvements in performance at a local level can result in degradations in performance at a global level, and **brittle**, meaning that small changes can have big consequences.

Richard's anomalies are summarized in the following theorem.

Theorem 11.4. *If a task set with fixed priorities, execution times, and precedence constraints is scheduled on a fixed number of processors in accordance with the priorities, then increasing the number of processors, reducing execution times, or weakening precedence constraints can increase the schedule length.*

Proof. The theorem can be proved with the example in Figure 11.14. The example has nine tasks with execution times as shown in the figure. We assume the tasks are assigned priorities so that the lower numbered tasks have higher priority than the higher numbered tasks. Note that this does not correspond to a critical path priority assignment, but it suffices to prove the theorem. The figure shows a three-processor schedule in accordance with the priorities. Notice that the makespan is 12.

First, consider what happens if the execution times are all reduced by one time unit. A schedule conforming to the priorities and precedences is shown below:

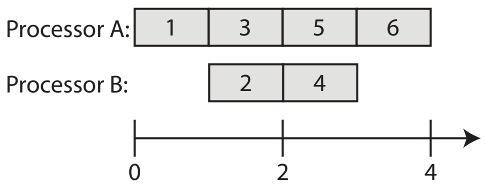


Figure 11.13: A two-processor parallel schedule for the tasks with precedence graph shown in Figure 11.7.

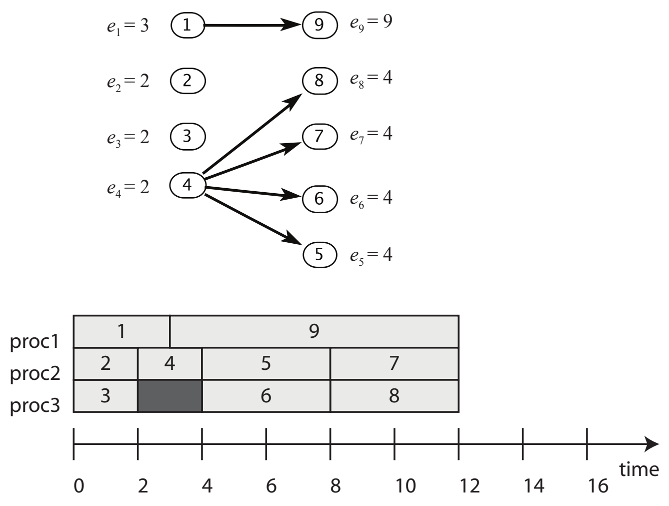
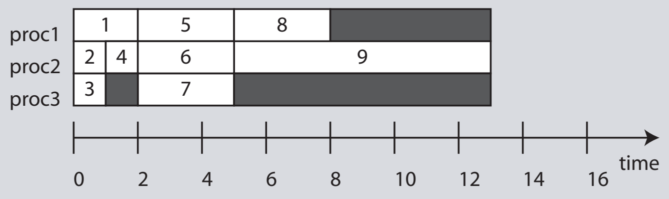
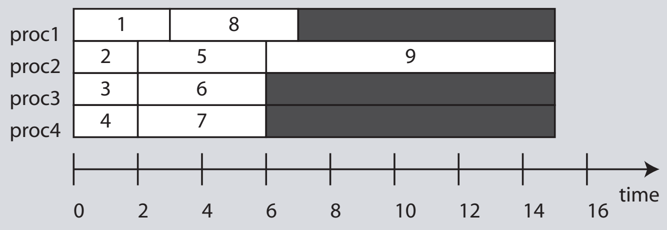


Figure 11.14: A precedence graph with nine tasks, where the lower numbered tasks have higher priority than the higher numbered tasks.



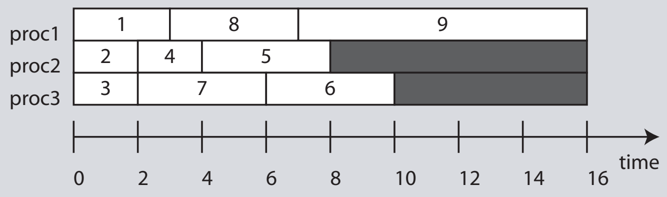
Notice that the makespan has *increased* to 13, even though the total amount of computation has decreased significantly. Since computation times are rarely known exactly, this form of brittleness is particularly troubling.

Consider next what happens if we add a fourth processor and keep everything else the same as in the original problem. A resulting schedule is shown below:



Again, the makespan has increased (to 15 this time) even though we have added 33% more processing power than originally available.

Consider finally what happens if we weaken the precedence constraints by removing the precedences between task 4 and tasks 7 and 8. A resulting schedule is shown below:



The makespan has now increased to 16, even though weakening precedence constraints increases scheduling flexibility. A simple priority-based scheduling scheme such as this does not take advantage of the weakened constraints. □

This theorem is particularly troubling when we realize that execution times for software are rarely known exactly (see Chapter 15). Scheduling policies will be based on approximations, and behavior at run time may be quite unexpected.

Another form of anomaly arises when there are mutual exclusion locks. An illustration is given in Figure 11.15. In this example, five tasks are assigned to two processors using a static assignment scheduler. Tasks 2 and 4 contend for a mutex. If the execution time of task 1 is reduced, then the order of execution of tasks 2 and 4 reverses, which results in an increased execution time. This kind of anomaly is quite common in practice.

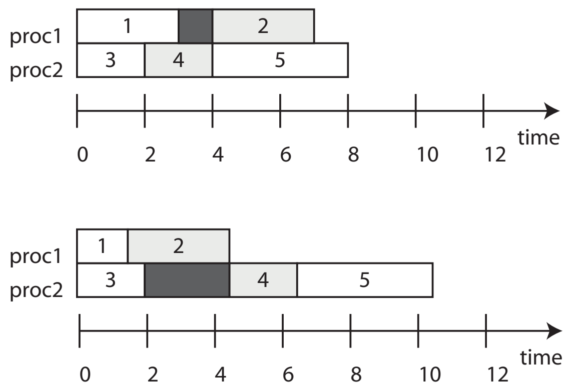


Figure 11.15: Anomaly due to mutual exclusion locks, where a reduction in the execution time of task 1 results in an increased makespan.

11.6 Summary

Embedded software is particularly sensitive to timing effects because it inevitably interacts with external physical systems. A designer, therefore, needs to pay considerable attention to the scheduling of tasks. This chapter has given an overview of some of the basic techniques for scheduling real-time tasks and parallel scheduling. It has explained some of the pitfalls, such as priority inversion and scheduling anomalies. A designer that is aware of the pitfalls is better equipped to guard against them.