Chapter 11

An Optimal Priority Inheritance Policy For Synchronization in Real-Time Systems

Ragunathan Rajkumar, Lui Sha,
John P. Lehoczky, and Krithi Ramamritham

Hard real-time systems require predictable timing behavior, and priority-driven preemptive scheduling is increasingly being used in these systems. Resources in these enviroiments should ideally be allocated to the highest-priority task. Priority inversion is a situation in which a higher-priority job is forced to wait for a lower-priority job. Priority inversion degrades system schedulability. Hence, priority inversion should be minimized in a hard real-time environment. Unfortunately, a direct application of synchronization primitives such as semaphores, monitors, and Ada rendezvous can cause uncontrolled priority inversion, a situation in which a low-priority job blocks a higher-priority job for an indefinite period of time. In this chapter we investigate policies belonging to the class of *priority inheritance policies* that minimize priority inversion. We develop a priority inheritance policy called the *Optimal Mutex Policy* (OMP) which has two desirable properties: deadlocks are avoided and the worst-case blocking duration of a job is bounded by the duration of execution of a single critical section of a lower-priority job.

11.1 Introduction

11.1.1 Real-Time Systems

Real-time systems operate under strict timing constraints and include applications such as avionics systems, space-related systems like the Space Shuttle and Space Station, production control, robotics, and defense systems. Timing constraints of different tasks in real-time systems can be either hard, soft, or non-existent.

A timing constraint is considered to be *hard* if it must be met at all times, or is considered to be *soft* if it must be met only most of the time. For example, the processing of a reactor temperature reading can have a hard deadline if it must be completed before the next reading becomes available. An operator query typically has a soft deadline with a desired average-case response time. In addition, background tasks such as on-line testing may have no associated timing constraints at all. The failure to meet hard deadlines in these systems can potentially lead to catastrophic results such as loss of life and/or property.

Real-time systems tend to be embedded systems which are not generally programmed by the end-user. Unlike traditional time-shared systems, tasks in hard real-time systems are known *a priori*. In particular, the worst-case behavior of tasks with hard deadlines and the average-case behavior of tasks with soft deadlines are reasonably well-tested and understood. Given a set of tasks and their associated timing constraints, two distinct approaches to the implementation of real-time systems are possible. One, called the *time-line* approach, is typified by the *cyclical executive*, where each segment of code to be executed is assigned a time slot for execution. The time-line is typically handcrafted such that the timing and logical constraints of the task set are met, and is repeatedly executed in cyclical fashion. However, this approach is very *ad hoc* in nature, and leads to very inflexible systems that are difficult to maintain and modify [18]. The other approach is the use of algorithmic techniques to schedule tasks using scheduling algorithms which can be mathematically modeled and analyzed [10]. In this chapter, we focus on this more powerful algorithmic approach to schedule real-time tasks to meet their timing constraints.

Real-time systems are becoming increasingly popular with the advent of faster and cheaper hardware which opens up newer application domains where automation is more reliable and cheaper. Task scheduling is a significant area of research in real-time computer systems. Both non-preemptive and preemptive scheduling algorithms have been studied [6, 8, 9, 14, 15, 16]. An important performance metric of many scheduling algorithms is the processor utilization below which tasks are guaranteed to meet their deadlines. For example, consider the rate-monotonic scheduling algorithm, which assigns a higher fixed priority to a task with a higher frequency [10] and is the optimal static priority algorithm for independent periodic tasks. A job (instance) of a periodic task must complete execution before the next job of the task arrives. The utilization of a task set is defined as the sum of all C_i/T_i, where C_i is the worst-case execution time of a task τ_i and T_i is its period. The rate-monotonic algorithm guarantees that it can schedule any periodic task set if its total utilization is less than $\ln 2$ (69%). If the periods of the tasks are harmonic, the utilization bound is 100%. It has also been shown that the rate-monotonic algorithm can schedule randomly generated periodic task sets up to 88% on the average [7].

11.1.2 The Resource-Sharing Problem

An important problem that arises in the context of priority-based real-time systems is the effect of blocking caused by the need for the synchronization of jobs that

share logical or physical resources. Mok [11] showed that the problem of deciding whether it is possible to schedule a set of periodic processes is NP-hard when periodic processes use semaphores to enforce mutual exclusion. One approach to the scheduling of real-time jobs when synchronization primitives are used is to try to dynamically construct a feasible schedule at runtime. Mok [11] developed a procedure to generate feasible schedules with a kernelized monitor, which does not permit the preemption of jobs in critical sections. It is an effective technique for the case where the critical sections are short.

In this chapter, we investigate the synchronization problem in the context of priority-driven preemptive scheduling. Unfortunately, a direct application of synchronization mechanisms like the Ada rendezvous, semaphores, or monitors can lead to uncontrolled priority inversion: a high-priority job being blocked by a lower-priority job for an indefinite period of time. Priority inversion in real-time systems cannot only cause deadlines to be missed at low levels of resource utilization but, perhaps more important, render these systems less predictable. In this chapter, we present an extension to the *priority inheritance policies* [17] and prove the properties of an optimal policy belonging to this family of policies. The priority inheritance policies, defined in the context of a uniprocessor, rectify the uncontrolled priority inversion problem that can result from an injudicious use of traditional synchronization primitives. The reader is encouraged to study related work in the context of earliest deadline scheduling by Chen and Lin [2], Baker [1], and Jeffay [4].

11.1.3 Organization of the Chapter

The chapter is organized as follows. We describe the priority inversion problem and review the basic concepts underlying the priority inheritance policies in Section 11.1.4. In Section 11.2, we define the *Optimal Mutex Policy* (OMP) and investigate its properties. We show that under the policy, the system becomes deadlock-free, and a job can be blocked for the duration of at most one critical section of a lower-priority job. We also present the impact of these policies on schedulability analysis when the rate-monotonic algorithm is used. Finally, Section 11.3 presents some concluding remarks.

11.1.4 The Concept of Priority Inheritance

Priority inversion is said to occur when a higher-priority job is forced to wait for the execution of a lower-priority job. A common situation arises when two jobs attempt to access shared data. If the higher-priority job gains access to the shared data first, the appropriate priority order is maintained. However, if the lower-priority data gains access first and then the higher priority job requests access to the shared data, the higher-priority job is blocked until the lower-priority job completes its access to the data.

Example 1: Let J_1, J_2, and J_3 be jobs listed in descending order of priority. Assume that J_1 and J_3 share data guarded by a mutex S. Suppose that at time t_1,

job J_3 locks S and enters its critical section. During J_3's execution of its critical section, J_1 arrives at time t_2 and preempts J_3 and begins execution. At time t_3, J_1 attempts to use the shared data and gets blocked. We might expect that J_1, being the highest-priority job, will be blocked no longer than the time for job J_3 to exit its critical section. However, the duration of blocking can, in fact, be unpredictable. This is because job J_3 can be preempted by the intermediate-priority job J_2. The blocking of J_3, and hence that of J_1, will continue until J_2 and any other pending intermediate jobs are completed.

The blocking duration in Example 1 can be unacceptably long. This situation can be partially remedied if a job is not allowed to be preempted within a critical section. However, this solution is appropriate only for short critical sections. For instance, once a low-priority job enters a long critical section, a higher-priority job which does not access the shared data structure may be needlessly blocked. Analogous problems exist with monitors and the Ada rendezvous. The priority inversion problem was first discussed by Lampson and Redell [5] in the context of monitors. They suggest that each monitor always be executed at a priority level higher than all tasks that would ever call the monitor. This solution has the same problem as the one discussed: a higher-priority job that does not share data may be unnecessarily blocked by a lower-priority job. The priority inversion problem in the context of earliest deadline scheduling has also been discussed by Clark [3]. The proposed solution is that a task with a longer deadline blocking a task with a shorter deadline promotes its deadline to that of the latter. This technique is referred to as *deadline promotion* and is analogous to the *basic priority inheritance policy* described in [17].

The use of priority inheritance policies is one approach to rectify the priority inversion problem inherent in existing synchronization primitives. The basic idea of priority inheritance policies is that when a job J blocks higher-priority jobs, it executes its critical section at the highest-priority level of all of the jobs it blocks. After exiting its critical section, job J returns to its original priority level. To illustrate this idea, we apply this policy to Example 1. Suppose that job J_1 is blocked by J_3. The priority inheritance policies stipulate that job J_3 execute its critical section at J_1's priority. As a result, job J_2 will be unable to preempt J_3 and will itself be blocked. When J_3 exits its critical section, it regains its original priority and will immediately be preempted by J_1. Thus, J_1 will be blocked only for the duration of J_3's critical section.

The concept of priority inheritance, as defined, allows us to develop a family of real-time synchronization policies based on when a job is defined to be blocked by a lower-priority job. For instance, the simplest priority inheritance policy stipulates that a lower-priority job inherit the priority of a higher-priority job when the latter tries to lock a mutex already locked by the lower-priority job. Such a policy is called the *basic priority inheritance policy* [17]. However, as we shall see, the basic priority inheritance policy can still lead to avoidable priority inversion and/or deadlocks. Our goal in this chapter is to develop a priority inheritance policy which leads to the minimum blocking duration for each job.

In all subsequent discussions, when a lower-priority job J_L prevents a higher-priority job J_H from executing, J_L is said to *block* J_H. When a higher-priority job

J_H preempts a lower-priority job J_L, J_H is *not* considered to be blocking J_L.

11.1.5 Assumptions and Notation

Before we investigate other priority inheritance policies, we define our terminology, introduce the notation used, and state the assumptions which apply in the following sections.

We assume a uniprocessor executing a fixed set of tasks. The highest-priority job eligible to execute is scheduled to run on the processor. A currently executing job is preempted by a higher-priority job that becomes eligible to execute. A *job* is a sequence of instructions that will continuously use the processor until its completion if it is executing alone on the processor. A *periodic task* is a sequence of the same type of job initiated at regular intervals. Each task is assigned a fixed priority, and every job of the same task is assigned that task's priority. If two jobs are eligible to run, the higher-priority job will be run. Jobs with the same priority are executed according to a FCFS discipline by order of job arrival time.

Notation: J_i denotes a job, namely an instance of a periodic task τ_i. P_i, T_i, and C_i denote the current executing priority, period and the worst-case execution time of task τ_i, respectively. The assigned priority of a job J_i is the same as that of task τ_i and is denoted by $p(J_i)$.

We also assume that jobs J_1, J_2, ..., J_n are listed in descending order of assigned priority, with J_1 having the highest priority.

In this chapter, we develop policies assuming that each data structure shared among jobs is guarded by a mutex. However, the principle underlying the policies is also applicable when monitors or rendezvous are used for the synchronization of jobs.

Similar to the common assumption in real-time systems that there exists a worst-case execution time of a job, we assume that there is a worst-case execution time within a critical section. We also assume that the mutexes that can be locked by a critical section are known *a priori*. This assumption can be relaxed to obtain policies which approximate the policy developed in this chapter. A critical section of a task need not always be entered by any given job of the task. However, if a job is already within a critical section, the locking policy developed in this chapter assumes the worst case that all mutexes that may be potentially locked within the critical section *will* be locked by the job.

Notation: A mutex guarding shared data and/or a shared resource is denoted by S_i. $Lock(S_i)$ and $Unlock(S_i)$ denote the indivisible operations *lock* (wait) and *unlock* (signal), respectively, on the mutex S_i. The section of code beginning with the locking of a mutex and ending with the unlocking of the mutex is termed a *critical section*.

A job can have multiple critical sections that do not overlap, e.g., $\cdots Lock(S_1) \cdots Unlock(S_1) \cdots Lock(S_2) \cdots Unlock(S_2) \cdots$. A critical section can be nested, i.e., a job J_i may make nested requests for mutex locks, e.g., $\cdots, Lock(S_1) \cdots Lock(S_2)$

$\cdots$ *Unlock*(S_2) $\cdots$ *Unlock*(S_1) $\cdots$. In this case, critical section $z_{i,1}$ is bounded by *Lock*(S_1) and *Unlock*(S_1) and nests the critical section $z_{i,2}$. The phrase "the duration of an (outermost) critical section" refers to the execution time bounded by the outermost pair of *lock* and *unlock* operations, e.g.,, the execution time of the outermost critical section starting with *Lock*(S_1) and ending with *Unlock*(S_1). We shall use the terms "critical section" and "outermost critical section" interchangeably.

The jth critical section in job J_i is denoted by $z_{i,j}$ and corresponds to the code segment of job J_i between the jth *Lock* operation and its corresponding *Unlock* operation. The mutex that is locked and released by critical section $z_{i,j}$ is denoted by $S_{i,j}$. We write $z_{i,j} \subset z_{i,k}$ if the critical section $z_{i,j}$ is entirely contained in $z_{i,k}$. The worst-case duration of the execution of the critical section $z_{i,j}$, denoted by $d_{i,j}$, is the time required to execute $z_{i,j}$ when J_i executes on the processor alone.

We assume that critical sections are properly nested. That is, given any pair of critical sections $z_{i,j}$ and $z_{i,k}$, then either $z_{i,j} \subset z_{i,k}$, $z_{i,k} \subset z_{i,j}$, or $z_{i,j} \cap z_{i,k} = \emptyset$. In addition, we assume that a mutex may be locked at most once in a single nested critical section. This implies that a job will not attempt to lock a mutex that it has already locked and thus deadlock with itself. In addition, we assume that locks on mutexes will be released before or at the end of a job.

Definition: A job J is said to be blocked by the critical section $z_{i,j}$ of job J_i if J_i has a lower-priority than J but J has to wait for J_i to exit $z_{i,j}$ in order to continue execution.

Definition: A job J is said to be blocked by job J_i through mutex S if the critical section $z_{i,j}$ blocks J and $S_{i,j} = S$.

An important feature of the policy that we propose is that it is possible to determine the *schedulability bound* for a given task set when this policy is used. If the utilization of the task set stays below this bound, then the deadlines of all the tasks can be guaranteed. To develop such a bound, it becomes necessary to determine the worst-case duration of blocking that any task can encounter. This worst-case blocking duration will depend upon the particular policy in use, but the following approach will always be taken.

Notation: $\beta_{i,j}$ denotes the set of all critical sections of the lower-priority job J_j which can block J_i. That is, $\beta_{i,j} = \{z_{j,k} \mid j > i \text{ and } z_{j,k} \text{ can block } J_i\}$.[1] Since we consider only properly nested critical sections, the set of blocking critical sections is partially ordered by set inclusion. Using this partial ordering, we can focus our attention on the set of maximal elements of $\beta_{i,j}$, $\beta^*_{i,j}$. Specifically, we have $\beta^*_{i,j}$ $= \{z_{j,k} \mid (z_{j,k} \in \beta_{i,j}) \wedge (\nexists\ z_{j,m} \in \beta_{i,j} \text{ such that } z_{j,k} \subset z_{j,m})\}$. The set $\beta^*_{i,j}$ contains the outermost critical sections of J_j which can block J_i and eliminates redundant inner critical sections. For purposes of schedulability analysis, we will restrict attention to $\beta^*_i = \{\cup_{j>i}\ \beta^*_{i,j}\}$, the set of all outermost critical sections that can block J_i.

[1] Note that the second suffix of $\beta_{i,j}$ and the first suffix of $z_{j,k}$ correspond to job J_j.

11.2 The Optimal Mutex Policy

The basic priority inheritance policy [17] stipulates that when a job J attempts to lock a mutex S already locked by a lower-priority job J_L, J_L inherits J's priority until J_L releases the lock on S. However, this policy suffers from two problems. First, a job J could be blocked for the duration of $min(m, n)$ critical sections [17], where n is the number of lower-priority jobs that attempt to lock a mutex also accessed by tasks with a priority higher than or equal to $p(J)$ and m is the number of distinct mutexes that can be locked by lower-priority jobs. For instance, consider the following example.

Example 2: Suppose that J_1 needs to sequentially lock S_1 and S_2. Also suppose that J_2 preempts J_3 after J_3 has locked S_1. Later, J_2 locks S_2. Job J_1 arrives at this instant and finds that the mutexes S_1 and S_2 have been locked by the lower-priority jobs J_3 and J_2, respectively. As a result, J_1 would be blocked for the duration of two critical sections, once to wait for J_3 to release S_1 and again to wait for J_2 to release S_2. Thus, a job can be blocked for the duration of more than one critical section. We refer to this as *multiple blocking*.

Second, the policy does not avoid deadlocks. For instance, consider jobs J_1 and J_2. J_1 will make nested requests to lock mutexes S_1 and S_2 in that order. Conversely, J_2 will lock S_2 first and then S_1. Suppose that J_2 arrives first and locks S_2. However, before it locks S_1, J_1 arrives and preempts J_2. Then, J_1 locks unlocked mutex S_1. When J_1 attempts to lock S_2, it gets blocked and J_2 inherits J_1's priority. But a deadlock situation occurs when J_2 tries to lock S_1. Hence, explicit deadlock avoidance techniques like total ordering of mutex requests may have to be employed if the basic priority inheritance policy is used.

Intuitively, it can be seen that the basic priority inheritance policy runs into its problems for the following reason. An unlocked mutex is allowed to be locked at any instant irrespective of its relationship to the mutexes that have already been locked. Hence, when a higher-priority job arrives, it can find that several mutexes that it needs have been locked by lower-priority jobs. Furthermore, such uncontrolled locking can potentially cause a deadlock as well. This situation can be remedied by allowing mutexes to be locked only under selective conditions. In other words, if the locking of a mutex may cause multiple blocking to a higher-priority job, we should not allow the mutex to be locked. We use the information about the mutex needs of each job and the job priorities to decide whether the locking of a mutex can lead to multiple blocking and/or deadlock. Imposing conditions on the locking of a mutex is the essence of the proposed policy.

In this section, we develop the Optimal Mutex Policy (OMP). The policy not only minimizes the blocking encountered by a job to the duration of execution of a single critical section but also avoids deadlocks. In this section, we shall present the policy and prove its properties. We shall show that the locking conditions used by OMP are both necessary and sufficient to limit the worst-case blocking duration to a single critical section for any job. However, an implementation of this policy may be expensive. Suboptimal but computationally simpler policies are discussed in [13].

11.2.1 The Concept of the Optimal Mutex Policy

Definition: The *priority ceiling* of a mutex S is defined as the assigned priority of the highest-priority task that may lock S. The priority ceiling of a mutex S represents the highest-priority that a critical section guarded by S can inherit from a higher-priority job. In other words, if a job J locks the mutex S, the corresponding critical section of J can inherit at most a priority equal to the priority ceiling of S.

Notation: The priority ceiling of a mutex S_j is denoted by $c(S_j)$.

Definition: The *current critical section* of a job J refers to the outermost critical section that J has already entered.

Notation: When a job J requests the lock on an unlocked mutex S,

- S^* is a mutex with the highest-priority ceiling locked by jobs other than J. If there is no mutex currently locked, S^* is defined to be a dummy mutex S_{dummy} whose priority ceiling is less than the priorities of all jobs in the system. If there is more than one mutex in the system with the same priority ceiling, any one of them may be chosen. We shall later show that this choice is immaterial, and that there can be at most two such mutexes with the highest-priority ceiling.
- J^* is the job holding the lock on S^*. If S^* is the dummy mutex S_{dummy}, J^* can be represented by the *idle* process that runs when there is no active process ready to run.
- SL^* is the set of mutexes already locked by the current critical section of job J^*.[2]
- SR is the set of mutexes that the current critical section of J may lock later.[3] For convenience, both SL and SR are defined to be the empty sets when J is not inside a critical section. Also, once J is successful in obtaining the lock, SL includes S. Otherwise, J will be blocked and $S \in SR$.
- SR^* is the set of mutexes that will be locked by the current critical section of job J^*. If the current critical section of job J^* does not request any more nested mutex locks, $SR^* = \emptyset$.
- z is the (outermost) critical section that J has already entered, else the critical section that J is trying to enter.

Remark: For any given job J, $SL \cap SR = \emptyset$ and $SL \cup SR =$ set of mutexes that can be locked by the current critical section of J.

As already mentioned, OMP selectively grants locks on unlocked mutexes to requesting jobs. Suppose that job J requests the lock on an unlocked mutex S.

[2] SL stands for "mutexes locked."
[3] SR stands for "mutexes required" for completion of the current critical section.

OMP allows J to lock S if and only if at least one of the following conditions is true.

1. Condition C1: The priority of job J is greater than the priority ceiling of S^*, i.e., $p(J) > c(S^*)$.
2. Condition C2: The priority of job J is equal to the priority ceiling of S^* and the current critical section of J will not attempt to lock any mutex already locked by J^*, i.e., $(p(J) = c(S^*)) \wedge (SR \cap SL^* = \emptyset)$.
3. Condition C3: The priority of job J is equal to the priority ceiling of S and the lock on mutex S will not be requested by J^*'s preempted critical section, i.e., $(p(J) = c(S)) \wedge (S \notin SR^*)$.

If none of these conditions is true, job J is blocked and J^* inherits J's current executing priority. We refer to conditions C1, C2, and C3 as the *locking conditions*.

Under OMP, a job can be blocked for the duration of at most a single critical section, and deadlocks cannot occur. Before we prove these properties, we illustrate the policy with a few examples. We shall first apply OMP to the examples in the preceding section, where multiple blocking occurs for a job.

Example 3: A job J_1 needs to lock S_1 and S_2 sequentially, while J_2 needs to lock S_1, and J_3 needs to lock S_2. Hence, $c(S_1) = c(S_2) = p(J_1)$. At time t_0, J_3 locks S_2. At time t_1, J_2 preempts J_3 and later attempts to lock S_1. Now, $J^* = J_3$ and $S^* = S_2$. However, $p(J) < c(S_2)$ and $p(J) < c(S_1)$, so all three locking conditions are false. Hence, J_2 is blocked and J_3 inherits J_2's priority. When J_1 arrives and attempts to lock S_1, condition C2 is true (since J_1 does not make any nested requests for mutex locks and $SR = \emptyset$). Hence J_1 can obtain the lock on S_1. Later, when J_1 attempts to lock the locked mutex S_1, J_3 inherits J_1's priority. When J_3 releases S_2, it resumes its priority before acquiring S_2 (its original priority in this case). Then, J_1 preempts J_3 and locks S_2. J_1 now runs to completion followed by J_2 and J_3, respectively.

It can be seen that both jobs J_1 and J_2 had to wait for a lower-priority job J_3 for at most the duration a single critical section guarded by S_2. Since J_1 was blocked because it needed a mutex locked by another job, the blocking encountered by J_1 is called *direct blocking*. Direct blocking is necessary to guarantee the consistency of shared data. However, J_2 is blocked when J_3 inherits a priority higher than J_2. This type of blocking is referred to as *push-through blocking*. Push-through blocking is essential to avoid multiple blocking as illustrated in Example 3, and to avoid the uncontrolled priority inversion problem exhibited in Example 1.

Example 4: Consider the preceding example, where deadlocks could occur under the basic priority inheritance policy. Job J_2 locks the mutex S_2, and before it makes a nested request for mutex S_1, J_1 arrives and preempts J_2. We again have $c(S_1) = c(S_2) = p(J_1)$. However, when J_1 attempts to lock S_1, $p(J) = c(S^*) = c(S)$, but $SR = \{S_2\}$, $SL^* = \{S_2\}$, $SR^* = \{S_1\}$, and $S = S_1$, so that all locking conditions are false. Hence, the lock on S_1 is denied to J_1 and J_2 inherits J_1's priority. Thus, the deadlock is avoided.

We now provide an example that illustrates each of the locking conditions of OMP.

Example 5: Consider 5 jobs J_0, J_{1a}, J_{1b}, J_2, and J_3 in descending order of priority except that jobs J_{1a} and J_{1b} have equal priorities. There are three mutexes S_1, S_2, and S_3 in the system. Suppose the sequence of processing steps for each job is as follows:

$$\begin{aligned}
J_0 &= \{\cdots Lock(S_0)\cdots Unlock(S_0)\cdots\} \\
J_{1a} &= \{\cdots Lock(S_0)\cdots Unlock(S_0)\cdots\} \\
J_{1b} &= \{\cdots Lock(S_1)\cdots Unlock(S_1)\cdots\} \\
J_2 &= \{\cdots Lock(S_2)\cdots Lock(S_1)\cdots Unlock(S_1)\cdots Unlock(S_2)\cdots\} \\
J_3 &= \{\cdots Lock(S_1)\cdots Unlock(S_1)\cdots Lock(S_2)\cdots Unlock(S_2)\cdots\}.
\end{aligned}$$

Thus, $c(S_0) = p(J_0)$, $c(S_1) = p(J_{1b}) = p(J_{1a})$, and $c(S_2) = p(J_2)$

The sequence of events described below is depicted in Figure 11.1. A line at a low level indicates that the corresponding job is blocked or has been preempted by a higher-priority job. A line raised to a higher level indicates that the job is executing. The absence of a line indicates that the job has not yet arrived or has completed. Shaded portions indicate execution of critical sections. Suppose that:

- First, J_3 arrives and begins execution. At time t_0, it locks the unlocked mutex S_1 since there is no other mutex locked by another job.[4]
- At time t_1, J_2 arrives and preempts J_3.
- At time t_2, J_2 attempts to lock S_2. Since $p(J_2) < c(S_1)$, conditions C1 and C2 are false. But $p(J_2) = c(S_2)$ and $SR^*=\emptyset$. Hence, condition C3 is true and J_2 is allowed to lock S_2.
- At time t_3, J_0 arrives and preempts J_2.
- At time t_4, J_0 attempts to lock S_0. Now, $S^* = S_1$. However, $p(J_0) > c(S_1)$ and condition C1 is true. Hence, J_0 is granted the lock on S_0.
- At time t_5, J_0 releases the mutex S_0. J_{1a} arrives now but is unable to preempt J_0.
- At time t_6, J_0 completes execution. J_{1a}, which is eligible to execute, begins execution.
- At time t_7, J_{1a} tries to lock S_0. $S^* = S_1$. We have $p(J_{1a}) = c(S_1)$ and there is no nested request for mutex locks. Hence condition C2 is true, and the lock on S_0 is granted to J_{1a}.
- At time t_8, J_{1a} releases the mutex S_0.

[4]The locking can occur because the idle process has locked the dummy mutex S^* and $p(J_3) > c(S^*)$ by definition.

Figure 11.1 Sequence of Events Described in Example 5

- At time t_9, J_{1a} completes execution and J_2 resumes execution.
- At time t_{10}, J_2 attempts to lock the locked mutex S_1 and is blocked. J_3, which holds the lock on S_1, inherits J_2's priority and resumes execution.
- At time t_{11}, J_{1b} arrives and preempts J_3 executing at a lower-priority of $p(J_2)$.
- At time t_{12}, J_{1b} attempts to lock locked mutex S_1. J_{1b} is blocked, and J_3 now inherits J_{1b}'s priority.
- At time t_{13}, J_3 releases the mutex S_1 and resumes its original lowest priority. J_{1b} resumes execution and is now granted the lock on the mutex S_1, since condition C1 is satisfied w.r.t. S_2 locked by J_2.
- At time t_{14}, J_{1b} releases the mutex S_1.
- At time t_{15}, J_{1b} completes execution. J_2 resumes execution and locks S_1 since there is no mutex locked by a lower-priority job.
- At time t_{16}, J_2 releases the mutex S_1.
- At time t_{17}, J_2 releases the mutex S_2.
- At time t_{18}, J_2 completes execution and J_3 resumes.
- Finally, J_3 locks S_2, releases S_2, and completes execution at time t_{21}.

In the above example, jobs J_0 and J_{1a} do not encounter any blocking due to lower-priority jobs. J_{1b} is blocked by J_3 during the interval t_{12}–t_{13}, which corresponds to at most one critical section of J_3. J_2 is blocked by J_3 during the intervals t_{10}–t_{11} and t_{12}–t_{13}, which together correspond to at most one critical section of J_3.

11.2.2 Definition of the Optimal Mutex Policy

Having illustrated OMP with examples, we now formally define the policy.

1. Let J be the highest-priority job among the jobs ready to run. J is assigned the processor and let S^* be a mutex with the highest-priority ceiling of all mutexes currently locked by jobs other than job J. Let the job holding the lock on S^* be J^*. Before job J enters its critical section, it must obtain the lock on the mutex S guarding the shared data structure. If the mutex S is unlocked, job J will be granted the lock on S if and only if at least *one* of the locking conditions is true.
2. In this case, job J will obtain the lock on mutex S and enter its critical section. Otherwise, job J is said to be blocked by J^*. When a job J exits its critical section, the binary mutex associated with the critical section will be unlocked, and the highest-priority job, if any, blocked by job J will be awakened.

3. A job J uses its assigned priority unless it is in its critical section and blocks higher-priority jobs. If job J blocks higher-priority jobs, J *inherits* P_H, the executing priority of the highest-priority job blocked by J. When J exits its critical section, it resumes its previous priority. Finally, the operations of priority inheritance and of the resumption of original priority must be indivisible.

4. A job J, when it does not attempt to enter a critical section, can preempt another job J_L if its priority is higher than the priority, inherited or assigned, at which job J_L is executing.

11.2.3 Properties of the Optimal Mutex Policy

In this section, we prove that under OMP, each job may be blocked for at most the duration of one critical section of a lower-priority job and, furthermore, deadlocks are avoided.

We remind the reader that when the priority of a job J is being referred to, it always refers to the priority which J is currently executing (unless explicitly stated otherwise to be J's assigned lower-priority). Note that a job outside a critical section always executes at its own assigned priority, but a job's executing priority inside a critical section might change due to priority inheritance.

Lemma 11.2.1 *A job J can be blocked by a job J_L with an assigned lower priority only if J_L has entered and remains within a critical section when J arrives.*

Proof: It follows from the definition of OMP that if J_L is not in its critical section, it can be preempted by the higher-priority job J. Since priority inheritance is in effect and the highest-priority job that is ready will always be run, J_L cannot resume execution until J completes. The Lemma follows. □

Lemma 11.2.2 *Once a job J begins execution, a job with an equal assigned priority cannot begin until J completes.*

Proof: This lemma follows directly from the fact that priority inheritance is in effect, the highest-priority job is always run, and equal-priority ties are broken in FCFS order. □

Lemma 11.2.3 *When a job J executes, there can be at most one strictly lower-priority job J_L that has locked a mutex S_k such that $c(S_k) \geq p(J)$.*

Proof: Suppose that there exists another lower-priority job J_j that has locked a mutex S_j such that $c(S_j) \geq p(J)$. Without loss of generality, suppose that J_j locked S_j first. When J_L attempts to lock S_k, it finds that S_j has been locked by another job J_j, and that $p(J_L) < p(J) \leq min(c(S_k), c(S_j))$. Hence, locking conditions C1, C2, and C3 are false, and OMP will not permit J_L to lock S_k, contradicting our assumption. The Lemma follows. □

Lemma 11.2.3 also leads us to the following corollary.

Corollary 11.2.1 *Suppose that when J requests S, condition 1 is false (i.e., $c(S^*) \geq p(J)$). Then, J^* must be unique.*

Proof: This follows directly from Lemma 11.2.3. □

Remark: The above corollary gives us the following interesting result. When a job must be blocked, J^* needs to be identified as the blocking job. Corollary 11.2.1 shows that J^* is unique whenever $c(S^*) \geq p(J)$. When $c(S^*) < p(J)$, J^* may not be unique, but condition 1 will be true. As a result, J cannot be blocked by any of the J^*'s, and the non-uniqueness of J^* need not be resolved! In summary, if condition 1 is true, the mutex will be granted and the non-uniqueness of J^* need not be resolved. If not, J^* is guaranteed to be unique.

Remark: While J^* may not be unique when $c(S^*) < p(J)$, there can still be at most two jobs which have locked mutexes with the highest-priority ceiling. The following lemma proves this result.

Lemma 11.2.4 *There can be at most two jobs which can lock mutexes with the same-priority ceiling.*

Proof: Let J_L lock the mutex S_L. Let J be another job that locks mutex S with the same priority ceiling as S_L under OMP. We first show that $p(J) = c(S_L) = c(S)$. Since $c(S_L) = c(S)$, $p(J) \leq c(S_L)$. When J tries to lock S, there is at least one mutex (S_L) locked by another job. Hence $c(S^*) \geq c(S_L) \geq p(J)$. Therefore, condition C1 would evaluate to false. Since J does lock S, at least one of condition C2 or condition C3 must be true. In either case, we have $p(J) = c(S_L) = c(S)$.

Suppose that another job J' attempts to lock a mutex S' such that $c(S') = c(S_L) = c(S)$. There are three cases:

Case I: $p(J') > J$. In this case, J' cannot lock a mutex such that $c(S') = c(S_L)$: the definition of priority ceiling would be violated.

Case II: $p(J') = J$. However, by Lemma 11.2.2, J' cannot begin execution until J completes—a contradiction.

Case III: $p(J') < J$. In this case, when J' requests S', S_L and S are locked, and $c(S^*) \geq p(J)$. As a result, when J' tries to lock S', all three locking conditions evaluate to false. Hence, J' cannot lock S'.

The Lemma follows. □

We now show that the OMP avoids deadlocks and minimizes the worst-case priority inversion encountered by a job to the duration of a single critical section.

Lemma 11.2.5 *Suppose that job J enters a critical section z by obtaining the lock on mutex S because condition C1 of the* locking conditions *is true. Then, job J cannot be blocked by a lower-priority job until J completes.*

Proof: Since condition C1 is true when J requests the lock on S, $p(J) > c(S^*)$, whether S^* is unique or not. That is, no job with equal or higher-priority than J (including J) will lock the mutexes held by lower-priority jobs. Hence, no lower-priority job can block J or any other higher-priority job, and inherit a priority $\geq$

$p(J)$. Furthermore, no arriving job with priority lower than $p(J)$ can even preempt J. Thus, J cannot be blocked by lower-priority jobs before J completes. □

Remark: Lemma 11.2.5 provides the result that once a mutex S is locked by a job J because condition C1 is true, then *all* subsequent requests for mutex locks by job J will also satisfy condition C1, and hence all these locks will be granted.

Lemma 11.2.6 *Suppose that job J enters a critical section z by obtaining the lock on mutex S because condition C2 of the* locking conditions *is true. Then, job J cannot be blocked by a lower-priority job until J exits the critical section z.*

Proof: Since condition C2 is true when J requests the lock on S, $p(J) = c(S^*)$. By Corollary 11.2.1, J^* is unique. By Lemma 11.2.3, J^* is also the only job which has locked a mutex with priority ceiling $= p(J)$.

Also, no job with higher-priority than $p(J)$ will lock a mutex already locked by J^* or any preempted job with lower-priority than $p(J)$. Thus, J^* cannot inherit a priority higher than $p(J^*)$ unless it can lock additional mutexes. However, J^* can resume execution before J exits the critical section z only if J is blocked by J^*. However, the critical section z will not lock any mutexes already locked by J^*. Let the critical section z request a nested lock to mutex S. We again have $p(J) = c(S^*)$, and still the critical section z cannot lock any mutexes already locked by lower-priority jobs. Hence, OMP would allow J to lock S. Since this is true for all requests for mutex locks nested within the critical section z, job J will exit the critical section without being blocked by J^*. □

Remark: Lemma 11.2.6 provides the result that if a job J has locked the outermost mutex of a nested critical section because condition C2 is true, condition C2 will always be true for all subsequent nested requests for mutex locks within the critical section as well. Hence, no nested request to a mutex within this critical section will be blocked.

Definition: When a job J is blocked by the job J^*, let the mutex with the highest-priority ceiling locked by J^* be S^*. Then, S^* is said to be *used* to block J.

Lemma 11.2.7 *Suppose that job J enters a critical section z by obtaining the lock on mutex S, because condition 1 is false and condition C3 of the* locking conditions *is true. Then, the mutex S cannot be used by job J to block a higher-priority job.*

Proof: Since condition 1 is false, by Lemma 11.2.1, J^* must be unique. Since condition C3 is true when J requests the lock on S, $c(S) = p(J)$. Hence, no higher-priority job J_H will lock S. J^* is the only job that can inherit a priority higher than or equal to J, but J^*'s current critical section does not need S. Hence, J's critical section guarded by S cannot inherit a priority that is higher than or equal to J_H's priority. Hence, S cannot be used by job J to block job J_H. The Lemma follows. □

Definition: If job J_i is blocked by J_j and J_j, in turn, is blocked by J_k, J_i is said to be *transitively blocked* by J_k.

Lemma 11.2.8 *The optimal mutex policy prevents transitive blocking.*

Proof: Suppose that transitive blocking is possible. For some $i \geq 2$, let job J_i block job J_{i-1} and let job J_{i-1} block job J_{i-2}, i.e., job J_{i-2} is transitively blocked by job J_i. By Lemma 11.2.1, to block job J_{i-1}, job J_i must enter and remain in its critical section when J_{i-1} arrives at time t_0. Similarly, to block J_{i-2}, job J_{i-1} must enter and remain in its critical section when J_{i-2} arrives at time t_1. At time t_1, let the mutexes with the highest-priority ceilings locked by jobs J_i and J_{i-1} be S and S_n, respectively. Since job J_{i-1} is allowed to lock mutex S_n when job J_i has already locked S, one of the *locking conditions* must have been true. If one of conditions C1 and C2 were true, by Lemmas 11.2.5 and 11.2.6, job J_i will be unable to block job J_{i-1}. Since J_i does block job J_{i-1} by assumption, condition C3 must have been true when J_{i-1} locked S_n. However, according to Lemma 11.2.7, the mutex S_n cannot be used by job J_{i-1} to block job J_{i-2}, contradicting our assumption. The Lemma follows. □

Theorem 11.2.1 *The optimal mutex policy prevents deadlocks.*

Proof: First, by assumption, a job cannot deadlock with itself. Thus, a deadlock can be formed only by a cycle of jobs waiting for one another. Let the n jobs involved in this cycle be $J_1, \ldots, J_n$. Since a job not holding any mutexes cannot contribute to the deadlock, each of the n jobs must be in its critical section. By Lemma 11.2.8, the number of jobs in the blocking cycle can only be 2, i.e., $n = 2$. Suppose that job J_2's critical section was preempted by job J_1, which then enters its own critical section. For J_1 to enter its critical section, one of the *locking conditions* must be true. If conditions C1 and C2 were true, by Lemmas 11.2.5 and 11.2.6, job J_2 cannot block J_1. Hence, condition C3 must have been true and by Lemma 11.2.3, $J^* = J_2$. Since condition C3 is true, each of the critical sections of jobs J_1 and J_2 is guaranteed not to have mutually locked mutexes that are expected by the other. Hence a deadlock cannot occur. The Theorem follows. □

Remark: The above theorem leads to the useful result that programmers can write arbitrary sequences of nested requests for mutex locks when OMP is used. As long as each job does not deadlock with itself, the system is guaranteed to be deadlock-free.

We now prove that under OMP, a job can be blocked for at most the duration of one critical section of lower-priority jobs.

Theorem 11.2.2 *A job J can be blocked for at most the duration of one critical section of lower-priority jobs.*

Proof: When the job J arrives, by Lemma 11.2.3, there can exist at most a single job J' that has locked a mutex S_k such that $c(S_k) \geq p(J)$. If no such job exists, no lower-priority job can inherit a priority higher than J, and condition 1 will always evaluate to true when J requests a mutex. As a result, J will run to completion without being blocked.

If there does exist such a job, we have $J' = J^*$. That is, J' is the only lower-priority job that can inherit a priority higher than that of J. Suppose that J' does inherit a higher priority than J. By Lemma 11.2.8, J^* will exit its critical section without being blocked by a lower-priority job. Once J^* exits its critical section, by Lemma 11.2.1, J^* can no longer block J. Since there exists no other job that can block J and no arriving lower-priority job can block J, the Theorem follows. □

The following corollary can be derived from Theorem 11.2.2.

Corollary 11.2.2 *A job J which voluntarily suspends itself k times can be blocked for at most the duration of $k+1$ critical sections of lower-priority jobs.*

Proof: The Corollary follows from Theorem 11.2.2 and the fact that a job that suspends k times can be considered to be $k+1$ jobs. □

11.2.4 Necessity and Sufficiency of the *Locking Conditions*

We now prove that a worst-case blocking duration of a single critical section can be guaranteed if and only if the *locking conditions* of OMP are used.[5] In the preceding section, we have shown that the locking conditions are sufficient to avoid deadlocks and to reduce the blocking duration of a job to at most a single critical section.

We first prove that a lock on an unlocked mutex S can be granted to a job J only if at least one of the locking conditions is true in order to prevent deadlocks and obtain the worst-case blocking of a single critical section for each job.

Lemma 11.2.9 *A job can be blocked for the duration of more than one critical section if a lock on an unlocked mutex S is granted to a job J when the following two conditions are true:*

condition (a) $p(J) < c(S^*)$

condition (b) $p(J) < c(S)$

Proof: When J tries to lock S, suppose that conditions (a) and (b) are true. If $S = S^*$, J is attempting to lock a locked binary mutex S and has to be blocked. Hence, J can possibly be granted the lock on S only if $S \neq S^*$. Then, there exist jobs J_i and J_j with higher-priority than J such that J_i will lock S^* and J_j will lock S.

Only three cases arise.

Case I: $J_i = J_j = J_H$. In other words, there exists a higher-priority job J_H that will lock both S and S^*. If the lock on S is granted to J, J_H can arrive now and will find that both the mutexes S and S^* that it requires are locked. Hence, J_H will be blocked for the duration of two critical sections, once to wait for J to release S and again to wait for J^* to release S^*.

[5] For the worst-case blocking to be a single critical section, the system should be free from deadlocks since a deadlock contributes to prolonged blocking of two or more jobs.

Case II: $J_i \neq J_j$ and without loss of generality, J_i has higher-priority than J_j. Suppose that the lock on S is granted to J. Job J_j arrives now and preempts J. Immediately, J_j can be preempted by J_i. Job J_i will be block for one critical section, waiting for J^* to release S^*. However, this constitutes push-through blocking for J_j as well. When J_j resumes after J_i completes, J_j will be blocked for the duration of one more critical section, waiting for J to release S. Thus, job J_j can be blocked for the duration of two critical sections.

Case III: $J_i \neq J_j$, but both jobs have equal priority. Suppose that the lock on S is granted to J. Job J_j arrives now followed by J_i. However, J_i is unable to preempt J_j. J_j attempts to lock S and is blocked by J, which inherits J_j's priority until the release of S. This constitutes blocking for J_i as well. After J_j completes, J_i begins execution and will again be blocked by J^* when it attempts to lock S^*. Thus, J_i can be blocked for the duration of two critical sections.

Thus, if both conditions (a) and (b) are true, a job can be blocked for the duration of more than one critical section. □

Remark: Suppose that a worst-case blocking of at most a single blocking has to be ensured. Hence, when a mutex is requested, and conditions (a) and (b) are satisfied, a job should be blocked. Thus, for a job J to be granted the lock on a mutex S, the negation of Lemma 11.2.9 must hold. Since $p(J) > c(S)$ is not possible by definition, at least one of the following conditions must be true:

$p(J) \geq c(S^*)$

$p(J) = c(\mathrm{S})$

We refer to the above conditions as *necessary locking conditions*. Thus, Theorem 11.2.9 states that for a worst-case blocking duration of a single critical section, at least one of the necessary locking conditions must be true for a job to be granted the lock on a mutex. However, the necessary locking conditions are only necessary but not sufficient to guarantee a worst-case blocking of a single critical section.

Remark: If there are no nested requests for mutex locks at all, the *necessary locking conditions* are equivalent to the *locking conditions*. Thus, the additional checks in the *locking conditions* are needed to avoid deadlocks and to prevent a job from being blocked for multiple critical sections.

Remark: The *necessary locking conditions* provide us with the insight to construct policies that are computationally simpler but suboptimal. Note that if both the *necessary locking conditions* are false, then it follows that the *locking conditions* are also false.

Lemma 11.2.10 *Suppose that job J attempts to lock mutex S. If all three* locking conditions *are false, at least one of the following conditions must be true:*

(F1) $(p(J) < c(S)) \wedge (p(J) < c(S^*))$.[6]

(F2) $(p(J) < c(S)) \wedge (p(J) = c(S^*)) \wedge (SR \cap SL^* \neq \emptyset)$.

[6] That is, the *necessary locking conditions* are false.

(F3) $(p(J) = c(S)) \wedge (p(J) < c(S^*)) \wedge (S \in SR^*)$.

(F4) $(SR \cap SL^* \neq \emptyset) \wedge (S \in SR^*)$.

Proof: The Lemma follows directly from the negation of the *locking conditions*. □

Theorem 11.2.3 *Deadlock can occur or a job can be blocked for the duration of more than one critical section if the lock on a mutex S is granted to a job J when all the* locking conditions *are false, i.e.,* $(\neg C_1 \wedge \neg C_2 \wedge \neg C_3 \Rightarrow \exists$ *J, which can be blocked for more than one critical section)* $\vee$ $(\neg C_1 \wedge \neg C_2 \wedge \neg C_3 \Rightarrow \exists$ *a deadlock).*

Proof: Suppose that all the locking conditions are false. By Lemma 11.2.10, one of the following cases must be true.

Case I: (F1) The *necessary locking conditions* are false. It follows from Theorem 11.2.9 that at least one job can block for the duration of more than one critical section.

Case II: (F2) $(p(J) < c(S)) \wedge (p(J) = c(S^*)) \wedge (SR \cap SL^* \neq \emptyset)$. That is, there exists a job J_H with higher-priority than J that will try to lock S. Moreover, the current critical section of J will try to lock a mutex S_i that has already been locked by the current critical section of J^*. Suppose that the lock on S were granted to J. However, J_H can arrive now and later attempt to lock S already locked by J. In order to release S, J would need to lock S_i held by J^*. Consequently, J_H will be blocked until J releases S. But J will be blocked until J^* releases S_i. Effectively, J_H would be blocked for the duration of two critical sections. Thus, a job can block for the duration of multiple critical sections.

Case III: (F3) $(p(J) = c(S)) \wedge (p(J) < c(S^*)) \wedge (S \in SR^*)$. That is, there exists a higher-priority job J_H that will try to lock S^*. Moreover, the current critical section of J^* will try to lock S. Suppose that the lock on S is granted to J. However, J_H can arrive now and later attempt to lock S^* already locked by J^*. Consequently, J_H will be blocked until J^* releases S^*. But, J^* will be blocked until J releases S. Effectively, J_H would block for the duration of two critical sections. Thus, a job can block for the duration of multiple critical sections.

Case IV: (F4) $(SR \cap SL^* \neq \emptyset) \wedge (S \in SR^*)$. Clearly, if the lock on S were granted to J, jobs J and J^* will deadlock, with one waiting for the other to release a mutex. Since these jobs are deadlocked, two jobs will be blocked for an infinite duration of time.

Thus, if all the *locking conditions* were false and the lock on mutex S is granted to job J, in the worst case, a deadlock can occur or a job can block for the duration of multiple critical sections. □

The above theorem leads us to the necessity and sufficiency of the *locking conditions*.

Theorem 11.2.4 *The* locking conditions *are necessary and sufficient to obtain the worst-case blocking duration of a single critical section and to avoid deadlocks.*

Proof: The Theorem follows from Theorems 11.2.1, 11.2.2, and 11.2.3. □

11.2.5 Schedulability Analysis

OMP places an upper bound on the duration that a job can be blocked. This property makes possible the schedulability analysis of a task set using rate-monotonic priority assignment and OMP. We also show that it is possible that OMP can potentially lead to better schedulability than previously known priority inheritance policies since it is possible for a task τ_i to encounter a better worst-case blocking duration from lower-priority tasks under OMP. We quote below the following theorem due to Sha, Rajkumar, and Lehoczky [17].

Theorem 11.2.5 *A set of n periodic tasks can be scheduled by the rate-monotonic algorithm if the following conditions are satisfied [17]:*

$$\forall i, \quad 1 \leq i \leq n, \qquad \frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_n}{T_n} + \frac{B_i}{T_i} \leq i(2^{1/i} - 1)$$

where C_i, T_i, and B_i are the worst-case execution time, period, and worst-case blocking time of a periodic task, respectively.

Again, C_i is the computation time of periodic task τ_i and T_i is its period. B_i is the worst-case blocking encountered by jobs belonging to τ_i. Just like the priority ceiling policy, OMP also avoids deadlocks and produces a worst-case blocking of at most the duration of one critical section. Hence, the above theorems are applicable to OMP as well. The value of B_i for each job J_i is computed as follows.

Definition: Jobs J_i and J_j are said to be *active* together if J_i and J_j can both be ready to execute on the processor at some instant in time.

For instance, in Example 5, jobs J_2 and J_3 are said to be active together since both are ready to execute at time t_1. However, jobs J_{1a} and J_{1b}, in reality, can be the execution of a single instance of task τ_1 with an intervening suspension (mostly for communication or I/O activities). That is, job J_1 actually consists of two jobs J_{1a} and J_{1b}. Since J_{1a} always precedes the initiation of job J_{1b}, jobs J_{1a} and J_{1b} are *not* said to be active together.

A job can be *blocked* only by jobs with lower-priority. Then, a critical section z_j of a lower-priority job J_j guarded by a mutex $S_{j,k}$ can block J_i if

- $p(J_i) < c(S_{j,k})$.
- $p(J_i) = c(S_{j,k}) \wedge J_i$ (or an equal-priority job that is active with J_i) may lock $S_{j,k}$.

The set of maximal elements of $\beta_{i,j}$, $\beta^*_{i,j}$, is formed by eliminating those critical sections nested inside other elements of $\beta_{i,j}$. Then, B_i is equal to the length of the longest critical section in $\beta^*_i = \cup_{j>i} \beta^*_{i,j}$, the set of all outermost critical sections that can block J_i.

Worst-Case Superiority of OMP over Previous Policies

If only condition 1 of the OMP *locking conditions* is used, we obtain the priority ceiling policy (PCP) defined in [17], where the use of condition 1 is shown to be

a sufficient condition to obtain the deadlock prevention property and a worst-case blocking of at most one critical section of a lower-priority task.

Consider the use of the priority ceiling policy on Example 4. Both jobs J_2 and J_{1a} would not have been allowed to enter their respective critical sections when S_1 is locked. This example illustrates the fact that OMP can lead to lower worst-case blocking relative to the priority ceiling policy. In particular, consider the case where each job J_1 of task τ_1 voluntarily suspends itself once (say for I/O) such that it executes as sub-jobs J_{1a} and J_{1b}. Under the priority ceiling policy, both of these sub-jobs can be blocked when they try to lock S_0 and S_1, respectively. Hence, each job of τ_1 will be blocked for the duration of two critical sections of lower-priority tasks. However, under OMP, whenever J_{1a} tries to lock S_0, it will always find that either condition 1 or condition 2 of OMP will be true. Thus, J_{1a} is never blocked and B_1 under OMP will be shorter than in the case of the priority ceiling policy. As a result, it is possible for OMP to lead to a better worst-case schedulability analysis than PCP.

11.3 Conclusions

Synchronization primitives used in real-time systems should bound the blocking duration that a job can encounter. Unfortunately, a direct application of commonly used primitives like semaphores, monitors, and Ada rendezvous can lead to unbounded priority inversion, where a high-priority job can be blocked by a lower-priority job for an arbitrary amount of time. Priority inheritance policies solve this unbounded priority inversion problem and bound the blocking duration that a job can experience. We have presented an optimal priority inheritance policy that not only bounds the worst-case blocking duration of a job to that of a single critical section but also prevents deadlocks. It can also be shown that this policy is also optimal in the sense that no other priority inheritance policy can guarantee a better worst-case blocking duration [13]. Other approximations to the policy which can be easier to implement are also possible.

Acknowledgments

The authors wish to thank Ted Baker for pointing out that J^* may not be unique, and for his many comments on this chapter. The authors would also like to thank many reviewers, including Prashant Waknis and Joo Yong Kim, for their valuable comments. This chapter is based on work reported in [12, 13].

References

[1] Baker, T. P. A stack-based resource allocation policy for real-time processes. *IEEE Real-Time Systems Symposium*, December 1990.

[2] Chen, M. I., and Lin, K.-J. Dynamic priority ceilings: a concurrency control protocol for real-time systems. *UIUCDCS-R89-1511, Technical Report*, Department of Computer Science, University of Illinois at Urbana-Champaign, 1989.

[3] Clark, D. D. The structuring of systems using upcalls. *The Tenth Symposium on Operating System Principles*, 1985.

[4] Jeffay, K. Scheduling sporadic tasks with shared resources in hard real-time systems. *TR90-038, Technical Report*, Department of Computer Science, University of North Carolina at Chapel Hill, November 1989.

[5] Lampson, B. W., and Redell, D. D. Experience with processes and monitors in mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[6] Lehoczky, J. P., and Sha, L. Performance of real-time bus scheduling algorithms. *ACM Performance Evaluation Review, Special Issue*, 14(1), May 1986.

[7] Lehoczky, J. P., Sha, L., and Ding, Y. The rate monotonic scheduling algorithm: exact characterization and average-case behavior. *IEEE Real-Time Systems Symposium*, December 1989.

[8] Leinbaugh, D. W. Guaranteed response time in a hard real-time environment. *IEEE Transactions on Software Engineering*, January 1980.

[9] Leung, J. Y., and Merrill, M. L. A note on preemptive scheduling of periodic, real time tasks. *Information Processing Letters*, 11(3):115–118, November 1980.

[10] Liu, C. L., and Layland, J. W. Scheduling algorithms for multiprogramming in a hard real time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[11] Mok, A. K. Fundamental design problems of distributed systems for the hard real time environment. *Ph.D. thesis*, M.I.T., 1983.

[12] Rajkumar, R. Task synchronization in real-time systems. *Ph.D. thesis*, Carnegie Mellon University, August 1989.

[13] Rajkumar, R., Sha, L., Lehoczky, J. P., and Ramamritham, K. An optimal priority inheritance protocol for real-time synchronization. *Technical Report*, IBM Thomas J. Watson Research Center, 1991.

[14] Ramaritham, K., and Stankovic, J. A. Dynamic task scheduling in hard real-time distributed systems. *IEEE Software*, July 1984.

[15] Sha, L., Lehoczky J. P., and Rajkumar, R. Solutions for some practical problems in prioritized preemptive scheduling. *IEEE Real-Time Systems Symposium*, 1986.

[16] Sha, L., and Goodenough, J. B. Real-time scheduling theory and ada. *Computer*, May 1990.

[17] Sha, L., Rajkumar, R., and Lehoczky, J. P. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, pages 1175–1185, September 1990.

[18] SofTech Inc. Designing real-time systems in ada. *Final Report 1123-1*, SofTech, Inc., January 1986.