AN OPTIMAL PRIORITY INHERITANCE PROTOCOL
for
REAL-TIME SYNCHRONIZATION

R. Rajkumar, L. Sha, J.P. Lehoczky
and K. Ramamritham

COINS Technical Report 88-98

October 17,1988

An Optimal Priority Inheritance Protocol for Real-Time Synchronization

Ragunathan Rajkumar[1], Lui Sha[2,3] and John P. Lehoczky[4]
[1]Department of Electrical and Computer Engineering
[2]Department of Computer Science
[3]Software Engineering Institute
[4]Department of Statistics
Carnegie Mellon University

and

Krithi Ramamritham
Department of Computer and Information Science
University of Massachusetts

17 October 1988

This work was sponsored in part by the Office of Naval Research under contracts N00014-84-K-0734 and N00014-85-K-0398, in part by Naval Ocean System Center under contract N66001-87-C-0155, and in part by the Federal Systems Division of IBM Corporation under University Agreement YA-278067.

Table of Contents

Abstract

In priority-driven preemptive scheduling, resources should, ideally, always 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, which is the processor utilization below which all job deadlines are guaranteed to be met. 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 paper, we investigate protocols belonging to the class of *priority inheritance protocols* that minimize priority inversion. We develop a priority inheritance protocol called the *Semaphore Control Protocol* which has two properties: deadlocks are avoided and the worst-case blocking duration of a job is reduced to the duration of execution of a single critical section. The protocol is optimal in the sense that the protocol embeds necessary and sufficient conditions to obtain these two desirable properties. Finally, we consider implemenation issues and present other protocols which are computationally simpler than the semaphore control protocol but are suboptimal.

1. Introduction

1.1. Background

Hard real-time systems operate under rigid time-constraints and include such applications as avionics systems, production control, robotics, and defense systems. Jobs in such systems must meet hard deadlines and failure to do so might lead to system failure. The scheduling of jobs in hard real-time systems has been a significant area of research in real-time computer systems. Both non-preemptive and preemptive scheduling algorithms have been studied [2, 3, 4, 5, 6, 9, 10]. An important problem that arises in the context of such real-time systems is the effect of blocking caused by the need for the synchronization of jobs that share logical or physical resources. Mok [7] 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 run-time. Mok [7] 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. Zhao, Ramamritham and Stankovic [12, 13] investigated the use of heuristic algorithms to generate feasible schedules given specific resource requirements of tasks. Their heuristic has a high probability of success in the generation of feasible schedules.

In this paper, we investigate the synchronization problem in the context of priority-driven preemptive scheduling, an approach used in many real-time systems. 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 can not only cause deadlines to be missed at low levels of resource utilization but, perhaps more importantly, render these systems to be less predictable. In this paper, we develop the family of *priority inheritance protocols* suggested by Sha, Rajkumar and Lehoczky [11] and prove the properties of an optimal

protocol belonging to this family. The priority inheritance protocols, defined in the context of a uniprocessor, rectify the uncontrolled priority inversion problem that can result from an unjudicious use of traditional synchronization primitives.

The paper is organized as follows. We describe the priority inversion problem and present the basic concepts underlying the priority inheritance protocols in Section 1.2. In Section 2, we define the *Semaphore Control Protocol* and investigate its properties. We show that under the protocol, 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 protocols on schedulability analysis when the rate-monotonic algorithm is used. In Section 3, we develop approximations to the semaphore control protocol which can be implemented very efficiently and describe an O(1) implementation of the semaphore control protocol. Finally, Section 4 presents some concluding remarks.

1.2. The Concept of Priority Inheritance

Priority inheritance protocols are an approach to rectify the priority inversion problem when synchronization mechanisms are in use. *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 semaphore 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 [1] in the context of monitors. They suggest that each monitor be always 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 use of priority inheritance protocols is one approach to rectify the priority inversion problem inherent in existing synchronization primitives. The basic idea of priority inheritance protocols is that when a job J blocks higher priority jobs, it executes its critical section at the highest priority level of all of the blocked jobs. After exiting its critical section, job J returns to

its original priority level. To illustrate this idea, we apply this protocol to Example 1. Suppose that job J_1 is blocked by J_3. The priority inheritance protocols 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 be immediately 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 protocols based on when a job is defined to be blocked by a lower priority job. For instance, the simplest priority inheritance protocol stipulates that a lower priority job inherit the priority of a higher priority job when the latter tries to lock a semaphore already locked by the lower priority job. Such a protocol is called the *basic priority inheritance protocol* [11]. However, as we shall see, the basic priority inheritance protocol can still lead to avoidable priority inversion and/or deadlocks. Our goal in this paper is to develop a priority inheritance protocol 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.

1.3. Assumptions and Notation

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

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.

Notation: J_i denotes a job, namely an instance of a periodic task τ_i. P_i and T_i denote the priority and period 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 $p(J_i)$.

In all our discussions below, we assume that jobs $J_1, J_2, \cdots, J_n$ are listed in descending order of priority with J_1 having the highest priority.

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

Notation: A binary semaphore guarding shared data and/or a shared resource is denoted by S_i. $\mathbf{P}(S_i)$ and $\mathbf{V}(S_i)$ denote the indivisible operations *lock* (wait) and *unlock* (signal) respectively on the binary semaphore S_i. The section of code beginning with the locking of a semaphore and ending with the unlocking of the semaphore is termed a *critical section.*

A job can have multiple critical sections that do not overlap, e.g. $\{ \cdots \mathbf{P}(S_1) \cdots \mathbf{V}(S_1) \cdots \mathbf{P}(S_2) \cdots \mathbf{V}(S_2) \cdots \}$. A critical section can be nested, i.e. a job J_i may make nested requests for

semaphore locks, e.g. $\{\cdots, \mathbf{P}(S_1)\cdots\ \mathbf{P}(S_2)\cdots\ \mathbf{V}(S_2), \cdots, \mathbf{V}(S_1), \cdots\}$. In this case, critical section $z_{i,1}$ is bounded by $\mathbf{P}(S_1)$ and $\mathbf{V}(S_1)$ 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 $\mathbf{P}(S_1)$ and ending with $\mathbf{V}(S_1)$. We shall use the terms "critical section" and "outermost critical section" interchangeably.

The j^{th} critical section in job J_i is denoted by $z_{i,j}$ and corresponds to the code segment of job J_i between the j^{th} **P** operation and its corresponding **V** operation. The semaphore 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 duration of the execution of the critical section $z_{i,j}$, denoted $d_{i,j}$, is the time 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 $z_{i,j} \cap z_{i,k} = \varnothing$. In addition, we assume that a semaphore may be locked at most once in a single nested critical section. This implies that a job will not attempt to lock a semaphore that it has already locked and thus deadlock with itself. In addition, we assume that locks on semaphores 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 semaphore S, if the critical section $z_{i,j}$ blocks J and $S_{i,j} = S$.

Finally, we assume a uni-processor executing a fixed set of tasks. Specifically, we assume that the tasks in the system as well as their semaphore needs are known *a priori*. However, we assume that *no* relative timing information is available about the instants at which tasks will attempt to enter their critical sections. This assumption implies that only syntactic information regarding the semaphore locks needed by each job is available, since timing information is difficult to obtain, and can vary with stochastic execution requirements of tasks and even minor software changes.

An important feature of the protocol that we propose is that it is possible to determine the *schedulability bound* for a given task set when this protocol 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 protocol 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, $\beta_{i,j} = \{z_{j,k} \mid j>i$ and $z_{j,k}$ can block $J_i\}$.[1]

---

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

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 $\beta^*_{i,j} = \{z_{j,k} \mid (z_{j,k} \in \beta_{i,j}) \wedge (\sim\exists\, z_{j,m} \in \beta_{i,j}$ 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.

2. The Semaphore Control Protocol

The basic priority inheritance protocol [11] stipulates that when a job J attempts to lock a semaphore 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, such a protocol suffers from two problems. First, a job J could be blocked for the duration of $min(m, n)$ critical sections, where n is the number of lower priority jobs that attempt to lock a semaphore with a priority ceiling higher than or equal to $p(J)$ and m is the number of distinct semaphores 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_2. Later, J_2 locks S_1. Job J_1 arrives at this instant and finds that the semaphores S_1 and S_2 have been respectively locked by the lower priority jobs J_3 and J_2. 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.

Secondly, the protocol does not avoid deadlocks. For instance, consider jobs J_1 and J_2. J_1 makes nested requests to lock semaphores S_1 and S_2 in that order. J_1 locks S_2 and 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 semaphore S_1. When J_2 attempts to lock S_2, it gets blocked and J_2 inherits J_1's priority but a deadlock situation occurs. Hence, explicit deadlock avoidance techniques like total ordering of semaphore requests may have to be employed if the basic priority inheritance protocol is used.

Intuitively, it can be seen that the basic priority inheritance protocol runs into its problems for the following reason. An unlocked semaphore is allowed to be locked at any instant irrespective of its relationship to the semaphores that have been already locked. Hence, when a higher priority job arrives, it can find that several semaphores 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 semaphores to be locked only under selective conditions. In other words, if the locking of a semaphore may cause multiple blocking to a higher priority job, we should not allow the semaphore to be locked. We use the information about the semaphore needs of each job and the job priorities to decide whether the locking of a semaphore can lead to multiple blocking and/or deadlock. Imposing conditions on the locking of a semaphore is the essence of the proposed protocol.

In this section, we develop a priority inheritance protocol, called the *semaphore control protocol.* The protocol 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 protocol and prove its properties. We shall show that the locking conditions used by the semaphore control protocol 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 protocol may be expensive. Implementation issues are considered in Section 3.1. Suboptimal but computationally simpler protocols are also discussed.

2.1. The Concept of the Semaphore Control Protocol

Definition: The *priority ceiling* of a semaphore S is defined as the priority of the highest priority task that may lock S. The priority ceiling of a semaphore 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 semaphore 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 semaphore 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 semaphore S,

- S^* is the semaphore with the highest priority ceiling locked by jobs with lower priority than J. If there is no semaphore currently locked, S^* is defined to be a dummy semaphore S_{dummy} whose priority ceiling is less than the priorities of all jobs in the system.
- J^* is the job holding the lock on S^*. If S^* is the dummy semaphore 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 semaphores already locked by the current critical section of job J^*.[2]
- SR is the set of semaphores that the current critical section of J may lock later.[3] Note that SL and SR are relevant to a job only if it is already inside a critical section. 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 semaphores 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 semaphore 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 semaphores that can be

---

[2] SL stands for "semaphores locked".

[3] SR stands for "semaphores required" for completion of the current critical section.

locked by the current critical section of J.

As already mentioned, the semaphore control protocol selectively grants locks on unlocked semaphores to jobs. Suppose that job J requests the lock on an unlocked semaphore S. The semaphore control protocol 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 semaphore already locked by J^*, i.e. $(p(J) = c(S^*)) \wedge (SR \cap SL^* = \varnothing)$.
3. **Condition C3**: The priority of job J is equal to the priority ceiling of S and the lock on semaphore 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 priority. We refer to conditions C1, C2 and C3 as the *locking conditions*.

Under the semaphore control protocol, 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 protocol with a few examples. We shall first apply the semaphore control protocol to the examples in the previous 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 the 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 semaphore locks and $SR=\varnothing$). Hence J_1 can obtain the lock on S_1. Later, when J_1 attempts to lock the locked semaphore S_1, J_3 inherits J_1's priority. When J_3 releases S_2, it resumes its original priority, and J_1 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 semaphore 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 previous example where deadlocks could occur under the basic priority inheritance protocol. Job J_2 locks the semaphore S_2 and before it makes a nested request for semaphore 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 $S=S_1$ such 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 the semaphore control protocol.

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 semaphores S_1, S_2 and S_3 in the system. Suppose the sequence of processing steps for each job is as follows:

$$J_0 = \{ \cdots, \mathbf{P}(S_0), \cdots, \mathbf{V}(S_0), \cdots \}$$

$$J_{1a} = \{ \cdots, \mathbf{P}(S_0), \cdots, \mathbf{V}(S_0), \cdots \}$$

$$J_{1b} = \{ \cdots, \mathbf{P}(S_1), \cdots, \mathbf{V}(S_1), \cdots \}$$

$$J_2 = \{ \cdots, \mathbf{P}(S_2), \cdots, \mathbf{P}(S_1), \cdots, \mathbf{V}(S_1), \cdots, \mathbf{V}(S_2), \cdots \}$$

$$J_3 = \{ \cdots, \mathbf{P}(S_1), \cdots, \mathbf{V}(S_1), \cdots, \mathbf{P}(S_2), \cdots, \mathbf{V}(S_2), \cdots \}$$

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

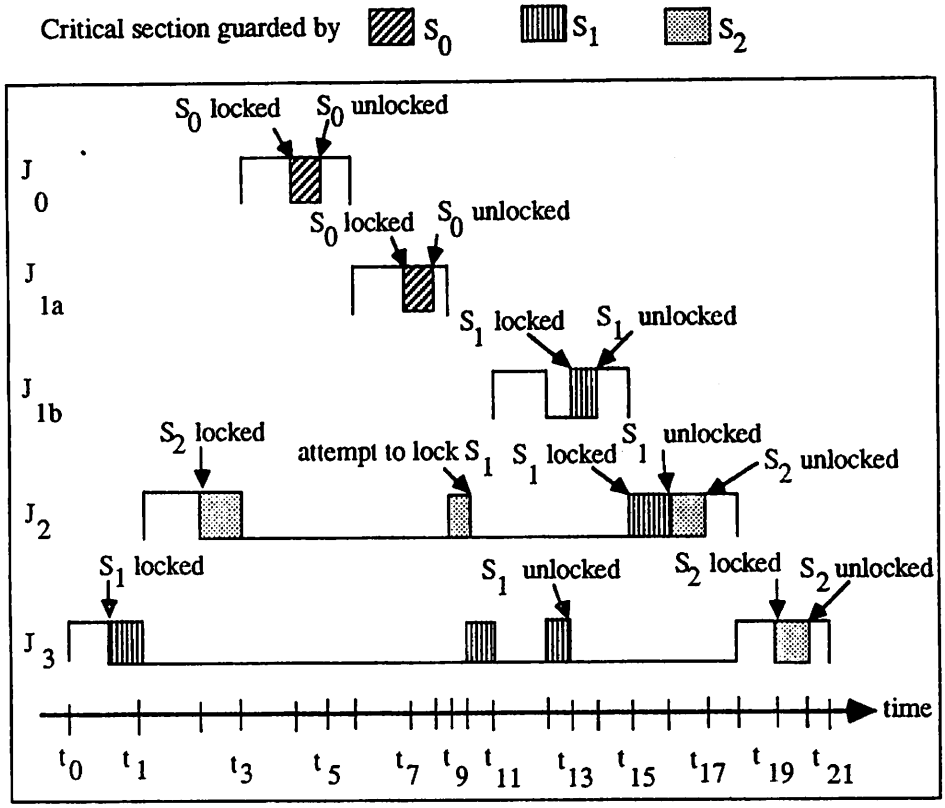

Figure 2-1: Sequence of Events described in Example 5.

The sequence of events described below is depicted in Figure 2-1. 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

- At time t_0, J_3 arrives and begins execution. Then it locks the unlocked semaphore S_1 since there is no other semaphore 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 semaphore 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 semaphore 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 semaphore S_0.
- At time t_9, J_{1a} completes execution and J_2 resumes execution.
- At time t_{10}, J_2 attempts to lock the locked semaphore 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 semaphore S_1. J_{1b} is blocked, and J_3 now inherits J_{1b}'s priority.
- At time t_{13}, J_3 releases the semaphore S_1 and resumes its original lowest priority. J_{1b} resumes execution and is now granted the lock on the semaphore S_1, since condition C1 is satisfied w.r.t. S_2 locked by J_2.
- At time t_{14}, J_{1b} releases the semaphore S_1.
- At time t_{15}, J_{1b} completes execution. J_2 resumes execution and locks S_1 since there is no semaphore locked by a lower priority job.
- At time t_{16}, J_2 releases the semaphore S_1.
- At time t_{17}, J_2 releases the semaphore 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}.

---

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

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 $\{t_{12}\text{-}t_{13}\}$ which corresponds to at most one critical section of J_3. J_2 is blocked by J_3 during the intervals $\{t_{10}\text{-}t_{11}\}$ and $\{t_{12}\text{-}t_{13}\}$ which together correspond to at most one critical section of J_3. We shall draw some more conclusions from this example in Section 3.1.

We now provide an intuitive explanation about the roles played by the three *locking conditions*, C1, C2 and C3. The properties of the semaphore control protocol resulting from the use of the three conditions are formally developed in subsequent sections.

Condition C1, $p(J) > c(S^*)$, checks whether the critical section z will be executed at a strictly higher priority than all preempted critical sections, taking priority inheritance into account. By definition, a preempted critical section z_j can inherit a priority at most equal to the highest priority ceiling of the semaphore(s) that z_j has already locked. Since, under condition C1, J's priority is greater than the highest priority J^* can inherit, condition C1 ensures a priority ordering of critical section executions. Such a priority ordering of critical section executions prevents deadlocks [11]. That is, critical sections are always executed in an order that is consistent with a pre-defined total ordering namely the priority ordering of the tasks.

In addition, such a total ordering of critical section executions also leads to the worst-case blocking of a single critical section. If a high priority job J_H can be blocked by two lower priority jobs J_M and J_L, then J_L would inherit J_H's priority which is higher than that of the critical section of J_M. This contradicts the total ordering property. To ensure total ordering, J_H will not be allowed to enter its critical section.

Although condition C1 is sufficient to induce a prioritized total ordering of critical section execution, it is not necessary. Together, conditions C1 and C2 form the necessary and sufficient conditions for total ordering. Condition C2 checks whether $p(J)=c(S^*)$ which implies that a preempted critical section can inherit at most a priority equal to $p(J)$. However, this priority inheritance can take place only if J is blocked by J^*. If critical section z does not attempt to lock a semaphore already locked by J^*, J cannot be blocked by J^* until z is exited. This ensures that the critical section z of J can still be executed at a strictly higher priority than those of preempted critical sections. Hence, the total ordering of the priorities of critical sections is maintained, and deadlocks are avoided. This condition, along with condition C1, again ensures that only one preempted critical section of J^* can inherit a priority higher than or equal to that of a higher priority job J. If a preempted critical section of J^* can inherit a priority equal to that of J, then, when condition 2 is true, the priority inheritance cannot occur until J exits its current critical section.

Finally, it turns out that the prioritized total ordering is itself a sufficient condition for obtaining the properties of freedom from deadlock and a worst-case blocking duration of one critical section. The addition of condition C3 completes the necessary and sufficient conditions for these two desired properties. Condition C3 checks whether J is the highest priority job that may lock S. At this point, there can be at most one lower priority job which can inherit a higher priority than J, and this job is J^*[5]. Since no higher priority job will lock S, the locking of S by J cannot

[5]This is because we allow a job J_i with priority less than $c(S^*)$ to lock a semaphore S_j only if $p(J_i) = c(S_j)$, i.e. J_i cannot inherit a priority higher than its assigned one.

block a higher priority job. Since J^* can inherit a priority higher than $p(J)$, the condition also checks whether S is required by J^*'s preempted critical section. This means that granting the lock on S to job J cannot block J^* even if J^* inherits a higher priority than J. For instance, suppose that J_1 locks S_1, J_2 locks S_2, and J_3 locks S_1 and then locks S_2 in a nested fashion. Let J_3 lock S_1 first. If J_2 is allowed to lock S_2 now, J_1 may arrive now. When J_1 blocks for S_1, J_3 inherits J_1's priority but would itself block for S_2. Job J_2 would now inherit J_1's priority but J_1 would be blocked for the duration of two critical sections. Under condition C3, J_2 would not be allowed to lock S_2. That is, when condition C3 is true, the locking of S can neither block a higher priority job nor block J^*. Hence, the locking of S can neither lead to deadlock nor contribute to additional blocking for other tasks. This condition does *not* ensure that critical sections will be executed at higher priority levels than preempted critical sections. However, it allows a critical section to be entered if it does not cause deadlock or multiple blocking.

2.2. Definition of the Semaphore Control Protocol

Having illustrated the semaphore control protocol using examples, we now formally define the protocol.

1. Let J be the highest priority job among the jobs ready to run. J is assigned the processor and let S^* be the semaphore with the highest priority ceiling of all semaphores 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 first obtain the lock on the semaphore S guarding the shared data structure. If the semaphore S is unlocked, job J will be granted the lock on S if and only if at least *one* of the following conditions is true:
   a. **Condition C1**: The priority of job J is greater than the priority ceiling of S^*, i.e. $p(J) > c(S^*)$.
   b. **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 semaphore already locked by J^*, i.e. $(p(J) = c(S^*)) \wedge (SR \cap SL^* = \varnothing)$.
   c. **Condition C3**: The priority of job J is equal to the priority ceiling of S and the lock on semaphore S will not be requested by J^*'s preempted critical section, i.e. $(p(J) = c(S)) \wedge (S \notin SR^*)$.
2. In this case, job J will obtain the lock on semaphore 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 semaphore 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 priority of the highest priority job blocked by J. When J exits its critical section, it resumes its original 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.

2.3. Properties of the Semaphore Control Protocol

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

Lemma 1: A job J can be blocked by a lower priority job J_L, only if J_L has entered and remains within a critical section when J arrives.

Proof: It follows from the definition of the semaphore control protocol 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 2: A job J can be blocked by a job J_L, only if the priority of job J is no higher than the highest priority ceiling of all the semaphores that are locked by job J_L when J arrives.

Proof: Suppose that when J arrives, the priority of job J is higher than the highest priority ceiling of all the semaphores that are currently locked by any lower priority job J_L. By the definition of the semaphore control protocol, job J can always preempt the execution of job J_L even if J attempts to enter its critical section. This is because condition C1 of the *locking conditions* becomes true. Since priority inheritance is in effect and the highest priority ready job will always be run, J_L cannot resume execution until J completes. Hence the Lemma follows.

Remark: Lemma 2 is necessary but not sufficient for a lower priority job to block J.

Lemma 3: When a job J arrives, there can be at most one strictly lower priority job J_L that has locked a semaphore S_k such that $c(S_k) \geq p(J)$.

Proof: Suppose there exists another lower priority job J_j that has locked a semaphore S_j such that $c(S_j) \geq p(J)$. Without loss of generality, let J_L have higher priority than J_j. Since J_j has lower priority, by Lemma 1, J_j must have locked S_j first. When J_L attempts to lock S_k, it finds that $p(J_L) < p(J) \leq min[c(S_k), c(S_j)]$. Hence, locking conditions C1, C2 and C3 are false and the semaphore control protocol will not permit J_L to lock S, contradicting our assumption. The Lemma follows.

Lemma 4: Suppose that job J enters a critical section z by obtaining the lock on semaphore 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, $p(J) > c(S^*)$. That is, no job with equal or higher priority than J will lock the semaphores held by lower priority jobs. Hence, from Lemmas 1 and 2, J^* cannot block a higher priority job J_H and therefore J^* cannot inherit a higher priority than J before J completes. 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 4 provides the result that once a semaphore S is locked by a job J because condition C1 is true, then *all* subsequent requests for semaphore locks by job J will also satisfy condition C1 and hence all these locks will be granted.

Lemma 5: Suppose that job J enters a critical section z by obtaining the lock on semaphore 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, $p(J) = c(S^*)$. Also, by Lemma 3, J^* is the only job which has locked a semaphore with priority ceiling = $p(J)$. Hence, no job with higher priority than $p(J)$ will lock a semaphore 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 semaphores. 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 semaphores already locked by J^*. Let the critical section z request a nested lock to semaphore S. We again have $p(J) = c(S^*)$, and still the critical section z cannot lock any semaphores already locked by lower priority jobs. Hence, the semaphore control protocol would allow J to lock S. Since this is true for all requests for semaphore locks nested within the critical section z, job J will exit the critical section without being blocked by J^*.

Remark: Lemma 5 provides the result that if a job J has locked the outermost semaphore of a nested critical section because condition C2 is true, condition C2 will always be true for all subsequent nested requests for semaphore locks within the critical section as well. Hence, no nested request to a semaphore within this critical section will be blocked.

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

Lemma 6: Suppose that job J enters a critical section z by obtaining the lock on semaphore S, because condition C3 of the *locking conditions* is true. Then, the semaphore S cannot be used by job J to block a higher priority job.

Proof: 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 7: The semaphore control protocol 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 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 semaphores 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 semaphore 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 4 and 5, 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 6, the semaphore S_n cannot be used by job J_{i-1} to block job J_{i-2}, contradicting our assumption. The Lemma follows.

Theorem 8: The semaphore control protocol 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 $\{J_1, \cdots, J_n\}$. Since a job not holding any semaphores cannot contribute to the deadlock, each of the n jobs must be in its critical section. By Lemma 7, 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 4 and 5, job J_2 cannot block J_1. Hence, condition C3 must have been true and by Lemma 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 semaphores 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 semaphore locks when the semaphore control protocol 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 the semaphore control protocol, a job can be blocked for at most the duration of one critical section of lower priority jobs.

Theorem 9: 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 Lemmas 2 and 3, there can exist at most a single job J^* that can block J. If J is blocked by J^*, J^* inherits J's priority. By Lemma 7, J^* will exit its critical section without being blocked by a lower priority job. Once J^* exits its critical section, by Lemma 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.

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 the semaphore control protocol are used.[6] In

[6]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.

the previous 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 semaphore 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 10: A job can be blocked for the duration of more than one critical section if a lock on an unlocked semaphore 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 semaphore 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 semaphores 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^*.

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 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 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 semaphore 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 semaphore S, the negation of Lemma 10 must hold. Since $p(J) > c(S)$ is not possible by definition, at least one of the following conditions

must be true:

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

We refer to the above conditions as *necessary locking conditions*. Thus, Theorem 10 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 semaphore. 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 semaphore 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 protocols that are computationally simpler but suboptimal[7]. Note that if both the *necessary locking conditions* are false, then it follows that the *locking conditions* are also false.

Lemma 11: Suppose job J attempts to lock semaphore S. If all three *locking conditions* are false, at least one of the following conditions must be true:

(F1) $(p(J) < c(S)) \wedge (p(J) < c(S^*))$.[8]
(F2) $(p(J) < c(S)) \wedge (p(J) = c(S^*)) \wedge (SR \cap SL^* \neq \emptyset)$.
(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 12: Deadlock can occur or a job can be blocked for the duration of more than one critical section if the lock on a semaphore 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, one of the following cases must be true.

Case I: (F1) The *necessary locking conditions* are false. It follows from Theorem 10 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 semaphore 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

---

[7]See Section 3.1.

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

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 semaphore. 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 semaphore 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 13: 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 8, 9 and 12.

2.5. Schedulability Analysis

The semaphore control protocol 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 the semaphore control protocol.

We quote below the following theorems due to Sha, Rajkumar and Lehoczky [11].

Theorem 14: A set of n periodic tasks using the *priority ceiling protocol* can be scheduled by the rate-monotonic algorithm if the following conditions are satisfied [11]:

$$\forall i,\ 1 \le i \le n, \quad \frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_i}{T_i} + \frac{B_i}{T_i} \le i(2^{1/i} - 1).$$

Theorem 15: A set of n periodic tasks using the priority ceiling protocol can be scheduled by the rate-monotonic algorithm for all task phasings if

$$\forall i,\ 1 \le i \le n, \qquad \min_{(k,l) \in Y_i} \left[\sum_{j=1}^{i-1} U_j \frac{T_j}{lT_k} \left\lceil \frac{lT_k}{T_j} \right\rceil + \frac{C_i}{lT_k} + \frac{B_i}{lT_k} \right] \le 1$$

where $Y_i = \{(k,l) \mid 1 \le k \le i,\ l = 1, \cdots, \lfloor T_i/T_k \rfloor\}$, and B_i is the worst-case blocking time for τ_i. [11]

In both theorems, 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 protocol, the semaphore control protocol 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 the semaphore control protocol as well. An example of the application of Theorem 15 can be seen in [11].

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 initation 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 semaphore $S_{j,k}$ 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.

3. Application Issues

In this section, we consider approximations to the semaphore control protocol and describe one possible implementation of the semaphore control protocol.

3.1. Approximations to the Semaphore Control Protocol

The implementation of the semaphore control protocol requires the maintenance of *SL* and *SR* for the jobs in progress as well as keeping track of J^* and S^*. While the latter is rather simple, the former might require either compile-time support or additional information from the programmer when a critical section entry is attempted. Further, the checks involved in computing conditions 2 and 3 may make this a complex protocol. In this section, we present a set of protocols that are computationally simpler than the semaphore control protocol. All these protocols avoid dead-

locks and minimize the worst-case blocking of each job to the duration of one critical section of a lower priority job. We present the impact of these protocols if used in the scenario described in Example 4 to show that these suboptimal protocols may result in more priority inversion than the optimal semaphore control protocol.

The Priority Ceiling Protocol

If only condition 1 of the *locking conditions* is used in the semaphore control protocol, we obtain the priority ceiling protocol. Sha, Rajkumar and Lehoczky [11] have shown that the use of condition 1 is a sufficient condition to obtain the properties of deadlock prevention and block-for-one-critical-section properties. Thus, a whole family of protocols with these properties can be developed by just including condition 1 of the *locking conditions* and obtaining smaller subsets of conditions 2 and 3. The priority ceiling protocol is easily implementable, since it is based on information that is not only readily available in real-time systems but also cheaply maintained.

Consider the use of the priority ceiling protocol 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 the semaphore control protocol can lead to lower priority inversion relative to the priority ceiling protocol.

The Priority Limit Protocol

We now define the *priority floor* of a semaphore as the priority of the lowest priority task that may lock this semaphore.

Under the *priority limit protocol* [8], a lock on a semaphore S can be granted to a job J if at least one of the following conditions is true:

- condition 1 of the *locking conditions* is true.
- $p(J) = c(S)$ and $floor(S) > p(J^*)$.

The second condition is referred to as the *priority floor condition* and represents a sufficient condition that the semaphore S is not locked by preempted jobs.

It is easy to see that if the priority floor condition is true, then condition 3 of the *locking conditions* is true. Hence the priority limit protocol is a better approximation to the semaphore control protocol than the priority ceiling protocol. Its advantage is that it is based on identical information as is the priority ceiling protocol and hence is also easily implementable. If the priority limit protocol were used in Example 4, it performs identical to the priority ceiling protocol. However, if another job J_4 had locked S_1 instead of J_3 and J_4 does not lock S_2, the priority limit protocol would allow J_2 to lock S_2. The priority ceiling protocol, on the contrary, would not grant the lock.

The Job Control Protocol

In the *job control protocol*, a lock on a semaphore S is granted to a job J if at least one of the following conditions is true:

- condition 1 of the *locking conditions* is true.
- $p(J) = c(S)$ and the semaphore S will not be locked by J^*.

This condition is referred to as the job control condition and is a much stronger condition than the priority floor condition in approximating condition 3 of the *locking conditions*. Its drawback is that one needs to maintain the complete list of semaphores that may be locked by each job.

The job control protocol also performs identical to the priority ceiling and limit protocols on Example 4. However, the job control protocol is, indeed, more powerful than the priority limit protocol. For example, suppose that J_3 never locks S_2 but a lower priority job J_4 might. Then, the priority limit protocol would *not* allow J_2 to lock S_2 when S_1 is locked by J_3 but the job control protocol will.

Efficient Priority Inheritance Protocols

The semaphore control protocol as well as the three simpler protocols just discussed can produce a blocking duration of at most a single critical section. Also they prevent deadlocks. We refer to protocols with these properties as *efficient priority inheritance protocols*. Theorem 14 is applicable to all these efficient priority inheritance protocols. Obviously, it should be possible to develop other suboptimal protocols belonging to this class. The performance of these simpler suboptimal protocols is itself an interesting study, but one that is outside the scope of this paper.

3.2. An Implementation of the Semaphore Control Protocol

We outline below a possible implementation of the semaphore control protocol. A fundamental requirement for the evaluation of the locking conditions is the knowledge of semaphores that may be locked by a job. Such information can either be specified to the run-time system by the compiler or the programmer by means of parameters passed along with the **P**() call on entry to the outermost critical section.

In addition, the following data structures would be required. We do not need semaphore queues, because a job can be blocked by a job J_L even if they do not share any semaphores. Furthermore, the traditional ready queue is replaced by a single job queue *Job_Q*. Unlike traditional ready queue operations, jobs blocked by a semaphore due to the semaphore control protocol need not be taken off the *Job_Q*. That is, *Job_Q* is a priority-ordered list of jobs ready to run or blocked by the semaphore control protocol. We assume that the job at the head of *Job_Q* is currently executing on the processor. The priority ordering of the semaphore locking operations ensures that a job J will be ready to run when it reaches the head of *Job_Q*. We shall illustrate this below with an example.

The run-time system also maintains *S_Stack*, a push-down stack of locked semaphores ordered by priority ceilings and the inverse locking sequence. When the lock on a semaphore S is requested and a locking condition is true, the lock is granted and S is pushed onto *S_STACK* if the priority ceiling of S is higher than the priority ceiling of the semaphore at the top of the stack. When S is released, it is popped from the top of *S_STACK*, if present. If J is the currently executing job, the semaphore at the top of *S_STACK* locked by a job other than J is S^* and the job that has locked S^* is J^*. In other words, when a job is preempted, the top of *S_Stack* becomes S^*. Each semaphore S stores its own priority ceiling and the information about the job, if any, that holds the lock on S. Each job holds the information about the (outermost) critical section z it is currently *within*. The indivisible system calls *Lock_Semaphore* and *Release_Semaphore* maintain *Job_Q* and *S_Stack*. This implementation is illustrated by the following example.

Example 6: Consider the sequence of events in Example 5, illustrated in Figure 2-1. Initially, *Job_Q* = <> and *S_Stack* = <>, where < · · · > represents an ordered list and <> is the null list. The system actions from t_0 to t_{14} are presented below. Note that the operations on *S_Stack* are conditional push-pop operations. Also, the head of *Job_Q* is the currently executing task.

- At time t_0, J_3 arrives and begins execution. $Job\_Q = < J_3 >$. When J_3 locks S_1, *S_Stack* is empty and hence, $S\_Stack = < S_1 >$.
- At time t_1, J_2 preempts J_3. $Job\_Q = < J_2, J_3 >$. $S^* = S_1$ and $J^* = J_3$.
- At time t_2, J_2 locks S_2. Since S_2's priority ceiling is less than the priority ceiling of S_1 at the head of the *S_Stack*, S_2 is *not* pushed onto *S_Stack*.
- At time t_3, J_0 arrives. $Job\_Q = < J_0, J_2, J_3 >$.
- At time t_4, J_0 locks S_0. Now, $c(S_0) > c(S_1)$ and $S\_Stack = < S_0, S_1 >$. Still, $S^* = S_1$ and $J^* = J_3$.
- At time t_5, J_0 releases S_0. $S\_Stack = < S_1 >$.
- At time t_6, J_0 completes and J_{1a} arrives. $Job\_Q = < J_{1a}, J_2, J_3 >$.
- At time t_7, J_{1a} locks S_0. $S\_Stack = < S_0, S_1 >$.
- At time t_8, J_{1a} releases S_0. $S\_Stack = < S_1 >$.
- At time t_9, J_{1a} completes. $Job\_Q = < J_2, J_3 >$.
- At time t_{10}, J_2 attempts to lock S_1 and is blocked by J_3. Job J_3 inherits J_2's priority. Recall that among equal priority jobs, FIFO priority order is maintained. Since J_3 entered *Job_Q* before J_2, its priority is now higher than that of J_2 and J_3 reaches the head of *Job_Q*. $Job\_Q = < J_3, J_2 >$. Now, S^* = *null* and J^* = *null*. In reality, S^* and J^* need not be reset to *null*. If a job J is executing and any semaphore S that J has locked is already on *S_Stack*, all subsequent locking requests of J will be granted. This is because, if S is on *S_Stack*, condition 1 was true when S was locked and hence, from Lemma 4, J cannot be blocked any longer. Hence, maintaining S^* is redundant. If S is not on *S_Stack*, S^* is the semaphore at the top of *S_Stack*.
- At time t_{11}, J_{1b} arrives. $Job\_Q = < J_{1b}, J_3, J_2 >$. Again, $S^* = S_1$ and $J^* = J_3$.
- At time t_{12}, J_{1b} is blocked by J_3. $Job\_Q = < J_3, J_{1b}, J_2 >$. S^* and J^* become null again.
- At time t_{13}, J_3 releases S_1 (*S_Stack* = <>) and J_{1a} locks S_1. $Job\_Q = < J_{1b}, J_2, J_3 >$ *and* $S\_Stack = < S_1 >$. S^* and J^* remain null.

Evaluating the Locking Conditions

The dynamic information about S^* is maintained as described above. The tests for ($SR \cap SL^* = \emptyset$) and ($S \notin SR^*$) are carried out as follows. For each (outermost) critical section z, maintain 3 bit-vectors SR_{orig}, $SR_{current}$ and SL which are n-bit long, where n is the number of semaphores. Thus, each semaphore S is assigned a position within the bit-vector and a bit-vector with the position corresponding to S = 1, else 0 is the *bit-representation* of S. The empty set $\emptyset$ is the 0 bit-vector. The SR_{orig} bit-vector contains the semaphores to be locked by z and can either be

created by the compiler or passed as an argument by the programmer. The k^{th} bit of SR_{orig} is 1 if z requires S_k and 0 otherwise. The bit-vectors $SR_{current}$ and SL represent the dynamic values of SR and SL required by the protocol and are maintained by the protocol. If J is not within z, $SR_{current} = SR_{orig}$ and SL is a null bit-vector.

Whenever J locks a semaphore S_i, $SL := SL \vee$ (bit-representation of S_i) and $SR_{current} := SR_{current} \oplus$ (bit-representation of S_i) where $\vee$ and $\oplus$ are the bit-wise logical *or* and *exclusive-or* operations.

The locking conditions can then be evaluated as follows:

- The evaluation of Condition 1 ($p(J) > c(S^*)$) is straightforward since J is the executing task and S^* is the semaphore at the head of *S_Stack*.
- The first conjunct of Condition 2 ($p(J) = c(S^*)$) can be similarly evaluated. The second conjunct ($SR \wedge SL^* = \varnothing$) can be tested by the bit-wise logical *and* operation between $SR_{current}$ and SL^*, and comparing against the null bit-vector.
- Evaluating the first conjunct of Condition 3 ($p(J) = c(S)$) is again straightforward. The second conjunct ($S \notin SR^*$) is tested by the *exclusive-or* operation between the bit-representation of S and $SR^*_{current}$ and comparing against the null bit-vector.

If the number of semaphores in the system is such that one hardware register is not sufficient for each SR and SL bit vectors, the logical operations on each of the registers can be done in parallel, if possible. If one hardware register is sufficient, an O(1) implementation results.

4. Conclusion

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 protocols solve this unbounded priority inversion problem and bound the blocking duration that a job can experience. We have presented an optimal priority inheritance protocol that not only bounds the worst-case blocking duration of a job to that of a single critical section but also prevents deadlocks. Finally, we have also presented approximations to the protocol which can be easier to implement.

References

[1] Lampson, B.W., Redell, D.D.
Experience with Processes and Monitors in Mesa.
Communications of the ACM 23(2):105-117, February, 1980.

[2] Lehoczky, J. P. and Sha, L.
Performance of Real-Time Bus Scheduling Algorithms.
ACM Performance Evaluation Review, Special Issue Vol. 14, No. 1 , May, 1986.

[3] Lehoczky, J. P., Sha, L. and Strosnider, J.
Enhancing Aperiodic Responsiveness in A Hard Real-Time Environment.
IEEE Real-Time System Symposium , 1987.

[4] Leinbaugh, D. W.
Guaranteed Response Time in a Hard Real-Time Environment.
IEEE Transactions on Software Engineering , Jan. 1980.

[5] Leung, J. Y. and Merrill M. L.
A Note on Preemptive Scheduling of Periodic, Real Time Tasks.
Information Processing Letters 11 (3):115 - 118, Nov. 1980.

[6] Liu, C. L. and Layland J. W.
Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment.
JACM 20 (1):46 - 61, 1973.

[7] Mok, A. K.
Fundamental Design Problems of Distributed Systems For The Hard Real Time Environment.
PhD thesis, M.I.T., 1983.

[8] Rajkumar, R., Lehoczky, J.P.
Task Synchronization in Real-Time Operating Systems.
Fifth IEEE Workshop on Real-Time Software and Operating Systems , May, 1988.

[9] Ramaritham K. and Stankovic J. A.
Dynamic Task Scheduling in Hard Real-Time Distributed Systems.
IEEE Software , July, 1984.

[10] Sha, L., Lehoczky, J. P. and Rajkumar, R.
Solutions for Some Practical Problems in Prioritized Preemptive Scheduling.
IEEE Real-Time Systems Symposium , 1986.

[11] Sha, L., Rajkumar, R. and Lehoczky, J. P.
Priority Inheritance Protocols: An Approach to Real-Time Synchronization.
To appear in IEEE Transactions on Computers , 1988.

[12] Zhao, W., Ramamritham, K., and Stankovic, J. A.
Scheduling Tasks with Resource Requirements in Hard Real-Time Systems.
IEEE Transactions on Software Engineering , May 1987.

[13] Zhao, W., Ramamritham, K. and Stankovic, J.
Preemptive Scheduling Under Time and Resource Constraints.
IEEE Transactions on Computers , Aug. 1987.