Priority Inheritance Protocols: An Approach to Real-Time Synchronization

LUI SHA, MEMBER, IEEE, RAGUNATHAN RAJKUMAR, MEMBER, IEEE, AND JOHN P. LEHOCZKY, MEMBER, IEEE

Abstract—**A direct application of commonly used synchronization primitives such as semaphores, monitors, or the Ada rendezvous can lead to uncontrolled priority inversion, a situation in which a higher priority job is blocked by lower priority jobs for an indefinite period of time. In this paper, we investigate two protocols belonging to the class of *priority inheritance protocols*, called the *basic priority inheritance protocol* and the *priority ceiling protocol*. We show that both protocols solve this uncontrolled priority inversion problem. In particular, the priority ceiling protocol reduces the worst case task blocking time to at most the duration of execution of a single critical section of a lower priority task. In addition, this protocol prevents the formation of deadlocks. We also derive a set of sufficient conditions under which a set of periodic tasks using this protocol is schedulable.**

Index Terms—**Priority inheritance, priority inversion, real-time systems, scheduling, synchronization.**

I. Introduction

THE SCHEDULING of jobs with hard deadlines has been an important area of research in real-time computer systems. Both nonpreemptive and preemptive scheduling algorithms have been studied in the literature [3], [4], [6]–[8], [10], [11]. 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 [9] 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 [9] 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 [14], [15] investigated the use of heuristic algorithms to generate feasible schedules. 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. The importance of this approach is underscored by the fact that Ada, the language mandated by the U.S. Department of Defense for all its real-time systems, supports such a scheduling discipline. 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. Such priority inversion poses a serious problem in real-time systems by adversely affecting both the schedulability and predictability of real-time systems. In this paper, we formally investigate the priority inheritance protocol as a priority management scheme for synchronization primitives that remedies the uncontrolled priority inversion problem. We formally define the protocols in a uniprocessor environment and in terms of binary semaphores. In Section II, we review the problems of existing synchronization primitives, and define the basic concepts and notation. In Section III, we define the basic priority inheritance protocol and analyze its properties. In Section IV, we define an enhanced version of the basic priority inheritance protocol referred to as the priority ceiling protocol and investigate its properties. Section V analyzes the impact of this protocol on schedulability analysis when the rate-monotonic scheduling algorithm is used and Section VI examines the implication considerations as well as some possible enhancements to the priority ceiling protocol. Finally, Section VII presents the concluding remarks.

II. The Priority Inversion Problem

Ideally, a high-priority job J should be able to preempt lower priority jobs immediately upon J's initiation. Priority inversion is the phenomenon where a higher priority job is blocked by lower priority jobs. A common situation arises when two jobs attempt to access shared data. To maintain consistency, the access must be serialized. If the higher priority job gains access first then the proper priority order is maintained; however, if the lower priority job gains access first and then the higher priority job requests access to the shared data, this higher priority job is blocked until the lower priority job completes its access to the shared data. Thus, *blocking* is a form of priority inversion where a higher priority job must wait for the processing of a lower priority job. Prolonged du-

Manuscript received December 1, 1987; revised May 1, 1988. This work was supported in part by the Office of Naval Research under Contract N00014-84-K-0734, in part by Naval Ocean Systems Center under Contract N66001-87-C-0155, and in part by the Federal Systems Division of IBM Corporation under University Agreement YA-278067.

L. Sha is with the Software Engineering Institute and the Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

R. Rajkumar is with IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598.

J. P. Lehoczky is with the Department of Statistics, Carnegie Mellon University, Pittsburgh, PA 15213.

IEEE Log Number 9037197.

0018-9340/90/0900-1175$01.00

rations of blocking may lead to the missing of deadlines even at a low level of resource utilization. The level of resource utilization attainable before a deadline is missed is referred to as the *schedulability* of the system. To maintain a high degree of schedulability, we will develop protocols that would minimize the amount of blocking. It is also important to be able to analyze the performance of any proposed protocol in order to determine the schedulability of real-time tasks that use this protocol.

Common synchronization primitives include semaphores, locks, monitors, and Ada rendezvous. Although the use of these or equivalent methods is necessary to protect the consistency of shared data or to guarantee the proper use of nonpreemptable resources, their use may jeopardize the ability of the system to meet its timing requirements. In fact, a direct application of these synchronization mechanisms can lead to an indefinite period of priority inversion and a low level of schedulability.

Example 1: Suppose that J_1, J_2, and J_3 are three jobs arranged in descending order of priority with J_1 having the highest priority. We assume that jobs J_1 and J_3 share a data structure guarded by a binary semaphore S. Suppose that at time t_1, job J_3 locks the semaphore S and executes its critical section. During the execution of job J_3's critical section, the high priority job J_1 is initiated, preempts J_3, and later attempts to use the shared data. However, job J_1 will be blocked on the semaphore S. We would expect that J_1, being the highest priority job, will be blocked no longer than the time for job J_3 to complete its critical section. However, the duration of blocking is, in fact, 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 period in Example 1 can be arbitrarily long. This situation can be partially remedied if a job in its critical section is not allowed to be preempted; however, this solution is only appropriate for very short critical sections, because it creates unnecessary blocking. For instance, once a low priority job enters a long critical section, a high priority job which does not access the shared data structure may be needlessly blocked. An identical problem exists in the use of monitors. The priority inversion problem was first discussed by Lampson and Redell [2] in the context of monitors. They suggest that the monitor be executed at a priority level higher than all tasks that would ever call the monitor. In the case of the Ada rendezvous, when a high priority job (task) is waiting in the entry queue of a server job, the server itself can be preempted by an independent job J, if job J's priority is higher than both the priority of the server and the job which is currently in rendezvous with the server. Raising the server priority to be higher than all its callers would avoid this particular problem but would create a new problem: a low priority job may unnecessarily block the execution of independent higher priority jobs via the use of the server.

The use of *priority inheritance protocols* is one approach to rectify the priority inversion problem in existing synchronization primitives. Before we investigate these protocols, we first define the basic concepts and state our assumptions. A *job* is a sequence of instructions that will continuously use the processor until its completion if it is executing alone on the processor. That is, we assume that jobs do not suspend themselves, say for I/O operations; however, such a situation can be accommodated by defining two or more jobs. In addition, we assume that the critical sections of a job are *properly* nested and a job will release all of its locks, if it holds any, before or at the end of its execution. 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. A *periodic task* is a sequence of the same type of job occurring at regular intervals, and an *aperiodic task* is a sequence of the same type of job occurring at irregular intervals. Each task is assigned a fixed priority, and every job of the same task is initially assigned that task's priority. If several jobs are eligible to run, the highest priority job will be run. Jobs with the same priority are executed in a FCFS discipline. When a job J is forced to wait for the execution of lower priority jobs, job J is said to be "blocked." When a job waits for the execution of high priority jobs or equal priority jobs that have arrived earlier, it is not considered as "blocked." We now state our notation.

Notation:

- J_i denotes a job, i.e., an instance of a task τ_i. P_i and T_i denote the priority and period of task τ_i, respectively.
- A binary semaphore guarding shared data and/or resource is denoted by S_i. $P(S_i)$ and $V(S_i)$ denote the indivisible operations *lock* (wait) and *unlock* (signal), respectively, on the binary semaphore S_i.
- 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 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 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 $z_{i,j} \cap z_{i,k} = \emptyset$. In addition, we assume that a semaphore may be locked at most once in a single nested critical section.

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 $S_{i,j} = S$.

In the next two sections, we will introduce the concept of priority inheritance and a priority inheritance protocol called the priority ceiling protocol. An important feature of this protocol is that one can develop a schedulability analysis for it in the sense that a schedulability bound can be determined. If the utilization of the task set stays below this bound, then the deadlines of all the tasks can be guaranteed. In order to create such a bound, it is necessary to determine the worst case duration of priority inversion that any task can encounter. This worst case blocking duration will depend upon the particular protocol in use.

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, $\beta_{i,j} = \{z_{j,k} | j > i \text{ and } z_{j,k} \text{ can block } J_i\}$.[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 reduce our attention to the set of maximal elements of $\beta_{i,j}$, $\beta^*_{i,j}$. Specifically, we have $\beta^*_{i,j} = \{z_{j,k} | (z_{j,k} \in \beta_{i,j}) \wedge (\sim \exists z_{j,m} \in \beta_{i,j}$ such that $z_{j,k} \subset z_{j,m})\}$.

The set $\beta^*_{i,j}$ contains the longest 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 $\beta^* = \bigcup_{j>i} \beta^*_{i,j}$, the set of all longest critical sections that can block J_i.

III. The Basic Priority Inheritance Protocol

The basic idea of priority inheritance protocols is that when a job J blocks one or more higher priority jobs, it ignores its original priority assignment and executes its critical section at the highest priority level of all the jobs it blocks. 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 job J_3. The priority inheritance protocol requires that job J_3 execute its critical section at job J_1's priority. As a result, job J_2 will be unable to preempt job J_3 and will itself be blocked. That is, the higher priority job J_2 must wait for the critical section of the lower priority job J_3 to be executed, because job J_3 "inherits" the priority of job J_1. Otherwise, J_1 will be indirectly preempted by J_2. When J_3 exits its critical section, it regains its assigned lowest priority and awakens J_1 which was blocked by J_3. Job J_1, having the highest priority, immediately preempts J_3 and runs to completion. This enables J_2 and J_3 to resume in succession and run to completion.

A. The Definition of the Basic Protocol

We now define the basic priority inheritance protocol.

1) Job J, which has the highest priority among the jobs ready to run, is assigned the processor. Before job J enters a critical section, it must first obtain the lock on the semaphore S guarding the critical section. Job J will be blocked, and the lock on S will be denied, if semaphore S has been already locked. In this case, job J is said to be blocked by the job which holds the lock on S. Otherwise, job J will obtain the lock on semaphore S and enter its critical section. When 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.

2) 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*** (uses) P_H, the highest priority of the jobs blocked by J. When J exits a critical section, it resumes the priority it had at the point of entry into the critical section.[2]

3) Priority inheritance is transitive. For instance, suppose J_1, J_2, and J_3 are three jobs in descending order of priority. Then, if job J_3 blocks job J_2, and J_2 blocks job J_1, J_3 would inherit the priority of J_1 via J_2. Finally, the operations of priority inheritance and of the resumption of original priority must be indivisible.[3]

4) A job J can preempt another job J_L if job J is not blocked and its priority is higher than the priority, inherited or assigned, at which job J_L is executing.

It is helpful to summarize that under the basic priority inheritance protocol, a high priority job can be blocked by a low-priority job in one of two situations. First, there is the ***direct*** blocking, a situation in which a higher priority job attempts to lock a locked semaphore. Direct blocking is necessary to ensure the consistency of shared data. Second, a medium priority job J_1 can be blocked by a low priority job J_2, which inherits the priority of a high priority job J_0. We refer to this form of blocking as ***push-through*** blocking, which is necessary to avoid having a high-priority job J_0 being indirectly preempted by the execution of a medium priority job J_1.

B. The Properties of the Basic Protocol

We now proceed to analyze the properties of the basic priority inheritance protocol defined above. In this section, we assume that deadlock is prevented by some external means, e.g., semaphores are accessed in an order that is consistent with a predefined acyclical order. Throughout this section, β^*_i refers to the sets of the longest critical sections that can block J_i when the basic priority inheritance protocol is used.

Lemma 1: A job J_H can be blocked by a lower priority job J_L, only if J_L is executing within a critical section $z_{L,j} \in \beta^*_{H,L}$, when J_H is initiated.

Proof: By the definitions of the basic priority inheritance protocol and the blocking set $\beta^*_{H,L}$, task J_L can block J_H only if it directly blocks J_H or has its priority raised above J_H through priority inheritance. In either case, the critical section $z_{L,j}$ currently being executed by J_L is in $\beta^*_{H,L}$. If J_L is not within a critical section which cannot directly block J_H and cannot lead to the inheritance of a priority higher than J_H, then J_L can be preempted by J_H and can never block J_H.

Lemma 2: Under the basic priority inheritance protocol, a high priority job J_H can be blocked by a lower priority job J_L for at most the duration of one critical section of $\beta^*_{H,L}$ regardless of the number of semaphores J and J_L share.

Proof: By Lemma 1, for J_L to block J_H, J_L must be currently executing a critical section $z_{L,j} \in \beta^*_{H,L}$. Once J_L exits $z_{L,j}$, it can be preempted by J_H and J_H cannot be blocked by J_L again.

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

[2] For example, when J executes $V(S_2)$ in $\{P(S_1), \cdots, P(S_2), \cdots, V(S_2), \cdots, V(S_1)\}$, it reverts to the priority it had before it executed $P(S_2)$. This may be lower than its current priority and cause J to be preempted by a higher priority task. J would, of course, still hold the lock on S_1.

[3] The operations must be indivisible in order to maintain internal consistency of data structures being manipulated in the run-time system.

Theorem 3: Under the basic priority inheritance protocol, given a job J_0 for which there are n lower priority jobs $\{J_1, \cdots, J_n\}$, job J_0 can be blocked for at most the duration of one critical section in *each* of $\beta^*_{0,i}$, $1 \le i \le n$.

Proof: By Lemma 2, each of the n lower priority jobs can block job J_0 for at most the duration of a single critical section in each of the blocking sets $\beta^*_{0,i}$.

We now determine the bound on the blockings as a function of the semaphores shared by jobs.

Lemma 4: A semaphore S can cause push-through blocking to job J, only if S is accessed both by a job which has priority lower than that of J and by a job which has or can inherit priority equal to or higher than that of J.

Proof: Suppose that J_L accesses semaphore S and has priority lower than that of J. According to the priority inheritance protocol, if S is not accessed by a job which has or can inherit priority equal to or higher than that of J, then job J_L's critical section guarded by S cannot inherit a priority equal to or higher than that of J. In this case, job J_L will be preempted by job J and the lemma follows.

We next define $\zeta^*_{i,j,k}$ to be the set of all longest critical sections of job J_j guarded by semaphore S_k and which can block job J_i either directly or via push-through blocking. That is, $\zeta^*_{i,j,k} = \{z_{j,p} | z_{j,p} \in \beta^*_{i,j} \text{ and } s_{j,p} = S_k\}$.

Let $\zeta^*_{i,\cdot,k} = \bigcup_{j>i} \zeta^*_{i,j,k}$ represent the set of all longest critical sections corresponding to semaphore S_k which can block J_i.

Lemma 5: Under the basic priority inheritance protocol, a job J_i can encounter blocking by at most one critical section in $\zeta^*_{i,\cdot,k}$ for each semaphore S_k, $1 \le k \le m$, where m is the number of distinct semaphores.

Proof: By Lemma 1, job J_L can block a higher priority job J_H if J_L is currently executing a critical section in $\beta^*_{H,L}$. Any such critical section corresponds to the locking and unlocking of a semaphore S_k. Since we deal only with binary semaphores, only one of the lower priority jobs can be within a blocking critical section corresponding to a particular semaphore S_k. Once this critical section is exited, the lower priority job J_L can no longer block J_H. Consequently, only one critical section in β^*_i corresponding to semaphore S_k can block J_H. The lemma follows.

Theorem 6: Under the basic priority inheritance protocol, if there are m semaphores which can block job J, then J can be blocked by at most m times.

Proof: It follows from Lemma 5 that job J can be blocked at most once by each of the m semaphores.

Theorems 3 and 6 place an upper bound on the *total* blocking delay that a job can encounter. Given these results, it is possible to determine at compile-time the worst case blocking duration of a job. For instance, if there are four semaphores which can potentially block job J and there are three other lower priority tasks, J may be blocked for a maximum duration of three longest subcritical sections. Moreover, one can find the worst case blocking durations for a job by studying the durations of the critical sections in $\beta^*_{i,j}$ and $\zeta^*_{i,\cdot,k}$.

Still, the basic priority inheritance protocol has the following two problems. First, this basic protocol, by itself, does not prevent deadlocks. For example, suppose that at time t_1, job J_2 locks semaphore S_2 and enters its critical section. At time t_2, job J_2 attempts to make a nested access to lock semaphore S_1. However, job J_1, a higher priority job, is ready at this time. Job J_1 preempts job J_2 and locks semaphore S_1. Next, if job J_1 tries to lock semaphore S_2, a deadlock is formed.

The deadlock problem can be solved, say, by imposing a total ordering on the semaphore accesses. Still, a second problem exists. The blocking duration for a job, though bounded, can still be substantial, because a *chain* of blocking can be formed. For instance, suppose that J_1 needs to sequentially access S_1 and S_2. Also suppose that J_2 preempts J_3 within the critical section $z_{3,1}$ and enters the critical section $z_{2,2}$. Job J_1 is initiated 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 blocking chain is formed.

We present in the next section the priority ceiling protocol that addresses effectively both these problems posed by the basic priority inheritance protocol.

IV. The Priority Ceiling Protocol

A. Overview

The goal of this protocol is to prevent the formation of deadlocks and of chained blocking. The underlying idea of this protocol is to ensure that when a job J preempts the critical section of another job and executes its own critical section z, the priority at which this new critical section z will execute is guaranteed to be higher than the inherited priorities of all the preempted critical sections. If this condition cannot be satisfied, job J is denied entry into the critical section z and suspended, and the job that blocks J inherits J's priority. This idea is realized by first assigning a priority ceiling to each semaphore, which is equal to the highest priority task that may use this semaphore. We then allow a job J to start a new critical section only if J's priority is higher than all priority ceilings of all the semaphores locked by jobs other than J. Example 2 illustrates this idea and the deadlock avoidance property while Example 3 illustrates the avoidance of chained blocking.

Example 2: Suppose that we have three jobs J_0, J_1, and J_2 in the system. In addition, there are two shared data structures protected by the binary semaphores S_1 and S_2, respectively. We define the *priority ceiling* of a semaphore as the priority of the highest priority job that may lock this semaphore. Suppose the sequence of processing steps for each job is as follows.

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

$$J_1 = \{\cdots, P(S_1), \cdots, P(S_2), \cdots, V(S_2), \cdots, V(S_1), \cdots\}$$

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

Recall that the priority of job J_1 is assumed to be higher than that of job J_2. Thus, the priority ceilings of both semaphores S_1 and S_2 are equal to the priority of job J_1.

The sequence of events described below is depicted in Fig. 1. A line at a low level indicates that the corresponding job

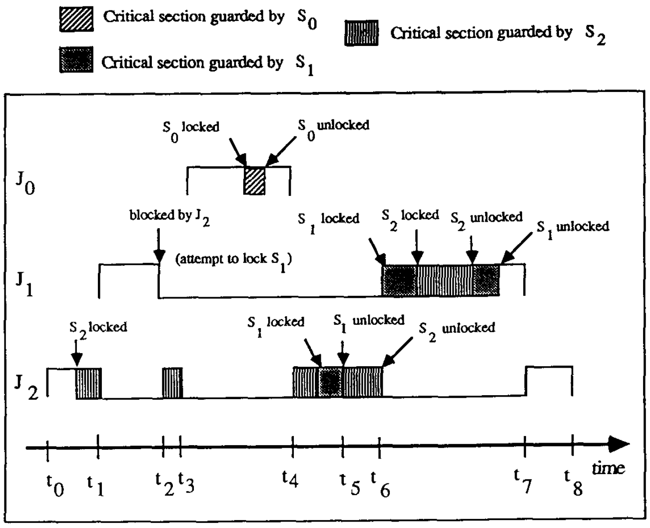


Fig. 1. Sequence of events described in Example 2.

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 been initiated or has completed. Shaded portions indicate execution of critical sections. Suppose that

- At time t_0, J_2 is initiated and it begins execution and then locks semaphore S_2.
- At time t_1, job J_1 is initiated and preempts job J_2.
- At time t_2, job J_1 tries to enter its critical section by making an indivisible system call to execute $P(S_1)$. However, the run-time system will find that job J_1's priority is *not* higher than the priority ceiling of *locked* semaphore S_2. Hence, the run-time system suspends job J_1 without locking S_1. Job J_2 now *inherits* the priority of job J_1 and resumes execution. Note that J_1 is blocked outside its critical section. As J_1 is not given the lock on S_1 but suspended instead, the potential deadlock involving J_1 and J_2 is prevented.
- At time t_3, J_2 is still in its critical section and the highest priority job J_0 is initiated and preempts J_2. Later, J_0 attempts to lock semaphore S_0. Since the priority of J_0 is higher than the priority ceiling of locked semaphore S_2, job J_0 will be granted the lock on the semaphore S_0. Job J_0 will therefore continue and execute its critical section, thereby effectively preempting J_2 in its critical section and not encountering any blocking.
- At time t_4, J_0 has exited its critical section and completes execution. Job J_2 resumes, since J_1 is blocked by J_2 and cannot execute. J_2 continues execution and locks S_1.
- At time t_5, J_2 releases S_1.
- At time t_6, J_2 releases S_2 and resumes its assigned priority. Now, J_1 is signaled and having a higher priority, it preempts J_2, resumes execution, and locks S_2. Then, J_1 locks S_1, executes the nested critical section, and unlocks S_1. Later it unlocks S_2 and executes its noncritical section code.
- At t_7, J_1 completes execution and J_2 resumes.
- At t_8, J_2 completes.

Note that in the above example, J_0 is never blocked because its priority is higher than the priority ceilings of semaphores S_1 and S_2. J_1 was blocked by the lower priority job J_2 during the intervals $[t_2, t_3]$ and $[t_4, t_6]$. However, these intervals correspond to part of the duration that J_2 needs to lock S_2. Thus, J_1 is blocked for no more than the duration of one critical section of a lower priority job J_2 even though the actual blocking occurs over disjoint time intervals. It is, indeed, a property of this protocol that any job can be blocked for at most the duration of a single critical section of a lower priority job. This property is further illustrated by the following example.

Example 3: Consider the example from the previous section where a chain of blockings can be formed. We assumed that job J_1 needs to access S_1 and S_2 sequentially while J_2 accesses S_2 and J_3 accesses S_1. Hence, the priority ceilings of semaphores S_1 and S_2 are equal to P_1. As before, let job J_3 lock S_1 at time t_0. At time t_1, job J_2 is initiated and preempts J_3. However, at time t_2, when J_2 attempts to lock S_2, the run-time system finds that the priority of J_2 is *not* higher than the priority ceiling P_1 of the *locked* semaphore S_1. Hence, J_2 is denied the lock on S_2 and blocked. Job J_3 resumes execution at J_2's priority. At time t_3, when J_3 is still in its critical section, J_1 is initiated and finds that only one semaphore S_1 is locked. At time t_4, J_1 is blocked by J_3 which holds the lock on S_1. Hence, J_3 inherits the priority of J_1. At time t_5, job J_3 exits its critical section $z_{3,1}$, resumes its original priority, and awakens J_1. Job J_3, having the highest priority, preempts J_3 and runs to completion. Next, J_2 which is no longer blocked completes its execution and is followed by J_3.

Again, note that J_1 is blocked by J_3 in the interval $[t_4, t_5]$ which corresponds to the single critical section $z_{3,1}$. Also, job J_2 is blocked by J_3 in the disjoint intervals $[t_2, t_3]$ and $[t_4, t_5]$ which also correspond to the same critical section $z_{3,1}$.

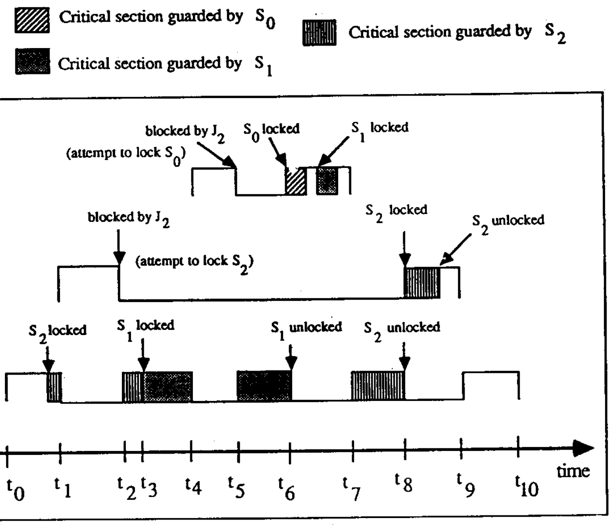


Fig. 2. Sequence of events described in Example 4.

B. Definition

Having illustrated the basic idea of the priority ceiling protocol and its properties, we now present its definition.

1) Job J, which has the highest priority among the jobs ready to run, 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. Before job J enters its critical section, it must first obtain the lock on the semaphore S guarding the shared data structure. Job J will be blocked and the lock on S will be denied, if the priority of job J is not higher than the priority ceiling of semaphore S^*.[4] In this case, job J is said to be blocked on semaphore S^* and to be blocked by the job which holds the lock on S^*. Otherwise, job J will obtain the lock on semaphore S and enter its critical section. 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.

2) 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 highest priority of the jobs blocked by J. When J exits a critical section, it resumes the priority it had at the point of entry into the critical section.[5] Priority inheritance is transitive. Finally, the operations of priority inheritance and of the resumption of previous priority must be indivisible.

3) 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.

We shall illustrate the priority ceiling protocol using an example.

[4] Note that if S has been already locked, the priority ceiling of S will be at least equal to the priority of J. Because job J's priority is not higher than the priority ceiling of the semaphore S locked by another job, J will be blocked. Hence, this rule implies that if a job J attempts to lock a semaphore that has been already locked, J will be denied the lock and blocked instead.

[5] That is, when J exits the part of a critical section, it resumes its previous priority.

Example 4: We assume that the priority of job J_i is higher than that of job J_{i+1}. The processing steps in each job are as follows:

Job J_0 accesses $z_{0,0}$ and $z_{0,1}$ by executing the steps

$$\{\cdots, P(S_0), \cdots, V(S_0), \cdots, P(S_1), \cdots, V(S_1), \cdots\},$$

job J_1 accesses only $z_{1,2}$ by executing

$$\{\cdots, P(S_2), \cdots, V(S_2), \cdots\},$$

and job J_2 accesses $z_{2,2}$ and makes a nested semaphore access to S_1 by executing

$$\{\cdots, P(S_2), \cdots, P(S_1), \cdots, V(S_1), \cdots, V(S_2), \cdots\}.$$

Note that the priority ceilings of semaphores S_0 and S_1 are equal to P_0, and the priority ceiling of semaphore S_2 is P_1. Fig. 2 depicts the sequence of events described below. Suppose that

- At time t_0, job J_2 begins execution and later locks S_2.
- At time t_1, job J_1 is initiated, preempts J_2, and begins execution.
- At time t_2, while attempting to access S_2 already locked by J_2, job J_1 becomes blocked. Job J_2 now resumes the execution of its critical section $z_{2,2}$ at its inherited priority of J_1, namely P_1.
- At time t_3, job J_2 successfully enters its nested critical section $z_{2,1}$ by locking S_1. Job J_2 is allowed to lock S_1, because there is no semaphore S^* which is locked by other jobs.
- At time t_4, job J_2 is still executing $z_{2,1}$ but the highest priority job J_0 is initiated. Job J_0 preempts J_2 within $z_{2,1}$ and executes its own noncritical section code. This is possible because P_0, the priority of J_0, is higher than P_1, the inherited priority level at which job J_2's $z_{2,1}$ was being executed.
- At time t_5, job J_0 attempts to enter its critical section $z_{0,0}$

by locking S_0, which is not locked by any job. However, since the priority of job J_0 is not higher than the priority ceiling P_0 of the locked semaphore S_1, job J_0 is blocked by job J_2 which holds the lock on S_1. This is a new form of blocking introduced by the priority ceiling protocol in addition to the direct and push-through blocking encountered in the basic protocol. At this point, job J_2 resumes its execution of $z_{2,1}$ at the newly inherited priority level of P_0.

- At time t_6, job J_2 exits its critical section $z_{2,1}$. Semaphore S_1 is now unlocked, job J_2 returns to the previously inherited priority of P_1, and job J_0 is awakened. At this point, J_0 preempts job J_2, because its priority P_0 is higher than the priority ceiling P_1 of S_2. Job J_0 will be granted the lock on S_0 and will execute its critical section $z_{0,0}$. Later, it unlocks S_0 and then locks and unlocks S_1.
- At time t_7, job J_0 completes its execution, and job J_2 resumes its execution of $z_{2,2}$ at its inherited priority P_1.
- At time t_8, job J_2 exits $z_{2,2}$, semaphore S_2 is unlocked, job J_2 returns to its own priority P_2, and job J_1 is awakened. At this point, job J_1 preempts job J_2 and J_1 is granted the lock on S_2. Later, J_1 unlocks S_2 and executes its noncritical section code.
- At time t_9, job J_1 completes its execution and finally job J_2 resumes its execution, until it also completes at time t_{10}.

The priority ceiling protocol introduces a third type of blocking in addition to direct blocking and push-through blocking caused by the basic priority inheritance protocol. An instance of this new type of blocking occurs at time t_5 in the above example. We shall refer to this form of blocking as *ceiling* blocking. Ceiling blocking is needed for the avoidance of deadlock and of chained blocking. This avoidance approach belongs to the class of pessimistic protocols which sometimes create unnecessary blocking. Although the priority ceiling protocol introduces a new form of blocking, the worst case blocking is dramatically improved. Under the basic priority inheritance protocol, a job J can be blocked for at most the duration of $\min(n, m)$ critical sections, where n is the number of lower priority jobs that could block J and m is the number of semaphores that can be used to block J. On the contrary, under the priority ceiling protocol a job J can be blocked for at most the duration of one longest subcritical section.

C. The Properties of the Priority Ceiling Protocol

Before we prove the properties of this protocol, it is important to recall the two basic assumptions about jobs. First, a job is assumed to be a sequence of instructions that will continuously execute until its completion, when it executes alone on a processor. Second, a job will release all of its locks, if it ever holds any, before or at the end of its execution. The relaxation of our first assumption is addressed at the end of this section. Throughout this section, the sets $\beta_{i,j}$, $\beta^*_{i,j}$, and β^*_i refer to the blocking sets associated with the priority ceiling protocol.

Lemma 7: A job J can be blocked by a lower priority 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 all lower priority jobs when J is initiated.

Proof: Suppose that when J is initiated, the priority of job J is higher than the highest priority ceiling of all the semaphores that are currently locked by all lower priority jobs. By the definition of the priority ceiling protocol, job J can always preempt the execution of job J_L, and no higher priority job will ever attempt to lock those locked semaphores.

Lemma 8: Suppose that the critical section $z_{j,n}$ of job J_j is preempted by job J_i which enters its critical section $z_{i,m}$. Under the priority ceiling protocol, job J_j cannot inherit a priority level which is higher than or equal to that of job J_i until job J_i completes.

Proof: Suppose that job J_j inherits a priority that is higher than or equal to that of job J_i before J_i completes. Hence, there must exist a job J which is blocked by J_j. In addition, J's priority must be higher than or equal to that of job J_i. We now show the contradiction that J cannot be blocked by J_j. Since job J_i preempts the critical section $z_{j,n}$ of job J_j and enters its own critical section $z_{i,m}$, job J_i's priority must be higher than the priority ceilings of all the semaphores currently locked by all lower priority jobs. Since J's priority is assumed to be higher than or equal to that of J_i, it follows that job J's priority is also higher than the priority ceilings of all the semaphores currently locked by all lower priority jobs. By Lemma 7, J cannot be blocked by J_j. Hence, the contradiction and the lemma follows.

Definition: Transitive blocking is said to occur if a job J is blocked by J_i which, in turn, is blocked by another job J_j.

Lemma 9: The priority ceiling protocol prevents transitive blocking.

Proof: Suppose that transitive blocking is possible. Let J_3 block job J_2 and let job J_2 block job J_1. By the transitivity of the protocol, job J_3 will inherit the priority of J_1 which is assumed to be higher than that of job J_2. This contradicts Lemma 8, which shows that J_3 cannot inherit a priority that is higher than or equal to that of job J_2. The lemma follows.

Theorem 10: The priority ceiling protocol prevents deadlocks.

Proof: First, by assumption, a job cannot deadlock with itself. Thus, a deadlock can only be formed by a cycle of jobs waiting for each other. Let the n jobs involved in the blocking cycle be $\{J_1, \cdots, J_n\}$. Note that each of these n jobs must be in one of its critical sections, since a job that does not hold a lock on any semaphore cannot contribute to the deadlock. By Lemma 9, the number of jobs in the blocking cycle can only be two, 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. By Lemma 8, job J_2 can never inherit a priority which is higher than or equal to that of job J_1 before job J_1 completes. However, if a blocking cycle (deadlock) is formed, then by the transitivity of priority inheritance, job J_2 will inherit the priority of job J_1. This contradicts Lemma 8 and hence the theorem follows.

Remark: Lemma 1 is true under the priority ceiling protocol.

Remark: Suppose that the run-time system supports the priority ceiling protocol. Theorem 10 leads to the useful result that programmers can write arbitrary sequences of properly nested semaphore accesses. As long as each job does not deadlock with itself, there will be no deadlock in the system.

Lemma 11: Let J_L be a job with a lower priority than that of job J_i. Job J_i can be blocked by job J_L for at most the duration of one critical section in $\beta^*_{i,L}$.

Proof: First, job J_i will preempt J_L if J_L is not in a critical section $z_{L,m} \in \beta^*_{i,L}$. Suppose that job J_i is blocked by $z_{L,m}$. By Theorem 10, there is no deadlock and hence job J_L will exit $z_{L,m}$ at some instant t_1. Once job J_L leaves this critical section at time t_1, job J_L can no longer block job J_i. This is because job J_i has been initiated and J_L is not within a critical section in $\beta^*_{i,L}$. It follows from Lemma 1 that job J_L can no longer block job J_i.

Theorem 12: A job J can be blocked for at most the duration of at most one element of β^*_i.

Proof: Suppose that job J can be blocked by $n > 1$ elements of β_i. By Lemma 11, the only possibility is that job J is blocked by n different lower priority jobs. Suppose that the first two lower priority jobs that block job J are J_1 and J_2. By Lemma 1, in order for both these jobs to block job J, both of them must be in a longest blocking critical section when job J is initiated. Let the lowest priority job J_2 enter its blocking critical section first, and let the highest priority ceiling of all the semaphores locked by J_2 be ρ_2. Under the priority ceiling protocol, in order for job J_1 to enter its critical section when J_2 is already inside one, the priority of job J_1 must be higher than priority ceiling ρ_2. Since we assume that job J can be blocked by job J_2, by Lemma 7 the priority of job J cannot be higher than priority ceiling ρ_2. Since the priority of job J_1 is higher than ρ_2 and the priority of job J is no higher than ρ_2, job J_1's priority must be higher than the priority of job J. This contradicts the assumption that the priority of job J is higher than that of both J_1 and J_2. Thus, it is impossible for job J to have priority higher than both jobs J_1 and J_2 and to be blocked by both of them under the priority ceiling protocol. The theorem follows immediately.

Remark: We may want to generalize the definition of a job by allowing it to suspend during its execution, for instance, to wait for I/O services to complete. The following corollary presents the upper bound on the blocking duration of a generalized job that might suspend and later resume during its execution.

Corollary 13: If a generalized job J suspends itself n times during its execution, it can be blocked by at most $n + 1$ not necessarily distinct elements of β^*_i.

V. Schedulability Analysis

Having proved the properties of the priority ceiling protocol, we now proceed to investigate the effect of blocking on the schedulability of a task set. In this section, we develop a set of *sufficient* conditions under which a set of periodic tasks using the priority ceiling protocol can be scheduled by the rate-monotonic algorithm, which assigns higher priorities to tasks with shorter periods and is an optimal static priority algorithm when tasks are independent [8]. To this end, we will use a simplified scheduling model. First, we assume that all the tasks are periodic. Second, we assume that each job in a periodic task has deterministic execution times for both its critical and noncritical sections and that it does not synchronize with external events, i.e., a job will execute to its completion when it is the only job in the system. Finally, we assume that these periodic tasks are assigned priorities according to the rate-monotonic algorithm. Readers who are interested in more general scheduling issues, such as the reduction of aperiodic response times and the effect of task stochastic execution times, are referred to [4] and [12].

We quote the following theorem also due to Liu and Layland which was proved under the assumption of independent tasks, i.e., when there is no blocking due to data sharing and synchronization.

Theorem 14: A set of n periodic tasks scheduled by the rate-monotonic algorithm can always meet their deadlines if

$$\frac{C_1}{T_1} + \cdots + \frac{C_n}{T_n} \le n(2^{1/n} - 1)$$

where C_i and T_i are the execution time and period of task τ_i, respectively.

Theorem 14 offers a sufficient (worst case) condition that characterizes the rate-monotonic schedulability of a given periodic task set. The following exact characterization was proved by Lehoczky, Sha, and Ding [5]. An example of the use of this theorem will be given later in this section.

Theorem 15: A set of n periodic tasks scheduled by the rate-monotonic algorithm will meet all their deadlines for all task phasings if and only if

$$\forall i,\ 1 \le i \le n, \qquad \min_{(k,l)\in R_i} \sum_{j=1}^{i} C_j \frac{1}{lT_k} \left\lceil \frac{lT_k}{T_j} \right\rceil = \min_{(k,l)\in R_i} \sum_{j=1}^{i} U_j \frac{T_j}{lT_k} \left\lceil \frac{lT_k}{T_j} \right\rceil \le 1$$

where C_j, T_j, and U_j are the execution time, period, and utilization of task τ_j, respectively, and $R_i = \{(k,l) | 1 \le k \le i,\ l = 1, \cdots, \lfloor T_i/T_k \rfloor\}$.

When tasks are independent of one another, Theorems 14 and 15 provide us with the conditions under which a set of n periodic tasks can be scheduled by the rate-monotonic algorithm.[6] Although these two theorems have taken into account the effect of a task being preempted by higher priority tasks, they have not considered the effect of a job being blocked by lower priority jobs. We now consider the effect of blocking. Each element in β_i is a critical section accessed by a lower priority job and guarded by a semaphore whose priority ceiling is higher than or equal to the priority of job J_i. Hence, β^*_i can be derived from β_i. By Lemma 7 and Theorem 12, job J_i of a task τ can be blocked for at most the duration of a single element in β^*_i. Hence, the worst case blocking time for J is at most the duration of the longest element of β^*_i. We denote this worst case blocking time of a job in task τ_i by B_i. Note that given a set of n periodic tasks, $B_n = 0$, since there is no lower priority task to block τ_n.

[6] That is, the conditions under which all the jobs of all the n tasks will meet their deadlines.

Theorems 14 and 15 can be generalized in a straightforward fashion. In order to test the schedulability of τ_i, we need to consider both the preemptions caused by higher priority tasks and blocking from lower priority tasks along with its own utilization. The blocking of any job of τ_i can be in the form of direct blocking, push-through blocking, or ceiling blocking but does not exceed B_i. Thus, Theorem 14 becomes

Theorem 16: 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:

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

Proof: Suppose that for each task τ_i the equation is satisfied. It follows that the equation of Theorem 14 will also be satisfied with $n = i$ and C_i replaced by $C_i^* = (C_i + B_i)$. That is, in the absence of blocking, any job of task τ_i will still meet its deadline even if it executes for $(C_i + B_i)$ units of time. It follows that task τ_i, if it executes for only C_i units of time, can be delayed by B_i units of time and still meet its deadline. Hence, the theorem follows.

Remark: The first i terms in the above inequality constitute the effect of preemptions from all higher priority tasks and τ_i's own execution time, while B_i of the last term represents the worst case blocking time due to *all* lower priority tasks for any job of task τ_i. To illustrate the effect of blocking in Theorem 16, suppose that we have three harmonic tasks: $\tau_1 = (C_1 = 1, T_1 = 2)$, $\tau_2 = (C_2 = 1, T_2 = 4)$, $\tau_3 = (C_3 = 2, T_3 = 8)$. In addition, $B_1 = B_2 = 1$. Since these tasks are harmonic, the utilization bound becomes 100%. Thus, we have "$C_1/T_1 + B_1/T_1 = 1$" for task τ_1. Next, we have "$C_1/T_1 + C_2/T_2 + B_2/T_2 = 1$" for task τ_2. Finally, we have "$C_1/T_1 + C_2/T_2 + C_3/T_3 = 1$" for task τ_3. Since all three equations hold, these three tasks can meet all their deadlines.

Corollary 17: A set of n periodic tasks using the priority ceiling protocol can be scheduled by the rate-monotonic algorithm if the following condition is satisfied:

$$\frac{C_1}{T_1} + \cdots + \frac{C_n}{T_n} + \max\left(\frac{B_1}{T_1}, \cdots, \frac{B_{n-1}}{T_{n-1}}\right) \le n(2^{1/n} - 1).$$

Proof: Since $n(2^{1/n} - 1) \le i(2^{1/i} - 1)$ and $\max(B_1/T_1, \cdots, B_{n-1}/T_{n-1}) \ge B_i/T_i$, if this equation holds then all the equations in Theorem 16 also hold.

Similar to the sufficient condition in Theorem 16, the conditions in Theorem 15 can be easily generalized. Specifically,

Theorem 18: A set of n periodic tasks using the priority ceiling protocol can be scheduled by the rate-monotonic algorithm for all task phasings if

$$\forall i,\ 1 \le i \le n,$$

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

where C_i, T_i, and U_i are defined in Theorem 15, and B_i is the worst case blocking time for τ_i.

Proof: The proof is identical to that of Theorem 16.

Remark: The blocking duration B_i represents the worst case conditions and hence the necessary and sufficient conditions of Theorem 15 become sufficient conditions in Theorem 18.

The following example helps clarify the use of Theorem 18. Consider the case of three periodic tasks:

- Task τ_1: $C_1 = 40$; $T_1 = 100$; $B_1 = 20$; $U_1 = 0.4$
- Task τ_2: $C_2 = 40$; $T_2 = 150$; $B_2 = 30$; $U_2 = 0.267$
- Task τ_3: $C_3 = 100$; $T_3 = 350$; $B_3 = 0$; $U_3 = 0.286$.

Task τ_1 can be blocked by task τ_2 for at most 20 units, while τ_2 can be blocked by task τ_3 for at most 30 time units. The lowest priority task, τ_3, cannot be blocked by any lower priority tasks. The total utilization of the task set ignoring blocking is 0.952, far too large to apply the conditions of Theorem 16. Theorem 18 is checked as follows:

1) Task τ_1: Check $C_1 + B_1 \le 100$. Since $40 + 20 \le 100$, task τ_1 is schedulable.
2) Task τ_2: Check whether either

$$\begin{array}{lll} & C_1 + C_2 + B_2 \le 100 & 80 + 30 > 100 \\ \text{or} & 2C_1 + C_2 + B_2 \le 150 & 120 + 30 \le 150. \end{array}$$

Task τ_2 is schedulable and in the worst case phasing will meet its deadline exactly at time 150.

3) Task τ_3: Check whether either

$$\begin{array}{lll} & C_1 + C_2 + C_3 \le 100 & 40 + 40 + 100 > 100 \\ \text{or} & 2C_1 + C_2 + C_3 \le 150 & 80 + 40 + 100 > 150 \\ \text{or} & 2C_1 + 2C_2 + C_3 \le 200 & 80 + 80 + 100 > 200 \\ \text{or} & 3C_1 + 2C_2 + C_3 \le 300 & 120 + 80 + 100 = 300 \\ \text{or} & 4C_1 + 3C_2 + C_3 \le 350 & 160 + 120 + 100 > 350. \end{array}$$

Task τ_3 is also schedulable and in the worst case phasing will meet its deadline exactly at time 300.

VI. Applications of the Protocol and Future Work

In this section, we briefly discuss the implementation aspects of the protocol as well as the possible extensions of this work.

A. Implementation Considerations

The implementation of the basic priority inheritance protocol is rather straightforward. It requires a priority queueing of jobs blocked on a semaphore and indivisible system calls *Lock_Semaphore* and *Release_Semaphore*. These system calls perform the priority inheritance operation, in addition to the traditional operations of locking, unlocking, and semaphore queue maintenance.

The implementation of the priority ceiling protocol entails further changes. The most notable change is that we no longer maintain semaphore queues. The traditional ready queue is replaced by a single job queue *Job_Q*. The job queue is a priority-ordered list of jobs ready to run or blocked by the ceiling protocol. The job at the head of the queue is assumed to be currently running. We need only a single prioritized job queue because under the priority ceiling protocol, the job with the highest (inherited) priority is always eligible to execute. Finally, the run-time system also maintains *S_List*, a list of currently locked semaphores ordered according to their priority ceilings. Each semaphore S stores the information of the job, if any, that holds the lock on S and the ceiling of S. Indivisible system calls *Lock_Semaphore* and *Release_Semaphore* maintain *Job_Q* and *S_List*. An example of the implementation can be seen in [13].

The function *Lock_Semaphore* could also easily detect a self-deadlock where a job blocks on itself. Since the run-time system associates with each semaphore the job, if any, that holds the lock on it, a direct comparison of a job requesting a lock and the job that holds the lock determines whether a self-deadlock has occurred. If such a self-deadlock does occur, typically due to programmer error, the job could be aborted and an error message delivered.

Suppose monitors are used for achieving mutual exclusion. We again assume that a job does not suspend until its completion when it executes alone on the processor. We also assume that the job does not deadlock with itself by making nested monitor calls. A job inside a monitor inherits the priority of any higher priority job waiting on the monitor. To apply the priority ceiling protocol, each monitor is assigned a priority ceiling, and a job J can enter a monitor only if its priority is higher than the highest priority ceiling of all monitors that have been entered by other jobs. Since the priority ceiling protocol prevents deadlocks, nested monitor calls will not be deadlocked. The implications of priority ceiling protocol to Ada tasking are more complicated and are beyond the scope of this paper. Readers who are interested in this subject are referred to [1].

B. Future Work

The priority ceiling protocol is an effective real-time synchronization protocol for it prevents deadlock, reduces the blocking to at most one critical section, and is simple to implement. Nonetheless, it is still a suboptimal protocol in that it can cause blocking to a job that can be avoided by enhancements to the protocol. Although a formal treatment of possible enhancements is beyond the scope of this paper, we would like to present the ideas of some possible enhancements to stimulate more research on this subject.

For example, we can define the *priority floor* of a semaphore, analogous to its priority ceiling, as the priority of the lowest priority job that may access it. Then, a job J can lock a semaphore S if its priority is higher than the priority ceiling of S *or* if the following conditions are true. The lock on S can also be granted if the priority of J is equal to the priority ceiling of S and the priority floor of S is greater than the highest priority preempted job. This latter condition, called the *priority floor condition*, ensures that neither a preempted job nor a higher priority job accesses S. This guarantees that deadlocks and chaining will be avoided. This protocol is called the *priority limit protocol*. The priority limit protocol eliminates the ceiling blocking that J_0 encounters at time t_5 in Example 4. Moreover, this protocol requires identical information as does the priority ceiling protocol and can be implemented with equal ease. However, the priority limit protocol does not improve the worst case behavior and hence the schedulability.

It is also possible to enhance the priority limit protocol by replacing the priority floor condition by the following condition. A job J can also be allowed to lock a semaphore S if the priority of J is equal to the priority of S and no preempted lower priority job accesses the semaphore S. This condition also guarantees avoidance of deadlock and chaining. This protocol is called the *job conflict protocol* and is better than the priority ceiling and priority limit protocols.[7] The job conflict protocol is, however, still a suboptimal protocol. It will be an interesting exercise to develop an optimal priority inheritance protocol, and then compare it to the priority ceiling protocol for both performance and implementation complexity.

VII. Conclusion

The scheduling of jobs with hard deadlines is an important area of research in real-time computer systems. In this paper, we have investigated the synchronization problem in the context of priority-driven preemptive scheduling. We showed that a direct application of commonly used synchronization primitives may lead to uncontrolled priority inversion, a situation in which a high priority job is indirectly preempted by lower priority jobs for an indefinite period of time. To remedy this problem, we investigated two protocols belonging to the class of *priority inheritance protocols*, called the *basic priority inheritance protocol* and the *priority ceiling protocol* in the context of a uniprocessor. We showed that both protocols solve the uncontrolled priority inversion problem. In particular, the priority ceiling protocol prevents deadlocks and reduces the blocking to at most one critical section. We also derived a set of sufficient conditions under which a set of periodic tasks using this protocol is schedulable by the rate-monotonic algorithm. Finally, we outlined implementation considerations for and possible extensions to this protocol.

Acknowledgment

The authors wish to thank D. Cornhill for his contributions on the priority inversion problems in Ada, J. Goodenough for his many insightful and detailed comments on this paper that helped us to clarify some of the key issues, and K. Ramamritham for his suggestions on the possible enhancements of this protocol. We would also like to thank H. Tokuda, T. Ess, J. Liu, and A. Stoyenko for their helpful comments. Finally, we want to thank the referees for their many fine suggestions.

References

[1] J. B. Goodenough and L. Sha, "The priority ceiling protocol: A method for minimizing the blocking of high priority Ada tasks," in *Proc. 2nd ACM Int. Workshop Real-Time Ada Issues*, 1988.

[2] B. W. Lampson and D. D. Redell, "Experiences with processes and monitors in Mesa," *Commun. ACM*, vol. 23, no. 2, pp. 105–117, Feb. 1980.

[7] This enhancement was suggested by Krithi Ramamritham.

[3] J. P. Lehoczky and L. Sha, "Performance of real-time bus scheduling algorithms," *ACM Perform. Eval. Rev.*, Special Issue, vol. 14, no. 1, May 1986.

[4] J. P. Lehoczky, L. Sha, and J. Strosnider, "Enhancing aperiodic responsiveness in a hard real-time environment," in *Proc. IEEE Real-Time Syst. Symp.*, 1987.

[5] J. P. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm—Exact characterization and average case behavior," in *Proc. IEEE Real-Time Syst. Symp.*, 1989.

[6] D. W. Leinbaugh, "Guaranteed response time in a hard real-time environment," *IEEE Trans. Software Eng.*, Jan. 1980.

[7] J. Y. Leung and M. L. Merrill, "A note on preemptive scheduling of periodic, real time tasks," *Inform. Processing Lett.*, vol. 11, no. 3, pp. 115–118, Nov. 1980.

[8] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.

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

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

[11] L. Sha, J. P. Lehoczky, and R. Rajkumar, "Solutions for some practical problems in prioritized preemptive scheduling," in *Proc. IEEE Real-Time Syst. Symp.*, 1986.

[12] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Task scheduling in distributed real-time systems," in *Proc. IEEE Industrial Electron. Conf.*, 1987.

[13] —, "Priority inheritance protocols: An approach to real-time synchronization," Tech. Rep., Dep. Comput. Sci., CMU, 1987.

[14] W. Zhao, K. Ramamritham, and J. A. Stankovic, "Scheduling tasks with resource requirements in hard real-time systems," *IEEE Trans. Software Eng.*, Apr. 1985.

[15] —, "Preemptive scheduling under time and resource constraints," *IEEE Trans. Comput.*, Aug. 1987.

Lui Sha (S'76–M'84) received the B.S.E.E. (Hons.) degree from McGill University, Montreal, P.Q., Canada in 1978 and the M.S.E.E. and Ph.D. degrees from Carnegie-Mellon University (CMU), Pittsburgh, PA, in 1979 and 1985.

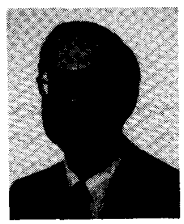
He was an engineer in the CMU Department of Computer Science from 1979 to 1984 and was a member of the Research Faculty in CMU CS department from 1985 to 1987. Since 1988 he has been a member of the Technical Staffs in the CMU's Software Engineering Institute, a member of Research Faculty in the CMU CS department, and a senior member of the Advanced Real-Time Technology (ART) project at CMU CS department. He is interested in developing analytical solutions for the problems in the construction of distributed real-time systems.

Dr. Sha is a member of the IEEE Computer Society.

Ragunathan Rajkumar (M'90) received the B.E. (Hons.) degree from the P.S.G. College of Technology, Coimbatore, India, and the M.S. and Ph.D. degrees from Carnegie Mellon University in 1986 and 1989, respectively.

He has been a Research Staff Member at the IBM Thomas J. Watson Research Center, Yorktown Heights, NY, since 1989. His interests lie in the area of real-time systems.

Dr. Rajkumar is a member of the IEEE Computer Society and the Association for Computing Machinery.



John P. Lehoczky (M'88) received the B.A. degree in mathematics from Oberlin College, Oberlin, OH, in 1965, and the M.S. and Ph.D. degrees in statistics from Stanford University, Stanford, CA, in 1967 and 1969, respectively.

He was an Assistant Professor of Statistics at Carnegie Mellon University, Pittsburgh, PA, from 1969 to 1973, Associate Professor from 1973 to 1981, and Professor from 1981 to the present. He has served as Head of the Department of Statistics since 1984. His research interests involve applied probability theory with emphasis on models in the area of computer and communication systems. In addition, he is a senior member of the Advanced Real-Time Technology (ART) Project in the Carnegie Mellon University Computer Science Department and is doing research in distributed real-time systems.

Dr. Lehoczky is a member of Phi Beta Kappa, a fellow of the Institute of Mathematical Statistics, and the American Statistical Association. He is a member of the Operations Research Society of America and the Institute of Management Science. He served as area editor of *Management Science* from 1981 to 1986.