Real-Time Synchronization Protocols for Multiprocessors

Ragunathan Rajkumar¹, Lui Sha² and John P. Lehoczky³

¹Department of Electrical and Computer Engineering
²Department of Computer Science
³Department of Statistics
Carnegie Mellon University

Abstract

The potential speedup of applications has motivated the widespread use of multiprocessors in recent years. Several mechanisms exist to synchronize tasks that execute on different processors, but share data and resources. In a hard real-time context, however, these synchronization mechanisms need to have bounded the blocking duration of a task waiting for a resource. Only then can deadlines of tasks be guaranteed. Unfortunately, a direct application of commonly used synchronization primitives like semaphores, monitors or message-passing mechanisms such as the Ada rendezvous can lead to uncontrolled priority inversion where a higher priority task is blocked by a lower priority task for an indefinite period of time. This paper studies the class of priority inheritance protocols for synchronizing tasks executing in parallel on multiprocessors. The multiprocessor priority ceiling protocol is a priority inheritance protocol that not only bounds the duration of blocking but also prevents deadlocks. These properties allow us to derive a set of sufficient conditions under which a set of periodic tasks using this protocol will be schedulable.

1. Introduction

The scheduling of jobs with hard deadlines has been an important area of research in real-time computer systems. Both non-preemptive and preemptive scheduling algorithms have been studied in the literature [5, 7, 8, 10, 12, 13]. 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, Ramamirtham and Stankovic [17, 18] investigated the use of heuristic algorithms to generate feasible schedules. Their heuristic has a high probability of success in the generation of feasible schedules. Dhall and Liu [2], and Mok [9] have considered scheduling algorithms for multiple processors assuming independent tasks.

In this paper, we investigate the synchronization problem in the context of priority-driven preemptive scheduling on shared memory multiprocessors. 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 even on uniprocessors by adversely affecting both the schedulability and predictability of real-time systems. The priority inheritance protocols [11, 14] constitute priority management schemes for synchronization primitives which remedy the uncontrolled priority inversion problem on uniprocessors. In this paper, we extend these protocols to multiprocessors. We formally define the protocols in a multiprocessor environment and in terms of binary semaphores. In Section 2, we summarize the problems of existing synchronization primitives on uniprocessors and review the priority ceiling protocol that effectively addresses these problems. In Section 3, we address the problems of synchronization primitives on multiprocessors, and present the multiprocessor priority ceiling protocol as a solution and analyze its properties. In Section 4, we generalize the multiprocessor priority ceiling protocol, and outline a task allocation scheme based on the generalized protocol. Finally, Section 5 presents the concluding remarks.

2. Task Synchronization on Uniprocessors

In this section, we provide an overview of the priority inversion problem on uniprocessors and discuss how this problem can be solved.

2.1. The Priority Inversion Problem

Priority inversion is the phenomenon where 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. 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. To maintain a high
degree of schedulability, we require protocols that would minimize the amount of blocking\(^2\). It is also important to be able to analyze the performance of such a protocol in order to determine the schedulability of real-time tasks that use this protocol.

The use of common synchronization primitives like semaphores, locks, monitors and Ada rendezvous is necessary to protect the consistency of shared data or to guarantee the proper use of non-preemptible resources. However, 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 duration of priority inversion.

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, job \( J_2 \) arrives, preempts \( J_3 \) and later attempts to use the shared data. However, job \( J_2 \) will be blocked on the semaphore \( S \). We would expect that \( J_1 \), being the highest priority job, will be blocked for a duration no longer than the time for job \( J_3 \) to exit its critical section. However, the duration of blocking is, in fact, unpredictable. This is because job \( J_2 \) can be preempted by the intermediate priority job \( J_2 \). The blocking of \( J_2 \) and hence that of \( J_2 \) 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 and the Ada rendezvous. The priority inversion problem was first discussed by Lampson and Redell [4] in the context of monitors. They suggest that the monitors be executed at a priority level higher than all tasks that would ever call the monitor.

We now define some of 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. 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. An aperiodic task can be serviced by means of a periodic server [6]. Each periodic task is assigned a fixed priority based on the rate-monotonic scheduling algorithm [8]. The rate-monotonic scheduling algorithm is an optimal static priority scheduling algorithm when tasks do not block each other and assigns higher priority to tasks with shorter periods. Every job of the same task is also assigned the task's priority. If two jobs are eligible to run, the higher priority job will be run. Jobs with the same priority are executed in a FCFS discipline. In all our discussions below, we assume that jobs \( J_1, J_2, \ldots, J_n \) of tasks \( T_1, T_2, \ldots, T_n \) respectively are listed in descending order of priority with \( J_1 \) having the highest priority. Each periodic task \( T_i \) has a period \( T_i \) and each job \( J_i \) has a computational requirement of \( C_i \) which has to be completed before the end of the task's period.

We now state the notation used to represent critical sections and job priorities.

**Notation:** We use the notation \( z_i \) to represent the \( k \)-th critical section in job \( J_i \) guarded by \( S_j \). This notation is necessary because job \( J_i \) can access the semaphore \( S_j \) a multiple number of times. However, we shall mostly use the notation \( z_i \) to denote an arbitrary member of the set \( \{ z_1, z_2, \ldots, z_k \} \). We shall also use the abbreviated notation \( z_i \) if the corresponding semaphore \( S_j \) is not of interest.

**Notation:** We use the notation \( P_i \) to denote the priority of job \( J_i \). Again, we assume that \( P_1 > P_2 > \cdots > P_n \).

**Notation:** We use the notation \( P(S_j) \) and \( V(S_j) \) to denote the indivisible operations wait and signal, respectively, on the binary semaphore \( S_j \).

We assume that each shared data structure is guarded by a binary semaphore and define the protocols presented in this paper in terms of binary semaphores. However, the idea is also applicable when monitors or rendezvous are used. We also assume that a job will not attempt to lock a semaphore 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. A job can have multiple critical sections that do not overlap, e.g. \{ \ldots \, P(S_1) \, V(S_1) \, \cdots \, P(S_k) \, V(S_k) \, \ldots \}. A critical section can be nested, i.e. a job \( J_i \) can make nested semaphore accesses, e.g. \{ \ldots \, P(S_1) \, \cdots \, P(S_2) \, V(S_2) \, \cdots \, V(S_k) \, \cdots \}. In this case, the critical section \( z_{i,j} \) is bounded by \( P(S_j) \) and \( V(S_j) \) and nests the critical section \( z_{i,j-1} \). In subsequent discussions, a nested critical section such as \( z_{i,j} \) will be referred to as a single (nested) critical section. That is, the terms "critical section" and "nested critical section" will be used interchangeably. Finally, in this section, we assume a uniprocessor executing a fixed set of known tasks. In the next section, we shall extend the protocol to multiprocessors.

### 2.2. Priority Inheritance Protocols

The use of priority inheritance protocols is one approach to rectify the priority inversion problem inherent in existing synchronization primitives [14]. 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 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 \( J_3 \) is blocked by job \( J_2 \). The priority inheritance protocol stipulates that job \( J_3 \) execute its critical section at job \( J_2 \)'s priority. As a result, job \( J_3 \) will not preempt job \( J_2 \) and will itself be blocked. That is, the higher priority job \( J_2 \) must wait for the critical section of lower priority job \( J_3 \) to be executed, because job \( J_3 \) "inherits" the priority of job \( J_2 \). Otherwise, \( J_3 \) will be indirectly preempted by \( J_2 \). When \( J_2 \) exits its critical section, it regains its assigned lowest priority and \( J_3 \) which was blocked by \( J_2 \) is awakened. Job \( J_3 \), having the highest priority, immediately preempts \( J_2 \) and runs to completion. This enables \( J_2 \) and \( J_3 \) to resume in succession and run to completion. On the other hand, suppose that job \( J_3 \) instead of \( J_2 \), shares the semaphore with \( J_2 \) and is blocked by job \( J_2 \). In this case, job \( J_3 \) executes its critical section at the priority of \( J_2 \). If job \( J_2 \) is ready to execute, it will preempt job \( J_3 \) and hence \( J_3 \). This is appropriate, since both job \( J_2 \) and \( J_3 \) should be preempted by job \( J_1 \). It was shown that if there are \( m \) lower priority tasks and these lower priority tasks access \( k \) distinct

---

\(^2\)See Section 3.3.
semaphores, a job $J$ can be blocked by at most $\min(m, k)$ critical sections.

The basic priority inheritance protocol, however, has the following two problems. First, this basic protocol, by itself, does not prevent deadlocks. 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 blockings can be formed. For instance, suppose that $J_2$ needs to sequentially access $S_1$ and $S_2$. Also suppose that $J_2$ preempts $J_1$ within the critical section $t_{1.1}$ and enters the critical section $t_{2.2}$. 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 $J_1$ and $J_2$. As a result, $J_1$ would be blocked for the duration of two critical sections, once to wait for $S_1$ to release $S_1$ and again to wait for $J_2$ to release $S_2$. Thus, a chain of blockings is formed.

The priority ceiling protocol [14] effectively addresses both these problems posed by the basic priority inheritance protocol. This protocol not only minimizes the blocking time of a job $J$ to at most the duration of one critical section but also prevents the formation of deadlocks. The goal of this protocol is to prevent the formation of mutual blocking and the formation of chained blockings. 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, the priority at which this new critical section will execute will be strictly higher than the priorities of all the preempted critical sections, taking the priority inheritance into consideration. If this condition cannot be satisfied, job $J$ is denied entry into the critical section $z$ and blocked. And the job that blocks $J$ inherits $J$'s priority. Example 2 illustrates this idea and the deadlock avoidance property while Example 3 illustrates the avoidance of chained blockings.

Definition: The priority ceiling of a semaphore is defined as the priority of the highest priority job that may lock this semaphore. The priority ceiling of a semaphore $S$ represents the highest priority at which a critical section guarded by $S$ can be executed.

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. Suppose the sequence of processing steps for each job is as follows:

$J_0 = \{ \ldots, \text{P}(S_0), \ldots, \text{V}(S_0), \ldots \}$
$J_1 = \{ \ldots, \text{P}(S_1), \ldots, \text{P}(S_2), \ldots, \text{V}(S_1), \ldots, \text{V}(S_2), \ldots \}$
$J_2 = \{ \ldots, \text{P}(S_2), \ldots, \text{P}(S_1), \ldots, \text{V}(S_1), \ldots, \text{V}(S_2), \ldots \}$

Recall that the priority of job $J_0$ 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 $J_0$.

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

- At time $t_0$, job $J_0$ enters its critical section by making an indivisible system call to execute $\text{P}(S_0)$. However, the run-time system will find that job $J_0$'s priority is not higher than the priority ceiling of locked semaphore $S_0$. Hence, the run-time system blocks job $J_0$ without locking $S_0$. Job $J_0$ now inherits the priority of job $J_0$, 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 blocked instead, the potential deadlock involving $J_1$ and $J_2$ is prevented.

- At time $t_4$, $J_2$ is still in its critical section and the highest priority job $J_0$ arrives and preempts $J_2$. Later, $J_0$ attempts to lock semaphore $S_2$. Since the priority of $J_0$ is higher than the priority ceiling of $S_2$, 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 $t_6$, $J_2$ 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 $S_1$ for there are no preempted critical sections.

- At time $t_7$, $J_0$ has exited its critical section and completes execution. Job $J_0$ resumes, since $J_1$ is blocked by $J_2$ and cannot execute. $J_0$ continues execution and locks $S_1$, and $J_2$ resumes.

- At time $t_8$, $J_2$ 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 $S_2$, and $J_0$ was blocked by the lower priority job $J_2$ during the intervals $[t_2, t_3]$ and $[t_4, t_5]$. However, these intervals correspond to part of the duration that $J_2$ needs to lock $S_2$. Thus, $J_0$ 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 dis-
joint 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 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 \( J_2 \) lock \( S_1 \) at time \( t_0 \). At time \( t_1 \), job \( J_2 \) arrives and preempts \( J_1 \). 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 \( J_2 \) resumes execution at \( J_3 \)’s priority. At time \( t_3 \), when \( J_3 \) is still in its critical section, \( J_1 \) arrives and finds that only one semaphore \( S_1 \) is locked. At time \( t_4 \), \( J_1 \) is blocked by \( J_2 \), which holds the lock on \( S_1 \). Hence, \( J_1 \) inherits the priority of \( J_2 \). At time \( t_5 \), job \( J_2 \) exits its critical section \( z_1 \). resumes its original priority and awakens \( J_1 \). Job \( J_1 \) having the highest priority, preempts \( J_2 \) and runs to completion. Next, \( J_2 \) which is no longer blocked complete its execution and is followed by \( J_3 \).

Again, note that \( J_1 \) is blocked by \( J_2 \) in the interval \([t_4, t_5]\) which corresponds to the single critical section \( z_1 \). Also, job \( J_2 \) is blocked by \( J_3 \) during the disjoint intervals \([t_2, t_3]\) and \([t_4, t_5]\) which also correspond to the same critical section \( z_1 \).

The priority ceiling protocol belongs to the class of pessimistic protocols which sometimes create unnecessary blocking. Nonetheless, this protocol dramatically improves the worst case blocking. It was proved that,

**Theorem 1:** The priority ceiling protocol prevents deadlocks. [14]

**Theorem 2:** A job \( J \) can be blocked for at most the duration of one critical section of any lower priority job. [14]

Theorem 2 is based on the assumption that a job does not suspend itself until completion once it begins execution. However, a job may voluntarily suspend itself, say, to delay itself for a specified amount of time. This leads us to the following corollary.

**Corollary 3:** A job \( J \), that suspends itself \( n \) times during its execution, can be blocked for the duration of at most \( n+1 \) critical sections of lower priority jobs.

### 3. Synchronization on Multiprocessors

In this section, we present a priority ceiling protocol that can be used on multiprocessors. In the analysis of scheduling algorithms, a fundamental notion is the concept of waiting time. The schedulability analysis originally developed by Liu and Layland [8] assumes that tasks are independent. Under this assumption a task only waits for equal priority tasks that arrive earlier or for higher priority tasks. In [11, 14], the effect of using semaphores in a uniprocessor is taken into the account through the concept of blocking. Blocking is defined as the duration that a job waits for the execution of lower priority tasks. In the case of multiple processors, the concept of waiting needs to be generalized to include remote blocking. When a job has to wait for the execution of a task of any priority assigned to another processor, it is said to experience remote blocking. One of our basic objectives is to minimize the remote blocking time in the design of multiprocessor synchronization protocols. Minimized blocking directly translates to additional schedulable utilization on each processor as we shall see in Section 3.3.

**Example 4:** Consider normal prioritized execution without priority inheritance in effect. Suppose that task \( \tau_1 \) is bound to processor \( p_1 \) and that tasks \( \tau_2 \) and \( \tau_3 \) are bound to processor \( p_2 \). Suppose that \( J_1 \) is executing on processor \( p_1 \) and wants to lock semaphore \( S \). But \( S \) is currently locked by job \( J_2 \) executing on processor \( p_2 \). Job \( J_1 \) is now said to encounter remote blocking. One might again expect that \( J_1 \) will be blocked only for the duration of \( J_2 \)'s execution of its critical section. However, if \( J_2 \) now preempts \( J_3 \) on processor \( p_2 \), \( J_1 \) will be blocked until \( J_2 \) completes (or blocks) to allow \( J_2 \) to resume and release \( S \). The blocking time of \( J_1 \) will continue until \( J_2 \) and any other intermediate priority jobs on \( p_2 \) complete execution or suspend themselves.

Again note that, with priority inheritance in effect in the above example, \( J_2 \) will inherit \( J_1 \)'s higher priority and complete its critical section without being preempted by \( J_3 \) or other intermediate priority jobs. Once \( J_2 \) releases \( S \), \( J_1 \) would be able to resume. However, let us now consider the following example.

**Example 5:** Suppose that tasks \( \tau_1 \), \( \tau_2 \), and \( \tau_3 \) are bound to processor \( p_2 \) and that task \( \tau_3 \) is bound to processor \( p_1 \). Job \( J_2 \) is executing on processor \( p_1 \) and attempts to lock semaphore \( S \). But \( S \) is currently locked by job \( J_1 \) executing on processor \( p_2 \). However, if \( J_3 \) now preempts \( J_4 \) on processor \( p_2 \), \( J_4 \) will be blocked until \( J_4 \) completes (or blocks). Thus, the blocking time of \( J_1 \) will continue until arriving higher priority jobs on \( p_2 \) (such as \( J_4 \) and \( J_2 \) ) complete execution or suspend themselves. Since \( \tau_1 \) and \( \tau_2 \) are periodic tasks, the blocking duration of \( J_1 \) can be extremely long.

Note that in Example 5, even the imposition of priority inheritance does not force any changes in the event sequence and the blocking duration of \( J_4 \) remains unchanged. It can also be easily seen that a direct use of the uniprocessor priority ceiling protocol does not induce any changes either. The blocking duration of \( J_4 \), unfortunately, can be a function of the entire execution times of jobs \( J_1 \) and \( J_2 \). This example illustrates the fact that the nature of remote blocking is very different from that of uniprocessor blocking. Blocking can be considered to be the duration that a task has to wait compared to the situation where no semaphores are present. In the absence of any data-sharing, \( J_4 \) would not have to wait at all and its blocking duration now becomes a function of execution times of tasks on other processors. Such a situation cannot be improved with either priority inheritance or the priority ceiling protocol. Hence the need to design a multiprocessor synchronization protocol.

Our goal is to design a multiprocessor synchronization protocol that bounds the remote blocking duration of a job as a function of the duration of critical sections of other jobs and not as a function of the duration of executing non-critical section code. In other words, we forbid situations where a job has to be blocked while another job executes outside a critical section. For instance, in Example 5, jobs \( J_1 \) and \( J_2 \) should not be allowed to execute outside their critical sections while job \( J_3 \) is blocked waiting for...
them to complete. The motivation behind this goal is the observation that a critical section is very small relative to the task execution time.

Another fundamental goal of our synchronization protocol is that whenever possible, we would let a lower priority job wait for a higher priority job. For example, if two jobs $J_H$ and $J_L$ are waiting for the release of a shared resource, the higher priority job $J_H$ will be allowed to access the resource first even if $J_L$ has been waiting for a longer duration. Such situations arise, for instance, when these two jobs are executing on two different processors and need a shared resource currently locked by a job on a third processor. Given a certain blocking duration $B$ for a job $J$ and a period $T$ for the corresponding task $\tau$, the ratio $B/T$ is a measure of schedulability loss due to blocking. Under the rate-monotonic scheduling algorithm, a lower priority job has a longer period and hence less schedulability loss results if we let the lower priority job wait. This objective is also reflected in our prioritized queues on the semaphores accessed by real-time tasks.

In summary, our objective is to minimize the schedulability loss due to blocking. The first goal is that the worst-case blocking duration $B$ of a job should be a function of critical section durations only and the second goal is to minimize the impact of $B$ by letting a lower priority job rather than a higher priority job experience blocking. Thus, our approach differs inherently from global synchronization techniques like those proposed by Gait [3] where tasks access semaphores on a FIFO basis.

3.1. The Multiprocessor Priority Ceiling Protocol

In this section, we present a priority ceiling protocol suitable for synchronizing tasks executing on multiprocessors.

Static Binding vs. Dynamic Binding

We assume that tasks are statically bound to a processor, i.e. once tasks are allocated to processors, each processor runs the same set of tasks. Each task, thus, runs on its host processor. We do not consider dynamic binding of jobs to processors for the following reason. Dhall and Liu [2] have shown that the rate-monotonic algorithm, which performs well on uniprocessors, behaves poorly for multiprocessors with dynamic binding. For example, consider the task set consisting of $m$-1 tasks, each with computational requirements of $2t$ and period 1, and a task with computation time 1 and period $1+\epsilon$. If we schedule this set of tasks on an $m$-1 processor system with dynamic binding, the highest priority $m$-1 tasks ready to run are scheduled on the $m$-1 processors. However, the first job of the task with period $1+\epsilon$ will miss its deadline. The utilization factor of this task set is $U = 2(m-1)+1/(1+\epsilon)$. As $\epsilon \to 0$, $U \to 1$, i.e. just $1+1/m-1$ of the available processor cycles. In contrast, with static binding, the task with period $1+\epsilon$ can be bound to dedicated processor and the other tasks statically bound to a different processor. The task set becomes schedulable with merely two processors. In general, with static binding, each processor can be utilized at least up to $ln 2$ (69%), the least upper bound of schedulable utilization when the rate-monotonic scheduling algorithm is used. An identical situation exists for dynamic priority scheduling algorithms such as the earliest deadline scheduling algorithm as well. Dynamic binding, in principle, could give better performance if certain combinations of task bindings are not allowed to occur, e.g. the above worst-case situation must be detected and avoided during run-time. Without sophisticated run-time evaluation, such bad combinations could be realized. In real-time systems, we typically cannot perform such combinatorial analyses during run-time and a static solution seems preferable.

Assumptions and Notation

We first state our assumptions and the notation used. For the sake of illustration of the protocol, we assume a multiprocessor with shared memory where each processor might also have local memory that is accessed only by its host processor. We also assume that static priority preemptive scheduling is used. In particular, we assume that tasks are assigned priorities according to the rate-monotonic scheduling algorithm. We also assume that at any given instant, the highest priority task ready to run is executing on each of the processors.

We assume that binary semaphores are used for task synchronization. A semaphore that is accessed by tasks allocated to different processors is referred to as a global semaphore, and a critical section guarded by a global semaphore is referred to as a global critical section (gcs). Similarly, a semaphore that is accessed by tasks allocated to a single processor is called a local semaphore, and a critical section guarded by a local semaphore is referred to as a local critical section (lcS). In this paper, we assume that global critical sections do not make any nested access to local semaphores and that local critical sections do not make any nested access to global semaphores. As we shall see, global semaphore accesses can potentially lead to high increases in blocking durations. Hence, we assume that efforts have been made to allocate tasks accessing the same semaphore to the same processor as far as possible. We describe a possible approach to task allocation in Section 4.1.

Synchronization and Application Processors

Jobs are first bound to processors. While a job $J$ always executes local critical sections and non-critical-section code on its host processor, its global critical sections are bound to and executed on a processor which can be other than $J$'s host processor. All global critical sections associated with the same global semaphore $S$ must be bound to the same processor. This means that each global semaphore $S_G$ is essentially bound to a synchronization processor $\rho_G$ and all jobs that need to lock $S_G$ need to execute the corresponding gcS on $\rho_G$. Thus, all gcS's

5Note that if there are no global semaphores in the system, the multiprocessor synchronization problem decomposes into multiple uniprocessor problems and the uniprocessor priority ceiling protocol can be used very effectively on each processor.

6Global critical sections are bound to a synchronization processor in order to isolate and bound the blocking effect due to global semaphores. Such a scheme also helps in porting to distributed systems where global shared resources can be handled in centralized locations and services offered via message-passing mechanisms between processors. Server tasks can replace specialized global critical sections code. If such a binding were not enforced and a job executes every gcS on its own host processor, complex sequences of events can take place and the worst-case blocking duration imposed by global semaphores can be extremely long. Furthermore, explicit deadlock avoidance techniques like partial ordering of semaphore accesses must be used. For instance, multiple requests for global semaphores may be outstanding from a single processor $\rho$. If several of these global semaphores are released simultaneously on other processors, only one gcS would be able to execute on $\rho$. As a result, jobs with lower priority than the ones ready to execute their gcS on $\rho$ would be needlessly blocked. A net effect is that excessive "coupling" exists between tasks executing on different processors leading to longer blocking durations.
will be executed on specially marked processors to isolate the blocking durations caused by global semaphores. A processor which executes global critical sections is called a synchronizer processor and processors which run application tasks only are called application processors. A synchronizer processor can also be running application tasks.

A job attempting to access a global semaphore on a synchronization processor can be considered to "migrate" to the synchronization processor for the execution of its global critical section. Once the job exits the global critical section, it can be considered to migrate back to its host processor. This allows us to refer to the thread of execution within a critical section by the job J on whose behalf the critical section is being executed. When a job is executing a gcs on a synchronization processor, its host processor is free and can be used to execute a lower priority job not in a gcs.

Global and Local Priority Ceilings

The priority ceiling of a semaphore S indicates the maximum priority at which a critical section guarded by this semaphore can execute.

Notation: The assigned priority of the highest priority task in the system is denoted by \( P_H \).

Under the uniprocessor priority ceiling protocol, no critical section will be executed at a priority higher than the highest priority of any task in the system. Hence, the priority ceiling of any local semaphore will be less than or equal to \( P_H \). However, on multiprocessors, global critical sections may need to execute at a priority higher than the highest assigned priority to tasks. The following theorem shows that this must be the case in order to obtain task blocking durations that are functions of critical section durations only.

Theorem 4: The remote blocking time of a job blocked on a global critical section will be a function of critical sections if and only if the global critical section cannot be preempted by jobs that are executing outside critical sections.

Proof: First, if global critical sections cannot be preempted by non-critical section code, then the waiting time for gcs can only be a function of critical sections. Second, if a gcs on a processor can be preempted by the non-critical section code of J, then a job on another processor blocked by this gcs will wait for the execution time of J.

Recall that our goal was to render the worst-case blocking duration of a job as a function of critical section durations only. Hence, by Theorem 4, when a job is waiting for a global semaphore \( S \), we should allow the gcs that will release \( S \) to execute at a priority higher than the assigned priority of the highest priority task running on that processor. It then becomes convenient to assume that this value of priority is higher than the assigned priorities of all tasks in the system. Since the priority ceiling of a semaphore \( S \) is the highest priority that a critical section guarded by \( S \) can execute, a global semaphore has to be assigned a priority ceiling which is higher than the highest assigned priority of all tasks in the system. Such a constraint does not apply to local semaphores. Thus, we need to define two types of priority ceilings, one for local semaphores and one for global semaphores.

Definition: The priority ceiling of a local semaphore \( S \) is defined to be the priority of the highest priority task that may lock the semaphore \( S \).

Definition: Let the priority of the highest priority task that accesses a semaphore \( S \) be denoted by \( P_S \). Then, the priority ceiling of a global semaphore \( S \) is defined such that

- The priority ceiling of \( S \) is higher than \( P_H \).
- If \( S \) and \( S \) are global semaphores and \( P_S > P_{S'} \), then the priority ceiling of \( S \) is greater than the priority ceiling of \( S \).

Normal Execution Priority of a GCS

Theorem 4 only requires that a gcs guarded by a global semaphore \( S \) be executed at higher priority than tasks executing outside critical sections when a remote task is blocked by \( S \). However, for ease of implementation, we always execute a gcs at higher priority than all tasks outside critical sections. The assigned priority at which a gcs executes is defined as follows:

Definition: The assigned priority of a global critical section \( z \) of job \( J \) executes is defined such that

- The priority is higher than \( P_H \).
- If the priority of \( J \) is greater than the priority of \( J \), then the priority assigned to gcs \( z \) is greater than the priority assigned to gcs \( z \).

The Multiprocessor Priority Ceiling Protocol

We now define the multiprocessor priority ceiling protocol that is used in each of the application processors and the synchronization processors.

- Each application processor runs the priority ceiling protocol on the set of tasks that it runs and the set of local semaphores bound to the application processor.
- Each global critical section on the synchronization processor normally executes at its assigned priority.
- The synchronization processor runs the priority ceiling protocol on the global critical sections (considering each thread of execution within a global critical section as a "job"), the set of application tasks (if any), and the set of global and local semaphores bound to the synchronization processor.

In the basic version of the protocol, we shall assume that all global critical sections are executed on a single synchronization processor \( P_G \) dedicated to execute global critical sections only. In the next section, we shall extend the protocol to include multiple synchronization processors and to add application tasks to synchronization processors.

---

6Such a set of conditions can be met, for example, as follows: Define the base priority ceiling \( P_B \) of a global semaphore as a fixed priority that is higher than \( P_H \). Then, the priority ceiling of a global semaphore can be the sum of the priority of the highest priority task accessing the semaphore and the base priority ceiling \( P_B \). This assignment maintains the ordering of the priority ceilings and ensures that the priority ceilings of all global semaphores are higher than \( P_H \).

7This set of conditions can also be met as follows. A global critical section \( z \) can be assigned a priority equal to the sum of the base priority ceiling \( P_B \) and \( P_H \), the priority of \( J \).
The following example illustrates the basic multiprocessor priority ceiling protocol.

**Example 6:** Suppose that there are two application processors $P_1$ and $P_2$, and one synchronization processor $P_G$. Let tasks $t_1$ and $t_2$ be bound to processor $P_1$, and tasks $t_3$ and $t_4$ be bound to processor $P_2$. Let local semaphores $S_1$ and $S_3$ be bound to $P_1$ and $P_2$ respectively. Let two global semaphores $S_G$ and $S_{G1}$ be bound to $P_G$. The jobs $J_1$, $J_2$, $J_3$, and $J_4$ have the following sequence of steps:

\[
J_1 = \{ \ldots, P(S_1), \ldots, V(S_1), \ldots, P(S_{G1}), \ldots, V(S_{G1}), \ldots \}
\]

\[
J_2 = \{ \ldots, P(S_2), \ldots, V(S_2), \ldots, P(S_G), \ldots, V(S_G), \ldots \}
\]

\[
J_3 = \{ \ldots, P(S_{G1}), \ldots, V(S_{G1}), \ldots, P(S_1), \ldots, V(S_1), \ldots \}
\]

\[
J_4 = \{ \ldots, P(S_3), \ldots, V(S_3), \ldots, P(S_{G1}), \ldots, V(S_{G1}), \ldots \}
\]

Hence, the local priority ceilings of local semaphores $S_1$ and $S_2$ are the priorities of jobs $J_1$ and $J_2$ respectively. Let us define a base priority ceiling $P_b = P_1 + 1$. The global semaphores $S_G$ and $S_{G1}$ are assigned priority ceilings $P_b + P_1$ and $P_b + P_3$ respectively. The global critical section of job $J_1$ is assigned a priority of $P_b + P_1$.

- At time $t_0$, job $J_1$ attempts to lock $S_1$ and since there is no other locked semaphore local to $P_1$, the local priority ceiling protocol grants the lock. Also, $J_2$ attempts to lock the global semaphore $S_{G1}$. The priority ceiling protocol on $P_G$ finds that the semaphore $S_{G1}$ has already been locked but that the assigned priority of $J_2$'s gcs is greater than the global priority ceiling of $S_{G1}$. Hence, $J_2$'s gcs is allowed to preempt $J_1$'s gcs on $P_G$. Meanwhile, $J_4$ resumes execution on the idle processor $P_2$.
- At time $t_0$, job $J_1$ completes its gcs and resumes execution on $P_1$. Job $J_2$ locks local semaphore $S_2$ since no other local semaphore is locked on $P_2$.
- At time $t_0$, job $J_2$ completes its gcs and releases $S_{G1}$ on $P_G$. Having higher priority, $J_3$ preempts $J_4$ and resumes execution on $P_2$. Job $J_3$ resumes execution of its gcs on $P_G$.
- At time $t_0$, job $J_4$ attempts to lock global semaphore $S_{G1}$. Since the assigned priority of the gcs is higher than the priority ceiling of $S_{G1}$, it preempts $J_3$ on $P_G$. Also, $J_2$ attempts to lock $S_2$. However, its priority is not higher than the priority ceiling of locked local semaphore $S_2$ and hence the lock is denied. Job $J_2$ is suspended, and $J_4$ inherits $J_2$'s priority and resumes execution in its gcs.
- At time $t_0$, job $J_1$ completes its gcs and resumes execution on $P_1$. Job $J_2$ resumes execution of its gcs on $P_G$. Job $J_4$ releases $S_2$ and resumes its original priority. Job $J_2$ resumes its execution and locks $S_2$.
- At time $t_0$, job $J_1$ completes on $P_1$ and $J_2$ releases $S_2$ on $P_2$.
- At time $t_0$, job $J_2$ completes and $J_4$ resumes execution on $P_2$.
- At time $t_0$, job $J_4$ attempts to lock $S_{G1}$ but is blocked since the assigned priority of its gcs is lower than the priority ceiling of locked semaphore $S_{G1}$.
- Between $t_0$ and $t_1$, job $J_2$ releases $S_{G1}$, locks and releases $S_1$, and finally completes. Job $J_4$ locks and releases $S_{G1}$ before completing.

3.2. Properties of the Basic Multiprocessor Priority Ceiling Protocol

The multiprocessor priority ceiling protocol prevents deadlocks and bounds the blocking duration of each task as a function of the critical section durations of other tasks. We shall prove these properties in this section.

**Theorem 5:** Under the multiprocessor priority ceiling protocol, deadlocks are avoided.

**Proof:** First, by assumption, a job cannot deadlock with itself. Hence, a job can only deadlock with some other job. Also, by assumption, global critical sections do not access local semaphores and vice-versa. Hence, access to global and local semaphores cannot occur within the same critical section. Since every global and local semaphore is accessed only by a single processor, deadlocks cannot occur across processor boundaries. The priority ceiling protocol which is used on each processor avoids deadlocks within a processor. The theorem follows.
We now consider the maximum blocking duration that each job can encounter.

Notation: Let $n^P_i$ represent the number of (outermost) global critical sections that a job $J_i$ executes before its completion. Again, a nested critical section like $\{P(S_1) \cdot \ldots \cdot P(S_j) \cdot \ldots \cdot V(S_r)\}$ is considered to be a single critical section.

Remark: Each time a job $J$ needs to enter a gcs, it suspends on its host processor $\varphi$. During this suspension, lower priority jobs on $\varphi$ can execute and lock a local semaphore. This locking can contribute to additional blocking to $J$ on the host processor, when it resumes (after the execution of its gcs at the synchronization processor). The next theorem provides a bound for this blocking duration at the local processor.

**Theorem 6:** A job $J_i$ bound to a processor $\varphi$ can be blocked for the durations of at most $n^P_i+1$ local critical sections of lower priority jobs bound to $\varphi$.

**Proof:** Job $J$ can be considered to be suspending itself when it attempts to enter a global critical section which will be executed on $\varphi$. The theorem follows from Corollary 3.

**Remark:** When a job $J$ attempts to enter a gcs $z$, it can be blocked by a locked global semaphore $S$ executing on the synchronization processor if the assigned priority of $z$ is not greater than the global priority ceiling of $S$. The next two theorems present an upper bound on this blocking duration.

**Theorem 7:** For every (outermost) gcs $J_i$ that $J_i$ enters on the synchronization processor $\varphi$, $J_i$ can be blocked for the duration of at most the duration of one global critical section of all lower priority jobs on $\varphi$.

**Proof:** The Theorem follows directly from Theorem 2 and the fact that each global critical section duration can be considered to be a job.

**Notation:** The sum of the durations that a job $J_i$ spends within global critical sections, when executing alone on the synchronization processor, is represented by $CS_i$.

**Theorem 8:** A job $J_i$ can wait for a duration of at most $(CS_i \times T_i/T_1)$ of a higher priority job $J_h$ on the synchronization processor $\varphi$ during its period $T_i$.

**Proof:** During the interval $T_i$, $J_i$ can execute for a maximum duration of $(CS_i \times T_i/T_1)$ in the synchronization processor. The Theorem follows.

**Effects of Deferred Execution**

Liu and Layland’s original schedulability analysis of the rate-monotonic scheduling algorithm [8] assumes that a periodic task arrives at periodic intervals and executes from initiation to completion without any suspensions. However, in the current scenario, a job can suspend itself waiting for a global semaphore to be released and this waiting duration can vary depending upon whether a global semaphore is locked and upon the requests from other jobs. As a result, a high priority job may be suspended for a long duration of time, then resume and complete its execution to just meet its deadline at the end of the period. The next job instantiation now occurs immediately which can now run to completion since all requests are satisfied. Thus, a high priority job $J_{H}$ can delay itself and then inflict “back-to-back hits” on a lower priority job $J_i$ that is initiated at the instant of $J_{H}$'s resumption. For instance, in the Liu and Layland analysis, the worst case occurs when $T_2/T_1 < 2$ and the lower priority job $J_i$ is preempted at most twice. However, in our context, even if $T_2/T_1 < 2$, job $J_i$ can be preempted 3 times as shown in Figure 3-2.

![Figure 3-2: The Invasive Effects of Deferred Execution](image)

A lower priority job $J_i$ in the worst case, encounters at most one additional preemption from $J_{H}$ than in the normal case with no suspensions. This bound is based upon Lemma 9 and the fact that, with no suspensions, a high priority job $J_{H}$ can be active $T_{2}/T_{1}$ times during the period $T_2$ of a lower priority task $J_i$.

**Lemma 9:** For all possible phasing and delays, a higher priority task $J_{H}$ can execute at most $T_{2}/T_{1} + 1$ times during the period $T_2$ of a lower priority task $J_i$.

**Proof:** The theorem follows directly from the fact that there can be at most $T_{2}/T_{1} + 1$ times that $J_{H}$ can be active during an interval $T_2$.

The effect of this worst-case additional preemption can be accounted for by considering either the maximum suspension delay of $J_i$ or the computation time of $J_{H}$ (whichever is smaller) as part of $B_i$, i.e., the additional invasiveness can be considered to be blocking for job $J_i$.

### 3.3. Schedulability Analysis

Theorems 6, 7 and 8 lead to sufficient conditions for a job $J$ bound to a processor $\varphi$ to meet its deadline.

**Theorem 10:** 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:

$$\forall i, 1 \leq i \leq n, \quad C_i \frac{1}{T_i} + C_{i+1} \frac{1}{T_{i+1}} + \ldots + C_n \frac{B_i}{T_n} \leq \frac{1}{T_i}$$

We test the above set of inequalities for each application processor $\varphi$ where $n$ is the number of tasks bound to $\varphi$, and $C_i$ and $T_i$ are the computation time and the period, respectively, of task $T_i$ bound to $\varphi$. The term $B_i$ is the maximum duration that a job $J$ bound to $\varphi$ can be blocked from initiation to completion. Since we use the priority ceiling protocol on each of the processors, Theorem 10 applies.

Suppose that $J_i$ is bound to a processor $\varphi$. Then $B_i$ is the sum of 1. Duration of $n^P_i+1$ local critical sections on $\varphi$ that can block
jobs not accessing a gcs. This can be used to improve the
(s) critical sections must be short relative to task periods in order to
The sum of the above three quantities yields
The schedulability of the synchronization processor
long gcs, the term
time of references to global semaphores relative to a task period.
also determined using the inequalities of Theorem
is the period of the task
is the total duration of gcs execution of job
Duration of global critical sections of higher priority jobs
bound to processors other than \( p \) if \( J_i \) accesses a global semaphore. That is, each job \( J_k \) which has higher priority
bound from executing. By Theorem 7, the upper bound of
first suspension. The first term in the
no
accesses a global semaphore. Again, a global
can
be replaced by

4. Duration of global critical sections of higher priority jobs
bound to processors other than \( p \) if \( J_i \) accesses a global semaphore. That is, each job \( J_k \) which has higher priority
than \( J_i \) is bound to a processor other than \( p \) and accesses a global semaphore, can contribute to a maximum blocking of
This follows from Theorem 8 and the following: A job not accessing a global semaphore does not
lock on a global semaphore allocated to a
accesses a gcs on \( p_{G_j} \). Such priority inversion can be avoided by allocating only the lowest priority jobs to synchronization processors.
Suppose that we have \( m \) synchronization processors. We disallow locks on global semaphores to be held across processor boundaries, i.e. a gcs executing on a synchronization processor \( p_{G_j} \) cannot request a nested lock on a global semaphore allocated to a
different synchronization processor. The primary reason behind this constraint is that such locking leads to excessive blocking durations with a cross-coupling between the processors. For example, consider a critical section on \( p_{G_j} \) attempting to lock a
semaphore \( S \) on \( p_{G_k} \). Tasks on \( p_{G_j} \) can now encounter remote
properties of the Generalized Protocol
Since we can now have multiple synchronization processors, their utilization will be generally low and application tasks can be
added to these synchronization processors. Hence, we must also consider the maximum blocking that can occur to tasks bound to
synchronization processors. Since tasks executed at the synchronization processor can be preempted by every global
critical section, the blocking time of task \( z_i \) due to global critical sections on the synchronization processor is

\[ B_i = \sum_{j \in R} C_j \times \left[ T_j / T_i \right] \]

where \( R \) is the set of tasks that may use a global semaphore. The
schedulability of the synchronization processor is determined as before. This computation of \( B_i \) is, again, pessimistic. If the
instants at which each job attempts to lock a global semaphore relative to the task's period are known, then \( B_i \) contains the
maximum duration of gcs accesses that can occur during any single period of \( T_{z_i} \).

Theorems 5 and 6 still hold under the generalized priority ceiling protocol. Theorems 7 and 8 are generalized in a straightforward
fashion for each synchronization processor \( p_{G_j} \).

4. The Generalized Multiprocessor Priority Ceiling Protocol
The basic multiprocessor priority ceiling protocol assumed the presence of a single synchronization processor. However, the
protocol definition permits the inclusion of multiple synchronization processors and the addition of application tasks to all
synchronization processors. In other words, a synchronization processor may execute some application tasks in addition to
gcs's. We refer to such an implementation as the generalized
multi-processor priority ceiling protocol. Under the generalized
multiprocessor priority ceiling protocol, gcs execution on any of the
synchronization processors will preempt an application task.

Remark: An application job \( J \) bound to a synchronization processor
\( p_{G_j} \) can be preempted by a lower priority job \( J_i \) bound to
another processor if \( J_i \) executes a gcs on \( p_{G_j} \). Such priority
inversion can be avoided by allocating only the lowest priority jobs to synchronization processors.

Notation: \( n_{G_j} \) represents the number of (outermost) global critical sections on \( p_{G_j} \), that a job \( J_i \) enters before its completion. We
have \( \sum_j n_{G_j} = n_{G} \).
Notation: The sum of the durations that a job \( J_i \) spends within global critical sections on processor \( \varphi_{G_i} \) when executing alone on the processor, is represented by \( CS_{G_i} \). Thus, we have \( \sum_j CS_{G_i} = CS_i \).

Remark: If a task does not access all the synchronization processors, its blocking would be less than in the case of a single synchronization processor. For example, suppose that a task \( \tau_i \) is bound to \( \varphi_1 \) accesses synchronization processor \( \varphi_{G_1} \) and a higher priority task \( \tau_k \) bound to \( \varphi_2 \) accesses synchronization processor \( \varphi_{G_2} \). \( \tau_i \)'s remote blocking time will not include \( \tau_k \)'s global critical sections, while it will if there is just a single synchronization processor. However, it must be noted that this absolute decrease in blocking is obtained by allocating a processor to be a synchronization processor when it could be used as an additional application processor. Thus, in this case, the computation of \( B_i \) of the job \( J_i \) assigned to \( \varphi_1 \) involves a smaller number of terms:

- If \( J_i \) is blocked by \( n_j^0 + 1 \) local critical sections of lower priority jobs on its host processor \( \varphi_1 \).
- If \( J_i \) is blocked for at most \( n_j^0 \) global critical sections of lower priority jobs on \( \varphi_{G_1} \). Sum this blocking for all synchronization processors.
- If \( J_i \) accesses a global critical section on \( \varphi_{G_1} \), \( J_i \) will be blocked for a duration of at most \( CS_{G_1} \times \left( \frac{1}{T_{\varphi_1}} \right) \) by a higher priority job \( J_k \) that accesses a global semaphore allocated to \( \varphi_{G_1} \). Sum this blocking for all blocking higher priority jobs that are not bound to \( \varphi_1 \) but access \( \varphi_{G_1} \) and over all \( \varphi_{G_1} \) accessed by \( J_i \).
- For each higher priority job \( J_i \) on \( \varphi_1 \) that suspends on global semaphores or for other reasons, add the term \( \min(C_1, l_k) \) to \( B_i \), where \( l_k \) is the maximum duration that \( J_k \) can suspend itself.

Once \( B_i \) has been computed for all \( i \), Theorem 10 can then be used to determine schedulability of each application processor and synchronization processors. The higher priority tasks for this computation will just be the higher priority tasks allocated to the host processor of \( \tau_i \).

4.1. Task Allocation

In this section, we illustrate a task allocation scheme to bind tasks to processors. The task allocation scheme is a complex optimization problem even for independent tasks [2]. However, tasks do share data and resources and the task allocation scheme must consider the blocking durations of each task to guarantee task deadlines. When the generalized multiprocessor priority ceiling protocol is used, the blocking duration for each task is bounded and should be considered by the task allocation scheme.

Traditional techniques for task allocation use graph-theoretic methods [15, 16]. In this approach, the system is modeled as a network in which tasks and processors are represented by nodes. Edges between task nodes represent inter-processor communication costs while edges between a task node and a processor node roughly represent the execution time. A graph-partitioning algorithm is then used to obtain a network cut with minimal weight. The complexity of this approach becomes exponential when the system contains more than two processors. Such graph partitioning algorithms cannot be used directly under our constraints. The methodology presented in [16], for instance, addresses the assignment of tasks, all ready at the same time, to processors and aims to minimize total execution times and communication times. In the current scenario, where tasks are periodic, the task assignment problem is not unlike a bin-packing problem [11].

The objective of the task allocation would be to determine the smallest number of processors which can schedule the tasks without missing deadlines. However, the determination of an optimal task allocation is beyond the scope of this paper and we shall just present an example of how the allocation could proceed:

**Example 7:** Let the set of tasks to be allocated be \( \{ \tau_1, \tau_2, \tau_3, \tau_4 \} \) and \( \tau_5 \) arranged in descending order of their priorities as assigned by the rate-monotonic scheduling algorithm. Let the set of semaphores in the system be \( S_1, S_2 \) and \( S_3 \). The tasks that access these semaphores are pictorially represented in Figure 4-1(a), i.e. all the tasks access \( S_1, S_2 \) and \( S_3 \) while \( \tau_5 \) access \( S_4 \). However, \( \tau_5 \)'s access of \( S_5 \) makes a nested access to \( S_3 \). Let us assume that at most two tasks can be scheduled on a single processor. The blocking duration is minimum when tasks that access the same semaphore are allocated to the same processor.

![Figure 4-1: The Task Allocation Procedure of Example 7.](image-url)

Our objective is to separate these relationships to form "closures", tasks within which can then be allocated to the same processor. Since global critical sections have higher priority than non-critical section code, this allows \( \tau_5 \) to access \( S_3 \) at higher priority,
on \( \phi_4 \) without causing any priority inversion. The partition \( \tau_3 \) can be allocated to a new processor \( \phi_3 \) or if schedulability criteria are met, it can be allocated to \( \phi_G \).

5. Conclusions

In this paper, we have addressed the problem of unbounded waiting time for access to global resources on shared memory multiprocessors. Such unbounded blocking duration can arise from priority inversion where a higher priority task has to wait for a lower priority task to complete access to a shared resource. The priority ceiling protocol is an efficient synchronization protocol for uniprocessors that not only prevents deadlocks but also limits the blocking duration of a task to at most the duration of one critical section of a lower priority task. The multiprocessor priority ceiling protocol is an extension of the priority ceiling protocol that can be applied to tasks executing in parallel on multiprocessors. Such a protocol prevents deadlocks and bounds the duration that a task has to wait for access to global data and/or resources. We have used the properties of this protocol to derive sufficient conditions to test whether task deadlines will be met. Finally, we have outlined a task allocation scheme to bind tasks to processors.

Acknowledgements

The authors would like to thank the referees for their helpful comments.

References


10Alternatively, it is possible to partition into \( \{ \tau_4 \} \) and \( \{ \tau_3 \} \) and allocate \( \delta_4 \) and \( \delta_3 \) to the latter.

269