Real-Time Synchronization Protocols for Shared Memory Multiprocessors

Ragunathan Rajkumar
IBM Thomas J. Watson Research Center
Yorktown Heights

Abstract

Multiprocessors have become popular in recent years due to hardware trends towards increasing performance with multiple processing elements. The speedups possible in multiprocessors are particularly attractive for real-time systems where additional computing power is in general desirable. Various synchronization mechanisms have been proposed and implemented in shared memory multiprocessors to synchronize tasks running on different processors. In hard real-time systems, any delays due to synchronization requirements must be bounded. Unfortunately, a direct use of common primitives like shared memory semaphores and message-based rendezvous in multiprocessors may cause a task to be blocked for an arbitrary duration of time. A multiprocessor protocol that bounds blocking durations on shared resources has been defined in [8]. Under this protocol, any given globally shared resource is controlled from a designated processor. This protocol is particularly well-suited for message-passing architectures, but does not exploit the availability of globally accessible shared memory in tightly coupled multiprocessors. In this paper, we define and analyze a priority-based synchronization protocol that explicitly uses shared memory primitives.

1. Introduction

Preemptive scheduling algorithms for uniprocessor real-time systems have been extensively studied in the literature [4, 6] and preemptive algorithms for multiple processors have also been investigated [1, 7]. A useful measure in real-time systems is schedulability, the resource utilization below which task deadlines can be guaranteed [6]. An important requirement in real-time systems is the need for tasks to share physical and logical resources. As a result, some synchronization mechanism must be provided to enforce the consistency constraints of the shared resources. Given that two or more tasks may share the same resource, one may need to block while the other task is using the resource. The blocking duration arising from these mutual exclusion requirements must be bounded in order to guarantee that system timing constraints will be satisfied. Mok [7] showed that the problem of deciding whether it is possible to schedule a set of periodic tasks is NP-hard when semaphores are used to enforce mutual exclusion.

The need to synchronize real-time tasks sharing physical and/or logical resources in a shared memory multiprocessor is the focus of this paper. In particular, we study the synchronization requirements in the context of a priority-driven preemptive scheduling discipline commonly used to implement real-time systems. It has been noted that a direct application of synchronization primitives like semaphores, monitors and the Ada rendezvous can lead to uncontrolled priority inversion where a high priority task can be blocked for an arbitrary duration of time [3, 10] by a lower priority task. A class of synchronization protocols called the priority inheritance protocols has been developed to rectify the unbounded priority inversion problem in the case of uniprocessors [10].

The priority inversion problem becomes much more severe in multiprocessor systems, and shall be considered in detail in Section 3. A real-time synchronization protocol for multiple processor environments has been proposed in [8]. This protocol is defined under the assumption that the responsibility for controlling any given globally shared resource is assigned to one particular processor. While multiple processors can control different global resources, only one processor controls all accesses to a given global resource. Hence, if a task requires the use of a global resource, it has to send a request to the controlling processor and obtain the results of the requested operation. Such a scheme is well-suited for message-passing architectures like distributed systems, but may be inefficient in the presence of shared memory. In this paper, we extend the work presented in [8] to make explicit use of the availability of shared memory in tightly coupled multiprocessors.

The paper is organized as follows. Section 2 briefly reviews a solution that has been proposed for bounding and minimizing synchronization delays in real-time systems [10]. In Section 3, we identify the waiting times introduced by synchronization requirements in multiple processor environments, and derive a set of goals for priority-based multiprocessor synchronization protocols. In Section 4, we study the underlying priority considerations for a shared memory synchronization protocol, and derive priority assignments to be used by the protocol. Section 5 defines and analyzes the properties of a shared memory synchronization protocol. Finally, Section 6 presents the concluding remarks.

2. Task Synchronization in Real-Time Systems

2.1. Synchronization Delays in Uniprocessors

On uniprocessors, a job, which is one instance of a task, can be delayed due to two factors: it can either be preempted by a higher priority job or wait for a lower priority job. Preemptions by higher priority jobs on the same processor are an integral part of the priority-driven preemptive scheduling discipline [6] and represent the intended operation. However, a job waiting for the execution of a lower priority job is undesirable, and the waiting time must be minimized or at least bounded. A common situation arises when two jobs attempt to access shared data. To maintain the consistency of the data, the two accesses must be serialized. If the higher priority job gains access first, 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, the higher priority job must wait until the lower priority job completes its access to the shared data. In other words, the higher priority job undergoes priority inversion. The duration of such priority inversion must be bounded and minimized. However, a direct use of synchronization primitives can actually jeopardize the deadlines of a real-time system. For instance, in the above example, suppose that the lower priority job gains access to the shared data first, and the higher priority job becomes blocked trying to access the shared data. The lower priority job, however, can be preempted by other medium jobs causing the higher priority job to block until all these jobs complete. Since the medium priority jobs can be periodic in nature, the blocking duration of the higher priority job can be unbounded.

2.2. The Priority Ceiling Protocol

The use of priority inheritance protocols is one approach to rectify the priority inversion problem inherent in existing synchronization primitives [10]. The basic idea of the priority inheritance protocols is that when a job J blocks higher priority jobs, it executes its critical section at the highest priority (until it unblocks J).

The priority inversion problem becomes much more severe in multiprocessor systems, and shall be considered in detail in Section 3. A real-time synchronization protocol for multiple processor environments has been proposed in [8]. This protocol is defined under the assumption that the responsibility for controlling any given globally shared resource is assigned to one particular processor. While multiple processors can control different global resources, only one processor controls all accesses to a given global resource. Hence, if a task requires the use of a global resource, it has to send a request to the controlling processor and obtain the results of the requested operation. Such a scheme is well-suited for message-passing architectures like distributed systems, but may be inefficient in the presence of shared memory. In this paper, we extend the work presented in [8] to make explicit use of the availability of shared memory in tightly coupled multiprocessors.

The paper is organized as follows. Section 2 briefly reviews a solution that has been proposed for bounding and minimizing synchronization delays in real-time systems [10]. In Section 3, we identify the waiting times introduced by synchronization requirements in multiple processor environments, and derive a set of goals for priority-based multiprocessor synchronization protocols. In Section 4, we study the underlying priority considerations for a shared memory synchronization protocol, and derive priority assignments to be used by the protocol. Section 5 defines and analyzes the properties of a shared memory synchronization protocol. Finally, Section 6 presents the concluding remarks.

2. Task Synchronization in Real-Time Systems

2.1. Synchronization Delays in Uniprocessors

On uniprocessors, a job, which is one instance of a task, can be delayed due to two factors: it can either be preempted by a higher priority job or wait for a lower priority job. Preemptions by higher priority jobs on the same processor are an integral part of the priority-driven preemptive scheduling discipline [6] and represent the intended operation. However, a job waiting for the execution of a lower priority job is undesirable, and the waiting time must be minimized or at least bounded. A common situation arises when two jobs attempt to access shared data. To maintain the consistency of the data, the two accesses must be serialized. If the higher priority job gains access first, 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, the higher priority job must wait until the lower priority job completes its access to the shared data. In other words, the higher priority job undergoes priority inversion. The duration of such priority inversion must be bounded and minimized. However, a direct use of synchronization primitives can actually jeopardize the deadlines of a real-time system. For instance, in the above example, suppose that the lower priority job gains access to the shared data first, and the higher priority job becomes blocked trying to access the shared data. The lower priority job, however, can be preempted by other medium jobs causing the higher priority job to block until all these jobs complete. Since the medium priority jobs can be periodic in nature, the blocking duration of the higher priority job can be unbounded.

2.2. The Priority Ceiling Protocol

The use of priority inheritance protocols is one approach to rectify the priority inversion problem inherent in existing synchronization primitives [10]. The basic idea of the priority inheritance protocols is that when a job J blocks higher priority jobs, it executes its critical section at the highest priority (until it unblocks J).

The priority inversion problem becomes much more severe in multiprocessor systems, and shall be considered in detail in Section 3. A real-time synchronization protocol for multiple processor environments has been proposed in [8]. This protocol is defined under the assumption that the responsibility for controlling any given globally shared resource is assigned to one particular processor. While multiple processors can control different global resources, only one processor controls all accesses to a given global resource. Hence, if a task requires the use of a global resource, it has to send a request to the controlling processor and obtain the results of the requested operation. Such a scheme is well-suited for message-passing architectures like distributed systems, but may be inefficient in the presence of shared memory. In this paper, we extend the work presented in [8] to make explicit use of the availability of shared memory in tightly coupled multiprocessors.

The paper is organized as follows. Section 2 briefly reviews a solution that has been proposed for bounding and minimizing synchronization delays in real-time systems [10]. In Section 3, we identify the waiting times introduced by synchronization requirements in multiple processor environments, and derive a set of goals for priority-based multiprocessor synchronization protocols. In Section 4, we study the underlying priority considerations for a shared memory synchronization protocol, and derive priority assignments to be used by the protocol. Section 5 defines and analyzes the properties of a shared memory synchronization protocol. Finally, Section 6 presents the concluding remarks.
delay itself for a specified amount of time. This leads to the following result presented in [10] as a corollary.

**Theorem 1:** 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 [10].

### 3. Multiple Processor Synchronization

#### 3.1. Assumptions and Notation

A job is a sequence of instructions that will continuously use the processor until its completion if it is executing alone on the processors. 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 [5]. Each periodic task is assigned a fixed priority based on the rate-monotonic scheduling algorithm [6]. The rate-monotonic scheduling algorithm assigns higher priority to tasks with shorter periods and is optimal for static priority scheduling when tasks do not block each other. Every job of the same task is also assigned the task’s priority. If two jobs are eligible to run, then 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 \( J_1, J_2, \ldots, J_m \) 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 \( J_1 \) has a computational requirement \( C_i \) which has to be completed before the end of the task’s period.

**Notation:** The notation \( P_i \) denotes the priority of job \( J_i \). We assume that \( P_1 > P_2 > \cdots > P_m \).

**Notation:** The notation \( P(S) \) and \( V(S) \) denote the indivisible operations wait and signal, respectively, on the binary semaphore \( S \).

We assume that each shared data structure is guarded by a binary semaphore and define the protocol presented in this paper in terms of binary semaphores. However, the idea is also applicable when monitors are used. We also assume that a job will not attempt to lock a semaphore it has already locked and thus deadlock itself. In addition, we assume that locks on semaphores will be released before or at the end of a job.

#### 3.2. Binding of Tasks to Processors

Given a set of \( m \) tasks to be executed on \( n \) processors, we have two options: to assign tasks to processors statically or dynamically. Static binding of tasks to processors is preferable. A task runs on its host processor when it is the highest priority ready task on that processor. This is known as static binding. Dhull and Liu [1] have shown that the rate-monotonic algorithm performs badly for multiprocessors with dynamic binding. For example, consider the task set consisting of \( m \) tasks, each with computational requirements of \( 2^i \) and period \( 1 \) unit, and a task with computational requirement \( 1 \) unit and period \( 1+i \). The identical \( m \) tasks will have higher priority under the rate-monotonic scheduling algorithm. If we schedule this set of \( m+1 \) tasks on \( m \) processors, the higher priority \( m \) tasks will begin execution on the \( m \) processors. However, the first instance of the task with period \( 1+i \) will miss its deadline. The utilization of this task set is \( U = 2^{m+1}/2^m \). As \( n \to 0 \), \( U \to 1 \), and a deadline can be missed with just \( 1/m \) of the available processor cycles being used. In contrast, with static binding, the task with period \( 1+i \) can be bound to a dedicated processor, and the other tasks statically bound to a different processor. The task set becomes schedulable with one processor. In general, there exist task sets which can be scheduled by static binding and not by dynamic binding, and vice-versa. However, with static binding, one can guarantee that each processor can be utilized at least \( \ln 2 \) (69%), the least upper bound of schedulable utilization for the rate-monotonic algorithm. Such an acceptable guarantee does not seem to be possible with dynamic binding, and sophisticated run-time evaluations may be necessary to avoid pathological cases like the one above. In real-time systems, one cannot normally perform potentially combinatorial analysis during run-time, and static binding is preferable. As a result, we assume that tasks are statically bound to processors.

Consider for a moment that tasks are independent and that they do not share any resource. Static binding then implies that in order to determine the schedulability of a task \( T \) on a processor \( P \), we need only consider its own computational requirements and its preemptions by higher priority tasks bound to processor \( P \). That is, given a static binding of tasks to processors, the problem of analyzing the schedulability of the \( m \) processors decomposes into \( m \) uniprocessor problems. When only tasks bound to the same processor share resources, the priority ceiling protocol can be used effectively on each processor.

#### 3.3. The Concept of Remote Blocking

Synchronization requirements can introduce much longer blocking delays when tasks execute in a multiple processor environment. Clearly, the concept of blocking delays on uniprocessors must now be extended to include delays caused by resource-sharing with tasks on remote processors. Assume that tasks are statically bound to processors. On any given processor \( P \), a task can be preempted by higher priority tasks on \( P \), blocked by lower priority tasks on \( P \), and in addition wait for tasks of any priority on remote processors to release required global resources. The determination of whether a task meets its deadlines must, therefore, consider not only preemption and blocking on its local processor but also the waiting time introduced by tasks of any priority on all remote processors. We shall refer to the latter as remote blocking. Now, remote blocking must also be minimized by a priority-based synchronization protocol. Let us now consider the following examples.

**Example 1:** Consider normal prioritized execution without priority inheritance in effect. Suppose that task \( T_1 \) is bound to processor \( P_1 \), and that tasks \( T_2 \) and \( T_3 \) are bound to processor \( P_2 \). As shown in Figure 3-1, 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 expect that \( J_1 \) will be blocked only for the duration of \( J_2 \)'s execution of its critical section. However, if \( J_2 \) preempts \( J_1 \) on processor \( P_2 \), \( J_1 \) will be blocked until \( J_1 \) 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.

With priority inheritance in effect, \( J_2 \) will inherit \( J_1 \)'s highest priority and complete its critical section without being preempted by \( J_2 \) or other intermediate priority jobs. Once \( J_2 \) releases \( S \), \( J_1 \) would be able to resume. However, consider the following example.

**Example 2:** Suppose that, as shown in Figure 3-2, tasks \( T_1 \) and \( T_2 \) are bound to processor \( P_1 \), and that task \( T_3 \) is bound to processor \( P_2 \). Job \( J_3 \) executes on processor \( P_2 \) and attempts to lock semaphore \( S \). But \( S \)
is currently locked by job J₂ executing on processor P₂. However, if J₁ now preempts J₂ on processor P₂, J₂ will be blocked until J₁ completes (or blocks). Thus, the blocking time of J₂ will continue until arriving higher priority jobs on P₂ (such as J₁ and other higher priority jobs) complete execution or suspend themselves. Since τ₁, for instance, is a periodic task, the blocking duration of J₂ can be extremely long.

In Example 2, even the enforcement of priority inheritance does not force any changes in the event sequence, and the blocking duration of J₂ 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 J₂ can be a function of the entire execution time of job J₁. 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 additionally compared to the situation where no semaphores are present. In the absence of any data-sharing, J₂ would not have to wait at all, and its blocking duration becomes a function of execution times of tasks on other processors. This situation cannot be improved by the direct use of the priority ceiling protocol.

A multiprocessor synchronization protocol must bound 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 remote job has to be blocked while a local job executes outside a critical section. For instance, in Example 2, job J₂ should not be allowed to execute outside its critical section while job J₁ is blocked waiting for it to complete. The motivation behind this goal is the observation that a critical section is short relative to task execution time. An implicit feature of the uniprocessor priority inheritance protocols is that whenever possible, a lower priority job always waits for a higher priority job. This is a direct result of the priority-based scheduling methodology, and must be adhered to in the multiple processor environment as well. For example, if two jobs J₃ and J₄ are waiting for the release of a shared resource, the higher priority job J₄ will be allowed to access the resource first even if J₃ 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 τ, 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 reflected in our prioritized queues on the semaphores accessed by real-time tasks.

It is worthwhile to recall that in a uniprocessor, the blocking duration \( B \) of a task includes only the time being blocked by lower priority jobs. However, when synchronization with tasks on remote processors is introduced, any blocking caused by remote tasks should also be considered as part of \( B \).

In summary, our objective is to minimize the schedulability loss due to blocking. The primary goal is that the worst-case blocking duration \( B \) of a job should be a function of critical section durations only, and the secondary goal is to minimize the impact of \( B \) by letting a lower priority job rather than a higher priority job experience blocking whenever possible. These two goals may seem conflicting in the sense that the achievement of the first may require that the critical section of a low priority job must execute at higher priority than a high priority job, which is contrary to the second goal. These two goals represent a prioritization of concerns in the following sense. If the first goal is not achieved, blocking times will be too long to be acceptable. Hence, this goal must be achieved prior to the second. Once the first goal has been achieved, the second helps to improve schedulability.

4. Priorities in a Shared Memory Protocol

4.1. Typical Multiprocessor Configuration

A typical example of a multiprocessor configuration in which this protocol can be used is presented in Figure 4-1. We consider a configuration with multiple processors and shared memory modules connected together by a backplane bus. Each processor has its own local memory and a data cache. The local memory in a processor contains the code and local data for all tasks bound to that processor, and the cache is used only to cache globally shared data. The local memory is essential to avoid bus contention for normal task execution which can not only slow down tasks but also lead to unpredictability. As a result, the shared memory is used only for accessing globally shared resources like data structures and devices. We also assume that a hardware mechanism such as bus snooping is used to maintain data coherence.
And, by assumption, all global semaphores shall reside in the shared memory space. 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 (loc). To avoid generating additional box traffic, all local semaphores shall reside in the local memory space of each processor. Note 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, for example, can be used very effectively on each processor. Since nested global critical sections can potentially lead to large increases in blocking durations, we shall assume in this section that global critical sections cannot nest other critical sections or be nested inside other critical sections. We shall consider the effects of nested global critical sections in Section 5.1.

4.3. Accessing Globally Shared Resources

Globally shared resources can be a significant source of blocking since multiple tasks allocated to different processors can attempt to access them simultaneously. A primary goal of the shared memory synchronization protocol is that the blocking duration of a task should only be a function of critical section execution times. As a result, a gcs may need to execute at a priority higher than any non-critical section code. Such a requirement arises from the following theorem.

**Theorem 2:** The remote blocking time of a job blocked on a gcs will be a function of critical sections if and only if the gcs cannot be preempted by jobs that are executing outside critical sections.

**Proof:** First, if a gcs cannot be preempted by non-critical section code, then the waiting time for a task to enter a 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 a remote processor blocked by this gcs will wait for the execution time of J. The Theorem follows.

4.4. Global and Local Priority Ceilings

Under the priority ceiling protocol, the priority ceiling of a semaphore S is the maximum priority at which a critical section guarded by this semaphore can execute.

**Notation:** The highest priority assigned to any task in the entire system is denoted by $P_{\text{if}}$

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 every local semaphore will be less than or equal to $P_{\text{if}}$. However, the same is not true of global critical sections. Suppose that a job $J$ on processor $P_i$ is waiting for a global semaphore $S_g$ held by job $J$ on processor $P_j$. Then, from Theorem 2, the gcs of $J$ must be allowed to execute at higher priority than any task executing non-critical sections on processor $P_j$. The priority of non-critical sections corresponds to the assigned priority of a task. As a result, a gcs on a remote processor can execute at a priority higher than any assigned task priority on $P_j$. In order to use uniform priority values on every processor for the sake of convenience, we can assume the following. Whenever a gcs needs to execute at higher priority than assigned task priorities, it executes at a higher priority than $P_{\text{if}}$.

Since the priority ceiling of a semaphore $S$ is the highest priority at which a critical section guarded by $S$ can execute, a global semaphore has to be assigned a priority ceiling greater than $P_{\text{if}}$. This 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 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 $S_i$ be denoted by $P_{S_i}$. Then, the priority ceiling of a global semaphore $S_g$ is defined such that:

- The priority ceiling of $S_g$ is greater than $P_{\text{if}}$

- If $S_g$ and $S_i$ are global semaphores and $P_{S_i} > P_{S_g}$, then the priority ceiling of $S_g$ is greater than the priority ceiling of $S_i$.

This set of conditions can be met, for example, as follows. Define the base priority ceiling $P_{\text{if}}$ of a global semaphore as a fixed priority greater than $P_{\text{if}}$. i.e. $P_{\text{if}} > P_{\text{if}}$. Then, the priority ceiling of a global semaphore $P_{S_g}$ can be given by $P_{\text{if}} + P_{\text{if}}$. This assignment maintains the ordering of the priority ceilings and ensures that the priority ceilings of all global semaphores are higher than $P_{\text{if}}$.

4.4. The Execution Priority of a GCS

In this section, we justify the need to execute any gcs at a fixed priority and then determine this priority. The priority of any job $J_i$'s gcs must be assigned such that it meets the following requirements. The blocking duration of any job must only be a function of critical section durations. In addition, if $J_i$ blocks $J_j$, priority must include $J_j$'s priority for the duration of blocking. We refer to these requirements as the gcs priority requirements.

By Theorem 2, a job within a gcs has to execute at a priority higher than $P_{\text{if}}$ only if it blocks a remote job. However, this dynamic priority change can be costly to implement in a multiple processor environment. Let us, therefore, assume that the gcs of a job $J_i$ always executes at a priority higher than $P_{\text{if}}$, such that blocking durations can only be a function of critical section durations. However, the priority inheritance policy requires that the gcs of a job still inherit the highest priority of jobs that it blocks. Again, this requires dynamic priority changes. Hence, it is desirable both from implementation and overhead points of view that a gcs is executed at a fixed priority. However, the benefits of priority inheritance must be still obtained. We therefore enforce the following priority assignment rule. The priority of a gcs of a job $J_i$ guarded by $S_g$ is at least $P_{S_g}$ and at most the global priority ceiling of $S_g$.

On a uniprocessor, a critical section can always be executed at a priority level equal to the priority ceiling of its associated semaphore. This is a good approximation of the priority ceiling protocol [9]. In the case of the synchronization protocol for message-based systems [8], every critical section guarded by $S$ can again possibly execute at a priority equal to the priority ceiling of $S$. In fact, it is suggested that a gcs guarded by a global semaphore $S_g$ always execute at a priority equal to the global priority ceiling of $S_g$ (8). However, in our case, not every gcs guarded by $S_g$ needs to execute at the priority ceiling of $S_g$.

Suppose that job $J_i$ is bound to $P_i$. Let $P_i$ be the highest priority job on processors other than $P_i$ that can lock $S_g$. Then, assign the fixed priority $P_{S_g} + P_i$ to a gcs of job $J_i$ guarded by $S_g$. This priority assignment satisfies the gcs priority requirements. First, it guarantees that every gcs executes at least at a priority of $P_{S_g} + P_i$. Therefore, by Theorem 2, blocking durations can only be a function of critical section durations and the first requirement is met. Next, suppose that a job $J_i$ on a processor other than $P_i$ blocks on $S_g$ held by $J_i$. The priority inheritance policy requires that $J_i$'s gcs inherit $J_i$'s priority. In general, $J_i$ must inherit the highest priority of the gcs's blocked by $J_i$. The remote job with the highest priority that can block on $S_g$ is $J_i$. Hence, the priority assigned to a gcs is given by $P_{S_g} + P_i$.

If a gcs always executes at its specified priority level, its priority need not be changed dynamically since priority inheritance to the highest possible level occurs as soon as the critical section is entered. Another important reason for this gcs priority assignment is as follows. The worst-case blocking time obtained with the priority ceiling protocol (where priority is inherited only after blocking occurs) is typically identical to that obtained by always executing critical sections at the maximum possible priority value. Guaranteeing the deadlines of real-time tasks requires the worst-case conditions to be met. As a result, our priority assignment scheme incurs no additional penalties, and a cheaper implementation becomes possible.

We illustrate our terminology and the concepts of global priority ceiling
and normal execution priority of a gcs in the following example. The binding of tasks and semaphores to processors is illustrated in Figure 4-2.

Example 3: Consider the 3-processor configuration shown in Figure 4-2. Tasks $\tau_1$ and $\tau_2$ are bound to processor $\varnothing_1$, tasks $\tau_3$ and $\tau_4$ to processor $\varnothing_2$, and tasks $\tau_5$, $\tau_6$, and $\tau_7$ to processor $\varnothing_3$. Jobs $J_i$ through $J_7$ execute the following sequence of steps:

1. Jobs $J_1$ and $J_2$ execute the following sequence of steps:
   - $J_1 = \text{,...,P}(S_1), \text{,...,P}(S_2), \text{,...,P}(S_3), \text{,...,P}(S_4), \text{,...,P}(S_5)$
   - $J_2 = \text{,...,P}(S_1), \text{,...,P}(S_2), \text{,...,P}(S_3), \text{,...,P}(S_4), \text{,...,P}(S_5)$
2. Jobs $J_1$ and $J_2$ execute the following sequence of steps:
   - $J_1 = \text{,...,P}(S_1), \text{,...,P}(S_2), \text{,...,P}(S_3), \text{,...,P}(S_4), \text{,...,P}(S_5)$
   - $J_2 = \text{,...,P}(S_1), \text{,...,P}(S_2), \text{,...,P}(S_3), \text{,...,P}(S_4), \text{,...,P}(S_5)$
3. Jobs $J_1$ and $J_2$ execute the following sequence of steps:
   - $J_1 = \text{,...,P}(S_1), \text{,...,P}(S_2), \text{,...,P}(S_3), \text{,...,P}(S_4), \text{,...,P}(S_5)$
   - $J_2 = \text{,...,P}(S_1), \text{,...,P}(S_2), \text{,...,P}(S_3), \text{,...,P}(S_4), \text{,...,P}(S_5)$
4. Jobs $J_1$ and $J_2$ execute the following sequence of steps:
   - $J_1 = \text{,...,P}(S_1), \text{,...,P}(S_2), \text{,...,P}(S_3), \text{,...,P}(S_4), \text{,...,P}(S_5)$
   - $J_2 = \text{,...,P}(S_1), \text{,...,P}(S_2), \text{,...,P}(S_3), \text{,...,P}(S_4), \text{,...,P}(S_5)$
5. Jobs $J_1$ and $J_2$ execute the following sequence of steps:
   - $J_1 = \text{,...,P}(S_1), \text{,...,P}(S_2), \text{,...,P}(S_3), \text{,...,P}(S_4), \text{,...,P}(S_5)$
   - $J_2 = \text{,...,P}(S_1), \text{,...,P}(S_2), \text{,...,P}(S_3), \text{,...,P}(S_4), \text{,...,P}(S_5)$

Semaphore $S_1$ is a local semaphore on $\varnothing_1$, there are no local semaphores on $\varnothing_2$ and $\varnothing_3$ are local semaphores on $\varnothing_3$. In addition, there are 2 global semaphores. Since the semaphores $S_0$ and $S_1$ can be held by jobs bound to different processors, these global semaphores must reside in shared memory. Define the base priority ceiling $P(J)$ for each job $J$.

5. Definition of the Shared Memory Protocol

The shared memory synchronization protocol is an extension of the uniprocessor priority ceiling protocol in that if the number of processors in the system is $P$, then the priority reduces to the priority ceiling protocol. The protocol is used on each processor in the system. Recall that we assume that there are no nested or overlapping global critical sections. Under this protocol, resources guarded by local semaphores are accessed using the uniprocessor priority ceiling protocol locally on each processor, and globally shared resources are accessed using read-modify-write instructions to obtain global semaphores. In addition, all global critical sections are executed at their predefined priority levels. The protocol is as follows.

1. A job uses its assigned priority unless it is within a critical section.
2. The uniprocessor priority ceiling protocol is used for all requests to local semaphores. This means the following: (a) When a job $J$ requests the local semaphore $S$ on processor $\varnothing$, let $S'$ be the semaphore with the highest priority ceiling of all local semaphores locked by jobs other than $J$ on processor $\varnothing$. (b) A job $J$ on a processor $\varnothing$ can obtain the local semaphore $S$ if its priority is higher than the priority ceiling of $S'$. Else, $J$ is said to be blocked by $S'$. In this case, $J$ inherits $S'$ priority until it releases $S'$. (c) A job $J$ can preempt another job $J'$ if its priority is higher than the priority of $J'$, assigned or inherited, at which job $J'$ is executing.
3. A job $J$ within a global critical section guarded by the global semaphore $S_0$ has the priority assigned to its gcs.
4. A job $J$ within a gcs can preempt another job $J'$ within a gcs if the assigned priority of $J$'s gcs is greater than that of $J'$'s gcs.
5. When a job $J$ requests a global semaphore $S_0$, $S_0$ can be granted to $J$ by means of an atomic transaction on shared memory, if $S_0$ is not currently held by another job.
6. If a request for a global semaphore $S_0$ cannot be granted, the job $J$ is added to a prioritized queue on $S_0$ without being preempted (i.e., $J$ holds the processor until it is inserted into the queue). The priority assigned to $J$ as the key for queue insertion is the normal priority assigned to $J$.
7. When a job $J$ attempts to release a global semaphore $S_0$, the highest priority job $J'_{\text{bus}}$ waiting for $S_0$ is signaled and becomes eligible for execution at $J'_{\text{bus}}$'s host processor at its gcs priority. If there is no job suspended on $S_0$, the semaphore is released.

We illustrate the shared memory synchronization protocol using the task set presented in Example 3.

Example 4: The system configuration of the task set is shown in Figure 5-1. Consider the following sequence of events shown in Figure 5-1:

- At $t=0$, jobs $J_1$ and $J_2$ are the only jobs eligible for execution on processors $\varnothing_1$ and $\varnothing_2$ respectively, and begin execution. Processor $\varnothing_3$ remains idle.
- At $t=1$, $J_1$ locks global semaphore $S_0$ on $\varnothing_1$ and begins execution at priority $P_1$. $J_2$ begins execution at priority $P_2$ on $\varnothing_2$. $J_3$ locks local semaphore $S_2$ (since there are no other locked local semaphores).
- At $t=2$, $J_1$ arrives but is unable to preempt $J_2$ executing its gcs. Job $J_2$ continues execution on $\varnothing_2$ until $t=5$. Job $J_2$ arrives at $t=3$ and immediately preempts $J_2$ executing within its gcs.
- At $t=3$, $J_2$ releases the global semaphore $S_0$ (since no other job is pending on the semaphore) and regains its original priority. As a result, $J_2$ preempts $J_2$ and begins execution on $\varnothing_1$. On $\varnothing_2$, $J_3$ attempts to lock the local semaphore $S_2$ but finds that its priority is not greater than the priority ceiling of the semaphore $S_2$ already locked by $J_2$. Hence, $J_3$ blocks, and $J_2$ resumes execution at the inherited priority of job $J_2$. 

---

Table 4-1: The Priority Ceilings of Semaphores in Example 3

<table>
<thead>
<tr>
<th>Semaphore</th>
<th>Priority Ceiling</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_0$ (local)</td>
<td>$P_1$</td>
</tr>
<tr>
<td>$S_1$ (local)</td>
<td>$P_2$</td>
</tr>
<tr>
<td>$S_3$ (local)</td>
<td>$P_3$</td>
</tr>
<tr>
<td>$S_0$ (global)</td>
<td>$P_0 + P_1$</td>
</tr>
<tr>
<td>$S_1$ (global)</td>
<td>$P_0 + P_2$</td>
</tr>
</tbody>
</table>

Table 4-2: Critical Section Priorities in Example 3

<table>
<thead>
<tr>
<th>Job</th>
<th>Critical Section Guarded by</th>
<th>Normal Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>$J_1$</td>
<td>$S_0$</td>
<td>$P_0$</td>
</tr>
<tr>
<td>$J_2$</td>
<td>$S_0$</td>
<td>$P_0 + P_1$</td>
</tr>
<tr>
<td>$J_3$</td>
<td>$S_1$</td>
<td>$P_2$</td>
</tr>
<tr>
<td>$J_4$</td>
<td>$S_1$</td>
<td>$P_0 + P_2$</td>
</tr>
<tr>
<td>$J_5$</td>
<td>$S_0$</td>
<td>$P_0$</td>
</tr>
<tr>
<td>$J_6$</td>
<td>$S_1$</td>
<td>$P_0$</td>
</tr>
<tr>
<td>$J_7$</td>
<td>$S_0$</td>
<td>$P_0 + P_1$</td>
</tr>
</tbody>
</table>

120
At J4, high priority and preempts releases at execution priority of J4 begins execution attempts to lock releases but is unable to preempt Hence, and are queued in priority order which releases 1=4, J5 locks global semaphore Blocked by local Preempted by global critical section of lower priority task Execution priority of the other. An instance of this occurs at t=7 on Job J4 is preempted by a higher priority gcs on a global semaphore, a lower priority job can lock a local semaphore on J4. If this semaphore is later required or has a sufficiently high priority ceiling, J4 can be blocked by means of this semaphore later. Similarly, while J7 is blocked on a global semaphore, a lower priority job can execute on the processor as shown at t=5 on Job J6 can resume, lock and release S3 and then complete on Job J2. Example 4 illustrates the following characteristics of the shared memory synchronization protocol. Jobs request and release local semaphores by using the uniprocessor priority ceiling protocol only on the local semaphores of each processor. Global semaphore requests do not follow any special protocol except that global critical sections execute at preassigned high priorities, and semaphore queues are priority-ordered. Any gcs executes at higher priority than all non-gcs code. Examples occur at t=2 on p3, and t=9 on p2. However, the gcs of one job can preempt the gcs of another if the execution priority of one gcs is higher than the execution priority of the other. An instance of this occurs at t=7 on p2. Jobs suspended on a semaphore are signaled in priority order as at t=11. When a higher priority job suspends on a global semaphore, a lower priority job can execute on the processor. In this section, we shall compute the worst-case waiting interval. However, both approaches can cause processor cycles to be lost which is generally unacceptable.

5.1. Determination of Task Waiting Times
In this section, we shall compute the worst-case waiting times that a job has to encounter before completion. The fundamental objective of the shared memory synchronization protocol is to obtain bounded waiting times for real-time tasks using shared memory primitives to access shared resources. The bounded waiting times in turn can be used to determine whether a set of real-time tasks running on a shared memory multiprocessor can meet their deadlines.

It can be easily seen that the shared memory synchronization protocol prevents deadlocks. The uniprocessor priority ceiling protocol avoids deadlocks on local shared resources on each processor. In addition, by assumption, global critical sections do not nest other critical sections and versa. The result follows.

Remark: If nested global critical sections are used, explicit partial ordering of global resources must be used to prevent deadlocks. When a job J4 bound to a processor p3 requests a global semaphore S3, it can be held by a lower priority job J2 on a different processor p2. However, the gcs of J2 can be preempted by a higher priority gcs on p2. As a result, J2 can be blocked until this higher priority gcs completes on p2, enabling J4 to resume and release S3. Such a processor p2, j x r is called a blocking processor for J4. In other words, a gcs on p2 is assigned a higher priority than another gcs on p3, guarded by S3, and S3 will be requested by J4 on p3.

In addition, we shall also use the following notation.

---

**Figure 5.1:** Sequence of events described in Example 4.

- At t=4, J1 locks the global semaphore S0, on p4. On p2, J2 arrives but is unable to preemp t J2 which continues execution. On p2, J2 releases S3 and regains its low priority. Job J2 preempts J4 and is now able to lock S2.
- At t=5, J1 continues execution of its gcs on p4 until t=7. On p2, J2 attempts to lock S3 but is blocked since J4 holds the semaphore. Hence, J2 begins execution. On p2, J4 arrives but is unable to preempt J2 which releases S2 and continues execution.
- At t=6, J2 is still blocked on p2. Job J2 locks the global semaphore S0 and begins execution of its gcs at priority P_H + P_S (See Table 4-2). On p2, J4 attempts to lock S3 and is blocked. As a result, J3 begins execution on p2. Currently, J3 and J4 are both blocked on S0 and are queued in priority order on S0.
- At t=7, J1 performs a V() operation on S0, and grants the semaphore to J3, the highest priority job pending on it. Then, J1 continues execution on p1. On p2, J4 is eligible to execute its gcs with an execution priority of P_H + P_S, which is higher than J4's gcs execution priority of P_H + P_S. Hence, J4 preempts J3 and begins execution of its gcs. On p2, J4 locks local semaphore S2.
- At t=8, J3 locks local semaphore S0 on p3. On p2, J4 needs to release S0 and grants the semaphore to J4. Job J4 also regains its normal priority and is preempted by the gcs of J4. On p2, J4 enters its gcs at a high priority and preempts J4 on p3.
- At t=9, J4 releases S1 and continues execution on p3. On p2, J4 releases S3, regains its lowest priority and is preempted by J2. On p2, J4 releases S2, regains its original priority and continues execution.
- At t=10, J3 completes execution on p3 and J4 resumes execution. On p2, J4 locks global semaphore S0 and assumes its gcs priority. On p2, J4 attempts to lock S3 and finds that its priority is higher than the priority ceiling of locked local semaphore S2. Hence, J4 is granted the semaphore and continues execution.

---

121
NCik: the Number of Global critical sections that a job Ik executes before its completion.

NHiJk: The Number of jobs with lower priority than Ik on the processor pJ.

(Jk): The set of jobs on Ik's blocking processor pJ with gcs having higher priority than Ik can directly block Ik.

NH(Jk): The Number of gcs's of a job Ik ∈ (Jk) with higher priority than a gcs on pJ, which can directly block Ik.

(GSik): The set of Global Semaphores each of which will be locked by both job Ik and guarding elements of (GSik).

NCik: The Number of critical sections entered by Ik and guarded by elements of (GSik).

The waiting time of a job Ik on processor pJ (in addition to preemption by higher priority jobs on pJ) consists of several factors. We enumerate and bound each of these blocking factors below.

1. Each time Ik needs a global semaphore, it can potentially suspend, letting lower priority jobs execute on pJ. During this suspension, lower priority jobs can lock local semaphores. This locking can contribute to additional blocking to Ik when it resumes execution on pJ.

At most NCik local critical sections of lower priority jobs can block a job Ik bound to a processor pJ. This follows from Theorem 1 and the fact that a job Ik can suspend itself when it attempts to enter a global critical section.

2. Each time Ik attempts to enter a gcs guarded by J0, it can find that J0 is currently held by a lower priority job.

At most NCik global critical sections of lower priority jobs can block a job Ik bound to a processor pJ. This follows from the fact that semaphore queues are priority-ordered.

3. Whenever Ik attempts to lock global semaphores, it can always be preceded by higher priority jobs requesting the same semaphores. Global critical sections of higher priority jobs bound to pJ itself can be considered to be normal preemption intervals for Ik.

At most NCik * [TfIk] global critical sections of a higher priority job Ik bound to pJ (or pJ) can blockIk on pJ. This follows directly from the fact that the number of instances of Ik, within Ik's period is bounded by [TfIk], and each of Ik's NCik gcs executions can execute before Ik is granted a global semaphore. In a sense, this blocking factor can be considered to be the remote preemption penalty for a job. That is, higher priority jobs on remote processors which need the same resources can preempt Ik, and Ik needs to have sufficient slack to take these preemptions.

4. Consider the set of blocking processors of Ik. Each of these processors contains higher priority gcs's which can preempt the gcs's of lower priority jobs directly blocking Ik.

For every Ik ∈ (Jk) on every blocking processor pJ, at most NH(Jk) * [TfIk] gcs's can block Ik. This again follows from the argument that there can only be [TfIk] instances of Ik during one period of Ik.

5. Each time Ik needs a global semaphore, it can potentially suspend, letting lower priority jobs execute on pJ. During this suspension, lower priority jobs can lock or queue up on global semaphores. These global critical sections can execute at higher priority than Pk and can therefore preempt Ik when it executes non-gcs code.

At most max(NCik + 1, 2NHiJk) global critical sections of every job Ik with lower priority than Ik on processor pJ can block Ik. This is due to the following. When Ik begins execution on pJ, a lower priority job Jk can have an outstanding request for a global semaphore. When this global semaphore is granted, Ik assumes higher priority than Ik within its gcs and can preempt Ik. This situation can repeat each time Ik suspends on a global semaphore. Hence, Ik can be blocked for at most NCik + 1 gcs's of Ik. In addition, since the period of a higher priority job Ik is shorter than the period of a lower priority job Jk, (i.e., there can be at most two executions of Ik within Jk's period. As a result, Ik can enter at most 2NCik gcs's to block Ik within its period. The bound follows.

This blocking factor can be rather large but can be expected not to occur in practice. Each job needs to execute non-critical section code before entering new critical sections. As a result, all critical sections of the lower priority jobs will be entered/requested during Ik's suspension(s). Thus, by considering actual time segments of execution of lower priority job, this blocking factor can be considerably reduced. In addition, there is some overlap between this blocking factor and the first two blocking factors above. For example, if a lower priority job Ik is within an i.e., it cannot be within a gcs to block Ik. Also, the gcs's of lower priority jobs on Ik's host processor pJ considered in this blocking factor need not be considered for inclusion in blocking factor 2.

The shared memory protocol does not change when global critical sections are nested inside one another. However, deadlocks need to be explicitly avoided, say, by imposing a partial ordering of resources. The blocking durations with nested critical sections can then become a major schedulability bottleneck. Also, the blocking factors for the protocol listed above are true only for non-nested global critical sections. This is because each nested global critical section not only increments NCik but can also rapidly increase the number of blocking processors via a transitive relationship. For instance, a job Jk holding a global semaphore on one processor can block on a nested semaphore held on another processor by another job, which in turn is blocked on another nested semaphore and so on. As a result, the list of blocking processors for the first job can include the list for the second job etc. In general, blocking factors 2, 3 and 4 scale up with nested critical sections. Another possible approach to analyze nested gcs's is to collapse nested critical sections into non-nested gcs's. This can be done by introducing semaphores which subsume the nested semaphores. For instance, a database transaction requiring two distinct objects may obtain two locks. Instead, a lock which provides access to both objects can be introduced. This is analogous to locking a larger section of the database for each transaction. The analysis of this situation is identical to the one above for non-nested critical sections.

Let Bi represent the worst-case waiting time that a job of a task τi can encounter. The term Bi for task τi under the shared memory protocol is given by the sum of two factors:

- The 5 blocking factors derived in Section 5.1.
- The schedulability analysis of the rate-monotonic scheduling algorithm [6] assumes that a periodic task arrives at periodic intervals and executes from initiation to completion without suspending itself. However, under the shared memory synchronization protocol, a job can block on a global semaphore and suspend itself. This results in a phenomenon called deferred execution, where each lower priority job, in the worst-case, can encounter an additional preemption from a higher priority job [5, 8]. This situation introduces a scheduling penalty which increases the magnitude of Bi.

5.2. Comparison of Multiple Processor Synchronization Protocols

Under the message-based synchronization protocol described in [8], all global critical sections guarded by a global semaphore Sg must be executed on the same specified processor. The first 3 blocking factors for shared memory synchronization protocol have their identical counterparts under the message-based synchronization protocol as well. The fourth blocking factor can be reduced in the message-based synchronization protocol by adding more synchronization processors, but the shared memory protocol can use these extra processors as additional processing resources. In addition, under the message protocol, all gcs's execute at the priorities equal to the global semaphore priority ceilings. Hence, the lesser blocking of the message-based protocol can be partially offset by the potentially lower assigned priorities to gcs's under the shared memory protocol.

The principal advantage of the message-based protocol is that it allows gcs's to be nested as long as locks do not cross processor boundaries. This
becomes possible because all gcs's guarded by the same semaphore are bound to the same processor. Under the shared memory protocol, the blocking factors rise rapidly when global critical sections are nested. This disadvantage of the shared memory protocol has to be weighed against its higher implementation efficiency in tightly coupled multiprocessors in contrast to the large overhead inherent in the message-passing protocol where every gcs of a job is generally executed in a remote processor.

5.3. Schedulability Analysis

Once the worst-case waiting time of a job \( J \) has been determined, the schedulability analysis of the multiprocessor system can proceed on a processor-by-processor basis. On each processor, we now have a set of periodic tasks each of which can be preempted by higher priority tasks and be blocked by the critical sections of lower priority tasks on the host processor and any task on other processors. The following theorem from [10] can be used to determine the schedulability of each processor.

**Theorem 3:** A set of \( n \) periodic tasks can be scheduled by the rate-monotonic algorithm if the following conditions are satisfied [10]:

\[
\forall i, 1 \leq i \leq n, \quad C_i \leq \frac{1}{1 - \sum_{j=1}^{n} S_j}
\]

\( S_j \) is the number of tasks bound to \( S_j \), and \( C_i \) and \( T_i \) are the computation time and the period, respectively, of each task bound to \( S_j \). The term \( B_i \) is the maximum duration that a job \( j \) bound to \( S_j \) can be blocked from initiation to completion as presented in Section 5.1.

5.4. Implementation Considerations

The local priority ceiling protocol can be implemented as suggested in [10]. The principal requirements of the shared memory synchronization protocol lie at the entry and exit points to global critical sections. Associated with each global semaphore is also a data structure in shared memory which represents a priority-ordered queue of tasks that suspend on the semaphore. This queue represents by itself another globally shared data structure, and can be guarded by a user-transparent semaphore. Any periodic task waiting on this semaphore queue must be atomic as well. Tasks, enforced as follows. Before a task enters a global critical section, it must first obtain the global semaphore \( S_g \) associated with the gcs. This can be done by performing an atomic read-modify-write on \( S_g \) located in the shared memory space. If the PO operation is successful, no further operations need be carried out. Else, if \( S_g \) is currently held by another task, the requesting task performs a busy-wait on the user-transparent semaphore \( S_x \) guarding the priority queue on \( S_g \) and then queues itself. To avoid the generation of excessive traffic on the backplane during any busy-wait, the task spins the other's cache entry until \( S_x \) is released [2]. A task releasing a global semaphore first obtains the semaphore \( S_x \) (with a busy-wait if necessary), picks the highest priority task off the queue (if any), awakens the task and transfers to it the lock on \( S_g \). If there are no tasks pending, it releases \( S_g \) and then \( S_x \). Also, a task is non-preemptible while it performs a busy-wait on the semaphore. This is such that task awakening and queuing are not unnecessarily postponed. The duration of busy-wait should be relatively short since it represents only the duration of adding an entry to (or deleting an entry from) a linked list.

The system cache and the cache-coherence scheme are used only to avoid generating unnecessary bus traffic while performing a busy-wait on locks to semaphore queues. If an interprocessor interrupt mechanism is available, the cache, the cache coherence requirement and the busy-wait situation can be replaced. When a job needs to busy-wait, it does not relinquish the processor but records its status in a field within the lock and disables all interrupts except interprocessor interrupts. The job and the processor suspend until an interprocessor interrupt arrives [2].

6. Conclusions

Resource sharing in a multiprocessor environment can lead to unbounded waiting time for accessing global resources. Hence, it is important that the waiting time to access a global resource is bounded and sufficiently short. In addition, the synchronization protocol must not incur considerable overhead, and must therefore utilize primitives that can be efficiently implemented in the underlying system. The priority ceiling protocol [10] and the message-based multiprocessor priority ceiling protocol [8] are priority-based synchronization protocols which bound the waiting time for resource accesses on uniprocessors and loosely coupled multiprocessors respectively. In this paper, we have defined a synchronization protocol for use in tightly-coupled multiprocessors where shared memory transactions can be implemented very efficiently. The properties of this protocol allow us to derive a set of sufficient conditions to determine whether real-time tasks using this protocol can meet their timing constraints. An underlying assumption of the protocol is that tasks are statically bound to processors. A task allocation scheme that can be used would be similar to that proposed in [8] and would attempt to allocate tasks with a high degree of resource sharing to the same processor(s). Since the task allocation is determined offline, the complexity of the allocation algorithm need not be a dominating factor in order to achieve a schedulable configuration with a small number of processors. Finally, we have compared its merits and demerits with respect to the message-based synchronization protocol. Variations to the protocol are possible. For example, the shared memory and message-based protocols can be mixed to reduce critical blocking factors and/or support nested critical sections.

**Acknowledgements**

The author would like to thank Lui Sha and Jane Liu for their comments.

**References**


123