An Improved Priority Ceiling Protocol to Reduce Context Switches in Task Synchronization[1]

Albert M.K. Cheng and Fan Jiang

Computer Science Department
University of Houston
Houston, TX, 77204, USA
http://www.cs.uh.edu

Technical Report Number UH-CS-05-23
November 5, 2005

Keywords: concurrent programming, context-switching, task synchronization, Priority Inheritance Protocol, Priority Ceiling Protocol

Abstract

Context switching between concurrent tasks is a pure operating system overhead which wastes CPU cycles. This paper describes a technique to reduce the number of context switches which are caused by task synchronization. Our method disallows a higher-priority task to preemptively seize the CPU if it will be blocked by a lower-priority task in the future. This protocol is applicable to real-time systems that use preemptive priority scheduling with binary semaphores to enforce mutual exclusion. We show that this protocol does not affect task completion times and so is especially suitable for synchronization in real-time systems where meeting tasks' timing constraints is more

[1] This material is based upon work supported in part by the National Science Foundation under Award Nos. CCR-9111563 and IRI-9526004, by the Texas Advanced Research Program under Grant No. 3652270, and by a grant from the University of Houston Institute of Space Systems Operations. This paper is an extended and refined version of a preliminary and shorter paper [1] presented at IEEE IPDPS 2001.

important than other factors such as task response time or throughput. This protocol can be combined with priority inheritance protocols to bound the duration of priority inversion while having the number of context switches reduced. Our simulation shows that about 10% to 20% of context switches can be avoided using our preemption protocol.

*This material is based upon work supported in part by the National Science Foundation under Award Nos. CCR-9111563 and IRI-9526004, by the Texas Advanced Research Program under Grant No. 3652270,and by a grant from the University of Houston Institute of Space Systems Operations. This paper is an extended and refined version of a preliminary and shorter paper [1] presented at IEEE IPDPS 2001.

An Improved Priority Ceiling Protocol to Reduce Context Switches in Task Synchronization*

Albert M.K. Cheng and Fan Jiang

Abstract

Context switching between concurrent tasks is a pure operating system overhead which wastes CPU cycles. This paper describes a technique to reduce the number of context switches which are caused by task synchronization. Our method disallows a higher-priority task to preemptively seize the CPU if it will be blocked by a lower-priority task in the future. This protocol is applicable to real-time systems that use preemptive priority scheduling with binary semaphores to enforce mutual exclusion. We show that this protocol does not affect task completion times and so is especially suitable for synchronization in real-time systems where meeting tasks' timing constraints is more important than other factors such as task response time or throughput. This protocol can be combined with priority inheritance protocols to bound the duration of priority inversion while having the number of context switches reduced. Our simulation shows that about 10% to 20% of context switches can be avoided using our preemption protocol.

Index Terms

concurrent programming, context-switching, task synchronization, Priority Inheritance Protocol, Priority Ceiling Protocol.

I. Introduction

In a real-time system, there are rigid time requirements on the operation of a processor or the flow of data. The correctness of computation depends on not only the results of computation but also the time at which outputs are generated. Processing done without obeying the defined time constraints is considered system failure. Thus, real-time systems are often used to monitor or control certain dedicated operations. For example, the controllers for the anti-breaking system and airbag in a car must operate and react within a short period of time during a life-threatening situation. The monitor for a chemical process also needs to respond in a real-time manner in order to catch every danger signs in a chemical plane. An exploration robot could also be monitored by a real-time system. It is certainly a requirement of such a system to evaluate the current surroundings in real-time in order to stop the robot soon enough before it falls into a ditch.

Modern computer systems allow multiple tasks to be loaded into memory and to be executed concurrently. Such systems are known as *multitasking* systems. CPU scheduling, which is the process of switching the CPU among tasks, is the basis of a multitasking operating system. Over the years, various scheduling algorithms have been developed with emphases on criteria such as CPU utilization, throughput, or task response time. In the area of real-time systems, the objective of CPU scheduling is to give *predictably* fast response to urgent or high-priority tasks. The predictability is usually obtained in terms of tasks meeting their timing requirements. Another measure of merit in real-time system scheduling is a high degree of *schedulability*, or the degree of resource utilization at or below which the timing requirements of tasks can be ensured [5].

A task may share common data with other tasks executing in the system. Concurrent access to shared data may result in data inconsistency. Mechanisms that force tasks to execute in a certain order so that data consistency is maintained are known as synchronization policies (or protocols). In a real-time environment, the need for tasks to synchronize with each other may cause some otherwise schedulable tasks to violate their timing constraints. Therefore, real-time task synchronization policies are developed to achieve data consistency with as little loss in schedulability as possible.

*This material is based upon work supported in part by the National Science Foundation under Award Nos. CCR-9111563 and IRI-9526004, by the Texas Advanced Research Program under Grant No. 3652270,and by a grant from the University of Houston Institute of Space Systems Operations. This paper is an extended and refined version of a preliminary and shorter paper [1] presented at IEEE IPDPS 2001.

In this paper, we consider the problem of reducing the number of context switches in a real-time system with priority-driven preemptive scheduling. Context switches occur whenever one task relinquishes its control of the CPU to the next task in the system. When tasks are required to synchronize with each other, more context switches will occur. For example, before using up its share of CPU time, task *P* may be denied accessing one of its shared resources by task *Q* which is currently using the shared data. Consequently, task *P* has to give up its control of the CPU even if it is more urgent than task *Q* and later resume execution when task *Q* finishes its critical section. Two context switches are caused by this simple synchronization schema. Such preemption may cause undesired high processor utilization, high energy consumption, or in some cases, even infeasibility. This is a pure operating system overhead which is not avoidable.

Our goal in this paper is to reduce the number of context switches caused by task synchronization in a real-time system. Our protocol is based on disallowing a higher-priority task to preemptively seize the CPU if it will be blocked by a lower-priority task in the future. We can apply this simple protocol regardless of the type of system being considered, real-time or non-real-time, thus making the proposed protocol applicable in any operating system to reduce context-switching overhead. Furthermore, the protocol can be incorporated into the family of priority inheritance protocols that are commonly used for real-time task synchronization, accomplishing the same result concerning blocking and deadlocks while reducing the number of context switches. Our simulation showed that about 10% to 20% of context switches can be avoided using our preemption protocol. Other systems that would benefit from our proposed protocol include portable battery-powered devices (such as personal digital assistants (PDAs) and cellular phones), which should continuously conserve power [8], [4], [7], [17] and further reduce operating system overhead.

A. Background

The problem of scheduling periodic tasks with hard deadlines equal to the task periods was first studied by Liu and Layland [5] in 1973. They obtained a necessary and sufficient condition for scheduling a set of independent periodic tasks in a uniprocessor system with preemption allowed. The scheduler employed was also shown to be optimal by Dertouzos [3] for arbitrary task sets (not necessarily periodic).

Task synchronization usually adds another level of difficulty for real-time system scheduling. Mok [6] showed that the problem of deciding whether it is possible to schedule a set of periodic processes is NP-hard when periodic tasks use semaphores to enforce mutual exclusion. In a real-time system with fixed-priority scheduling, the goals of the synchronization protocol include not only maintaining task mutual exclusion, but also bounding the duration of priority inversion, in which a higher-priority task is forced to wait for a lower-priority task to execute. In [13], Sha *et al* investigated two protocols that belong to the family of *priority inheritance protocols* for real-time synchronization: the *Basic Priority Inheritance Protocol* (PIP) and the *Priority Ceiling Protocol* (PCP). They showed that both protocols avoid uncontrolled (or an unpredictable duration of) priority inversion. They also proved that the Priority Ceiling Protocol prevents deadlocks and minimizes the priority inversion duration to at most one critical section. Furthermore, another protocol that applies the method of priority inheritance is introduced in [11] by Rajkumar *et al*, called the *Optimal Mutex Policy* (OMP). This policy is optimal in the sense that it provides the necessary and sufficient condition for limiting the blocking to a single critical section as well as avoiding deadlocks. It is an enhancement of the PCP but requires much more information of the system and the tasks in the system than the PCP does. The focus of this paper is on applying our context switch reduction technique to the PCP and the OMP. Earlier work in this area includes [2], which provides an extension to PCP, and uses early blocking to reduce context switches that are resulting from resource synchronization. Inspired by the work of [2], we extend the theory in such a way that it provides a complete analysis to show the efficacy of early blocking for implementing this technique with arbitrary policies, which are PCP and OMP. The analysis of OMP even shows that it guarantees a better worst-case blocking duration while reducing context switches.

Several prior works dealt with task preemptions. [14] aimed at reducing the number of preemptions in an existing fixed-priority schedule without considering task synchronization. Their method changed the attributes (i.e. priority) of each task instance independently in an off-line schedule to eliminate unnecessary preemptions. [15] used preemption threshold to limit task preemptions. Their focus was on the creation of a general model for both preemptive and non-preemptive scheduling model for a fixed-priority scheduler. Few attentions were placed on reducing the number of task preemptions. [16] used a technique that is similar to the one described in this paper

to reduce the number of expensive requests to lock semaphore. They implemented this technique on the priority inheritance protocol. In this paper we present implementations of this technique on both the priority ceiling protocol and the optimal mutex policy. We also prove the feasibility of this technique on both protocols.

II. General Idea

A preemptive priority scheduling algorithm performs a CPU preemption if the priority of a newly initiated task is higher than that of the currently running task. The lower-priority task has to wait for the higher-priority task to complete its execution before it can run on the CPU again. *Priority inversion* occurs when a higher-priority task is forced to wait for a lower-priority task to finish certain amount of execution. This situation can happen when the two tasks share common data and the lower-priority task gains access to the data first. To ensure the consistency of the shared data, the access must be serialized. Therefore, the higher-priority task has to wait for the lower-priority task to finish its segment of code *C* which uses the shared data (also known as a *critical section*). The higher-priority task is said to be blocked by the critical section *C* of the lower-priority task. If *C* is guarded by a binary semaphore *s*, then we may also say that the lower-priority task blocks the higher-priority one through *s*, and that blocking has occurred.

In the following analysis we assume that the shared data structures are guarded by binary semaphores. A task obtains and releases access to a shared data structure by *locking* and *unlocking* the guarding semaphore, respectively. Two tasks compete for one shared data structure if and only if they will attempt to lock the same semaphore. If the critical sections of a task overlap, we require that the task executes those sections in a first-in-last-out order. That is, overlapping critical sections should be properly nested. Loops in a task's code are assumed to be unrolled by the compiler and replaced by straight-line code. All locks required by a task are also known in advance. We also assume that tasks do not suspend themselves such as for I/O operations. In reality, a task that suspends itself *n* times can be regarded as *n* smaller tasks.

When blocking occurs, the CPU is switched from the higher-priority task to the lower-priority one and later when the lower-priority task leaves its critical section, the CPU is again switched back to the higher-priority one. Therefore, two context switches are caused by every blocking. We can simply avoid the unnecessary context switches by disallowing a higher-priority task to preemptively seize the CPU if it will be blocked by a lower-priority task in the future. The higher-priority task simply behaves in a very astute and patient manner when it foresees the future. Putting this idea in a common sense manner: "since my work will be interrupted by yours later, why don't you finish your interrupting part right now so that I can later proceed smoothly." Thus we have a *general principle* for reducing the number of context switches for priority-driven preemptive scheduling:

> A task *Q* is allowed to preempt the running task *P* on the CPU if and only if the priority of *Q* is higher than the priority of *P* and *Q* will not later be blocked by lower-priority tasks already in the system.

The most common form of blocking is that caused by synchronization of the critical sections. Thus to determine whether a task will be blocked later, we only need to check if it will attempt to lock a locked semaphore. This is our *preemption protocol* (PP) for task synchronization where binary semaphores are used:

> A task *Q* is allowed to preempt the running task *P* on the CPU if and only if the priority of *Q* is higher than the priority of *P* and none of the semaphores it will attempt to lock is currently locked by other tasks.

Under this protocol, a task will not be running on the CPU if it will later be blocked by the critical section of another (lower-priority) task. Since each blocking causes two CPU preemption and thus two context switches, we can reduce the number of context switches to what is minimally required.

A significant property of the PP is that the time at which a task completes will not be affected when this condition is being used, as shown in the following example:

Example 1: Suppose there are two tasks *P* and *Q*, with *Q* having a priority higher than *P*. In addition, there is one shared data structure protected by binary semaphore *s*.

Suppose task *P* arrives first and starts its execution on the CPU at time t_1. Later *P* locks semaphore *s*. At time t_2, *P* is preempted by some other task before leaving its critical section. Task *Q* then arrives and preempts the task on the CPU at time t_3. (*P* is not necessarily preempted by *Q* because there may be another task with priority higher than *P* which arrives before *Q* does. If there is not such an intermediate priority task, we assume that t_2 =

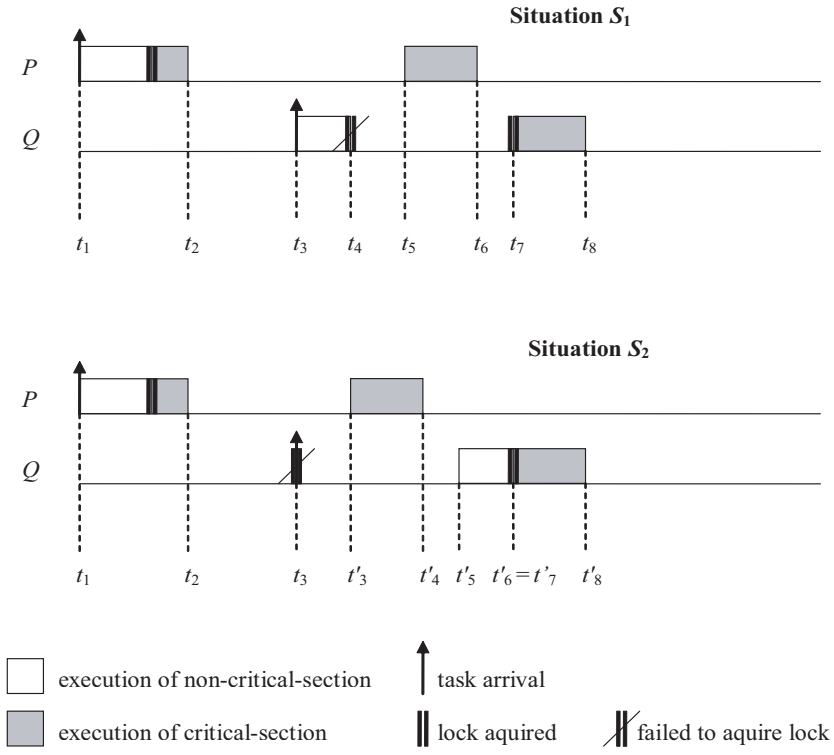

Fig. 1. An illustration of situation S1 and S2 in Example 1. Task R, which has higher priority than P and lower priority than Q, executes in time intervals $[t_2, t_3)$, $[t_4, t_5)$ of S1 and $[t_2, t_3)$, $[t_3, t'_3)$ of S2. Task T, which has higher priority than both P and Q, executes in time interval $[t_6, t_7)$ of S1 and $[t'_4, t'_5)$ of S2.

t_3.) At time t_4, *Q* tries to lock *s* and finds that *s* is already locked by *P*. Therefore the execution of *Q* has to be suspended and *Q* is blocked by task *P*. *P* resumes its running later at time t_5 and finishes its critical section at time t_6. *Q* will start again at time t_7 and finishes its execution at time t_8 The above situation will result from applying a traditional preemptive scheduling policy and will be referred to later as situation S_1.

Under our new preemption condition, when *Q* arrives at time t_3, it cannot preempt the running task on the CPU because *s* is locked by *P*. *P* will resume the execution of its critical section at time t'_3, and finish at time t'_4. *Q* will start execution at time t'_5 and will attempt to lock *s* at time t'_6. Task *Q* will be able to enter its critical section at time t'_7 and will leave it at time t'_8. We call this situation S_2.

Suppose that $t'_7 > t_7$. In both situations, the amounts of time spent on the executions of *P* and *Q* before *Q* enters its critical section are the same. Therefore, the only possible delay under the PP must be caused by another task that arrives before time t'_7. Tasks that arrives after time t'_7 need not be considered because by then any effect of the preemption protocol should be over.

If a task *R* whose priority is lower than that of *Q* but higher than that of *P* arrives before t'_7, then *R* will gain control of the CPU only if it arrives in the period $[t_2, t'_4)$ in situation S_2. Since by assumption that if *R* does not exist, then $t_2 = t_3$, it must be that $t_2 \leq t_3 < t_4$. If *R* arrives in the period $[t_2, t_3)$, i.e. , before the arrival of *Q*, the execution of *R* cannot cause any delay to *Q* in either situation. If *R* arrives in [t3, t4), *R* will cause a delay to *P*'s execution of its critical section and transitively a delay to the waiting task *Q* in S_2. However, in S_1, this delay will be imposed to *P* after *Q* suspends itself and waits for *P* at time t_4, because the priority of *P* is lower than that of *R*. Since *Q* is waiting for *P* and *P* for *R*, *R* is transitively causing a delay to *Q*. Therefore if an intermediate priority task can cause a delay to *Q* in S_2, then it can cause the same amount of delay to *Q* in S_1.

Notice that if *R* also requests *s*, the delay to *Q* will occur only in S_1 (if *R* arrives in $[t_2, t_6)$) because *R* is not allowed to preempt the task on the CPU under the new condition. *R* will finish at the same time in both situations, though, because it has to wait for *Q* to finish before it can lock *s*.

If a task *T* with a priority higher than that of *Q* arrives before time t'_7, *T* can always preempt the task running on the CPU in both situations as long as it does not require to lock *s*. Therefore the amount of delay caused to

both *P* and *Q* will be the same in both situations. On the other hand, suppose *T* will need *s*. The only thing that can affect the execution of *T* is the critical section of *P*, whereas the only thing that can affect the execution of the critical section of *P* is the non-critical section part of *Q*, and *only* in S_1. Thence the PP will not cause a delay to the completion of *T*. If *T* arrives before time t_7, *Q* is a non-factor on the execution of *T* and thence $t_7 = t'_7$. Since there is no task that arrives before t_7 can cause a delay to the start of *Q*'s critical section execution, it is impossible that $t_7 < t'_7$. Therefore, the time of completion of *Q* is not affected by the preemption protocol. Task *P* always has to start its section following the critical section after *Q* has completed. So its time of completion is not affected either.

In general, the preemption condition will be effective only if some lower-priority task is in its critical section when the higher-priority task arrives. The preemption protocol trades the execution time of the higher-priority task before it reaches its critical section with the remaining execution time of the lower-priority task's critical section that will cause blocking. Such a trade will not affect the time the higher-priority task enters into its critical section, neither will it cause a delay to the time when the lower-priority task starts to execute the segment of code right after its critical section. In fact, if the lower-priority task happens to finish its entire execution with the blocking-causing critical section, its time of completion will be moved up as a result of using the PP.

The following theorem concludes what we have examined so far:

Theorem 1: No task completion will be delayed as a result of applying the new preemption protocol in a uniprocessor system using preemptive priority scheduling and binary semaphore for task synchronization.

Proof In Example 1, we have shown the case in which three tasks cause interference to each other's execution and none of them will be delayed in their completions. A fourth task, if present, can be treated like *R* if its priority level is between that of any two tasks already being considered, or like *T* if its priority is higher than all tasks already being analyzed. Applying this strategy to all additional tasks that require consideration, we can see that none of the task completions will be delayed. □

In our preemption protocol, the execution of the higher-priority task is delayed until the lower-priority task has finished using the locked resource. This allows the CPU to work on the lower-priority task in the critical section without being switched to the higher-priority task, gets blocked and switched back to the lower-priority task, thus reducing the number of context switches. Hence, the above theorem shows that our general principle for context switch reduction does not delay task completion.

However, there is an additional cost caused by this scheme because, given a task, the scheme needs to know more information in advance, such as looking up the other lower-priority tasks which would use the same unit resource. In this case, there is a cost associated with the look-up steps though context switches are saved. Let n be the total number of tasks, and T_i be the name of every task where i is from 1 to n. The current task T_c has to look up the other m (m ¡ n) tasks, where m is the number of tasks that require the same resource T_c also requires and that have a priority lower than the priority of T_c. Thus, we now give the formula to describe the look-up cost introduced by this algorithm: $\sum_{k=1}^{m} T_k \cdot X_k$, where

$$X_k = \begin{cases} 1 & if\ task\ T_k\ uses\ the\ same\ resource\ that\ T_c\ requires; \\ 0 & otherwise; \end{cases}$$

This formula also applies even if there are multi-unit resources and their usages are nested. Therefore, in the worst case, assuming every task has to do the look-up step, there would be $\sum_{m=1}^{n-1} m = \frac{n(n-1)}{2}$ look-ups. Hence the time complexity for this algorithm is O(n^2) in the worst case.

III. Application to the Priority Inheritance Protocols

The characteristic of our preemption protocol with respect to task completion has a very interesting implication on the synchronization of real-time tasks. To remedy the problem of uncontrolled priority inversion in a real-time system that uses a fixed-priority preemptive scheduling algorithm, protocols that utilize priority inheritance have been developed. The fundamental idea of priority inheritance is to allow the lower-priority task to temporarily *inherit* the priority of a higher-priority one which it has blocked through the use of a semaphore. Once the lower-priority task releases (or *unlocks*) the semaphore that the higher-priority task requires, its priority returns to the level it was at before the blocking of the higher-priority task. In a system that uses our new preemption condition,

blocking is said to occur when a higher-priority task is denied CPU preemption by a lower-priority one under the preemption protocol. During that blocking period, if we let the lower-priority task assume the priority of the task it has blocked, we can achieve exactly the same results as those of the basic priority inheritance protocol given in [6]. What we save under the preemption protocol is the overhead associated with extra context switches resulting from not using the PP.

Note that our proof of Theorem 1 is one for a system that uses a traditional preemptive priority scheduler, although the basic idea behind the PP will always stay the same, that is, trading the initial segment of execution of the higher-priority task with the critical-section execution of the lower-priority task. In those cases where priority inheritance is applied, the result of no delay under the preemption protocol can be proven in a similar way.

Theorem 2: No task completion will be delayed as a result of applying the new preemption protocol in a uniprocessor system which uses a fixed-priority preemptive scheduler and also enforces priority inheritance.

Proof Let us consider again Example 1. If task *R* arrives in the system, it can never gain control of the CPU between *Q*'s initiation and completion because *P* will inherit the priority of *Q* if it is running anytime during that period. In other words, during the time *Q* is in the system, the priority of a task that actually executes on the CPU is at least as high as that of *Q*. Therefore, *R* will have an impact on the system only if it arrives before t_3. There are two situations to be considered here:

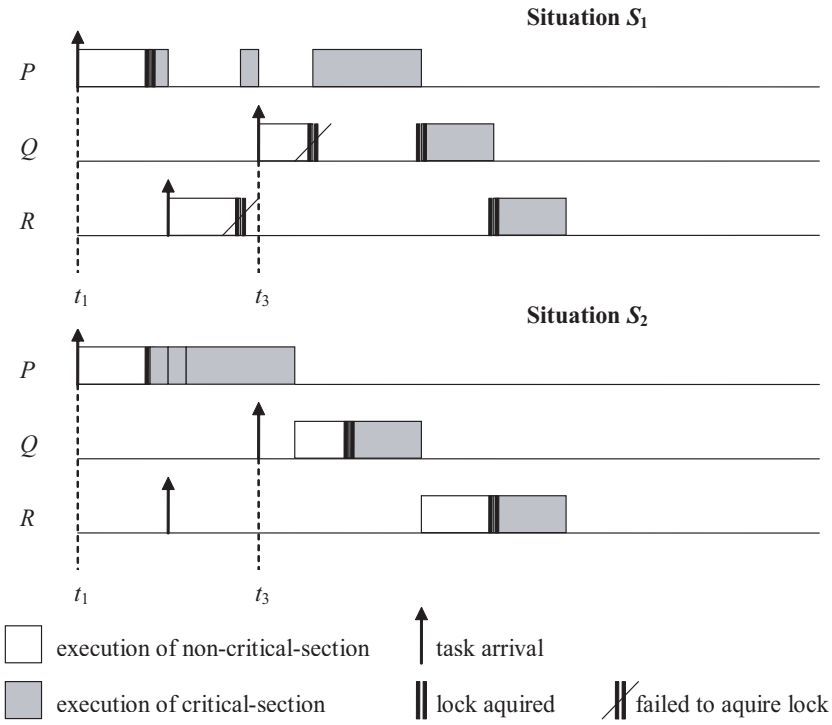

Fig. 2. An illustration of situation S1 and S2 in (1) below

(1) If *R* requires *s*, *R* can execute only the part before the critical section guarded by *s*, and then it will be blocked. Task *P* will resume its execution of the critical section and in the meantime task *Q* arrives. *R* can resume and finish only after *Q* has completed. This is what happens in situation S_1 shown in Fig. 2. In S_2, however, *R* will be blocked on its arrival and gain the control of the CPU only after *Q*'s completion. In both situations, the period between *R*'s initiation and completion includes *Q*'s entire execution, *P*'s execution of its remaining critical section, and *R*'s entire execution only. Thus *R*'s completion will not be delayed. An interesting point to be noticed is that *Q*'s completion is actually moved up in S_2 (shown in Fig. 2) because it has to wait for a shorter period of time for *P* to finish its critical section when *R*'s execution is moved down.

(2) If *R* does not lock *s*, it can run only for the period of time between its arrival and *Q*'s arrival (at time t_3). Then, if it is not finished, *R* can resume execution only after *Q* has completed. Therefore, *R* will finish at the same time in both situations regardless of whether it is able to finish before t_3.

Now suppose there is a task *T* whose priority is higher than that of *Q* and which arrives between the period from t_1 to t_8 (use the time-line in Fig. 1). Certainly, regardless of whether *T* locks *s*, *P* and *Q*'s portions after their respective critical sections will have to wait until *T* finishes all its execution. On the other hand, if *T* does not require to lock *s*, it will be able to start immediately upon arrival and finish without interruptions in both situations S_1 and S_2. If *T* does need *s*, then its completion may actually be advanced and never delayed in S_2. To see this, we first observe that $t_6 > t'_4$ because we assume that there is a preemption of *P* by *Q* in S_1 before t_6, but there is no such preemption in S_2. Thus, if T arrives at any time between t_2 and t'_4, it will experience the same amount of delay (to wait for *P* to finish the rest of its critical section) in both situations. Now if *T* arrives after t'_4 and before t_6, *T* will be blocked by *P* in S_1 but not so in S_2. Thus *T* can actually finish earlier in S_2 than in S_1. This is the same situation as in the condition (1) above where *R* and *Q* are considered. Here *T* is like *Q* and *Q* like *R* above. We have already shown that *R*'s completion above will not be delayed, so neither will *Q*'s here.

If more tasks are involved, a similar argument can be used to show that no delay will be caused under the preemption protocol. □

A. Implementation with the Priority Ceiling Protocol

The most effective and easily implemented protocol that uses priority inheritance is the Priority Ceiling Protocol. The *priority ceiling* of a binary semaphore is defined to be the priority of the highest-priority task that can lock this semaphore. The basic idea of the PCP is to deny a task's request for locking a semaphore if its priority is not higher than any of the ceilings of semaphores already locked by other tasks. This locking condition of the PCP ensures that whenever a task *P* locks a semaphore, no higher-priority task that arrives later will ever attempt to lock a semaphore locked by a task with priority lower than *P*'s. See [13] for a detailed discussion of the Priority Ceiling Protocol.

The advantage of the PCP is that the only additional information needed to implement the protocol is the ceilings of all semaphores in the system. To implement our new preemption protocol, however, we need to have a list of all semaphores that a task will ever request. This information can be stored in the process control block of that task. The task is checked according to the PP when it first arrives in the ready queue of the system.

The list of semaphores is not required if the preemption protocol is combined with the Priority Ceiling Protocol. Under the PCP, a task is blocked if it requires a semaphore and its priority is lower than the ceiling of some locked semaphore. This task can resume only after the blocking task releases its semaphore. As this kind of *ceiling blocking* is necessary to ensure certain important properties for real-time scheduling, our general principle for context switch reduction also applies in this situation.

We can define the combination of the PCP and the PP as follows:

> Let s^* be the semaphore with the highest priority ceiling of all semaphores currently locked by some tasks in the system. When task *P* arrives, it cannot preempt the currently running task on the CPU if its priority is lower than that of the running task, or its priority is not higher than the priority ceiling of s^* and it will later require to lock a semaphore. In the latter case, *P* is said to be blocked on semaphore s^* and *by* the task P^* which holds the lock on s^*. At the same time, P^* will inherit the priority of *P* until it finishes the critical section that causes the blocking on *P*. At that time P^* will resume its previous priority and the highest-priority task will be selected to run.

We call this the *priority ceiling preemption protocol* (PCPP). It takes advantage of the basic idea of the PCP which eliminates deadlocks and multiple blocking to a task, and at the same time reduces the number of context switches by letting the lower-priority tasks finish their blocking-causing critical sections first. This protocol can also be implemented without much overhead. In addition to the priority ceilings of semaphores, we only need to know whether a newly arrived task will use a semaphore. We are not even concerned with which semaphore(s) the task will use.

Corollary 3: Under the priority ceiling preemption protocol, no task completion will be delayed compared to when the Priority Ceiling Protocol is used.

A detailed proof will not be given for the above corollary as it is very similar to the one for Theorem 2. But intuitively, we see that the Priority Ceiling Protocol only changes the testing condition of blocking, not the nature of it. Since our preemption protocol is effective in situations where blocking occurs regardless of how it occurs, it

certainly works with the ceiling blocking of the Priority Ceiling Protocol. Hence the completion of tasks are not delayed as a result of applying the preemption protocol to the PCP.

Theorem 4: Under the priority ceiling preemption protocol, a task *P* can be blocked for the duration of at most one of the critical sections that can cause blocking to *P*.

Theorem 5: The priority ceiling preemption protocol prevents deadlocks.

Theorems 4 and 5 are derived directly from the results in [13]. Since under the priority ceiling preemption protocol tasks finish no later than under the Priority Ceiling Protocol, the maximum blocking each task endures under the PCPP cannot be longer than under the PCP. Thus we have Theorem 4. To obtain Theorem 5, we observe that the task which is requesting to lock a semaphore must be the currently-running task on the CPU. Two types of tasks may currently be in the system (but not running): those preempted during their execution and those denied starting their execution. By the preemption condition, the running task will never require to lock a semaphore that is already locked by some task of the first type, whereas a task of the second type has locked no semaphores (it has not even started its execution). Thus a task cannot wait for another task that is already waiting for it and consequently deadlock can never occur.

In addition to the advantage of achieving the same result as the PCP with a reduced number of context switches (our experimental results show a reduction of 10% to 20%), some tasks in a certain situation can finish earlier under the priority ceiling preemption protocol. The following example illustrates this point.

Example 2: Suppose that task *P* arrives at time 0, requires 6 units of CPU time for execution, and locks the semaphore *s* during time units 2–4 of its execution; task *Q* arrives at time 2, requires 4 units of CPU time, and locks *s* during its 3rd time unit; task *R* arrives at time 6, requires 4 units of CPU time, and locks semaphore s' during time units 2–3; and task *T* arrives at time 7, requires 4 unit of CPU time, and locks *s* (not s') during its 3rd time unit of execution. We also suppose that *T* has the highest assigned priority, *R* has the next highest, *Q* the 3rd highest, and *P* has the lowest priority among all four tasks. Suppose the priority ceiling of *s* is the same as the priority of *T*, while the priority ceiling of s' is higher than *T*'s priority. The different results of scheduling under the PCP and the PCPP are shown in Fig. 3.

In situation S_1, *Q* is able to preempt *P* when it arrives, while *Q* is blocked by *P* in situation S_2. The two time units of execution of *Q* in S_1 before task *R*'s arrival have caused *R* to finish 2 time units later in S_1 than in S_2. Notice also that there are 9 context switches in S_1 while there are only 5 in S_2.

When the locking condition of the Priority Ceiling Protocol is true, the condition of our Preemption Protocol must be true. This is because if the priority of the newly arrived task is higher than the priority ceiling of all semaphore, then it must be the highest priority task among all running tasks in the system. It is, therefore, allowed to lock any semaphore and does not cause extra context switch. Thus, when our preemption protocol is combined with the Priority Ceiling Protocol, its effectiveness in context switch reduction is retained.

A point to be noted is that according to the PCP, the task that is holding the semaphore with the highest priority ceiling is not necessarily the task currently executing on the CPU. Another task with a priority lower than the highest ceiling but having no semaphore requirement may be the running task. Under the PCPP, when a task arrives with an even higher priority but still no higher than the highest priority ceiling of an already locked semaphore, it may be blocked because it does require to lock certain semaphore(s). The blocking task (which holds the semaphore with the highest ceiling) of that newly arrived task should resume execution at this moment and preempt the running task. Consequently, one context switch is not saved in a situation like this. However, this context switch is the price we pay to ensure the worst case blocking property of the Priority Ceiling Protocol. Moreover, situations in which a task with intermediate priority and no semaphore requirement just comes in time to cause such a little disturbance are not as often as the occurrence of task blocking.

B. Implementation with the Optimal Mutex Policy

Another real-time task synchronization protocol that belongs to the family of priority inheritance protocols is the Optimal Mutex Policy [11]. The optimality of this policy lies in the fact that no other priority inheritance policy can guarantee a better worst-case blocking duration while still preventing deadlock. In the following we will look at whether our preemption protocol can be implemented with the Optimal Mutex Policy to have the number of context switches reduced while retaining the policy's properties.

Let *P* be the task that attempts to lock an unlocked semaphore *s*. Let s^* be a semaphore (also referred to as a *mutex* because its function is to ensure **mut**ual **ex**clusion) with the highest-priority ceiling locked by tasks other

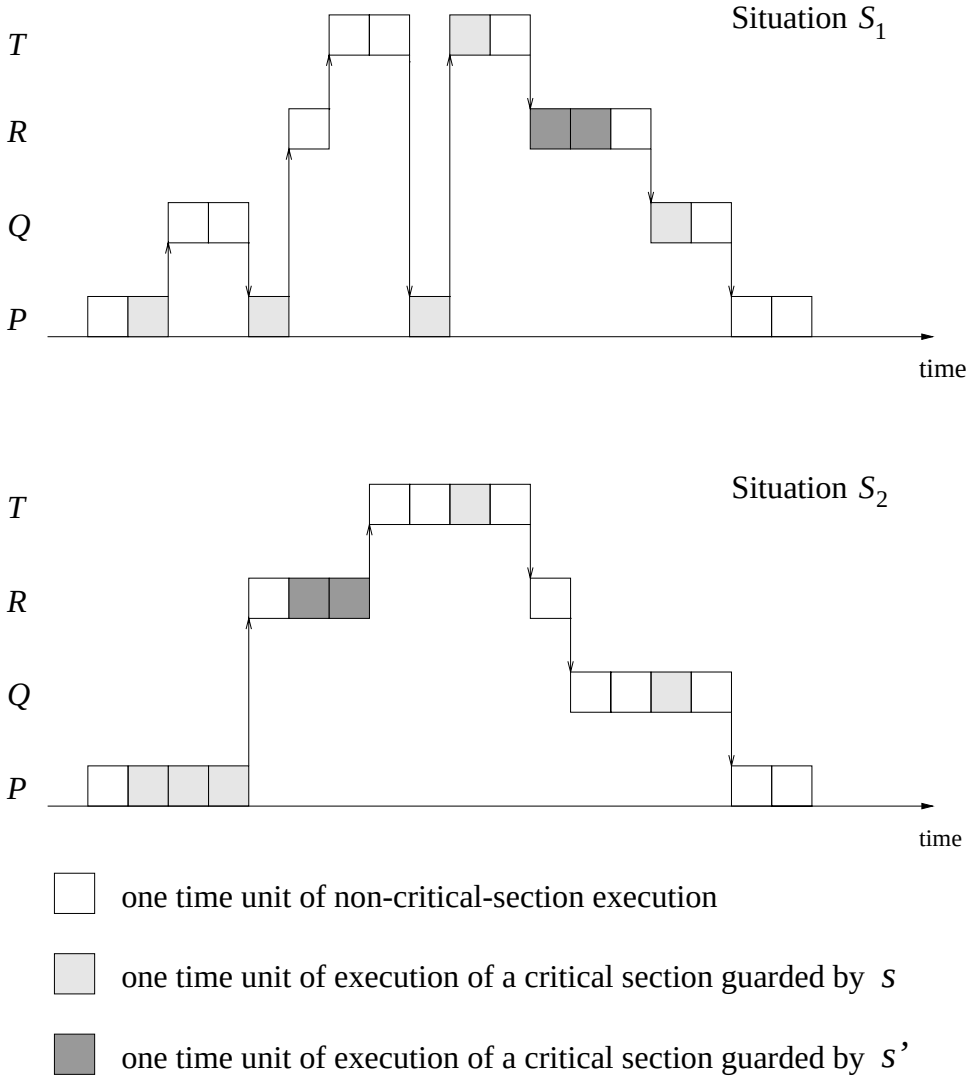

Fig. 3. An example that compares the PCPP with the PCP

than *P*, and let *Q* be the task holding the lock on s^*. The Optimal Mutex Policy allows task *P* to lock the unlocked mutex s if and only if at least one of the following conditions is true.

1) The priority of *P* is greater than the priority ceiling of s^*.
2) The priority of *P* is equal to the priority ceiling of s^* and the current critical section of *P* will not attempt to lock any mutex already locked by *Q*.
3) The priority of *P* is equal to the priority ceiling of *s* and the lock on mutex *s* will not be requested by *Q*'s preempted critical section.

These conditions are referred to as the *locking conditions* of the Optimal Mutex Policy. Condition 1 is basically the locking condition of the Priority Ceiling Protocol, and so we can implement with it our preemption protocol to save context switches. Conditions 2 and 3 require a closer examination.

One of the most significant properties of the Optimal Mutex Policy is that when a task *P* is executing on the CPU, there can be at most one strictly lower-priority task *Q* that has locked a semaphore s^* such that the priority ceiling of s^* is greater than or equal to the priority of *P* (see [11]). In other words, when *P* requests a semaphore and Condition 1 is false, then there must be a task *Q* that has locked s^* and *Q* is unique. Now suppose *P* successfully locks a semaphore because Condition 2 or 3 of the OMP is true. Then the priority of *P* must be greater than all the ceilings of semaphores which are locked by tasks other than *Q* (which has locked s^*). It follows that *P* will not attempt to lock those semaphores. The only semaphores left to be considered are those locked by *Q*. Condition 2 tells us that *P* will not attempt in the current critical section to lock those semaphores already locked by *Q*.

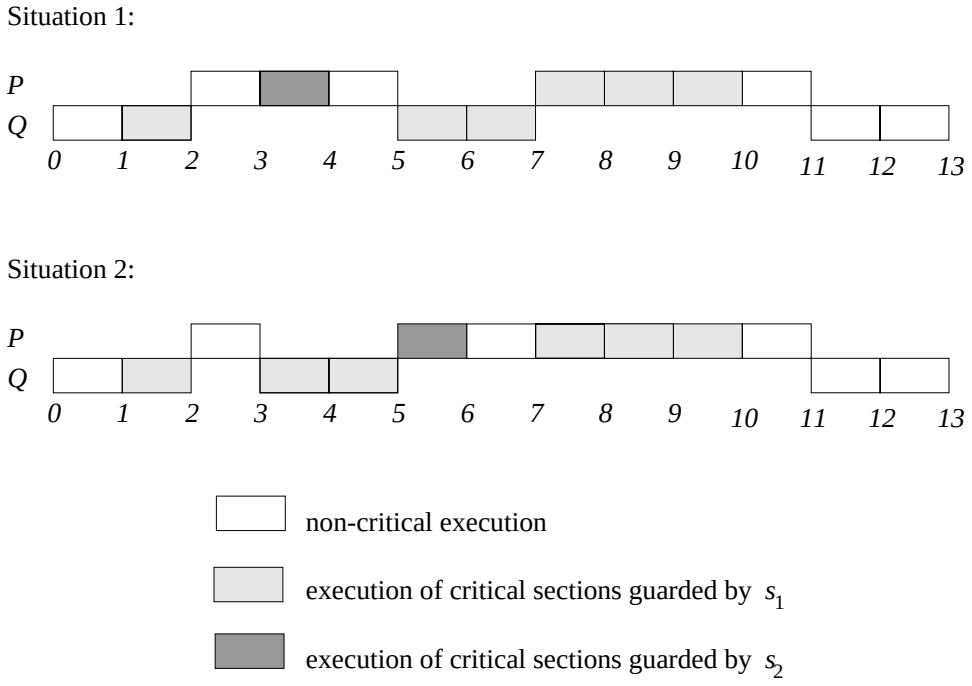

Fig. 4. A closer look at Condition 3 of the OMP

However, it does not say that *P* will not attempt to lock a semaphore already locked by *Q* in outer-most critical sections following the current one, if such critical sections exist. Therefore, *P* may still be blocked later (by *Q*) after it locks a semaphore because Condition 2 is true. To apply our preemption protocol, we must extend this condition to require that task *P*, during its whole execution period, will not attempt to lock any semaphore already lock by task *Q*. Since the above extension of Condition 2 is a logically sufficient condition for the original Condition 2, the properties associated with the original condition is retained after the extension. Moreover, the effect of Condition 2 is nil with respect to task completion when in the later stage of its execution, *P* does require a semaphore locked by *Q*.

Example 3: Suppose that task *Q* arrives at time 0, requires 6 units of CPU time to execute, and locks the semaphore s_1 during time units 2-4 of its execution; task *P* arrives at time 2, requires 7 units of CPU time, and locks semaphore s_2 during the second time unit and locks s_1 during time units 4–6 of its execution. Suppose the priority ceiling of s_1 is equal to the priority of task *Q*, which is higher than that of task *Q*.

When locking Condition 2 is used in Situation 1 of Fig. 4, *P* is able to lock s_2 and enter its critical section at time 3, because its priority is the same as the priority ceiling of s_1 and the current critical section of *P* will not request to lock any semaphore already locked by *Q*. At time 5, *P* requests s_1 and is blocked by *Q*. *P* can resume only after *Q* finishes its critical section at time 7 and finishes its own execution at time 11, when *Q* resumes and finishes at time 13.

In Situation 2, however, Locking Condition 2 is not in effect and *P* is blocked at time 3 when it attempts to lock s_2. *P* resumes at time 5 and finishes still at time 11. Task *Q*'s time of completion is not changed either. Thus we can see that the only improvement Locking Condition 2 can bring with respect to task completion is when the higher-priority task does not later lock a semaphore already locked by the lower-priority task.

As we can see from Example 3, any blocking experienced by the higher-priority task means that part of its execution can resume only after the blocking-causing critical section of the lower-priority task is finished. There will not be any difference in the completion times of the higher-priority task if the blocking does occur regardless of when it occurs.

Condition 3 of the OMP is somewhat troublesome. A task that locks a semaphore because Condition 3 is true may still be blocked if it later attempts to lock any semaphore with a priority ceiling higher than its priority. Similar to the extension of Condition 2, we can guarantee that the task will not be blocked once it enters its critical section under Condition 3. But this is achieved by requiring (suppose *P* and *Q* are defined as they are in the definition of

the OMP):

1) *P* does not need to lock a semaphore already locked by *Q*,
2) all the semaphores that *P* requires have priority ceilings equal to the priority of that task, and
3) the preempted critical section of *Q* will not attempt to lock any semaphore that *P* locks.

We call these the non-blocking requirements of Condition 3. The cost of implementing the non-blocking requirements may much outweigh the saving that we bring about from context switch reduction when we apply our preemption protocol. Furthermore, while Condition 3 can indeed move up the completion time of task *P* when *P* meets all of the non-blocking requirements, such an "improvement" may be made at the expense of the execution time of a potential task with a priority higher than that of *P*.

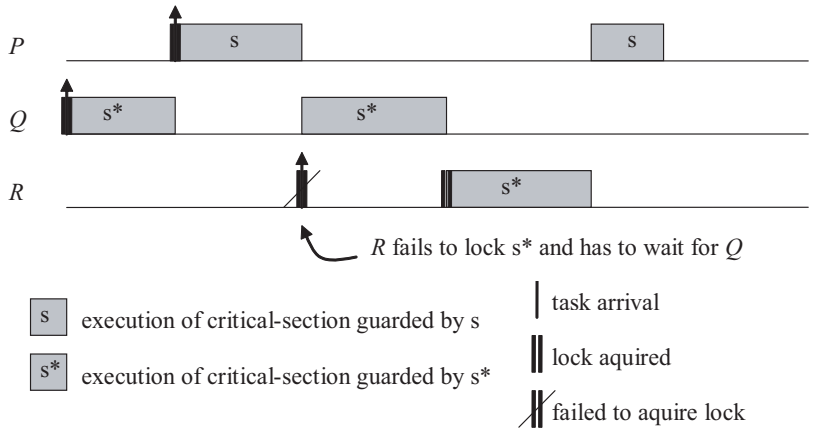

Fig. 5. An illustration for Example 4

Example 4: Suppose tasks *P* and *Q* are defined as in the definition of the OMP and *Q* has locked s^*. Suppose *P* starts out its critical section because Locking Condition 1 is false and Locking Condition 3 with non-blocking requirements is true. Certainly *P* does not have to wait before completion for *Q* to finish its critical section guarded by s^*.

If a task *R* with a priority higher than that of *P* arrives and requires to lock s^* (as in Fig. 5, since the priority of s^* is higher than the priority of *P*), *R* will be blocked and wait for *Q* to finish its critical section. *P* then is indirectly blocked by and waiting for *Q* to finish the critical section as well.

From the above example, we see under Condition 3 it is possible that a task like *R* arrives and offsets the benefit of this condition, whereas under Conditions 1 and 2 no such task (with priority higher than that of the current running task but no higher than the highest priority ceiling of locked semaphores) will exist. Consequently, Locking Condition 3, though constituting the optimality of the Optimal Mutex Policy, is undesirable to be implemented with our preemption protocol. Therefore, our sub-optimal mutex preemption protocol is stated as follows:

> Let s^* be the semaphore with the highest priority ceiling of all semaphores currently locked by some task *Q* in the system. When a task *P* arrives, it can preempt the running task *R* on the CPU only if *P*'s priority is higher than *R*'s and one of the following is true:
>
> 1) *P* has no semaphore requirement;
> 2) *P* will require to lock a semaphore and *P*'s priority is higher than the priority ceiling of s^*;
> 3) *P*'s priority is equal to the priority ceiling of s^* and *P* will not attempt to lock any semaphore already locked by *Q*.
>
> If *P*'s priority is higher than that of *R* but *P* is unable to preempt *R*, we say that *P* is blocked on s^* by task *Q*. According to the principle of priority inheritance, *Q* will resume its execution at *P*'s priority and return to its previous priority when it exits the critical section guarded by s^*.

This derived protocol does not affect most of the properties of the Optimal Mutex Policy regarding deadlocks and task blocking. A task will experience a blocking duration of at most one critical section of a lower-priority task and the system is deadlock-free. Most importantly, a considerable number of context switches are saved as a result.

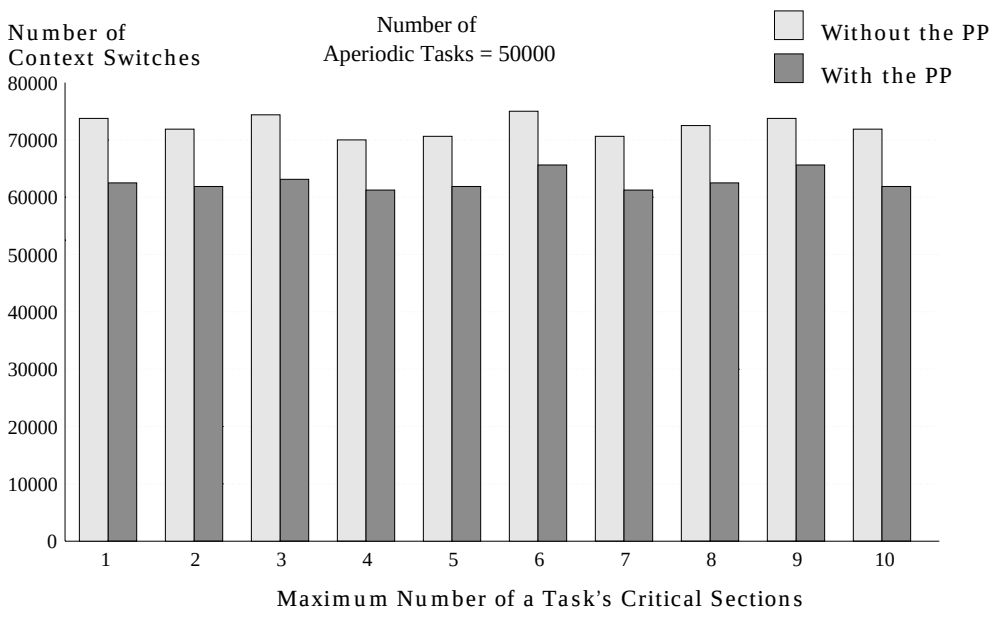

Fig. 6. Context switches comparison by using PCPP versus PCP

C. Simulation Results

We used simulation programs to study the effect of our derived protocol PCPP in comparison with the original protocol PCP.

First, we simulated a non-real-time system using a traditional preemptive scheduler. Tasks are randomly generated with exponential inter-arrival times. There are ten priority levels in the system and tasks are assigned a random priority on arrival. The number of semaphores in the system is also ten but tasks are all given a maximum number of outer-most critical sections that they will have. In the original system, tasks are blocked when they attempt to lock a locked semaphore while in the modified system with the preemption protocol in effect, tasks can only be blocked at the time of their arrival. The simulation result is shown in Fig. 6. The horizontal axis shows the maximum number of critical sections each task will need to execute. The relationship is weak between the number of critical sections that a task can have and the number of blockings caused by task synchronization. Our simulation showed that about 10% to 20% of context switches can be avoided using our preemption protocol.

In another simulation, we randomly generated one thousand sets of ten periodic real-time tasks. In each set, tasks are assigned relative priorities according to their periods. There are also ten semaphores in the system with randomly generated priority ceilings. Each task also has a maximum number of critical sections that it can execute. The semaphores to be locked by a task for its critical sections have priority ceilings equal to or higher than the priority of that task. The Priority Ceiling Protocol and the Priority Ceiling Preemption Protocol are implemented for comparison. The result is shown in Fig. 7 in terms of average percentage of context switches reduced by the PCPP compared to the PCP.

As we can see, the average percentage of context switch reduction ranges in the tens and twenties in either a real-time or a non-real-time environment. Since one context switch can be associated with each task arrival, the number of context switches will equal the number of tasks (or task instances if the task is periodic) when there is no preemption. Each preemption leads to one extra context switch when the preempted task resumes whereas each blocking causes two more context switches. The essence of our protocol is to combine preemption with blocking. The composite effect of the number of tasks, the number of critical sections each task has, the duration of critical sections, and the task period (or arrival time in non-real-time systems) determines the number of blockings that actually occur. In general, our protocol can save a large number of context switches, especially when the number of task blockings is substantial. For real-time scheduling, the more serious the problem of priority inversion, the more significant the effect of our protocol.

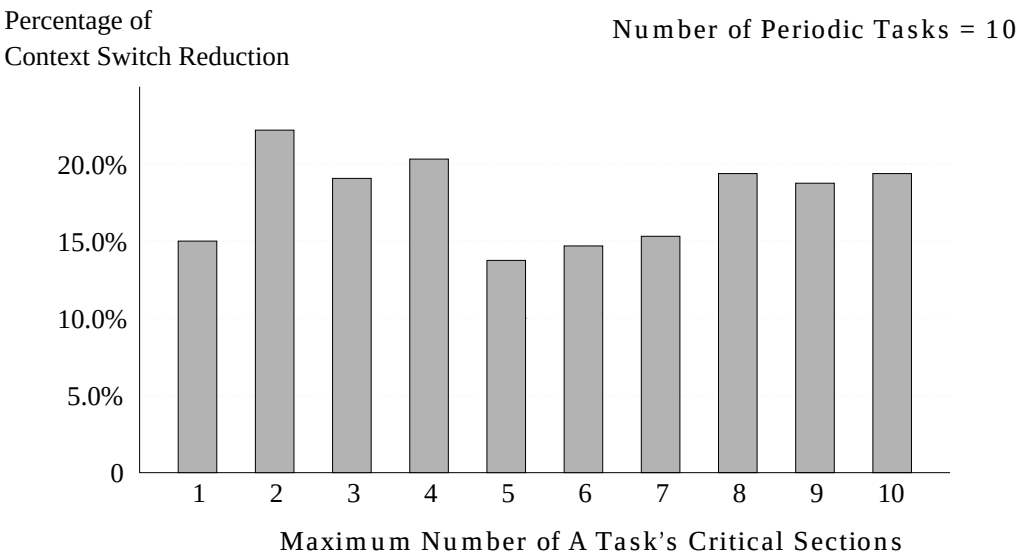

Fig. 7. Average percentage of context switches reduced by PCPP compared to PCP

IV. Concluding Remarks

A. Conclusion

In this paper, we have presented a new protocol for task preemption in a static-priority system where binary semaphores are used to enforce mutual exclusion. The underlying idea of this protocol is to allow to finish first any critical section that will cause blocking to a higher-priority task. As a result, we can achieve a reduction in the number of context switches used while causing no delay to task completion. We have also applied this preemption protocol to real-time synchronization where priority inheritance is used to avoid unbounded priority inversion. It is shown that the preemption protocol can be very easily implemented with the Protocol Ceiling Protocol and we call the combination of the two the Priority Ceiling Preemption Protocol. This new protocol uses fewer context switches than the Priority Ceiling Protocol while retaining the advantages of the PCP.

The preemption condition can also be implemented with other priority inheritance protocols such as the Optimal Mutex Policy. We have examined the three locking conditions of the OMP and determined which ones of the three that our preemption protocol can be applied to. The result is a protocol not as optimal as the OMP with respect to single critical-section blocking, but we have shown reasons not to have such an optimality.

We have performed simulations to compare the number of context switches occurred under the original protocols with the derived ones. Both real-time and non-real-time systems have been studied. The results are similar because the number of context switches that can be saved is dependent upon the number of task blocking, which in turn can be a function of many factors such as task initiation times and length of critical sections. The percentages of context switch savings normally range in the tens and twenties in either system environments.

B. Future Work

In this paper our focus is on the preemptive scheduling of static-priority tasks in a uniprocessor system where corresponding synchronization protocols are used. More simulation and evaluation work is still on the way. These include an implementation of PP with OMP, a simulation comparison of PP with PCP and OMP, an evaluation of the overhead of implementing PP in a real operating system, and a comparison with the results in [16]. Research on the techniques of context switch reduction can be extended into the following areas:

- In a system that assigns dynamic priorities to tasks such as those using the Earliest Deadline or the Least Laxity scheduling algorithms, more context switches are usually required than in a system that applies static-priority scheduling. However, dynamic-priority scheduling algorithms normally achieve higher CPU utilization than the static-priority ones. Investigation on the tradeoff between the CPU utilization and the system overhead due to context switches is needed to determine which algorithm has the best overall performance under various circumstances.

- Between the EDS and the LLS, it seems that the EDS requires a smaller amount of context switches than the LLS. Since in the EDS, the task's (relative) priority becomes fixed once it arrives in the system, we can determine the current priority ceiling of a semaphore by calculating the task arrival times and deadlines. Our preemption protocol may be applied to the EDS since it is effective as long as there are priority inversions. More studies are required to find a synchronization protocol for the EDS and apply our protocol to it.
- Under the Least Laxity algorithm, since a task's priority can still change after the task's arrival in the system, the concept and effect of blocking are very different than under other scheduling algorithms. We surmise that the effect of task *P* waiting for task *Q* is nil unless the section of *Q* that causes the waiting is the last execution section of *Q*, since the waiting task *P*'s priority will be increasing and *P* will have more "leverage" to gain more execution time on the CPU. If this is true, then as in our preemption protocol, we can simply allow for a longer duration of waiting in the middle of task execution so as to reduce the number of context switches.
- The proposed protocol assumes complete knowledge of the locks required by each task. Ongoing work attempts to extend the protocol to use partial or no knowledge of future locks required by each task. Furthermore, work is under way to modify the protocol to take advantage of current multithreaded systems.
- In a multiprocessor system, if task migration is not allowed, the concept of blocking is generalized to include the time a task spends on waiting for a task of any priority but on a different processor [9], [10]. This is so because if there is no data sharing among the tasks on different processors, a task cannot be blocked by any task of any priority on another processor [10]. The blocking of a task by another task on a different processor is referred to as *remote* blocking. When a task *P* is initiated on a processor, there may be a task *Q* on a different processor that has locked the semaphore *s* required by *P*. By the time *P* has executed to actually attempt to lock *s*, *Q* might have release it so that there won't be any blocking at all. On the other hand, two tasks running at the same time on different processors may require to lock the same semaphore. Thus one task may *potentially* block another when their resource requirements overlap and they are executing simultaneously. Since our context-switch reduction technique works only if blocking does take place without applying the technique, analysis and calculations must be performed on tasks to make certain that a priority inversion is going to occur before we can apply the technique to save context switches.
- When the same task can be executed on different processors, task migration replaces context switch as the greater operating system overhead. In other words, a preempted task may resume execution on a different processor than it was on before the preemption. To reduce task migration overhead, research is needed to find a multiprocessor scheduler that does not preempt tasks and switch them to a different processor unless it is absolutely necessary.

V. Acknowledgement

Thanks go to Chen Feng and Bin Lu for help in revising and improving the paper. Also, thanks go to the reviewers for the very detailed reviews and helpful suggestions.

References

[1] F. Jiang and A. M. K. Cheng, "A Context Switch Reduction Technique for Real-Time Task Synchronizati on," *Proc. IEEE-CS Intl. Parallel and Distributed Processing Symp.*, San Francisco, CA, CD format, May 2001.

[2] T. Baker, "A Stack-Based Resource Allocation Policy for Real-time Processes," *Proc. IEEE Real-Time Systems Symposium*, Dec. 1990.

[3] M. L. Dertouzos, "Control robotics: The procedural control of physical processes," *Proc. IFIP Cong.*, pp. 807-813, 1974.

[4] C.-H. Hsu, U. Kremer, and M. Hsiao, "Compiler-directed dynamic voltage/frequency scheduling for energy reduction in mircoprocessors," *Proceedings of the 2001 international symposium on Low power electronics and design*, ACM Press, pp. 275-278, 2001.

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

[6] A. K. Mok, "Fundamental design problems of distributed systems for the hard reak time environment," *PhD Thesis*, M.I.T., 1983.

[7] P. Pillai and K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*, ACM Press, 2001, pp. 89-102.

[8] A. Qadi, S. Goddard and S. Farritor, "A Dynamic Voltage Scaling Algorithm for Sporadic Tasks," *Proc. IEEE Real-Time Systems Symposium*, Dec. 2003.

[9] R. Rajkumar, "Real-time synchronization protocols for shared memory multiprocessors," *Proc. 10th IEEE Int. Conf. on Distributed Computing Systems*, pp 116-123, May 1990.

[10] R. Rajkumar, L.Sha and J. Lehoczky, "Real-time synchronization protocols for multiprocessors," *Proc. 9th IEEE Real-Time Systems Symposium*, pp. 259-269, Dec. 1988.

[11] R. Rajkumar, L. Sha, J. P. Lehoczky, and K. Ramamritham, “An optimal priority inheritance policy for synchronization in real-time systems,” *Advances in Real-Time Systems*, pp. 249-271, New Jersey: Prentice Hall, 1995.
[12] I. Rhee and G. R. Martin, “A scalable real-time synchronization protocol for distributed systems,” *Proc. IEEE Real-Time Systems Symposium*, pp. 18-27, 1995.
[13] L. Sha, R. Rajkumar and J. P. Lehoczky, “Priority inheritance protocols: an approach to real-time synchronization,” *IEEE Transactions on Computers*, pp. 1175-1185, Sept. 1990.
[14] Radu Dobrin. and Gerhard Fohler, “Reducing the Number of Preemptions in Fixed Priority Scheduling,” *EuroMicro Conference on Real-Time Systems*, 2004.
[15] Yun Wang, and Manas Saksena, “Scheduling Fixed-Priority Tasks with Preemption Threshold,” *IEEE Real-Time and Embedded Technology and Applications Symposium*, 1999.
[16] Khawar M. Zuberi, and Kang G. Shin, “An Efficient Semaphore Implementation Scheme for Small-Memory Embedded Systems,” *IEEE Real-Time Technology and Applications Symposium*, 1997.
[17] Yan Wang, and A. M. K. Cheng, “A Dynamic-Mode DVS Algorithm under Dynamic Workloads,” *Proc. IEEE-CS Real-Time and Embedded Technology and Applications Symposium WIP Session*, San Francisco, March 2005. Also as invited paper, ACM Special Interest Group on Embedded Systems (SIGBED) Review, Volume 2, Number 2, April 2005.