Perfecting Preemption Threshold Scheduling for Object-Oriented Real-Time System Design: From the Perspective of Real-Time Synchronization

Saehwa Kim
School of Electrical Engineering and Computer Science
Seoul National University
Seoul 151-742, Korea
+82-2-880-8370
ksaehwa@redwood.snu.ac.kr

Seongsoo Hong
School of Electrical Engineering and Computer Science
Seoul National University
Seoul 151-742, Korea
+82-2-880-8357
sshong@redwood.snu.ac.kr

Tae-Hyung Kim
Dept. of Computer Science and Engineering
Hanyang University
Ansan, Kyunggi-Do 425-791, Korea
+82-31-400-5668
tkim@cse.hanyang.ac.kr

ABSTRACT

In spite of the proliferation of object-oriented design methodologies in contemporary software development, their application to real-time embedded systems has been limited because of the practitioner's conservative attitude toward handling timing constraints. In fact, this conservative attitude is well-grounded because traditional priority-based scheduling techniques cannot be straightforwardly integrated into them. The automated implementation from the object-oriented real-time designs usually incurs a large number of tasks which, under traditional priority-based scheduling techniques, does not scale well due to excessive preemption overheads. Recently, preemption threshold scheduling was introduced to reduce run-time multi-tasking overhead while improving schedulability by exploiting non-preemptibility as much as possible. Unfortunately, the preemption threshold scheduling cannot be directly adopted into the object-oriented design methods due to the lack of real-time synchronization.

In this paper, we present the essential basis of real-time synchronization for preemption threshold scheduling. Specifically, we integrate the priority inheritance protocol, the priority ceiling protocol, and the immediate inheritance protocol into preemption threshold scheduling. We also provide their schedulability analyses. Consequently, the integrated scheme, which minimizes worst-case context switches, is appropriate for the automated implementation of real-time object-oriented design models

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management – *scheduling*.

General Terms

Theory

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

LCTES'02–SCOPES'02, June 19-21, 2002, Berlin, Germany.

Copyright 2002 ACM 1-58113-527-0/02/0006…$5.00.

Keywords

Preemption threshold scheduling, real-time synchronization, priority inheritance protocols, priority ceiling protocol, object-oriented real-time system design.

1. INTRODUCTION

Developing a real-time embedded system is a sophisticated task. The monolithic approach to developing large and/or complex computer systems is coming to an end in the era of contemporary software development. Nonetheless, intrinsic difficulties arising from timing constraints in the real-time embedded applications still tie up practitioners with a rather conservative approach to developing embedded systems. Many early real-time system researchers have attached importance to timing analysis at design time in order to derive a set of feasible tasks [11, 13, 14]. This has been quite successful and real-time programmers are able to safely and reliably derive a feasible set of tasks in advance.

However, as real-time embedded systems get complex and sophisticated to meet the increased degree of safety, reliability, and performance requirements, it becomes inevitable for real-time system designers to rely on systematic software design methodologies during system development. Among a wide variety of software design methodologies, object-oriented design methodologies have become dominant and popular since they allow for easy software maintenance, software reuse, and component-based coding of complex real-time systems that may evolve over time. A recent trend in embedded systems development even tries to hit the lines of hardware by harnessing software to implement physical layer functions in pursuit of its apparent advantages: re-usability of hardware resources, ease of upgrades, flexibility to build a dual- or multiple-standards implementation, and so on. Software defined radio (SDR) is one of those representative approaches.

To cope with the additional complexity, the object-oriented design methodology should be seamlessly integrated into traditional real-time schedulability analysis techniques, so that embedded system programmers can be equipped with systematic software design methodologies and CASE tools as well. Traditional preemptive fixed priority scheduling cannot be directly used in this integration largely owing to its excessive run-time overhead. Preemption threshold scheduling (PTS) improves both run-time overhead and schedulability, but still cannot be directly used to this end either,

since real-time synchronization problems remain unsolved. In this paper, we present an essential basis for their integration by perfecting preemption threshold scheduling [1, 2, 12] for real-time synchronization.

1.1 Background

Two primary thrusts in embedded systems research are how to enhance schedulability by devising elaborate scheduling techniques and how to integrate schedulability analysis techniques into modern software engineering design and implementation methodologies. Successfully applied scheduling techniques have been based on preemptive fixed priority scheduling in most real-time systems developments since Liu and Layland introduced it in their pioneering paper [11]. However, dynamic priority schedulers can achieve higher schedulability than fixed ones, and non-preemptive schedulers incur less run-time overhead. This means that in fine-grain or medium-grain thread-based processing, which is the case especially in multi-threaded implementations from real-time object-oriented designs, the context switching overhead may overshadow the benefits of multi-threading and preemptive scheduling.

To alleviate the run-time overhead problem while improving schedulability, Lamie at Express Logic, Inc. introduced the notion of preemption threshold [12]. The preemption threshold regulates the degree of "preemptiveness" in fixed-priority scheduling. If the threshold of each task is the same as its original priority, then it is equivalent to the preemptive fixed priority scheduling. If each task has the highest threshold value in a system, then it becomes a non-preemptive scheduling. As Wang and Saksena illustrated in [1, 2], the schedulability of a task set under preemptive scheduling does not imply that the task set is also schedulable under non-preemptive scheduling and vice versa. Thus, the preemption threshold scheduling is a good complement to preemptive fixed-priority scheduling. It improves schedulability, withholds unnecessary preemptions, reduces the number of tasks since a group of non-preemptive tasks can be regarded as a unit, and eventually helps allow for scalable real-time system design.

In the meantime, there have been several research activities to integrate schedulability analysis techniques into object-oriented design methodologies [15, 16] based on the ROOM (Real-time Object Oriented Modeling) methodology [18]. The integration is to provide many other advantages of object-oriented design methods and software engineering tools for the automated implementation of real-time embedded control systems from ROOM-based design models. The most difficult part of this integration is how to automatically produce a thread-based implementation from a real-time object model with timing constraints. Saksena, et al. initiate such a method using one-to-one mapping between objects and tasks for schedulability [15] and improve the performance using preemption threshold scheduling to reduce the adverse effects of context switching in an automated implementation [2].

1.2 Approaches and Contributions

A real-time system can be viewed as a collection of concurrent objects that cooperate with each other. Each object may participate in multiple system functions, and as a result, is subject to multiple timing constraints. In this sense, there are two extremes in mapping between objects and tasks. One is to map all objects into a single task as in RoseRT for UML-RT [19] and ObjecTime for ROOM [18]. Although this is a practical approach to reduce blocking time due to priority inversion, the system cannot be analyzed by the fixed-priority scheduling theory. The alternative is to map each object into a single task in order to perform timing analysis within the object-oriented design techniques as in [15]. Since multi-tasking in this multi-threaded implementation is expensive, the preemption threshold scheduling is adopted in [1].

In our previous work, we presented a systematic schedulability-aware method that can generate a multi-threaded implementation from a given real-time object-oriented design model [3, 4]. Unlike the above mentioned approaches, the mapping relationship between objects and tasks is not biased to many-to-one or one-to-one in our approach. Tasks are rather automatically identified from a set of objects. Our method is a three-step process: (1) deriving scenarios (end-to-end computation units), (2) identifying logical threads, and then (3) deriving physical threads. Logical threads are mapped from mutually exclusive scenarios and assigned priorities and preemption thresholds that guarantee schedulability. To reduce the number of tasks, those logical threads are partitioned into a mutually exclusive group of non-preemptive ones. A group of non-preemptive logical threads is in fact a physical thread.

Since our approach is primarily based upon real-time synchronization under the preemption threshold scheduling, it needs to be addressed in a comprehensive manner. Note that an object-oriented design produces a number of object locks for consistency of object states and for maintenance of the run-to-completion semantics of a finite state machine inside each object. Unfortunately, real-time synchronization under the preemption threshold scheduling has not been considered yet. In this paper, we consider the problem of integrating three real-time synchronization schemes into the preemption threshold scheduling: they are the basic priority inheritance protocol (BPI) [5], the priority ceiling protocol (PCP) [5], and the immediate priority inheritance protocol (IIP) [9]. In doing so, we introduce the notion of effective priority inheritance and define priority ceiling and preemption threshold ceiling for reducing run-time overhead. We also investigate their schedulability analyses.

The remainder of the paper is organized as follows. Section 2 describes the task model with proper definitions for the discussion. In Section 3, we identify the priority inversion problem under PTS and present the revised BPI protocol under PTS. In Sections 4 and 5, we present the properties of PCP and IIP under PTS, and the schedulability analyses of the proposed protocols, respectively. We conclude this paper in Section 6.

2. TASK MODEL

The task model used here is very similar to that used in [1, 2, 5]. We assume a uniprocessor environment and allow only properly nested mutexes. We further assume a system with a fixed set of tasks, each of which has a fixed period, known worst-case execution time, fixed priority, and preemption threshold. We denote a higher priority with a larger value since this befits the intuitive meaning of being a higher threshold. The notations and their descriptions used throughout the paper are summarized in Table 1.

Table 1. Summary of notations for the task model.

Notation	Description
τ_i	A task
T_i	The period of task τ_i
C_i	The worst-case execution time of task τ_i
π_i	The fixed-priority of task τ_i
γ_i	The preemption threshold of task τ_i
p_i	The effective priority of task τ_i
M_i	A mutex (binary semaphore)
$P(M_i)$, $V(M_i)$	Indivisible lock and unlock operation of mutex M_i
$\varphi(M_i)$	The ceiling of mutex M_i
Ψ_i	The set of tasks that may use mutex M_i
Φ_i	The set of mutexes that task τ_i may use
ξ_i	The set of mutexes that are currently locked by task τ_i
$z_{i,k}$	The duration of critical section of task τ_i guarded by mutex M_k
β_i	The blocking time of task τ_i due to synchronization
B_i	The PTS blocking time of task τ_i
S_i	The start time of task τ_i
F_i	The finish time of task τ_i
R_i	The response time of task τ_i

Under PTS, each task has a preemption threshold in addition to its regular priority. Note that it is meaningful to assign a task a preemption threshold that is no less than its regular priority since a preemption threshold is used as an effective run-time priority to control unnecessary preemptions. Since the effective priority of a task is changed at run-time due to priority inheritance and task dispatching under PTS, it is desirable to precisely define it. Conceptually, the effective priority of a task is the priority that is used by the kernel scheduler for selecting a task to be dispatched. Under PTS, effective priorities vary according to task states. We define it in an operational manner as below.

- Effective priority p_i of τ_i =
  π_i if τ_i is released in its period and not yet dispatched;
  otherwise, $\max(\gamma_i, p_1, p_2, \ldots, p_j)$ such that $\tau_1, \tau_2, \ldots, \tau_j$ are tasks blocked by τ_i.

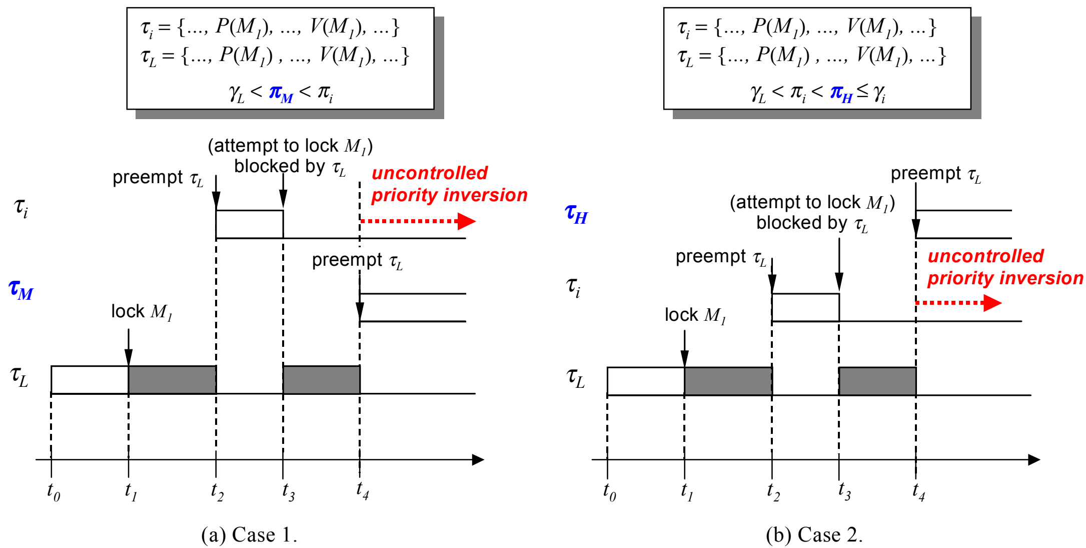
In traditional priority-based preemptive scheduling, tasks may experience blocking due to synchronization. Under PTS, tasks may encounter another type of blocking which we name PTS blocking. Task τ_i is said to be in PTS blocking if it is blocked by a lower priority task whose preemption threshold is higher than π_i. We denote the duration of PTS blocking by B_i while the duration of other types of blocking by β_i.

3. PREVENTING THE PRIORITY INVERSION PROBLEM

Without using priority inheritance protocols, catastrophic priority inversion problems cannot be rectified in a real-time system that uses synchronization primitives. The basic priority inheritance protocol (BPI) prevents preemptions that eventually cause priority inversion. In this section, we identify the priority inversion problem under PTS, formulate the BPI protocol under PTS, and then describe its properties.

3.1 Priority Inversion Problem under PTS

The priority inversion problem in traditional priority-based scheduling occurs when a medium priority task preempts a lower priority task that blocks a higher priority task. Under PTS, we reformulate the original problem into the *effective priority inversion* problem by substituting a priority of a task with its effective priority. The rationale behind this formulation is obvious since an effective priority takes the place of a priority under PTS. In this formulation, given that task τ_i is blocked by τ_L, priority inversion occurs (1) when a medium priority task τ_M with $\gamma_L < \pi_M < \pi_i$ preempts τ_L; or (2) when another task τ_H with $\pi_i < \pi_H \leq \gamma_i$ preempts τ_L. These cases are illustrated in Figure 1 (a) and (b), respectively.



(a) Case 1. (b) Case 2.

Figure 1. Priority inversion problem under PTS.

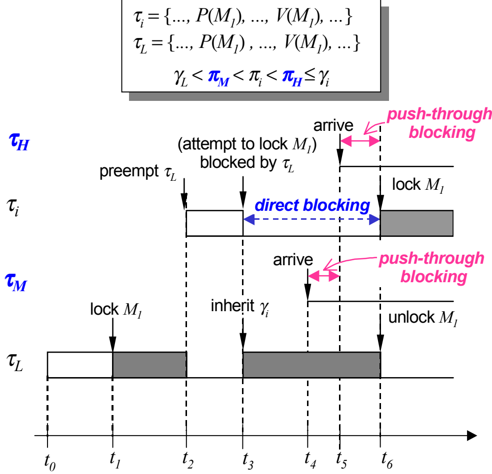


Figure 2. Prevention of priority inversion by BPI under PTS.

3.2 BPI under PTS

We define the BPI protocol under PTS using effective priorities as below.

Protocol 1: BPI under PTS.

- *Inheritance of effective priorities*.
  When task τ_H is blocked by task τ_L with $\pi_H > \gamma_L$, task τ_L inherits p_H from task $\tau_{H.}$
- *Recovery of effective priorities*.
  When task τ_L exits from a critical section, task τ_L recovers p_L that it had before entering that critical section.

We can easily see that this definition of BPI prevents preemptions that eventually cause priority inversion. Figure 2 shows an example that follows Protocol 1. Since neither does τ_M nor τ_H preempt τ_i, priority inversion is avoided.

3.3 Properties of BPI under PTS

BPI under PTS bears similar properties with the original BPI in [5]. In the original protocol, a task can be blocked in one of three forms of blocking: (1) *direct blocking*, (2) *push-through blocking*, and (3) *transitive blocking*. In our protocol, a task can be blocked additionally in PTS blocking. Direct blocking occurs when a higher priority task attempts to lock a mutex locked by a lower priority task. It is to ensure the consistency of a non-preemptible shared resource. Push-through blocking occurs under PTS when a medium priority task attempts to preempt a lower priority task that is blocking a higher priority task, as described below.

- *Push-through blocking under PTS*.

  Consider three tasks τ_L, τ_M, τ_H with $\gamma_L < \pi_M$, $\gamma_L < \pi_H$, and $\pi_M \leq \gamma_H$. If task τ_M is blocked by task τ_L that already blocked τ_H, this situation is referred to as push-through blocking under PTS.

In the above definition, task τ_M falls into push-through blocking since the effective priority of task τ_L is greater than that of τ_M due to effective priority inheritance. Push-through blocking is introduced to prevent preemptions that cause priority inversion. In Figure 2, tasks τ_M and τ_H are blocked in push-through blocking during the period of (t_4, t_5) and (t_5, t_6), respectively.

Finally, transitive blocking occurs when task τ_H is blocked by τ_M which, in turn, is blocked by another task τ_L. Figure 3 (a) shows an example for transitive blocking. During the period of (t_6, t_7), τ_M is blocked by τ_L while blocking τ_H. Therefore, τ_H is indirectly blocked by τ_L in a transitive manner. As such, transitive blocking occurs when mutexes are accessed in a nested fashion.

Additionally, a task can encounter *chained blocking*. Chained blocking (or a chain of blocking) is said to occur if a task is repeatedly blocked when it enters its critical sections. An example for chained blocking is illustrated in Figure 4 (a) where task τ_H is blocked during the period of (t_6, t_7) and blocked again during (t_9, t_{10}). Note that chained blocking is not a form of blocking, but refers to a situation where a task is blocked more than once.

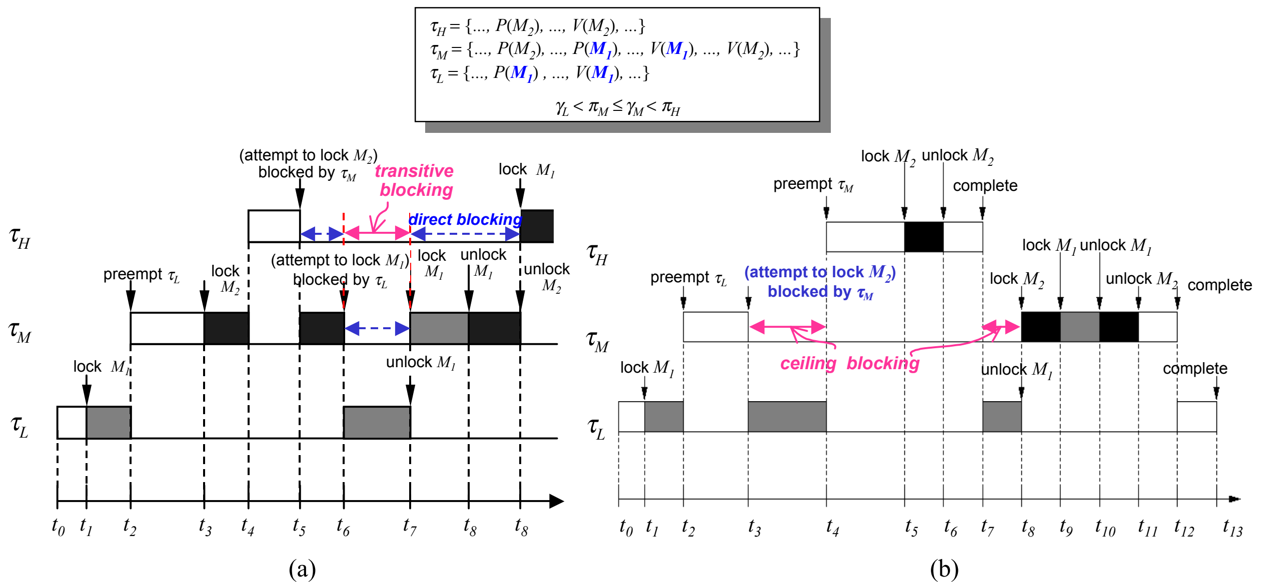


Figure 3. (a) Transitive blocking in BPI, (b) transitive blocking prevention in PCP.

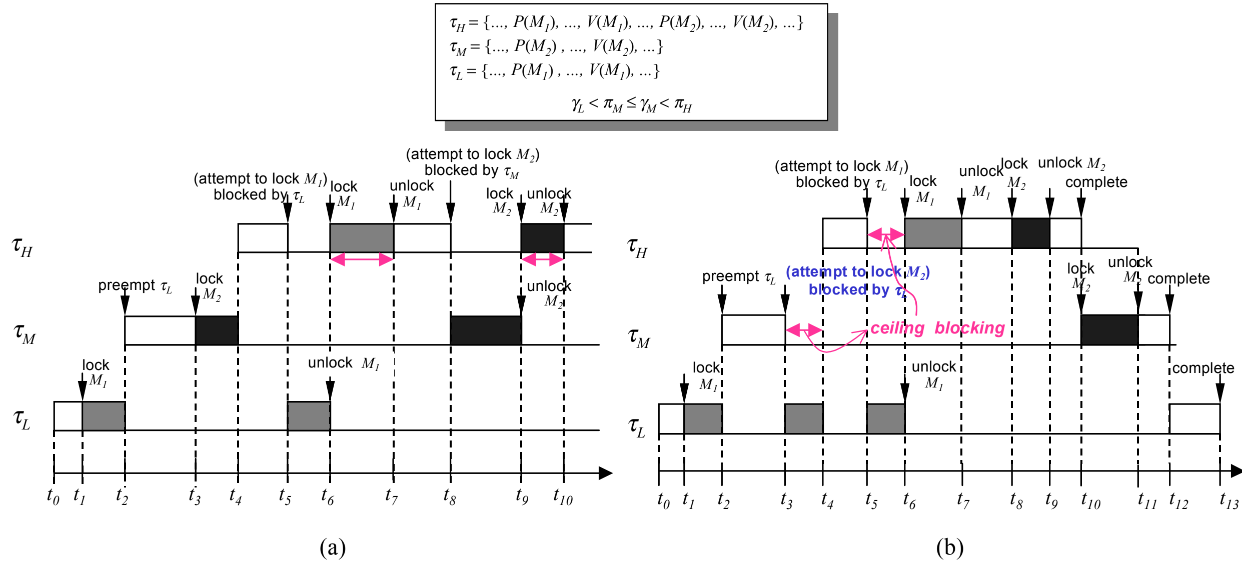


Figure 4. (a) Chained blocking in BPI, (b) chained blocking prevention in PCP.

Note that the BPI protocol alone does not prevent deadlock. Deadlock may occur when multiple tasks try to access nested mutexes in a circular manner. Figure 5 shows an example for deadlock. In this example, both tasks τ_H and τ_L need to acquire two mutexes M_1 and M_2. As shown, τ_L waits for τ_H to release M_2, and τ_H waits for τ_L to release M_1 simultaneously, while τ_L and τ_H are holding M_1 and M_2, respectively. Therefore, a deadlock occurs at time t_5.

4. PREVENTING DEADLOCK, TRANSITIVE BLOCKING, AND CHAINED BLOCKING

Although BPI prevents priority inversion by means of push-through blocking, it may incur deadlock and excessively long blocking delay due to transitive and chained blocking, as discussed in Section 3.3. The priority ceiling protocol (PCP) was introduced to solve these problems [5]. In this section, we define two versions of PCP under PTS: one with priority ceilings and the other with preemption threshold ceilings. We refer to the former as PC-PCP and the latter as PTC-PCP. Then we investigate their properties, compare them, and present schedulability analysis.

4.1 PCP under PTS with Priority Ceilings

The underlying idea of the original PCP algorithm is to allow a task to lock a mutex only if it can make sure that all mutexes that the task and its higher priority tasks may use are not locked by lower priority tasks. If not, it forces the task to wait for those mutexes to be unlocked. To realize this idea, Rajkumar et al. introduced the notion of a priority ceiling and associated it with each mutex [5].

We define PCP under PTS with priority ceilings by combining an

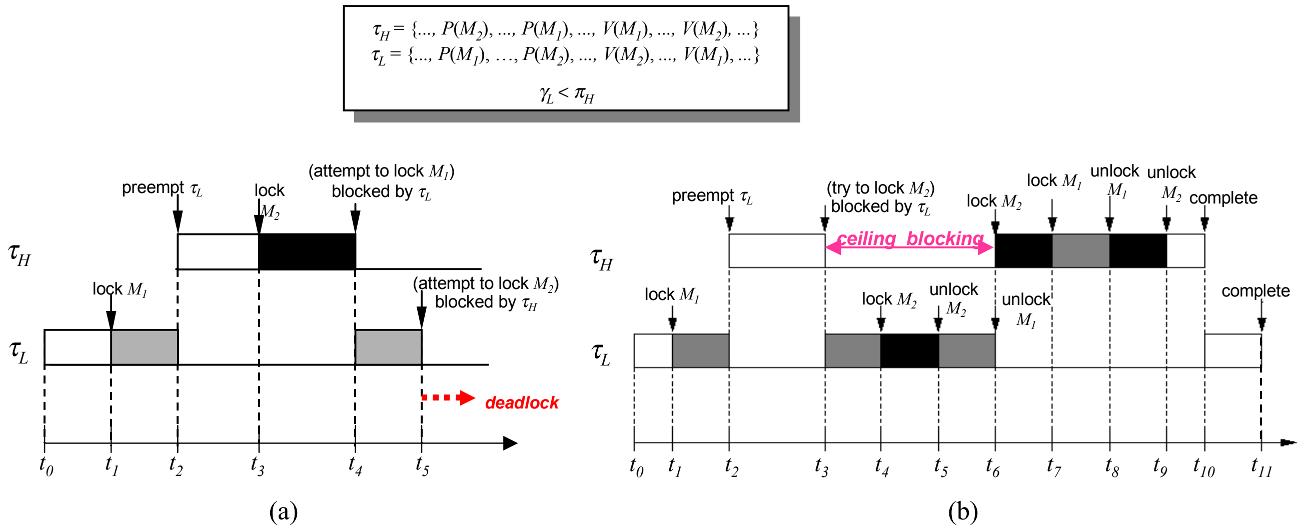


Figure 5. (a) Deadlock in BPI, (b) deadlock prevention in PCP.

offline ceiling protocol and an online locking protocol in a similar manner as in [6].

Protocol 2: PC-PCP.

- *Offline ceiling protocol*
  CP. Each mutex M_i is assigned a priority ceiling whose value is given by $\varphi(M_i) = max\{\pi_j | \tau_j \in \Psi_i\}$, where Ψ_i is a set of tasks that may use mutex M_i.
- *Online locking protocol*
  LP1. A *system ceiling*, which is a system-wide scheduling attribute, is dynamically set to the maximum of priority ceilings of all mutexes being locked in the system.
  LP2. In order for task τ_i to enter its critical section, π_i should be higher than the system ceiling.
  LP3. *Inheriting effective priorities*
  If high priority task τ_H is blocked by low priority task τ_L with $\pi_H > \gamma_L$, task τ_L inherits p_H from task τ_H as its effective priority.
  LP4. *Recovering effective priorities*
  When task τ_L exits from a critical section, task τ_L recovers p_L that it had before entering that critical section.

In this definition of PCP under PTS, effective priorities are used for priority inheritance (as specified in **LP3** and **LP4**), while regular priorities are used for ceilings of mutexes (as specified in **CP** and **LP2**). It is quite straightforward to use effective priorities for priority inheritance since it is a mere adoption of BPI to solve the priority inversion problemWe now show that PC-PCP prevents deadlock, transitive blocking, and chained blocking. To begin with, we introduce a locking rule that can guarantee the avoidance of deadlock, transitive blocking, and chained blocking. Then, we show our PCP definition satisfies this rule.

Locking Rule 1. Task τ_i is allowed to lock a mutex only if all mutexes that τ_H may use are not locked by lower priority tasks, where $\pi_H \geq \pi_i$.

Theorem 1. Conforming to Locking Rule 1 guarantees the prevention of deadlock, transitive blocking, and chained blocking.

Proof.
Conforming to Locking Rule 1 ensures that a task cannot enter its critical section until all mutexes it may use are unlocked. Therefore, circular waiting cannot occur, and thus deadlock is prevented.

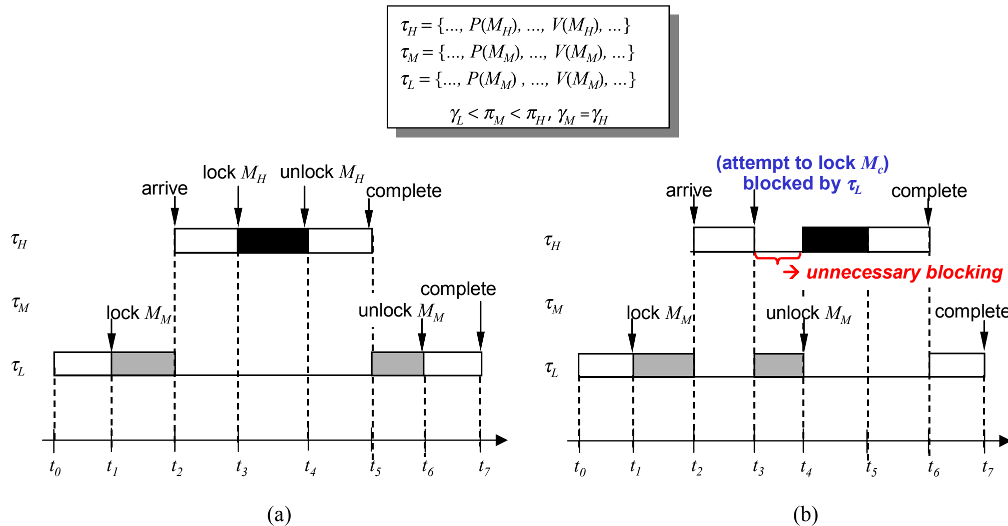
To show that conforming to Locking Rule 1 guarantees the prevention of transitive and chained blocking, we show that a task is blocked at most once by at most one task under the Locking Rule 1. Suppose that task τ_i attempts to lock a mutex, while there is preempted task τ_L that has locked mutexes that may be used by higher priority or preemption threshold task τ_H where $\pi_H \geq \pi_i$. Then, the Locking Rule 1 forces task τ_i to get blocked first and wait for τ_L to unlock those mutexes. Consequently, there is always 'at most only one' such task that locks mutexes that may be used by τ_H where $\pi_H \geq \pi_i$. Accordingly, whenever a task arrives, it sees at most one (lower priority) task that has acquired mutexes it may use. Therefore, it is guaranteed that a task is blocked (to acquire its mutexes) 'at most once' (when it first tries to lock any mutex) and by 'at most one' lower priority task. Therefore, transitive and chained blocking cannot happen.

Theorem 2. PC-PCP conforms to Locking Rule 1.

Proof.
Suppose that the PC-PCP does not conform to Locking Rule 1. Then, task τ_i may lock a mutex when there is a locked mutex M_H that may be used by task τ_H where $\pi_H \geq \pi_i$. However, according to **CP** of Protocol 2, the priority ceiling of M_H, $\varphi(M_H)$, is equal to π_H. When task τ_i attempts to lock any mutex, the system ceiling equals $\varphi(M_H) = \pi_H$ by **LP1**. Therefore, τ_i should be blocked, since π_i is not higher than π_H by **LP2**. This is a contradiction.

Figure 3 (b), Figure 4 (b), and Figure 5 (b) illustrate how PCP prevents transitive blocking, chained blocking, and deadlock, respectively.

We have seen that in PCP, a task can be blocked only when it tries to enter its critical section for the first time: once a task successfully acquires any of its mutexes, it will never get blocked again. This obviously implies that a task can be blocked at most once by at most one lower priority task.



(a) (b)

Figure 6. Examples: PCP under PTS using (a) priority ceilings, and (b) preemption threshold ceilings.

In PCP under PTS, a task can encounter one of four forms of blocking: (1) *direct blocking*, (2) *push-through blocking*, (3) *ceiling blocking* [5], and (4) *PTS blocking*. A task encounters ceiling blocking when it attempts to enter a critical section with its priority not greater than the system ceiling. Figure 3 (b), Figure 4 (b), and Figure 5 (b) show examples of ceiling blocking in PTS. This helps prevent deadlock, transitive blocking, and chained blocking.

4.2 PCP under PTS with Preemption Threshold Ceilings

It is not obvious why we used regular priorities to define ceilings of mutexes instead of preemption threshold ceiling in Section 4.1. In fact, we can use preemption threshold ceilings for that purpose and define another version of PCP under PTS, accordingly. The result is PTC-PCP. It is the same as PC-PCP of Protocol 2 except that the priorities of **CP1** and **LP2** of Protocol 2 are replaced with preemption thresholds.

Locking Rule 2. Task τ_i is allowed to lock a mutex only if all mutexes that τ_H may use are not locked by lower priority tasks, where $\gamma_H \geq \gamma_i$.

Theorem 3. Conforming to Locking Rule 2 guarantees the prevention of deadlock, transitive blocking, and chained blocking.

Proof.
This can be proved in a manner similar to Theorem 1 by substituting priorities with preemption thresholds.

Theorem 4. PTC-PCP conforms to Locking Rule 2.

Proof.
This can be proved in a manner similar to Theorem 2 by substituting priority ceilings of Theorem 2 with preemption threshold ceilings and Locking Rule 1 with Locking Rule 2.

4.3 Comparison between PC-PCP and PTC-PCP

Now we will show that PC-PCP performs better than PTC-PCP with respect to run-time costs and the average response time of a task set. The latter may (1) incur unnecessary context switches and (2) lead to longer response times due to unnecessary blockings.

In most cases of practical interest, a task has a preemption threshold that is no less than its fixed priority so as to avoid unnecessary context switches. In [1], a task is even assigned the maximum possible preemption threshold. This is indeed a desirable heuristic to reduce unnecessary preemptions in preemptive fixed-priority scheduling.

However, this heuristic may not work well in PTC-PCP. In PTC-PCP, with this heuristics, the system ends up with a large number of tasks assigned the maximum preemption threshold, which often forces high priority tasks to seriously suffer from ceiling blocking. Figure 6 demonstrates the situation. Suppose there are three tasks τ_L, τ_M, and τ_H with $\gamma_L < \pi_M < \pi_H$ and $\gamma_M = \gamma_H$. At t_1, task τ_L acquires M_M, and then task τ_H arrives and preempts task τ_L at t_2. At t_3, it attempts to acquire M_H. At that point in time, the system ceiling is equal to $\varphi(M_M)$.

If we adopt priority ceilings (PC-PCP) as depicted in Figure 6 (a), the system ceiling at t_3 is π_M. Since $\pi_H > \pi_M$, task τ_H acquires M_H and continues to run at t_3. On the other hand, if we adopt preemption threshold ceilings (PTC-PCP) as depicted in Figure 6 (b), the system ceiling at t_3 is γ_M. Since γ_H is not greater than γ_M, task τ_H gets blocked at t_3 and resumes at t_4. Thus, two extra context switches occur at t_3 and t_4. Moreover, the response time of task τ_H becomes longer in Figure 6 (b) than in Figure 6 (a) due to unnecessary blocking at t_3.

When running task τ_i tries to enter its critical section in PTS, there is a case where τ_H is blocked unnecessarily only in PTC-PCP: when the preempted task τ_L with $\gamma_L < \pi_H$ has acquired mutex M_l, which may be requested by τ_l with $\pi_l < \gamma_H$ and $\gamma_l \geq \gamma_H$. This type of ceiling blocking does not contribute to preventing deadlock, transitive blocking or chained blocking since this blocking is for mutexes that may be requested by lower priority tasks. For every such case, there are two additional context switches in PTC-PCP as compared to PC-PCP. Note that such cases tend to be very frequent due to maximum preemption threshold assignment policy in PTS.

From this observation, we recommend PC-PCP for obvious reasons: to avoid unnecessary context switches and to reduce the response times of higher priority tasks.

4.4 Schedulability Analysis of PC-PCP

For the schedulability analysis of PC-PCP, we adopt the worst-case response time analysis in [10]. Under PTS, the sets of interfering higher priority tasks before and after a task gets the CPU, are different. Therefore, for each task τ_i, we should analyze its start time S_i and finish time F_i separately. The equations for the response time analysis of PCP under PTS for task τ_i are as follows.

$$\beta_i = \begin{cases} 0 & \text{if } \Phi_i = \phi \\ \max\limits_{\forall j, \gamma_j < \pi_i} \{ z_{j,k} :: \varphi(M_k) \geq \pi_i \} & \text{otherwise} \end{cases} \tag{1}$$

$$B_i = \max_{\forall j, \gamma_j \geq \pi_i > \pi_j} \{ C_j + \beta_j \} \equiv C_{Li} + \beta_{Li} \tag{2}$$

$$S_i^{n+1} = \max\{B_i, \beta_i\} + \sum_{\forall j, \pi_j > \pi_i} \left(1 + \left\lfloor \frac{S_i^n}{T_j} \right\rfloor \right) \cdot C_j \tag{3}$$

$$\delta_i = \begin{cases} 0 & \text{if } B_i = 0 \text{ or } \beta_{Li} \neq 0 \\ 1 & \text{otherwise} \end{cases} \tag{4}$$

$$F_i^{n+1} = S_i + C_i + \sum_{\forall j, \pi_j > \gamma_i} \left(\left\lceil \frac{F_i^n}{T_j} \right\rceil - \left(1 + \left\lfloor \frac{S_i}{T_j} \right\rfloor \right) \right) \cdot C_j + \beta_i \cdot \delta_i \tag{5}$$

$$R_i = F_i \tag{6}$$

In PCP, a task may be blocked once in one of three forms of blocking: push-through, direct, and ceiling blocking. The worst case blocking duration for these types of blocking for task τ_i (β_i) is formulated in Equation (1) where $z_{j,k}$ is the duration of the critical section of task τ_j guarded by mutex M_k. Note that if the set of mutexes that task τ_i may use is empty (Φ_i =0), the blocking time under PCP is zero. Additionally, in PTS, a task can experience PTS blocking. The worst-case duration of PTS blocking for task τ_i, is formulated in Equation (2).

The start time of task τ_i (S_i) is formulated in Equation (3): a task may be blocked once in *either* PTS blocking *or* push-through blocking before its execution since (1) both blockings are caused by the same running task when the task arrives and (2) the sets of tasks that can cause PTS blocking and push-through blockings are mutually exclusive, as shown in Equations (1) and (2).

The finish time of task τ_i (F_i) is formulated in Equation (5) where δ_i is calculated from Equation (4): task τ_i can be blocked once again during its execution, in either direct or ceiling blocking. As shown in Equation (4), this type of blocking ($\beta_i \cdot \delta_i$) becomes zero (1) if PTS blocking cannot occur ($B_i = 0$) or (2) if the PCP blocking time of the task causing the PTS blocking is not zero ($\beta_{Li} \neq 0$). Note that if task τ_i has encountered push-through blocking, or the task that caused PTS blocking (τ_{Li}) has encountered PCP blocking ($\beta_{Li} \neq 0$), direct or ceiling blocking cannot occur.

5. MINIMIZING CONTEXT SWITCHES

In PCP, a task can be blocked once when it attempts to enter a critical section. Alternatively, if we give a task a chance to be blocked when it is released for the first time in its period, it will never be blocked again during its execution. If we choose to use this approach, we can reduce two context switches associated with each blocking of a task in PCP. This alternative, which is already widely used in practice, is called the immediate inheritance protocol (IIP) or the PCP emulation protocol with flag _POSIX_THREAD_PRIO_PROTECT in IEEE POSIX 1003.1c [9]. In this section, we define two versions of IIP under PTS: one with priority ceilings and the other with preemption threshold ceilings. We refer to the former as PC-IIP and the latter as PTC-IIP. Then we investigate their properties, compare them, and present schedulability analysis.

5.1 IIP under PTS with Priority Ceilings

The underlying idea of the original IIP algorithm is to allow a task to *start its execution* only if it can make sure that all mutexes that the task and its higher priority tasks may use are not locked by lower priority tasks. If not, it forces the task to wait for those mutexes to be unlocked.

IIP bears many similarities and benefits with PCP. It can prevent priority inversion, deadlock, transitive blocking, and chained blocking. Moreover, it can reduce the number of context switches without sacrificing the worst-case response time.

As in PCP, we can define two versions of IIP: PC-IIP and PTC-IIP. PC-IIP uses effective priorities for priority inheritance and regular priorities for priority ceilings of mutexes. Its offline ceiling protocol and online locking protocol are presented below.

Protocol 3: PC-IIP.

- *Offline ceiling protocol*
    - **CP.** Each mutex M_i is assigned a priority ceiling given by $\varphi(M_i) = max\{\boldsymbol{\pi_j} | \tau_j \in \Psi_i\}$ where Ψ_i is a set of tasks that may use mutex M_i.
- *Online locking protocol*
    - **LP1.** *Inheriting ceiling*
      If task τ_c acquires a mutex, its effective priority is set to $p_c = max\{p_c , \varphi(M_i) \mid M_i \in \xi_c\}$ where ξ_c is a set of mutexes that are currently locked by τ_c.
    - **LP2.** *Recovering effective priorities*
      When task τ_c exits from a critical section, task τ_c recovers p_c that it had before entering that critical section.

While the offline ceiling protocol (**CP**) is the same as that of PC-PCP, the locking protocol (**LP1** and **LP2**) is much more simplified. This is because IIP forces priority inheritance whenever a task locks a mutex while PCP does so only when a task blocks a higher priority task. It is obvious that our PC-IIP definition prevents priority inversion, deadlock, transitive blocking, and chained blocking since IIP is merely a special case of PCP. Later, we compare IIP with PCP and discuss the advantages of IIP over PCP.

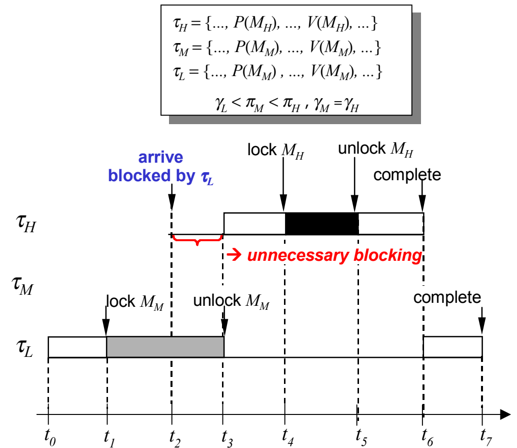


Figure 7. Example: IIP under PTS with preemption threshold ceiling.

5.2 IIP under PTS with Preemption Threshold Ceilings

As mentioned above, the IIP under PTS can be defined with either priority ceilings (PC-IIP) or preemption threshold ceilings (PTC-IIP). We compare these two versions of IIP definitions using the same example task set shown in Section 4.3. Figure 6 (a) can be reused to illustrate IIP with priority ceilings. Figure 7 shows an example of PTC-IIP. As in PCP under PTS, PTC-IIP yields extra blockings, thus makes the response times of higher priority tasks longer. On the other hand, note that the context-switching overhead remains the same in the two versions since blocking in IIP does not accompany context switches. As in PCP, we recommend PC-IIP over PTC-IIP.

5.3 IIP vs. PCP

The most striking difference between IIP and PCP is that a task in IIP can encounter only ceiling blocking: there is neither direct blocking nor push-through blocking in IIP. The number of context switches incurred by IIP can be as low as a half of the number incurred by PCP. The best efficiency under IIP occurs when every task in the system enters at least one critical section. On the other hand, if quite a few tasks do not enter a critical section at all, IIP may yield extra blockings.

In either case, the number of context switches in IIP is always lower than in PCP. This is because blocking in IIP does not accompany context switches since it always occurs before a task starts execution. However, it is also true that the extra blockings encountered by high priority tasks increase their average response times in IIP.

5.4 Schedulability Analysis of PC-IIP

We provide the schedulability analysis for PC-IIP. In IIP under PTS, a task can be blocked only once before its start time. Since a task can be blocked even if it does not enter any critical section before its start time, the equation for blocking time β_i in IIP is formulated as Equation (7). On the other hand, since a task is not blocked during its execution, the PTS blocking in IIP under PTS is formulated as Equation (8) and the finish time is formulated as Equation (9). Without these, the formulations for S_i and R_i in Equations (4) and (6) in Section 4.3 are adopted as they are.

$$\beta_i = \max_{\forall j, \gamma_j < \pi_i} \{ z_{j,k} :: \varphi(M_k) \geq \pi_i \} \tag{7}$$

$$B_i = \max_{\forall j, \gamma_j \geq \pi_i > \pi_j} \{ C_j \} \tag{8}$$

$$F_i^{n+1} = S_i + C_i + \sum_{\forall j, \pi_j > \gamma_i} \left(\left\lceil \frac{F_i^n}{T_j} \right\rceil - \left(1 + \left\lfloor \frac{S_i}{T_j} \right\rfloor \right) \right) \cdot C_j \tag{9}$$

6. CONCLUSIONS

In spite of the proliferation of object-oriented design methodologies in contemporary software development, their application to real-time embedded systems has been limited for the lack of proper real-time scheduling theory that can be seamlessly integrated into these methods. Specifically, popular preemptive fixed priority scheduling cannot be directly used in real-time object-oriented design methodologies largely owing to its excessive run-time overhead. The preemption threshold scheduling has been recently suggested to this end since it improves both run-time overhead and schedulability. Unfortunately, it lacks real-time synchronization capabilities. To solve this problem, in this paper, we have adopted three well-known real-time synchronization protocols and integrated them into the preemption threshold scheduling. They are the basic priority inheritance protocol (BPI), the priority ceiling protocol (PCP), and the immediate inheritance protocol (IIP). We have also presented their schedulability analyses.

Since each task under PTS has two scheduling attributes, it is not intuitive during the integration which scheduling attribute should be used for priority inheritance and for priority ceilings of mutexes. To clarify this problem, we have introduced the notion of the effective priority, which is conceptually a task priority that is used by the kernel scheduler for selecting a task to be dispatched. With this, we have identified the priority inversion problem under PTS and presented the BPI protocol under PTS that avoids such priority inversion.

Since BPI alone cannot prevent deadlock, we have also presented PCP and IIP under PTS. In these protocols, we used effective priorities for priority inheritance and regular priorities for priority ceilings of mutexes. We have shown that two protocols with priority ceilings yield the smaller number of context switches and shorter response times for higher priority tasks than those with preemption threshold ceilings. We have also shown that IIP further reduces the number context switches compared to PCP.

Currently, we are implementing a RoseRT-based CASE tool that is capable of deriving tasks from an object-oriented design model. We are integrating into the CASE tool the preemption threshold scheduling and the IIP algorithm proposed in this paper. We are also conducting extensive experiments to show the viability of our approach. The results look promising.

7. ACKNOWLEDGEMENTS

The authors would like to thank Jamison Allen Masse for his fruitful comments on drafts of this article. The comments from the anonymous reviewers further improved the quality. The work reported in this paper is supported in part by MOST under the National Research Laboratory (NRL) grant 2000-N-NL-01-C-136, by Automation and Systems Research Institute (ASRI), and by Automatic Control Research Center (ACRC).

8. REFERENCES

[1] M. Saksena and Y. Wang. Scalable real-time system design using preemption thresholds, In Proceedings of IEEE Real-Time Systems Symposium, pp. 25 -34, 2000.

[2] Y. Wang and M. Saksena. Scheduling fixed priority tasks with preemption threshold. In Proceedings of IEEE Real-Time Computing Systems and Applications Symposium, pp. 328-335, 1999.

[3] S. Kim, S. Cho, and S. Hong. Schedulability-aware mapping of real-time object-oriented models to multi-threaded implementations. In Proceedings of International Conference on Real-Time Computing Systems and Applications, pp. 7-14, 2000.

[4] S. Kim, S. Hong, and N. Chang, Scenario-based implementation architecture for real-time object-oriented models, In Proceedings of IEEE International Workshop on Object-oriented Real-time Dependable Systems, 2002.

[5] R. Rajkumar, L. Sha and J. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. In IEEE Transactions on Software Engineering, vol. 39, pp. 1175-1185, 1990.

[6] M. Chen and K. Lin. Dynamic priority ceilings: A concurrencly control protocol for real-time systems. In Real-Time Systems Journal, vol. 2, pp. 325-346, 1990.

[7] T. P. Baker. A stack-based resource allocation policy for real-time processes. In *Proceedings of IEEE Real-Time Systems Symposium*, pp. 191-200, 1990.

[8] T. P. Baker. Stack-based scheduling of real-time processes. In *Real-Time Systems Journal*, vol. 3, num. 1, pp. 67-99,1991.

[9] Institute for Electrical and Electronic Engineers. In IEEE Std. 1003.1c-1995 POSIX Part 1: System application program interface - amendment 2:threads extension, 1995.

[10] K. Tindell, A. Burns, and A. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. In *Real-Time Systems Journal*, vol. 6, pp. 133-151, 1994.

[11] C. Liu and J. Layland, Scheduling algorithm for multiprogramming in a hard real-time environment. In Journal of the ACM, vol. 20(1), pp. 46-61, Jan. 1973.

[12] W. Lamie, Preemption-Threshold, White Paper, Express Logic Inc., Available at http://www.threadx.com/wppreemption.html

[13] M. Harbour, M. Klein, and J. Lehoczky, Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority, In *Proceedings of IEEE Real-Time Systems Symposium*, pp. 116-128, Dec. 1991.

[14] J. Lehoczky, L. Sha, and Y. Ding, The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *Proceedings of IEEE Real-Time Systems Symposium*, pp. 166-171, Dec. 1989.

[15] M. Saksena, P. Freeman, and P. Rodziewicz, Guidelines for Automated Implementation of Executable Object Oriented Models for Real-Time Embedded Control Systems, In *Proceedings of IEEE Real-Time Systems Symposium*, pp. 240-251, Dec. 1997.

[16] M. Saksena, A. Ptak, P. Freeman, and P. Rodziewicz, Schedulability Analysis for Automated Implementations of Real-Time Object-Oriented Models, In *Proceedings of IEEE Real-Time Systems Symposium*, pp. 92-102, Dec. 1998.

[17] H. Gomma, Software Design Methods for Concurrent and Real-Time Systems, *Addison-Wesley*, 1993.

[18] B. Selic, G. Gullekson, and P. T. Ward, Real-Time Object-Oriented Modeling. *John Wesley and Sons*, 1994.

[19] Rational Software, http://www.rational.com.