Dual ceiling protocol for real-time synchronization under preemption threshold scheduling

Saehwa Kim

Information Communications Engineering Major, Hankuk University of Foreign Studies, Yongin-si, Gyeonggi-do, 449-791, Republic of Korea

A B S T R A C T

The application of object-oriented design methods to real-time embedded systems is seriously hindered by the lack of existing real-time scheduling techniques that can be seamlessly integrated into these methods. Preemption threshold scheduling (PTS) enables a scalable real-time system design and thus has been suggested as a solution to this problem. However, direct adoption of PTS may lead to long priority inversion since object-oriented real-time systems require synchronization considerations in order to maintain consistent object states. In this paper, we propose the dual ceiling protocol (DCP) in order to solve this problem. While DCP exploits both priority ceilings and preemption threshold ceilings, this is not a straightforward integration of existing real-time synchronization protocols for PTS. We present the rationale for the locking conditions of DCP and show that it leads to the least blocking and response times by comparison with other real-time synchronization protocols. We also present its blocking properties and schedulability analyses. We implemented PTS and DCP in a real-time object-oriented CASE tool and present the associated experimental results, which show that the proposed protocol is a viable solution that is superior to other real-time synchronization protocols for PTS.

© 2010 Elsevier Inc. All rights reserved.

1. Introduction

Real-time embedded systems are becoming extremely complex and sophisticated with the broadening of their application domain, due to the rapid convergence of automotive, consumer electronics, telecommunication, and computer technologies. In order to meet the enhanced safety, reliability, and performance requirements of such real-time embedded systems, it is inevitable that real-time embedded system developers will exploit well-founded systematic methods for system design, synthesis, and tuning at various stages of system development. Particularly, in the late stages, real-time embedded system developers need rigorous design analysis and implementation techniques including real-time scheduling theory and run-time system modeling, since violations of performance requirements and resource constraints in complex real-time embedded systems often appear in the late phases of system development. Without those rigorous techniques, real-time embedded system developers must resort to a series of labor intensive and erroneous system tuning processes. Such processes include hand-tweaking of task code, re-assigning task priorities, and re-implementing problematic tasks as a last resort.

Recently, preemption threshold scheduling (PTS) [1–5] has attracted the attention of real-time system practitioners, since it facilitates system tuning processes. PTS is an extension of preemptive fixed-priority scheduling; each task has an extra scheduling attribute, called a preemption threshold, in addition to a priority. The preemption threshold of a task is its run-
time priority, which is maintained from when the task is dispatched until its execution terminates, so it regulates the degree of preemptiveness in fixed priority scheduling. If the threshold of each task is the same as its original priority, then PTS is equivalent to preemptive fixed priority scheduling, and if each task has the highest threshold in a system, it is equivalent to non-preemptive scheduling. PTS is very effective in system tuning processes, since it enhances real-time schedulability, eliminates unnecessary preemptions, reduces the number of tasks since a group of non-preemptive tasks can be regarded as a single task, and enables a scalable real-time system design [1].

However, PTS cannot be directly used in complex real-time embedded systems, since real-time synchronization problems remain unsolved. To solve this problem, we proposed various real-time synchronization protocols while investigating various real-time synchronization problems under PTS [2]. Specifically, we integrated the priority ceiling protocol (PCP) [3] into PTS. Since tasks in PTS have dual scheduling attributes, it is not obvious how these two scheduling attributes should be used for locking conditions. We proposed two protocols, namely PC-PCP (PCP with priority ceiling) and PTC-PCP (PCP with preemption threshold ceiling) [2]; the former employs priority ceilings while the latter employs preemption threshold ceilings. We also showed that PC-PCP is better than PTC-PCP in terms of task response times, since the latter may incur unnecessary blockings [2].

In this paper, we propose a more advanced but still practical real-time synchronization protocol, namely dual ceiling protocol (DCP). DCP exploits both priority ceilings and preemption threshold ceilings but it is not a straightforward integration of PC-PCP and PTC-PCP. We present the rationale for the locking conditions of DCP by comparison with those of other protocols, and obtain insights into the underlying principles of real-time synchronization protocols. We show that DCP leads to the least blocking and minimum response times compared to other real-time synchronization protocols. We also provide its blocking properties and schedulability analyses based on worst-case response time analyses.

We implemented PTS and DCP in a real-time object-oriented CASE tool and present the associated experimental results. The implementation established that the implementation complexity is the same as that of PC-PCP and PTC-PCP. The runtime overhead is also the same as that of the other protocols. As a task set application, we used an industrial private branch exchange (PBX) system [4,5], which was also implemented in a real-time object-oriented language based on UML 2.0 [6]. The results show that DCP leads to least blocking times for tasks and also leads to reduced response times of higher priority tasks.

1.1. Related work

The notion of preemption thresholds was introduced by Lamie and a complete scheduler mechanism was implemented in the ThreadX kernel from Express Logic [7] and the SSX kernel from REALOGY [8]. Saksena and Wang formulated PTS with its schedulability analysis and accompanying characteristics such as a non-preemptive relationship [19]. As PTS became known as an effective scheduling policy that can solve the schedulability problem of preemptive fixed priority scheduling, there were various research activities that aimed to apply PTS to real-time embedded systems. Representative examples are solution space search for priorities and preemption thresholds [10], fault tolerance mechanisms [11], low-power scheduling [12], and application to SoC [13].

Real-time synchronization protocols under PTS were addressed in [13] and in our previous work [2]. While [13] applied the stack-based resource allocation policy (SRP) [14] to PTS in the context of dynamic-priority scheduling, this protocol reduces to PC-PCP and PTC-PCP of our previous work [2] when it is applied to fixed-priority scheduling. In dynamic-priority scheduling, there is no fixed priority for each task and thus there is no way to determine the mutex ceilings. To solve this problem, SRP exploits the preemption level, which must be assigned to each task as a kind of virtual fixed priority. Preemption levels are defined such that a task with a lower preemption level cannot preempt a task with a higher one [14]. According to this definition, either priorities or preemption thresholds in PTS can be used as preemption levels. Consequently, if priorities are allowed to be preemption levels, then SRP becomes equivalent to PC-PCP; if preemption thresholds are allowed to be preemption levels, then SRP becomes equivalent to PTC-PCP. On the other hand, PC-PCP and PTC-PCP [2] are also inferior to the protocol proposed in this paper, since they lead to more blockings and larger response times; this is investigated in this paper.

The protocol proposed in this paper, namely DCP, is an extension of the effective priority inheritance protocol (EPI) in [2]. EPI is an adapted version of the basic priority inheritance protocol (BPI) in [3] for PTS. While EPI solves the uncontrolled priority inversion problem in PTS, as shown in [2], it can lead to deadlocks as with BPI. DCP is also an extension of the priority ceiling protocol (PCP) in [3] for PTS. To the best of our knowledge, PCP has been extended for PTS only in our previous work [2].

There have been several research activities directed towards integrating schedulability analysis techniques into object-oriented design methodologies [15–17] based on the ROOM (real-time object oriented modeling) methodology [18]. The goal of such integration is the automated synthesis of implementations that adhere to timing constraints from real-time object models. Saksena, et al. proposed a method that uses a one-to-one mapping between objects and tasks for schedulability [19] and improved the performance via PTS to reduce the adverse effects of context switching in the automated implementation [17]. In our previous work, we presented a systematic schedulability-aware method that can generate a multi-thread implementation from a given real-time object-oriented design model [4,5]. Unlike the aforementioned approaches, the mapping relationship between objects and tasks is not biased to many-to-one or one-to-one in our approach. Instead, feasible task sets are automatically identified from a set of objects. Since such an approach is primarily based on real-time synchro-
Table 1
Summary of notations for the task model.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \tau_i )</td>
<td>A task</td>
</tr>
<tr>
<td>( T_i )</td>
<td>The period of task ( \tau_i )</td>
</tr>
<tr>
<td>( C_i )</td>
<td>The worst-case execution time of task ( \tau_i )</td>
</tr>
<tr>
<td>( p_i )</td>
<td>The fixed-priority of task ( \tau_i )</td>
</tr>
<tr>
<td>( pt_i )</td>
<td>The preemption threshold of task ( \tau_i )</td>
</tr>
<tr>
<td>( \epsilon_i )</td>
<td>The effective priority of task ( \tau_i )</td>
</tr>
<tr>
<td>( M_i )</td>
<td>A mutex (binary semaphore)</td>
</tr>
<tr>
<td>( P(M_i) )</td>
<td>The priority ceiling of mutex ( M_i )</td>
</tr>
<tr>
<td>( pt(M_i) )</td>
<td>The preemption threshold ceiling of mutex ( M_i )</td>
</tr>
<tr>
<td>( M )</td>
<td>A mutex with the largest priority ceiling of all mutexes that are currently locked by any tasks except the currently running task</td>
</tr>
<tr>
<td>( \overline{M} )</td>
<td>A mutex with the largest preemption threshold ceiling of all mutexes that are currently locked by any tasks except the currently running task</td>
</tr>
<tr>
<td>( d_{i,k} )</td>
<td>The worst-case execution time of the critical section of task ( \tau_i ) protected by mutex ( M_k )</td>
</tr>
<tr>
<td>( B_i )</td>
<td>The worst-case PTS blocking time of task ( \tau_i )</td>
</tr>
<tr>
<td>( \beta_i )</td>
<td>The worst-case synchronization blocking time of task ( \tau_i )</td>
</tr>
<tr>
<td>( L_i )</td>
<td>Priority level- ( i ) busy period</td>
</tr>
<tr>
<td>( S_i(q) )</td>
<td>The start time of ((q+1))-th instance of task ( \tau_i ) in ( L_i )</td>
</tr>
<tr>
<td>( F_i(q) )</td>
<td>The finish time of ((q+1))-th instance of task ( \tau_i ) in ( L_i )</td>
</tr>
<tr>
<td>( R_i )</td>
<td>The worst-case response time of task ( \tau_i )</td>
</tr>
</tbody>
</table>

The synchronization under PTS, it needs to be comprehensively addressed. Note that an object-oriented design produces a number of object locks to maintain the consistency of object states and the run-to-completion semantics of the finite state machine for each object.

The remainder of the paper is organized as follows. Section 2 describes the task model and presents the necessary definitions for the discussion. Section 3 describes the protocol specification of DCP and Section 4 shows the blocking properties of DCP. In Section 5, we present the rationale for the locking conditions of DCP by comparison with those of other real-time synchronization protocols. Section 6 presents the schedulability analysis algorithm for the proposed protocol. Section 7 describes the results of an empirical study to evaluate our analyses. We conclude the paper in Section 8.

2. Task model

The task model is the same as the one used in traditional real-time scheduling [3,9,20] except that each task has a preemption threshold as its scheduling attribute, in addition to its priority. Specifically, we assume a uniprocessor environment and we allow only properly nested mutexes. We also assume a system with a fixed set of tasks, each of which has a fixed period, a known worst-case execution time, a fixed priority, and a preemption threshold. A higher priority is denoted by a larger value, as befits the intuitive meaning of a higher threshold. The notations and associated descriptions used in this paper are summarized in Table 1.

Under PTS, each task \( \tau_i \) has a preemption threshold \( pt_i \), in addition to its regular priority \( p_i \). Note that it is meaningful to assign a task a preemption threshold not 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, a precise definition is desirable. Conceptually, the effective priority \( \epsilon_i \) of a task \( \tau_i \) 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. It has the following operational definition:

Effective priority \( \epsilon_i \) of task

\[
\epsilon_i = p_i \text{ if } \tau_i \text{ is released in its period and not yet dispatched;}
\]

\[
\text{otherwise, } \max(pt_i, p_1, p_2, \ldots, p_j) \text{ such that } \tau_1, \tau_2, \ldots, \tau_j \text{ are tasks blocked by } \tau_i.
\]

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 \( \tau_i \) is said to be in PTS blocking if it is blocked by a lower priority task with a preemption threshold higher than \( p_i \). We denote the duration of PTS blocking by \( B_i \) while we denote the duration of synchronization blocking by \( \beta_i \).

3. Protocol specification of DCP

We define DCP under PTS with priority ceilings by combining an offline ceiling protocol and three online protocols in a similar manner to that described in [21].
Protocol DCP.

- **Offline ceiling protocol.** Each mutex $M_i$ is assigned a priority ceiling $p(M_i)$ and a preemption threshold ceiling $pt(M_i)$ as follows:

$$p(M_i) = \max\{p_j | \tau_j \text{ is such a task that may require mutex } M_i\}$$

$$pt(M_i) = \max\{pt_j | \tau_j \text{ is such a task that may require mutex } M_i\}.$$  

- **Online effective priority inheritance protocol.** When the currently executing task $\tau_i$ is blocked by task $\tau_j$, the effective priority of task $\tau_i$ is set as follows:

$$ep_i = ep_j.$$  

- **Online effective priority recovery protocol.** When the currently executing task $\tau_i$ exits its critical section, the effective priority of task $\tau_i$ is set as follows:

$$ep_i = \max(pt_i, ep_k)$$

where task $\tau_k$ is any task that is still blocked by task $\tau_i$.

- **Online locking protocol.** When the currently executing task $\tau_i$ tries to lock an unlocked mutex, the following rule is applied; let mutex $\overline{M}$ be a mutex with the maximum priority ceiling of all mutexes that are currently locked by any tasks except task $\tau_i$. Task $\tau_i$ is allowed to lock a mutex only when the following condition is true:

$$C1 \lor C2$$

where $C1: p(\overline{M}) < p_i$ and $C2: pt(\overline{M}) < pt_i$.

If the condition is false, then task $\tau_i$ is blocked by the task that has locked mutex $\overline{M}$.

The second and third parts of DCP are the effective priority inheritance protocol (EPI) that we previously proposed in [2]. We have shown that EPI can prohibit the uncontrolled priority inversion problem under PTS in [2]. Note that the online effective priority recovery protocol is different from the one presented in [2], which corrects for the following anomaly: all of the existing priority recovery protocols in [2,3,22,23] state that when task $\tau_i$ exits from a critical section, task $\tau_i$ recovers the priority that "it had before entering that critical section". This statement is problematic since any higher priority task may have been blocked after task $\tau_i$ entered that critical section. In such a case, the effective priority of task $\tau_i$ may have been raised to more than the priority it had before entering that critical section, thus such a raised priority must be maintained after task $\tau_i$ exits that critical section. Fig. 1(a) shows this case. At time $t_6$ when task $\tau_1$ exits its critical section, task $\tau_1$ recovers its priority at time $t_2$, which is $ep_1$. Then, task $\tau_2$ preempts task $\tau_1$ while task $\tau_3$ is still waiting for task $\tau_1$ to unlock mutex $M_1$, thus there is an uncontrolled priority inversion [3] after time $t_6$. Fig. 1(b) illustrates how DCP prevents such a priority inversion.
The online locking protocol does not use the notion of the system ceiling mentioned in [2,14]. The system ceiling is defined as a system-wide scheduling attribute that is dynamically set to the maximum priority ceiling of all currently locked mutexes in the system. If we adopt the notion of a system ceiling, we can introduce a system priority ceiling and a system preemption threshold ceiling; the system priority ceiling is defined in the same manner as the system ceiling in [2] and the system preemption threshold ceiling is defined similarly by replacing priority ceilings with preemption threshold ceilings. The left term $p(M)$ of condition C1 is in fact the system priority ceiling. However, note that the left term $pt(M)$ is not the system preemption threshold ceiling. This is because mutex $M$ may not be the mutex with the maximum preemption threshold ceiling of all currently locked mutexes. As will be discussed in Section 5, this is the reason that DCP is superior to the existing ceiling protocols for PTS.

4. Blocking properties of DCP

In this section, we discuss the blocking properties of DCP. As with other real-time synchronization protocols such as PC-PCP and PTC-PCP, DCP prevents deadlock and multiple synchronization blockings. We prove these properties in the following subsections.

4.1. Prevention of deadlock

While EPI can effectively solve the uncontrolled priority inversion problem [2], it can incur a deadlock situation, as illustrated in Fig. 2(a). In this example, both tasks $\tau_1$ and $\tau_2$ require mutex $M_1$, both tasks $\tau_2$ and $\tau_3$ require mutex $M_2$, and both tasks $\tau_3$ and $\tau_1$ require mutex $M_3$. As shown, task $\tau_3$ waits for task $\tau_2$ to release mutex $M_2$ at time $t_6$, task $\tau_2$ waits for task $\tau_1$ to release mutex $M_1$ at time $t_7$, and task $\tau_1$ waits for task $\tau_3$ to release mutex $M_3$ at time $t_8$, while tasks $\tau_1$, $\tau_2$, and $\tau_3$ are holding $M_1$, $M_2$, and $M_3$, respectively. Therefore, circular waiting occurs at $t_8$ and thus a deadlock occurs. Fig. 2(b) shows how DCP prevents such a deadlock. The following theorem proves that DCP always prevents a deadlock.

**Theorem 1.** DCP prevents deadlock.

**Proof.** We show that circular waiting cannot occur in DCP. Circular waiting cannot occur if each task does not enter its critical section until all mutexes it can use are unlocked. If any mutex that task $\tau_i$ can use is locked, it follows that $p(M) \geq p_i$ and $pt(M) \geq pt_i$. That is, both conditions C1 and C2 are false and thus C1 $\lor$ C2 is also false. This means that each task does not enter its critical section while any mutex it can use is locked in DCP. Therefore, circular waiting cannot occur and thus deadlock is prevented. \(\square\)

4.2. Prevention of multiple synchronization blockings

Fig. 3(a) shows an example of multiple synchronization blockings in EPI. In this example, both tasks $\tau_3$ and $\tau_4$ are blocked by task $\tau_1$ during period $(t_6, t_7)$ and they are also blocked by task $\tau_2$ during period $(t_9, t_{10})$. Note that multiple synchronization blockings refer to a situation where a task is blocked by more than two tasks which execute their critical sections. Fig. 3(b) shows how DCP prevents such multiple synchronization blockings where both tasks $\tau_3$ and $\tau_4$ are blocked only by task $\tau_1$ during period $(t_6, t_7)$. The following theorem proves that DCP always prevents multiple synchronization blockings.
Theorem 2. DCP prevents multiple synchronization blockings.

Proof. We prove the theorem by contradiction. Suppose that a task encounters twice the synchronization blocking in DCP. Without the loss of generality, suppose that after task $\tau_i$ has arrived, two lower priority tasks $\tau_1$ and $\tau_2$ sequentially execute their critical sections for mutexes $M_1$ and $M_2$. Tasks $\tau_1$ and $\tau_2$ can execute after task $\tau_i$ has arrived only when they inherit the effective priority of a task $\tau_j$ such that $p_j \geq p_i$. This involves two cases: 1) task $\tau_2$ inherits $\text{ep}_j$ and then $\tau_1$ inherits $\text{ep}_j$ transitively and 2) tasks $\tau_1$ and $\tau_2$ inherit $\text{ep}_j$ independently.

Case 1: transitive effective priority inheritance. This case implies that task $\tau_2$ locks mutex $M_2$ and then tries to lock mutex $M_1$. When task $\tau_2$ locks mutex $M_2$, mutex $M_1$ has been locked by task $\tau_1$ and thus it follows that $p(M) \geq p_j$ and $pt(M) \geq pt_j$. That is, both conditions $C_1$ and $C_2$ are false and thus $C_1 \lor C_2$ is also false. Accordingly, task $\tau_2$ cannot lock mutex $M_2$. This is a contradiction.

Case 2: independent effective priority inheritance. This case implies that task $\tau_j$ with $p_j \geq p_i$ requires mutexes $M_1$ and $M_2$. When task $\tau_2$ locks mutex $M_2$, mutex $M_1$ has been locked by task $\tau_1$ and thus it follows that $p(M) \geq p_j$ and $pt(M) \geq pt_j$. That is, both conditions $C_1$ and $C_2$ are false and thus $C_1 \lor C_2$ is also false. Accordingly, task $\tau_2$ cannot lock mutex $M_2$. This is a contradiction.

5. Rationale for DCP locking conditions

It is not obvious why we used both priority and preemption threshold to define ceilings of mutexes in DCP. Nor is it obvious why DCP does not exploit the notion of system ceilings as in [2,14]. In fact, we can use only a priority ceiling or only a preemption threshold ceiling while exploiting the notion of system ceilings for this purpose. The former protocol with a priority ceiling is PC-PCP, and the latter one with a preemption threshold ceiling is PTC-PCP, as presented in [2]. In this section, we show that DCP is superior to both PC-PCP and PTC-PCP, since DCP results in the least blocking.

The weaker the locking condition, the lower the blocking probability. This is because the locking condition allows any currently running task to successfully proceed with its execution without blocking. Consider the simple locking condition $C_0$, which is that any mutex is not locked. This locking condition is trivial, since there is no reason for a task to be blocked if no mutex is locked. In fact, the locking condition $C_0$ was introduced in a real-time synchronization protocol named NPCS (non-preemptive critical section) [24]. As a real-time synchronization protocol, NPCS prevents deadlock and multiple synchronization blockings [3]. However, this protocol is sub-optimal in that the probability of unnecessary blocking is too high. Thus, PCP was introduced, such as in [3]. In fact, the real-time synchronization protocol has evolved to have a weaker locking condition, while still preventing deadlock and multiple synchronization blockings.

The locking conditions of NPCS, DCP, PC-PCP, and PTC-PCP are as follows:

- NPCS: $C_0$,
- DCP: $C_1 \lor C_2$,
- PC-PCP: $C_1$,
- PTC-PCP: $C_3$. 

**Fig. 3.** (a) Multiple synchronization blockings in EPI, (b) multiple synchronization blockings prevention in DCP.
Fig. 4. Relationships of locking conditions: C0 for NPCS, C1 for PC-PCP, C1, C2 for DCP, and C3 for PTC-PCP.

\[ \tau_1 \rightarrow \{p(M_1), \ldots, p(M_n)\} \]
\[ \tau_2 \rightarrow \{p(M_1), \ldots, p(M_n)\} \]
\[ \tau_3 \rightarrow \{p(M_1), \ldots, p(M_n)\} \]

Fig. 5. (a) Unnecessary blocking in PC-PCP and PTC-PCP, (b) prevention of unnecessary blocking in DCP.

where C0: any mutex is not locked.

C3: \( p_t(M) < p_t \) and mutex \( M \) is the mutex with the maximum preemption threshold ceiling of all mutexes that are currently locked by any tasks except task \( \tau_i \).

Fig. 4 shows the relationships of these locking conditions. Note that C3 \( \Rightarrow \) C2 since \( p_t(M) \geq p_t(M) \). Fig. 4 shows that DCP has the weakest locking condition of all other protocols. Accordingly, it implies that DCP results in lesser blocking than both PC-PCP and PTC-PCP. We can introduce a new locking condition C1 \( \lor \) C3 by directly integrating PC-PCP and PTC-PCP. However, Fig. 4 also implies that this condition causes more blocking, since C1 \( \lor \) C3 is a subset of the locking conditions of DCP.

Fig. 5 illustrates how DCP prevents unnecessary blockings in PC-PCP and PTC-PCP. Fig. 5(a) shows that task \( \tau_2 \) is blocked by task \( \tau_1 \) during period \( (t_3, t_4) \) since mutex \( M_1 \) is locked in PC-PCP or PTC-PCP. Although mutex \( M_1 \) is required by task \( \tau_3 \) that has a higher priority than task \( \tau_2 \), \( p_t(M) \) is no higher than \( p_t(M) \) and thus task \( \tau_2 \) cannot preempt task \( \tau_3 \). Therefore, blocking task \( \tau_2 \) is entirely unhelpful in preventing deadlock or multiple synchronization blockings. This blocking also causes unnecessary context switches between task \( \tau_2 \) and \( \tau_1 \) and increases the response time of task \( \tau_2 \). Since there is no decrease in the response time of task \( \tau_1 \) while the response time of task \( \tau_2 \) increases, this kind of unnecessary blocking increases the average response times of tasks. Fig. 5(b) shows how DCP prevents such unnecessary blocking.

6. Schedulability analysis

In this section, we present schedulability analysis algorithms for DCP based on the worst-case response time analysis [25]. In DCP, the worst-case response time \( R_i \) is always less than or equal to the value calculated from the following equations:

\[ B_i = \max\{|C_j| \mid \forall \tau_j, p_t \geq p_i \geq p_j\} \]  
(1)

\[ \beta_i = \max\{d_{j,k} \mid \forall \tau_j, p_t < p_i \} \cap \{\forall M_k, (p(M_k) \geq p_i) \land (p_t(M_k) \geq p_t)\} \]  
(2)

\[ L_i = B_i + \beta_i + \sum_{\forall j, \ p_j \geq p_i} \left[ L_j \cdot C_j \right] \]  
(3)

\[ q = 0, 1, \ldots, \left[ \frac{L_i}{T_j} \right] \]  
(4)
\[ S_i(q) = B_i + \beta_i + q \cdot C_i + \sum_{j, p_j > p_i} \left( 1 + \left[ \frac{S_j(q)}{T_j} \right] \right) \cdot C_j \]  
(5)

\[ F_i(q) = S_i(q) + C_i + \sum_{j, p_j > p_i} \left\{ \left[ \frac{F_j(q)}{T_j} \right] - \left( 1 + \left[ \frac{S_j(q)}{T_j} \right] \right) \right\} \cdot C_j \]  
(6)

\[ R_i = \max \{ F_i(q) - q \cdot T_i \} \]  
(7)

where \( L_i \) is a busy period [25] of task \( \tau_i \) and \( d_{j,k} \) is the worst-case execution time of the critical section of task \( \tau_j \) protected by mutex \( M_k \).

The schedulability analysis for the worst-case task response time under PTS without synchronization blocking was presented in [9] and its error was corrected in [11]. If we set the worst-case synchronization blocking time \( \beta_i \) to zero in the above equations, then the equations reduce to those presented in [11]. Therefore, we only show the formulations regarding \( \beta_i \): how it is formulated and how \( \beta_i \) is included in the other equations.

First, we show why \( \beta_i \) is formulated as in Eq. (2). From Theorem 2, at most one task \( \tau_j \) blocks task \( \tau_i \) during one outermost critical section \( d_{j,k} \). Since task \( \tau_j \) must be able to preempt \( \tau_j \), it follows that \( \forall \tau_j, p_{t_j} < p_i \). Task \( \tau_j \) can be blocked in two different ways. (1) Directly, when it tries to enter a critical section or (2) transitively, via task steps 2 and 4 are independent of \( \beta_i \).

Proof. We show that the set of tasks that can preempt task \( \tau_i \) before task \( \tau_i \) starts its execution is a superset of the set of tasks that can preempt task \( \tau_i \) after task \( \tau_i \) starts its execution. Let the former task set be \( A \) and the latter be \( B \). A task \( \tau_j \) from \( A \) is a task that satisfies \( p_j > p_i \) while a task \( \tau_k \) from \( B \) is a task that satisfies \( p_k > p_i \). Since each task \( \tau_i \) satisfies \( p_{t_i} \geq p_i \) under PTS, it flows that \( A \supseteq B \) and thus the theorem has been proved. \( \Box \)

Therefore, \( \beta_i \) is transitively included in \( F_i(q) \), since it is included in \( S_i(q) \) and \( S_i(q) \) is included in \( F_i(q) \). Since PTS blocking and synchronization blocking can occur independently, \( \beta_i \) should be added independently with \( B_i \), which implies Eqs. (5) and (6).

7. Experimental evaluation

In this section, we evaluate the blocking and response time performance of DCP. We compare DCP with other real-time synchronization protocols, which are specifically PC-PCP and PTC-PCP [2]. We integrated PTS and PC-PCP, PTC-PCP, and DCP into a real-time object-oriented CASE tool, which is IBM Rational RoseRT [26]. The integration established that the implementation complexity of DCP is the same as that of PCP, which proves that DCP is practical. Specifically, the condition C1 is implemented in the same manner as in PCP-PCP. The condition C2 is implemented with a system threshold ceiling as in PTC-PCP; its value is set only when the system priority ceiling is updated to \( p(M_i) \), while its value is set to \( pt(M_i) \).

As a task set application, we used an industrial private branch exchange (PBX) system, which was presented in [4,5]. The system was implemented in a real-time object-oriented language based on UML 2.0 [6], and the development environment was IBM Rational RoseRT [26]. To simplify the presentation, we use a simplified task set; only three wireless phone extensions were supported. The task set is presented in Table 2; the preemption thresholds were assigned according to the maximum preemption threshold assignment algorithm presented in [9].
Table 2
Task Set for PBX system.

<table>
<thead>
<tr>
<th>Task ID</th>
<th>Period</th>
<th>Deadline</th>
<th>WCET</th>
<th>Priority</th>
<th>Threshold</th>
<th>Mutex accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8000</td>
<td>8000</td>
<td>400</td>
<td>1</td>
<td>2</td>
<td>{P(M_3), V(M_3)}</td>
</tr>
<tr>
<td>2</td>
<td>8500</td>
<td>8500</td>
<td>420</td>
<td>2</td>
<td>2</td>
<td>{P(M_4), V(M_4), P(M_2), V(M_2)}</td>
</tr>
<tr>
<td>3</td>
<td>5000</td>
<td>5000</td>
<td>300</td>
<td>3</td>
<td>3</td>
<td>{P(M_3), V(M_2)}</td>
</tr>
<tr>
<td>4</td>
<td>4000</td>
<td>4000</td>
<td>500</td>
<td>4</td>
<td>8</td>
<td>{P(M_3), V(M_2)}</td>
</tr>
<tr>
<td>5</td>
<td>10000</td>
<td>10000</td>
<td>400</td>
<td>5</td>
<td>8</td>
<td>{P(M_4), V(M_4)}</td>
</tr>
<tr>
<td>6</td>
<td>6000</td>
<td>3000</td>
<td>900</td>
<td>6</td>
<td>6</td>
<td>{P(M_1), V(M_1), P(M_2), V(M_2)}</td>
</tr>
<tr>
<td>7</td>
<td>6000</td>
<td>2500</td>
<td>900</td>
<td>7</td>
<td>7</td>
<td>{P(M_1), V(M_1), P(M_2), V(M_2)}</td>
</tr>
<tr>
<td>8</td>
<td>6000</td>
<td>2000</td>
<td>900</td>
<td>8</td>
<td>8</td>
<td>{P(M_1), V(M_1), P(M_2), V(M_2)}</td>
</tr>
</tbody>
</table>

Fig. 6. (a) Average blocking times, (b) average response times, (c) maximum blocking times, and (d) maximum response times.

Fig. 6 (a) and (b) show the average blocking and response time of each task. The results show that the average blocking and response times of higher priority tasks 5–8 were reduced in DCP by comparison with the other protocols. Specifically, the average blocking times of higher priority tasks 5–8 were reduced by 39.4% and 42.8% by comparison with PC-PCP and PTC-PCP, respectively. The average response times of these tasks were also reduced by 13.2% and 14.7% by comparison with PC-PCP and PTC-PCP, respectively. On the other hand, we can see that the average response time of task 4 was increased by 72.4%. This is because the lower priority tasks experience a greater interference time, since the higher priority tasks in DCP are less blocked and thus they interfere with the lower priority tasks more.

Fig. 6 (c) and (d) show the maximum blocking and response time of each task. The results are similar to the average blocking and response time of each task. The major difference is that the maximum response time of task 4 was dramatically increased by 168% in DCP by comparison with both PC-PCP and PTC-PCP. This is because it is inevitable that the response times of the lower priority tasks increase as the response times of the higher priority tasks decrease, as in our explanation of the average case.

8. Conclusion

Although object-oriented design methods are widely used in contemporary software development, their application to real-time embedded systems has been limited due to the lack of traditional real-time scheduling techniques that can be seamlessly integrated into these methods. Preemption threshold scheduling (PTS) has been suggested as a solution since it
improves both run-time overhead and schedulability. However, direct adoption of PTS may lead to long priority inversion since object-oriented real-time systems require synchronization considerations to maintain consistent object states.

We proposed the dual ceiling protocol (DCP) to solve this problem. DCP is a real-time synchronization protocol for PTS, which leads to least blocking and worst-case response times by comparison with known real-time synchronization protocols for PTS. DCP exploits both priority ceilings and preemption threshold ceilings, but it is not a straightforward integration of PC-PCP and PTC-PCP, which are existing real-time synchronization protocols for PTS. We presented the rationale for the locking conditions of DCP by comparison with the locking conditions of other protocols. We also provided its blocking properties and schedulability analyses based on worst-case response time analyses.

We have implemented PTS and DCP in a real-time object-oriented CASE tool and presented the associated experimental results. This work is based on our previous implementation of a CASE tool that is capable of deriving tasks from an object-oriented design model [4,5]. This tool is an extension of the RoseRT CASE tool [26]. We showed that the implementation complexity of DCP is the same as that of PC-PCP and PTC-PCP and the run-time overhead is the same as that of the other protocols. The experimental results showed that DCP leads to least blocking times for tasks, which also leads to reduced response times of higher priority tasks.

References