Real-Time Synchronization Protocols

CS 537 Scheduling Algorithms

Steve Jankly

6/7/2015
Abstract

In real-time systems, there is a situation that can occur when low priority jobs block high priority tasks. This duration is scrutinized and evaluated in this paper. The unbounded priority inversion problem is analyzed. In an ideal situation, the higher priority job should win if two jobs compete for the CPU, however, this is not the case in many situations. There were milestones in the 1980s that offered solutions, and Caccamo and Sha [1] offers a newer approach.
# Table of Contents

Abstract ................................................................................................................................. 2

1 Introduction .......................................................................................................................... 4
   1.1 Real-Time Systems ......................................................................................................... 4
   1.2 Unbounded Priority Inversion Problem ......................................................................... 4

2 Concepts / Priority Inheritance Approach ........................................................................ 6
   2.1 Notations ....................................................................................................................... 6
   2.2 Assumptions and Lemmas ............................................................................................ 6

3 Static Priority Real-Time Synchronization Protocols ............................................................. 7
   3.1 Basic Priority Inheritance Protocol ............................................................................... 7
   3.2 Priority ceiling protocol ............................................................................................... 7
   3.3 Schedulability Analysis ................................................................................................. 7
   3.4 Stack Resource Policy .................................................................................................. 8
   3.5 Resource Sharing Among Hybrid Tasks ....................................................................... 8
   3.6 Example Usage of CBS Server with Resource Constraints ........................................ 10

5 Conclusion .......................................................................................................................... 12

References .............................................................................................................................. 13
1 Introduction

1.1 Real-Time Systems

What is a real-time system? They are systems that must guarantee response within specified time constraints (i.e. “deadlines”). Some examples are mission critical systems, QNX, and iOS. They could reside in your watch, or even in your car. In a system, if the total correctness of an operation depends not only upon its logical correctness, but also upon the time in which it is performed, that system is considered real-time [1]. Both real-time systems and their deadlines have consequences if missing a deadline. There are three types: hard, firm, and soft. Missing a hard deadline results in total system failure. For firm deadlines, missing an infrequent deadline is tolerable, but the system’s quality of service might be degraded, and the usefulness of a result is zero after its deadline. For soft deadlines, after its deadline, the usefulness of a result is degraded which lowers the system’s quality of service. Missing deadlines may have serious implications in real-world scenarios.

1.2 Unbounded Priority Inversion Problem

There is a situation that can occur in real-time systems when low priority jobs block high priority tasks, and the duration of this is scrutinized and evaluated. The unbounded priority inversion problem is analyzed. In an ideal situation, the higher priority job should win if two jobs compete for the CPU, however, this is not the case in many situations. There were milestones in the 1980s from Lampson and Redell [7] that offered solutions, and Caccamo and Sha [1] offer a newer approach.

Priority inversion is the situation when a higher-priority job J is blocked by lower-priority jobs. An ideal situation is when two jobs are competing for CPU; the higher priority job should win. Now, this is not always ideal. For example, if a semaphore locked by low priority
job, a high priority job is delayed when it wants to lock that semaphore. Another example is when two jobs try to access shared data. This is a common occurrence. See Figure 1 for an example of this.

Figure 1 depicts three jobs, J1, J2, and J3, from highest to lowest priority. J1 and J3 share a data structure guarded by the binary semaphore S. At time t1, J3 locks S. At time t2, J1 preempts J3. However, J1 cannot lock S and blocks. At time t3, J2 preempts J3. But J1 cannot lock S and still blocks. Since J2 came into the picture, there is an indeterminate amount of time until J2 completes. This is a problem, because J3 and in turn J1 have to wait until J2 completes at t4. At t5, J3 was completed already and J1 can now lock S and access the data. This shows that the duration of the blocking of J1 is unpredictable, since J3 can be preempted by J2 and other intermediate jobs. J3 (and therefore J1) would still block until J2 and other intermediate jobs complete.

Now, regarding unbounded priority inversion, why is it called “unbounded”? The reason is because the blocking period not bounded by the critical section duration. This is the case in the
previous example, which includes the execution time of J2. There are ways to avoid this. First, if a job in its critical section is executed at the highest priority level (so that it cannot be preempted), it creates unnecessary blocking. This is only good for short critical sections. An improved solution is given by Lampson and Redell [7]: A monitor should be executed at the highest priority of any task that will ever use this monitor. When a task enters the monitor, the priority is temporarily increased to that of the monitor.

### 2 Concepts / Priority Inheritance Approach

#### 2.1 Notations

Some notations used in this paper are as follows:

- A periodic task
  - $\tau_i$
- A job (an instance of $\tau_i$)
  - $J_i$
- Priority of job $J_i$
  - $\pi_i$
- A pair of jobs
  - $\{J_i, J_j, i < j\}$ (assume job $J_i$ has higher priority)

#### 2.2 Assumptions and Lemmas

There are three assumptions for the priority inheritance approach. The first assumption is that a fixed priority is assigned to each job. Second, a job will use its assigned priority unless it is in a critical section. Third, when a job executes inside a critical section, priority could be changed by pairs of actions that raise and lower this job’s priority, and a job’s priority will never be lowered below its assigned priority.

There are two lemmas for the priority inheritance approach:

**Lemma 1:**

Job $J$ can be blocked by a lower priority job $J_L$:

Only if when $J$ becomes ready, $J_L$ is inside its critical section
Real-Time Synchronization Protocols

*Lemma 2:* When there is no deadlock:

A job $J$ can be blocked by a lower-priority job $J_L$ for at most the duration of one
outermost nested critical section of $J_L$, regardless of the number of semaphores $J$ and $J_L$
share.

## 3 Static Priority Real-Time Synchronization Protocols

### 3.1 Basic Priority Inheritance Protocol

When job $J$ blocks one or more higher-priority jobs, it executes its critical section at the
highest priority level among all the jobs that it blocks. After exiting its critical section, job $J$
returns to its assigned priority $\pi$.

*Theorem 1:* A job $J$ can be blocked by each of the lower priority jobs that may use a semaphore at
most once.

### 3.2 Priority ceiling protocol

The priority ceiling of a semaphore $S$ is defined as the maximum priority of all the jobs
that may ever use this semaphore. The equation is as follows:

$$ceil(S) = \max_i \{\pi_i | \tau_i \text{ may lock } S\}$$

The PCP has two useful properties. It prevents deadlocks, and a job can be blocked by lower
priority jobs at most once.

### 3.3 Schedulability Analysis

Liu & Layland (1973) proved that for a set of $n$ periodic tasks with unique periods, a
feasible schedule that will always meet deadlines exists if the CPU utilization is below a specific
bound. However, this depends on the number of tasks. This Liu & Layland bound can be extended by incorporating the effect of blocking time.

### 3.4 Stack Resource Policy

Stack Resource Policy, or SRP, is a concurrency control protocol proposed by Baker. In SRP, bound the priority inversion phenomenon in static as well as dynamic priority systems. Under EDF (earliest deadline first), only jobs from tasks with long relative deadline cannot preempt jobs from tasks with short relative deadline. The relative deadline of a job is the duration between a job’s deadline and its arrival time.

**Lemma 3:**

Let job $J_i$ have relative deadline $D_i$, $J_j$ have relative deadline $D_j$, and $D_i < D_j$

From EDF: job $J_i$ can never block the execution of $J_j$

This lemma basically explains that from EDF, a job cannot block another job with longer relative deadline. The case when jobs with longer relative deadlines block jobs with shorter relative deadlines, is the only case to be considered when studying blocking under EDF. Now, a period tasks that has a shorter period has jobs that have shorter relative deadlines. Blocking actions with EDF and RMS are equivalent logically due to the fact that jobs from a task can only be blocked by jobs from tasks with longer periods.

### 3.5 Resource Sharing Among Hybrid Tasks

These days, there are different levels of criticality for current real-time applications. Usually these tasks need to share common resources in exclusive mode. The system has to prevent unbounded blocking to ensure predictable execution of hard tasks. Priority inversion situations can cause unbounded blocking. The previous classic solutions did not consider hybrid
task sets (hard and soft). There is a more efficient method for combining hard and soft activities with resource constraints from Caccamo and Sha. There are some assumptions here:

- Consider: hybrid task set with
  - Hard periodic tasks
  - Soft aperiodic tasks, that must be executed on a uniprocessor system
  - Tasks are preemptable and are scheduled using the earliest deadline first (EDF) algorithm
  - Hard and soft tasks may share a set R of mutually exclusive resources accessed using the SRP

Periodic tasks are jobs that are regularly activated. Aperiodic tasks have irregular job activation, and job arrival times are not known beforehand. Soft tasks do not have deadlines, but are assigned one anyways. Hard tasks have hard deadlines.

Preventing budget exhaustion inside critical sections is of interest. One way is mutually exclusive shared resource access, where for a capacity based server, problems arise if the server exhausts its budget when a task is inside a critical section. To prevent long blocking delays, a job which exhausts its budget should be allowed to continue executing with the same deadline. It uses extra budget until leaving the critical section. Lastly, a job is dynamically partitioned into chunks. Each chunk’s execution is time less than the available bandwidth. A chunk can never be inside a critical section.

Dynamic preemption levels can solve some problems. So far, fixed relative deadline and static preemption levels do not permit to provide an easy and efficient solution to the problem. Use dynamic preemption levels for aperiodic tasks. This is simpler more elegant to solve the problem of sharing resources with CBS and SRP. CBS stands for constant bandwidth server. It is
Real-Time Synchronization Protocols

one of the most used algorithms for implementing resource reservation upon deadline-based schedulers. SRP stands for stack resource policy. It is used for accessing shared resources when using Earliest Deadline First (EDF) scheduling.

There are some requirements for the aperiodic server. There are two rules:
1. Each job chunk must have a minimum relative deadline known beforehand
2. A task must never exhaust its budget when it is inside a critical section

Additionally, CBS must be modified from its original definition. Now it guarantees that each job chunk has a minimum relative deadline equal to the server period. An extra rule is added prevent a task from exhausting its budget when it is using a shared resource. The new definition is as follows [1]:

1. Each server $S_i$ is characterized by a budget $c_i$ and by an ordered pair $(Q_i, T_i)$, where $Q_i$ is the maximum budget and $T_i$ is the period of the server. At each instant, a fixed deadline $d_{i,k}$ is associated with the server. At the beginning $\forall i, d_{i,0} = 0$.
2. When a job $\tau_i, j$ arrives and the server is idle, the server generates a new deadline $d_{i,k} = \max(r_{i,j}, d_{i,k-1}) + T_i$ and $c_i$ is recharged at the maximum value $Q_i$.
3. Whenever a served job executes, the budget $c_i$ is decreased by the same amount.
4. When the server is active and $c_i$ becomes equal to zero, the server budget is recharged at the maximum value $Q_i$ and a new server deadline is generated as $d_{i,k} = d_{i,k-1} + T_i$.
5. Whenever a served job $\tau_i, j$ tries to access a critical section, if $c_i < \zeta_i$ (where $\zeta_i$ is the duration of the longest critical section of job $\tau_i, j$ such that $\zeta_i < Q_i$), a budget replenishment occurs, that is $c_i = c_i + Q_i$ and a new server deadline is generated as $d_{i,k} = d_{i,k-1} + T_i$.

3.6 Example Usage of CBS Server with Resource Constraints
This example details the usage of the CBS Server with resource constraints. Consider a task set that consists of an aperiodic job $J_1$, handled by a CBS. The max budget $Q_s = 4$, the
Real-Time Synchronization Protocols

server period $T_s = 8$. There are two periodic tasks $\tau_1$ and $\tau_2$. They share two resources $R_a$ and $R_b$. $J_1$ and $\tau_1$ share resource $R_b$. $\tau_1$ and $\tau_2$ share resource $R_a$. Figure 2 shows the parameters of the task set. Figure 3 shows the schedule produced.

### Table: Parameters of the Task Set

<table>
<thead>
<tr>
<th>Task</th>
<th>Type</th>
<th>$Q_s$ or $C$</th>
<th>$T_s$ or $T$</th>
<th>$R_a$</th>
<th>$R_b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$J_1$</td>
<td>Soft aperiodic</td>
<td>4</td>
<td>8</td>
<td>—</td>
<td>3</td>
</tr>
<tr>
<td>$\tau_1$</td>
<td>Hard periodic</td>
<td>2</td>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>Hard periodic</td>
<td>6</td>
<td>24</td>
<td>3</td>
<td>—</td>
</tr>
</tbody>
</table>

*Figure 2: Parameters of the Task Set in this Example*

Job $J_1$ arrives at time $t = 3$. The first chunk $H_{1,1}$ receives a deadline $d_{1,1} = a_{1,1} + T_s = 11$ (from the CBS algorithm). At that time, $\tau_2$ is already inside a critical section on resource $R_a$. $H_{1,1}$ of job $J_1$ is able to preempt it. Preemption level is $\pi_{1,1} = 1/8 > \Pi(H_{1,1})$. At time $t = 6$, $J_1$ tries to access a critical section. Residual budget = 1 is not sufficient to complete the whole critical section. So, the deadline is postponed and the budget is replenished.

The next chunk $H_{1,2}$ of $J_1$ starts at time $a_{1,2} = 6$ with deadline $d_{1,2} = 19$. The chunk $H_{1,2}$ of $J_1$ cannot start. Its preemption level $\pi_{1,2} = 1/13 < \Pi(H_{1,2})$. Then $\tau_2$ executes until the end of its critical section. When its system ceiling becomes zero, $J_1$ is able to preempt $\tau_2$. When $J_1$ frees resource $R_b$, $\tau_1$ starts executing. Note that each chunk can be blocked for the duration of one
critical section (at most) by the preemption test & once started will never be blocked for resource contention.

5 Conclusion

Real-time systems require guaranteed response within specified time constraints, or, deadlines. If a deadline is missed, it may have dire consequences, depending on the system. The unbounded priority inversion problem is one situation that can occur in real-time systems when a high priority job is blocked by a low priority job. The proposal by Caccamo and Sha in their paper [1] offers a newer approach to this problem. A hybrid task set with preemptable tasks is assumed. Budget exhaustion prevention is key. Mutually exclusive shared resource access, along with budget usage relaxations is one way. Jobs are partitioned into chunk, and cannot reside in critical sections. Use dynamic preemption levels for aperiodic tasks. The Constant Bandwidth Server (CBS) scheduling algorithm is modified to deal with job chunks and preventing tasks from exhausting budgets. Overall, this newer method is effective in dealing with the unbounded priority inversion problem.
Real-Time Synchronization Protocols

References


