Computer Control:
Task Synchronisation in Dynamic Priority Scheduling

Sérgio Adriano Fernandes Lopes
Department of Industrial Electronics
Engineering School
University of Minho
Campus de Azurém
4800 Guimarães - PORTUGAL
email: sfl@dei.uminho.pt

António José Pessoa de Magalhães
SAIC - DEMEGI
Faculty of Engineering
University of Porto
Rua dos Bragas
4050 Porto - PORTUGAL
email: apmag@fe.up.pt

Abstract
Due to common resource protection, most real-time tasks have non-preemptive sections. Such sections, called critical sections, rise several problems to real-time scheduling theory. Namely, deadlock avoidance and bounded blocking time. Different and widely mentioned solutions exist for this problem in the context of fixed priority scheduling. However, solutions for the same problem but in the context of totally dynamic scheduling, although much more interesting, are seldom referred in the current literature. This paper surveys those solutions and illustrates their philosophies, providing thus a considerable help for real-time systems designers who develop or intent to develop their applications upon EDF or other totally dynamic scheduling algorithm.

Keywords: Discrete Digital Control Systems, Computer Control, Real-Time Scheduling and Task Synchronization.

1 Introduction
Real-time systems are those that must perform correctly not only the value but also in the time domain. Computer controllers are probably the most known real-time systems. This is because a controller must obey to some time constraint in sensing the environment and perform control actions accordingly. There are many areas of investigation in real-time systems. One of the most interesting and challenging is the scheduling theory. This is particularly true when tasks have to synchronise their executions.

The dynamic scheduling algorithms are known for a long time. Liu and Layland [1] showed that Rate Monotonic (RM) and Earliest Deadline First (EDF) are optimal for fixed and dynamic priority systems, respectively. Unfortunately, Liu and Layland conclusions only apply for independent tasks. Yet, in a multi-tasking control system, tasks are likely to share data or some physical or logical device that cannot be arbitrarily accessed. Tasks that perform in this way are said to be dependent, in the sense that tasks execution depend on each other. In this scenario, the system must provide some protection mechanism around shared resources. Otherwise, resource corruption will occur due to tasks' concurrent accesses to shared resources. Synchronisation mechanisms (like semaphores, monitors, etc.) can protect shared resources. However, they define critical sections in the tasks' code. That is, tasks have portions of code that are non-preemptive, from the point of view of a task that intents to enter an already accessed resource.

Synchronisation poses serious problems in dynamic scheduling. Dependent tasks may easily miss their deadlines. Such scenarios are totally unacceptable in a hard real-time environment. Yet, finding the optimal scheduling in a preemptive system with resource sharing is known to be an NP-hard problem[2]. However, dynamic priority schemes are much advantageous: namely, they provide good processor efficiency. Therefore, a considerable research effort has been done is the context of dependent tasks dynamic scheduling, and solutions have been found. Unfortunately, those solutions do not seem to have had much echo on real-time systems designers. This is probably because such solutions are very dispersed in the current literature and no survey seems to exist.

This paper surveys and illustrates three solutions for the dynamic scheduling of dependent tasks. A special emphasis is placed on the EDF algorithm. By doing this, the paper collects the most common solutions to the stated problem and present them in an organised way. Section 2 describes the problem and the solutions that can...
solve it in fixed priority systems. Sections 3, 4 and 5, survey three synchronisation algorithms for dynamic priority scheduling. Namely Dynamic PCP, Stack Resource Protocol (SRP) and Interruptible Critical Sections (ICS), respectively. Section 6 summarises the most important conclusions.

## 2 Synchronisation Problems and Fixed Priority Solutions

The two major problems that arise in dynamic scheduling systems are multiple priority inversion and deadlock. Priority inversion happens when a high priority task has to wait for a lower priority task to release a shared resource. Figure 1 shows an example of multiple priority inversion. In here, a intermediate priority task $\tau_2$ preempts a lower priority task $\tau_3$ during the blocking of the higher priory task $\tau_1$ by $\tau_3$. The up arrows denote tasks' execution request times and the down ones denote tasks' execution completion times.

The deadlock problem happens when at least two tasks block simultaneously waiting for any other to release a shared resource. This scenario is illustrated in Figure 2 by $\tau_2$ and $\tau_3$, while $\tau_1$ (not sharing any resource with any of the other tasks) is normally executed.

Two solutions for these problems are presented by Sha, Rajkumar and Lehoczky [3] in the context of fixed priority scheduling, if critical sections are protected by binary semaphores. These are the Priority Inheritance Protocol (PIP) and the Priority Ceiling Protocol (PCP). The former states that when a lower priority task executing a critical section blocks a higher priority task, it transitively abandons its actual priority and inherits the priority of the blocked task. The latter defines a priority ceiling for each semaphore, and states that a task can only enter a critical section if its actual priority is higher than the priority of all semaphores' ceilings currently locked by other tasks. The priority ceiling of each semaphore equals the priority of the highest priority task that can use it. By using PCP, both multiple priority inversion and deadlock are prevented. Furthermore, chained blocking is also avoided.

Whist the PCP is not directly applied to dynamic priority scheduling, it is a good point of departure. Moreover, it can be easily adapted to totally dynamic scheduling as it will be shown in the next section.

## 3 Dynamic PCP

The Dynamic PCP, firstly presented by Chen and Lin [4], is an extension of PCP to dynamic priority scheduling. It assumes that a task priority is given by the deadline of its current execution if it is pending; or by the deadline of its next execution, otherwise. Since tasks priorities change with time, so do the priority ceilings of the semaphores controlling the access to shared resources.

In priority dynamic algorithms we need to distinguish between a task's priority and the priority of its executions. The former, called original priority and denoted $p(t)$, depends only on the deadlines of its sequence of executions. The latter, called augmented priority and denoted $p^*(t)$, depends also on the priority inheritance mechanism.

The Dynamic PCP is given by two rules:
At instant \( t \), the priority ceiling of a semaphore \( S \), \( c(t) \), is the original priority of the highest priority task \( \tau_p \) that locks, or may lock, \( S \) at time \( t \) or later. That is, \( c(t) = p_{HF}(t) \).

When the execution of a task \( \tau \) tries to lock \( S \), it gets the lock if \( p^*(t) > c_{HF}(t) \), where \( S_{HF} \) is the semaphore with highest priority ceiling among all the semaphores locked by others executions at that time. Otherwise, the task's execution is suspended, and the execution of task \( \tau \), which currently locks \( S_{HF} \), inherits \( p^*(t) \) until it frees \( S_{HF} \).

This solution, besides solving the problems previously discussed, possesses an interesting property: if a task is blocked, that will occur in its first critical section. Therefore, the lock condition only needs to be verified for the first semaphore lock of each task. Figure 3 shows how this algorithm applies (\( d(t_i) \) denotes the priority associated with a deadline at instant \( t_i \), according to EDF).

\[
\begin{align*}
t_0: & \quad p_1^* = d(t_7), p_2^* = d(t_9) \text{ and } p_3^* = d(t_8). \quad \text{and } p_3 = d(t_9), p_1 = d(t_7), p_2 = d(t_8) \text{ and } p_3 = d(t_8). \\
t_1: & \quad p_2 > p_3. \\
t_2: & \quad p_5 > p_3, \quad p_3 = d(t_8). \\
t_3: & \quad p_3^* = d(t_9), \quad p_2^* = d(t_8) > p_3^* = d(t_9). \\
t_4: & \quad p_1^* = d(t_7) > p_2^* = d(t_8), \quad p_1^* > c_2. \\
t_5: & \quad p_1 = d(t_8+T_1), \quad c_1 = d(t_8+T_1). \\
t_6: & \quad p_2 = d(t_8+T_2), \quad c_2 = d(t_8+T_2), \quad p_3 = d(t_8+T_3).
\end{align*}
\]

Figure 3 - Deadlock Prevention with Dynamic PCP.

Chen and Lin show that applying Dynamic PCP synchronisation, a set of \( n \) periodic tasks are schedulable by the EDF if

\[
\sum_{i=1}^{n} \frac{C_i + B_i}{T_i} \leq 1,
\]

where, \( C_i \) denotes maximum execution time, \( T_i \) is the period and \( B_i \) is the priority inversion bound, of task \( t_i \).

## 4 Stack Resource Policy

Baker [5] presents another solution that extends the PCP in three very useful ways:

- multi-unit resources – allowing other more general resource protection than simple binary semaphores do;
- support for various scheduling policies – EDF, RM, Deadline Monotonic and combinations of those;
- runtime stack sharing – resulting in memory savings, so important in real-time systems.

Besides its priority, each task \( \tau \) has also an associated fixed preemption level \( \pi \). A task \( \tau \) may only preempt another task \( \tau_i \) if \( \pi = \pi_j \). The preemption levels must be defined in such a way that the priority concept is not violated (see Baker [5]). One possible definition is ordering them inversely with respect to the relative deadlines.

Each resource \( R_r \) has a preemption ceiling, \( c_r(v_r) \), that depends on the number of units currently available of that resource. The definition is:

\[
c_r(v_r) \text{ is the maximum between zero and the preemption levels of all the tasks' executions that may be blocked when there are } v_r \text{ units of } R_r \text{ available.}
\]

Denoting by \( \mu_r(\tau) \), the maximum needs in units of a resource \( R_r \) by a task \( \tau \), the previous definition can be stated more formally:

\[
c_r(v_r) = \max(\{0\} \cup \{v_r: v_r \leq \mu_r(\tau)\}).
\]
For simplicity, it is also convenient to define a global ceiling of the system $\pi$:

$$\pi = \max (c_r, r = 1, \ldots, m),$$

where $m$ is the number of resources.

Now is then possible to define the SRP: the request for execution of a task $\tau_i$ is blocked until $\pi_i > \pi$.

Figure 4 exemplifies the application of this synchronisation algorithm in the context of the EDF. Three tasks $\tau_1$, $\tau_2$, and $\tau_3$ and three resources $R_1$, $R_2$, and $R_3$ are considered. The tasks’ preemption levels are $\pi_1 = 3$, $\pi_2 = 2$ and $\pi_3 = 1$. The number of units of each resource are $NR_1 = 3$, $NR_2 = 1$ and $NR_3 = 3$. Finally, the maximum needs of each resource by each tasks are given in Table 1. According to these parameters, preemption ceilings of each resource for each number of available units are given in Table 2.

<table>
<thead>
<tr>
<th>Resource</th>
<th>$c_r(0)$</th>
<th>$c_r(1)$</th>
<th>$c_r(2)$</th>
<th>$c_r(3)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1$</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$R_2$</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$R_3$</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 1 - Maximum resource needs of the tasks represented in Figure 4.

<table>
<thead>
<tr>
<th>Resource</th>
<th>$\mu_r(\tau_1)$</th>
<th>$\mu_r(\tau_2)$</th>
<th>$\mu_r(\tau_3)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1$</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>$R_2$</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$R_3$</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2 - Preemption ceilings for the tasks represented in Figure 4.

Baker also proves that a set of $n$ tasks (periodic and aperiodic), with relative deadlines equal to the period ($D_i = T_i$), is schedulable by the EDF if

$$\forall k, \sum_{i=1}^{k} \frac{C_i}{T_i} + \frac{B_k}{T_k} \leq 1.$$  (4)

It is worth noticing that this result is better than the one provided by the Dynamic PCP (compare with equation 1), in the senses that there is only one blocking term in the sum. Also worth noting is that the runtime stack sharing can be achieved simply by defining its preemption ceiling as zero. As it never blocks any execution, it may be ignored in the computation of the priority inversion bounds $B_k$. 
5 Interruptible Critical Sections

So far we’ve seen two pessimistic synchronisation protocols, in the sense that they cause blocking. Yet, optimistic (i.e. non-blocking) concurrency control exist. One of these techniques is the ICS protocol presented by Johnson [6]. The advantages of the ICSs include: no high priority task ever waits for a lower priority one and the synchronisation protocol is independent of the scheduling algorithm. However, ICSs have their own limitations: their purpose is to protect shared data (objects) and it takes a little more overhead for the operating system during context switches.

The ICSs are based on Restartable Atomic Sequences (RASs). A RAS is a sequence of code that is re-executed from the beginning if it is interrupted by a context switch.

Each operation on the shared object (executed in an ICS) computes its modifications in a private set of records obtained from a global stack of records. To commit the operation the ICS has to:

- remove from the global stack the records used;
- add to the global stack the garbage records;
- link the new records to the shared data structure by changing the value of a pointer.

The operation is committed by the execution of a decisive instruction. Every shared object has a commit record and a flag that indicates if its commit record is valid or invalid.

The execution of an ICS is:

If a previous operation has left a valid commit register, it executes the respective writes to the shared object (the change is asserted – cleaning phase) and sets the flag invalid.

Computes its changes in the private set of records and writes them in the commit register.

Sets the commit register flag valid (decisive instruction).

Figure 5 illustrates this method applied to the case of a linked list.

```
struct CommitRecordElement {
  word *lhs, rhs;
} CommitRecord[MAX];
boolean valid;
record *GlobalRecordStack;

ICS () {
  record *current, *GarbageHead, *GarbageTail;
  word instruction;
  RAS {
    if (Valid) { instruction = 0;
      while (instruction < 3 && CommitRecord[instruction].lhs != NULL) {
        *(CommitRecord[instruction].lhs) = CommitRecord[instruction].rhs;
        instruction ++;
      } valid = FALSE; }
    current = GlobalRecordStack;   // iniciates list pointers
    GarbageHead = GarbageTail = NULL;  // computes modifications to the data structure
    CommitRecord[0].lhs = &(GarbageTail->next);  // save modifications in CommitRecord
    CommitRecord[0].rhs = current;
    CommitRecord[1].lhs = GlobalRecordStack;
    CommitRecord[1].rhs = GarbageHead;
    CommitRecord[2].lhs = CriticalLink;
    CommitRecord[2].rhs = CriticalLinkValue;
    valid = TRUE;    // commits the operation
  }
}
```

Figure 5 - ICS Applied to a Linked List Data Structure.
By ensuring that no high priority task is blocked by a lower priority one, this algorithm prevents priority inversion. Figure 6 illustrates this property; it assumes that, before $t_1$, the last operations upon $O_1$ and $O_2$ were accomplished by $\tau_1$ and $\tau_2$, respectively.

![Figure 6 - Priority Inversion Prevention with ICSs.](image)

The schedulability analyses for this synchronisation algorithm is not known by the time of this writing. The only bounding stated by Johnson in [6] is that, under the worst possible scenario, a task executes twice each critical section. Therefore, we have:

$$C_i \leq E_i + 2 \sum_{j=1}^{n_i} I_{i,j}$$

where, $E_i$ is the maximum execution time of the task $\tau_i$ excluding the duration of critical sections, and $I_{i,j}$ is time spent executing the $j_{th}$ critical section of task $\tau_i$.

6 Conclusion

General purpose synchronisation primitives that may lead a task to unpredictable blocking times or deadlocks are no solution for real-time systems. Feasible solutions to this problem exist. Moreover, they are widely refereed in the current literature, but mostly in the context of fixed priority scheduling. Consequently solutions for real-time task synchronisation in the context of dynamic scheduling are mostly unknown to most real-time systems designers. This paper has intended to fill this gap by surveying, illustrating and comparing the most important solutions presented in the specialised literature. We hope this can help real-time community in designing modern and realistic control systems based on dynamic algorithms.

References


