Patterns which help to avoid conflicts over shared resources in time-triggered embedded systems which employ a pre-emptive scheduler

Huiyan Wang, Michael J Pont and Susan Kurian
Embedded Systems Laboratory, University of Leicester,
University Road, LEICESTER LE1 7RH, UK.
hw79@le.ac.uk; M.Pont@le.ac.uk; sk183@le.ac.uk
http://www.le.ac.uk/eg/embedded/

Abstract

This paper is concerned with the use of patterns to support the development of software for reliable, resource-constrained, embedded systems. The specific focus is on systems with a time-triggered architecture in which task pre-emption can occur. The paper introduces one new abstract pattern (CRITICAL SECTION), and four new design patterns (DISABLE TIMER INTERRUPT, RESOURCE LOCK, PRIORITY INHERITANCE PROTOCOL and IMPROVED PRIORITY CEILING PROTOCOL).

Acknowledgements

We thank Juergen Salecker, who provided numerous useful suggestions on earlier drafts of this paper during the shepherding process.

This work is supported by a DTA award to Huiyan Wang from the UK Government (EPSRC).

Copyright

Introduction

We are concerned with the development of embedded systems for which there are two (sometimes conflicting) constraints. First, we wish to implement the design using a low-cost microcontroller, which has – compared to a desktop computer – very limited memory and CPU performance. Second, we wish to produce a system with extremely predictable timing behaviour.

To support the development of this type of software, we have previously described a "language" consisting of more than seventy patterns for time-triggered (TT) embedded systems (e.g. see Pont, 2001). Work began on these patterns in 1996, and they have since been used it in a range of industrial systems and numerous university research projects (e.g. see Mwelwa et al., 2007; Kurian and Pont, 2007).

The patterns presented in this paper mark the start of a new area of work. Unlike the majority of papers we have presented at previous PLoP conferences, the work described here is based on the use of a pre-emptive scheduler. This brief paper describes one new abstract pattern (CRITICAL SECTION), and four new design patterns (DISABLE TIMER INTERRUPT, RESOURCE LOCK, PRIORITY INHERITANCE PROTOCOL and IMPROVED PRIORITY CEILING PROTOCOL). The solutions presented in this set of patterns are from published papers: they have been adapted (as necessary) to work with TT architectures and documented in pattern format.

Pattern structure

As our experience with our pattern language for TT systems has grown, we have felt that there were ways in which the overall architecture could be improved in order to make the collection easier to use, and to reduce the impact of future changes. We are therefore in the process of re-working the collection into different pattern categories: Abstract Patterns, Design Patterns and Pattern Implementation Examples (PIEs). Very briefly, abstract patterns address common design decisions faced by developers of embedded systems and form entry points for the collection. Design Patterns have a sharper solution on specific solutions. PIEs provide implementation-specific details (Kurian and Pont, 2006).

References

Kurian, S. and Pont, M.J. (2006) "Restructuring a pattern language which supports time-triggered co-operative software architectures in resource-constrained embedded systems". Paper presented at the 11th European Conference on Pattern Languages of Programs (EuroPLoP 2006), Germany, July 2006.

Kurian, S. and Pont, M.J. (2007) "Maintenance and evolution of resource-constrained embedded systems created using design patterns", *Journal of Systems and Software*, **80**(1): 32-41.

Mwelwa, C., Athaide, K., Mearns, D., Pont, M.J. and Ward, D. (2007) "Rapid software development for reliable embedded systems using a pattern-based code generation tool". *SAE Transactions: Journal of Passenger Cars (Electronic and Electrical Systems)*, **115**(7): 795-803.

Pont, M.J. (2001) "Patterns for Time-Triggered Embedded Systems: Building Reliable Applications with the 8051 Family of Microcontrollers", Addison-Wesley / ACM Press. ISBN: 0-201-331381.

CRITICAL SECTION

{abstract pattern}

Context

- You are developing an embedded system using a computer system with CPU and / or memory resources which are – compared with typical desktop designs – rather limited.
- Your system employs a single CPU.
- Your system employs a TT SCHEDULER [Pont et al., 2007: this conference].
- Your system supports task pre-emption.
- Predictable system behaviour is a key design requirement: in particular, predictable task timing is a concern.

Problem

How can you avoid conflicts over shared resources during the execution of critical sections?

Background

We provide some relevant background material in this section.

Scheduling and TT architectures

For general background information about scheduling (and scheduling of time-triggered systems in particular), please refer to the pattern TT SCHEDULER [Pont et al., 2007: this conference]. TT SCHEDULER provides background information on key concepts such as TTC, TTH and TTRM scheduling.

Shared resources and critical sections

Our focus in this pattern will be on TTH and TTRM designs in which task pre-emption can occur. Our particular concern will be with the issue of resources which may be accessed by more than one task at the same time. Such "shared resources" may – for example - include areas of memory (for example, two tasks need to access the same global variable) or hardware (for example, two tasks need to access the same analogue-to-digital converter). The code which accesses such shared resources is referred to as a "critical section".

Suppose that there are two tasks, $Task_A$ and $Task_B$ in a system, which are illustrated in Figure 1. There is one shared resource. In the figure, N represents the normal section and C represents the critical section (that is, the section which involve access to a shared resource). From t1 to t4, $Task_A$ and $Task_B$ are attempting to run "simultaneously".

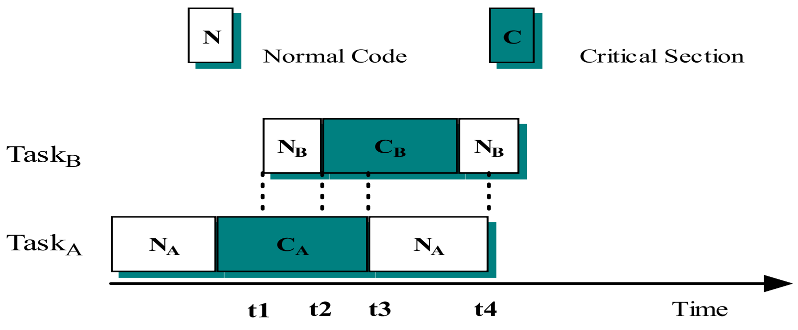


Figure 1 Two Tasks with a shared resource

In a TTC system, a task cannot be pre-empted by another task and the two tasks shown in Figure 1 are scheduled as shown in Figure 2. As in this example, there are no conflicts caused by the shared resources in TTC systems.

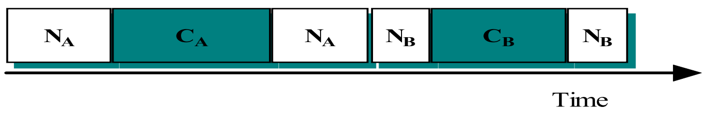


Figure 2 Two tasks scheduled using a TTC scheduler

However, if the same tasks are scheduled in a pre-emptive system, there may be conflicts. For example, suppose the priority of $Task_B$ is higher than that of $Task_A$. $Task_B$ will then pre-empt $Task_A$ at time t1 while it is running the critical section (Figure 3).

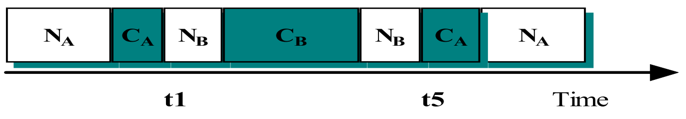


Figure 3 Two tasks scheduled in a pre-emptive system

Assume that the shared resource in the above example is some shared data (for example, numerical data stored in an array). The two tasks write and read the data in the critical section. $Task_B$ pre-empts $Task_A$ at t1 while it is reading the data: we will assume that $Task_A$ has read from half of the array at the time it is interrupted. We will further assume that $Task_B$ $Task_B$ updates all of the data values in the array. After $Task_B$ finishes, $Task_A$ then continues, reading the remaining values from the second half of the array: it then processes a combination of "new" and "old" data, possibly leading to erroneous results (e.g. see Kalinsky, 2001).

In general, tasks must share data and / or hardware resources. However, the system designer must ensure that each task has exclusive access to the shared resources to avoid conflicts, data corruption or "hanging tasks" (Pont, 2001; Labrosse, 2002; Laplante, 2004).

What is a resource lock?

A lock is the most common way to protect shared resources.

Before entering a critical section, a semaphore is checked. If it is clear, the resource is available. The task then sets the semaphore and uses the resource. When the task finishes with the resource, the semaphore is cleared.

Resource locking in this way requires care but is comparatively straightforward to implement. and affects only those tasks that need to take the same semaphore (Simon, 2001).

The main drawback is that it causes priority inversion in a priority based system (Sha, 1990; Burns, 2001; Renwick, 2004).

What is priority inversion?

In a priority-based system, each task is assigned a priority. In a TTC design, the scheduler will – when deciding which task to run next – always run the task with the highest priority (and this task will then run to completion). In a pre-emptive system, a high-priority task may interrupt a lower-priority task while it is executing.

Priority inversion can occur in pre-emptive designs when resource locks are used. For example, suppose that a low-priority task is using a resource. The resource will be locked. If a high-priority task is then scheduled to run (and use the resource) it will not be able to do so: in effect, the low-priority task will be given greater priority than the high-priority task.

For example, Figure 4 shows an intended operation sequence for two tasks, Task_H and Task_L, sharing a critical section C. Figure 5 shows how the priority inversion takes place. When Task_L owns C, and Task_H attempts to access it (at t_3), Task_H is blocked and has to wait until time t_4 before it can run.

Please note that this is sometimes called “bounded priority inversion” (Burns, 2001; Renwick, 2004) or “controlled priority inversion” (Locke, 2002). In this case, the blocking time of Task_H will not exceed the duration of the critical section C of Task_L.

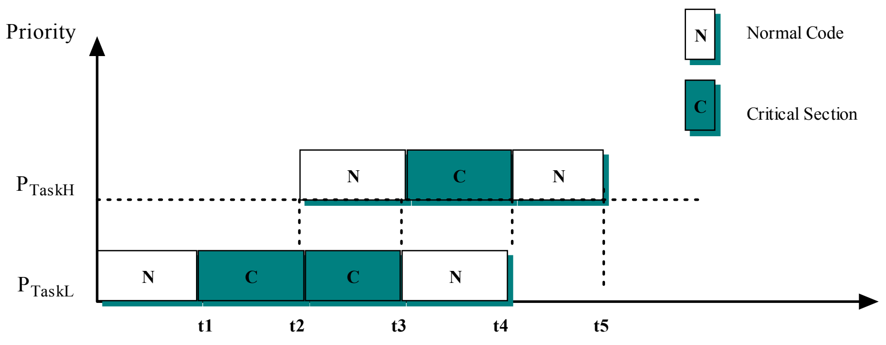


Figure 4 Operation Sequences of TaskH and TaskL

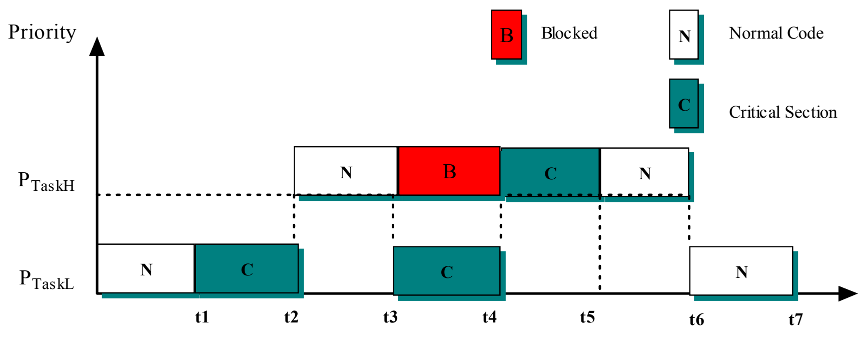


Figure 5 Bounded Priority Inversion

We further suppose that Task$_M$ (with "medium" priority) pre-empts Task$_L$ when Task$_H$ is blocked by Task$_L$, the owner of the shared resource at this time. Task$_H$ then has to wait until Task$_M$ relinquishes control of the processor and Task$_L$ completes the critical section. For example, see Figure 6: here, at t$_4$, Task$_M$ pre-empts Task$_L$.

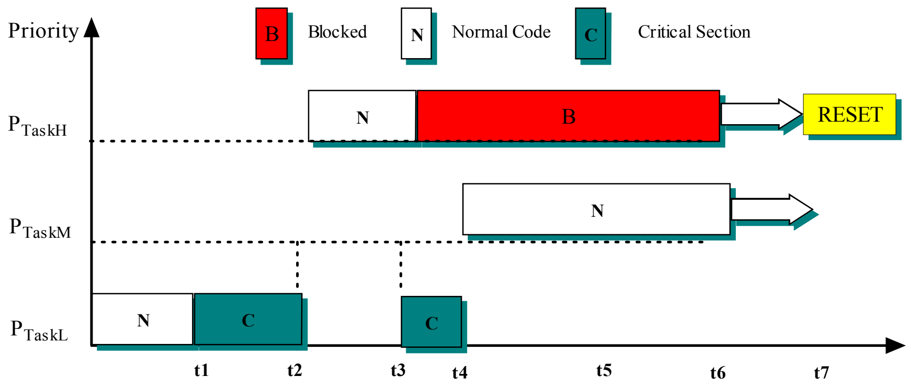


Figure 6 Unbounded Priority Inversion

In these circumstances, the worst-case waiting time for Task$_H$ is the sum of the worst-case execution times of Task$_M$ and the critical section of Task$_L$. This is called unbounded priority inversion (Renwick, 2004). If Task$_M$ runs for a long time (or "for ever"), Task$_H$ is likely to may miss its deadline, with potentially serious consequences (shown as a system reset in Figure 6).

Unbounded priority inversion can be particularly problematic. For example, in 1997, the Mars Pathfinder mission nearly failed because of an undetected (unbounded) priority inversion (Jones, 1997).

What is deadlock?

As noted above, a locking mechanism may lead to priority inversion. However, this is not the only problem which is introduced by the use of locking mechanisms.

For example, suppose that Task$_H$ is waiting for a resource held by Task$_L$, while Task$_L$ is simultaneously waiting for a resource held by Task$_H$: neither task is able to proceed and – as a result - a *deadlock* is formed.

As an example, Figure 7 shows two tasks $Task_H$ and $Task_L$ which share two resources (via C1 and C2): in this case, it is assumed that C1 is nested within C2 in $Task_L$ and that C2 is nested within C1 in $Task_H$

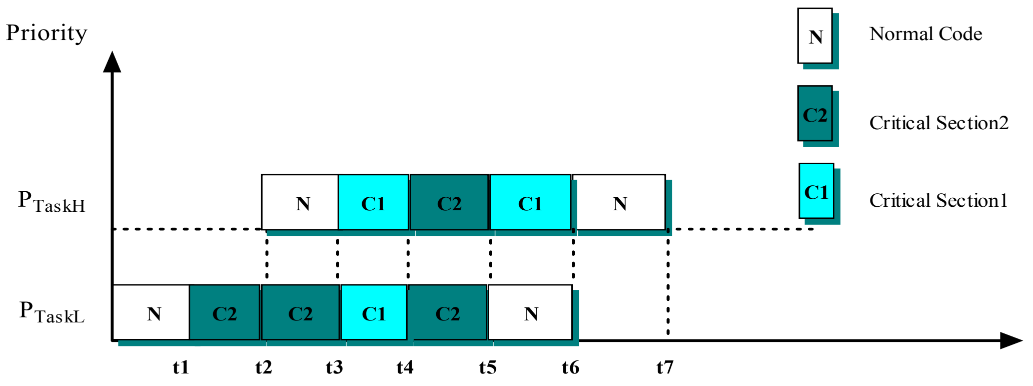


Figure 7 Operation Sequences of TaskH and TaskL

In Figure 8, at t_1, $Task_L$ locks C2, at t_2 $Task_H$ pre-empts $Task_L$ and starts to run, locks C1 at t_3, then requires C2 at t_4. Due to the fact that $Task_L$ has locked C2, $Task_H$ is blocked and $Task_L$ resumes running at t_4. At t_5, $Task_L$ requires C2 which is locked by $Task_{H,}$ and both tasks are blocked.

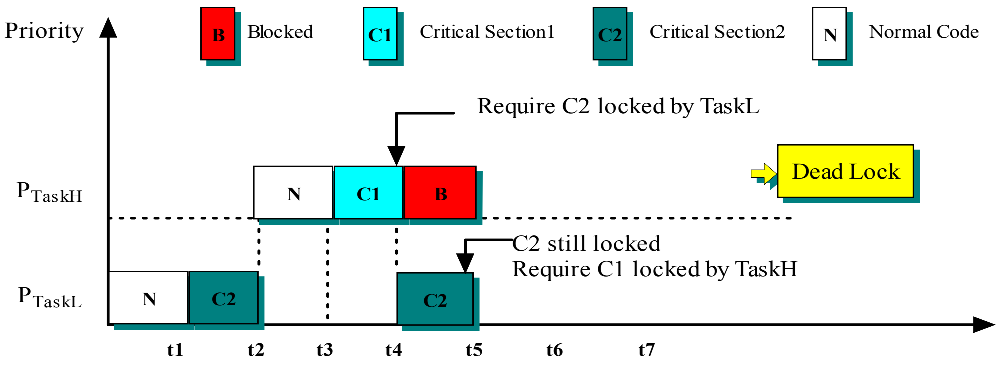


Figure 8 Deadlock: Operation Sequences of Tasks without Priority Protocols

What is chained blocking?

Locking mechanisms can also cause a phenomenon known as chained blocking.

This is best explained by means of an example. Suppose that $Task_H$ is waiting for a resource held by $Task_M$, while $Task_M$ is waiting for a resource held by $Task_L$, and so on (Figure 9). $Task_H$ needs to sequentially access resources C3 and C2. $Task_M$ accesses C2 (with nested C1) and $Task_L$ accesses C1.

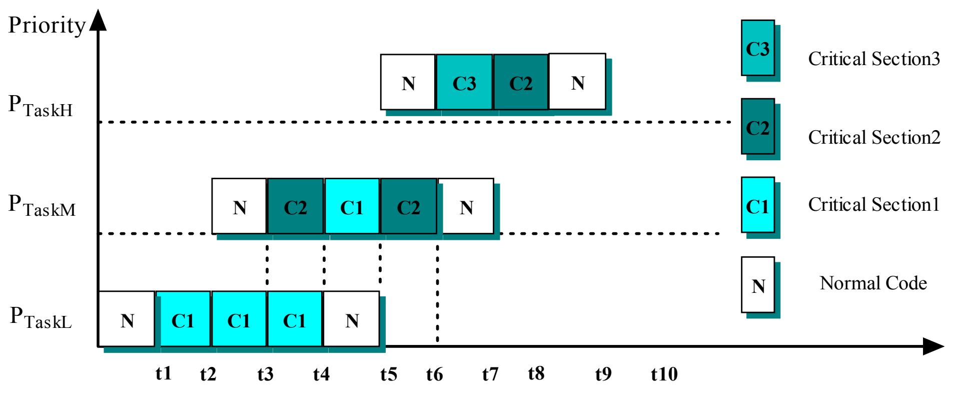


Figure 9 Operation Sequences of Three Tasks

Figure 10 shows that $Task_M$ is blocked by $Task_L$ at t_4 when it requires access to C1 (which is locked by $Task_L$). $Task_H$ is blocked by $Task_M$ at t_7 when it requires access to C2 (which is locked by $Task_M$). Therefore $Task_H$ is blocked for the duration of two critical sections (it has to wait for $Task_L$ to release C1, and then wait for $Task_M$ to release C2). As a result, a *blocking chain* is formed (Sha, 1990).

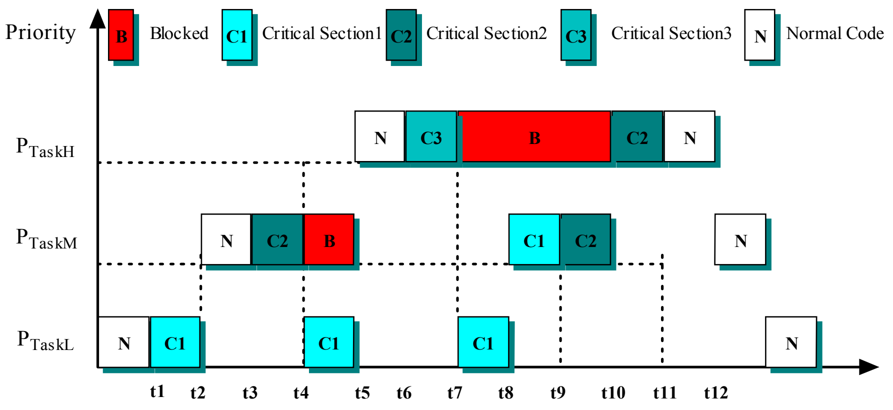


Figure 10 A blocking chain: Operation Sequences of Three Tasks without priority protocols

Solution

This pattern is intended to help answer the question: “How can you avoid conflicts over shared resources during the execution of critical sections?”

In general, the answer to this question is straightforward: you need to ensure that only one task attempts to access each shared resource at any time. There are two common ways of achieving this in a TT system architecture:

1. As noted in “Background”, you can avoid conflicts over shared resources in a time-triggered system if you use a TTC scheduler. This solution avoids the need for any of the mechanisms outlined in “Related patterns and alternative solutions”.
2. You can disable interrupts and / or using a locking mechanism (probably in conjunction with a protocol that will help you avoid priority inversions). These solutions are outlined in “Related patterns and alternative solutions”.

Of these solutions, the first is the simplest and generally the most effective. No matter what you do in a pre-emptive design to protect your shared resources, they will still be shared and only one task can use them at a time. **As such, any form of protection mechanism provides only a partial solution to the problems caused by multi-tasking.**

Consider an example. If the purpose of Task A is to read from an ADC, and Task B has locked the ADC when the Task A is invoked, then Task A cannot carry out its required activity. Use of locks, or any other mechanism, will not solve this problem; however, they may prevent the system from crashing.

Please note that there may – in some circumstances – be two further options for you to consider:

1. Pre-runtime scheduling (e.g. Xu and Parnas, 1990). Use of a "pre-run time schedule design[1]" may allow you to adapt your pre-emptive system schedule in order to ensure that – even with pre-emption – there are never conflicts over shared resources. Such techniques are not trivial to implement and are beyond the scope of the present paper.
2. Planned pre-emption. Adi and Pont (2005) have described an approach called "planned pre-emption" which avoids the need for locking mechanisms in TTH scheduler designs.[2]

Related patterns and alternative solutions

This pattern is an abstract pattern, which provides background knowledge related to shared resources in embedded systems.

The following patterns describe some solutions to avoid shared resources conflicts and priority inversion:

DISABLE TIMER INTERRUPT

Disable interrupt is the simplest and fastest approach considered in this paper. However it may affect the response times of all other tasks in the system.

RESOURCE LOCK

Lock is the most common way to protect shared resources because it affects only those tasks that need to take the same semaphore. However, basic use of locking mechanisms can give rise to problems of priority inversion.

PRIORITY INHERITANCE PROTOCOL

The Priority Inheritance Protocol is intended to address problems with priority inversion.

[1] It can be argued that any form of static schedule (e.g. most TTC schedules) could be described as a "pre-runtime schedules". However, this phrase is usually used to refer to static designs involving pre-emption for which a detailed modelling process is carried out prior to program execution.

[2] Planned Pre-emption will be described in a future pattern. It is not considered further in this paper. Please see Maaita and Pont (2005) for further details.

IMPROVED PRIORITY CEILING PROTOCOL

The Improved Priority Ceiling Protocol is intended to address problems with priority inversion, deadlock and chained blocking.

Reliability and safety implications

If a system is to implement in pre-emptive architecture, applying the mechanisms discussed in this pattern will generally help you to construct a reliable and safe embedded system.

Overall strengths and weaknesses

☺ Being aware of the need to safeguard critical sections can help to increase system reliability

☹ Inappropriate use of locking mechanisms and related techniques may increase system complexity without increasing reliability

DISABLE TIMER INTERRUPT

{pattern}

Context

- You are developing an embedded system using a computer system with CPU and / or memory resources which are – compared with typical desktop designs – rather limited.
- Your system employs a single CPU.
- Your system employs a TT SCHEDULER [Pont et al., 2007: this conference].
- Your system supports task pre-emption.
- Predictable system behaviour is a key design requirement: in particular, predictable task timing is a concern.

Problem

What is the simplest way of ensuring safe access to shared resources in your system?

Background

We provide some relevant background material in this section.

What is a shared resource?

For background information about shared resources, please refer to the abstract pattern CRITICAL SECTION [this paper].

The role of interrupts in TT systems

In general, an interrupt is a signal that is used to inform the processor that an event has occurred. Such event may include a timer overflow, completion of an A/D conversion or arrival of data in a serial port.

In TT systems, we only have a single interrupt source, linked to a timer overflow[3].

Solution

This pattern is intended to describe the simplest way of avoiding conflicts over shared resources in a TT system which involves task pre-emption.

As noted in Background, only a single interrupt is enabled in a time triggered system. This interrupt will be used to drive the scheduler (Pont, 2001). If we disable this interrupt, the scheduler will be disabled.

[3] It is possible – using a Super Loop – to create very simple TTC designs which involve no interrupts (at all). Such architectures are not suitable for use with pre-emptive task sets and are not considered in this set of patterns: see Kurian and Pont (2007) for further details.

This gives us a simple mechanism to avoid conflicts over resources, as follows:

- When a task accesses a shared resource, it disables the timer interrupt.
- When the task has finished with the resource it re-enables the timer interrupt.
- During the time that our task is using the shared resource, the scheduler is disabled. This means that no context switch can occur, and no other task can attempt to gain access to the resource.

Overall, this is a very simple (and fast) way of dealing with issues of shared resources in a TT design. However, it may have an impact on all the tasks in the system. Therefore, interrupts should be disabled as little as possible (and for a very short period of time).

Related patterns and alternative solutions

The pattern CRITICAL SECTION provides general background material on mechanisms for dealing with shared resources in TT systems which involve task pre-emption.

The following patterns describe some alternative ways of handling conflicts over shared resourcess:

- RESOURCE LOCK
- PRIORITY INHERITANCE PROTOCOL
- IMPROVED PRIORITY CEILING PROTOCOL

Reliability and safety implications

Disabling the interrupt of a system affects the response times of the interrupt routine and of all other tasks in the system. It is not safe if it keeps interrupt disable for long time. However, if the critical section is very short (e.g. we wish to access a single global variable), it is a fast and easy solution.

Overall strengths and weaknesses

☺ Easy to implement

☺ Faster than other protection mechanisms, such as locks

☹ Increases interrupt latency

☹ May decrease system’s ability to respond to external events

☹ Need carefully recognise the situation in which interrupts should be disabled

RESOURCE LOCK

{pattern}

Context

- You are developing an embedded system using a computer system with CPU and / or memory resources which are – compared with typical desktop designs – rather limited.
- Your system employs a single CPU.
- Your system employs a TT SCHEDULER [Pont et al., 2007: this conference].
- Your system supports task pre-emption.
- Predictable system behaviour is a key design requirement: in particular, predictable task timing is a concern.

Problem

How can you implement a resource lock for your embedded system?

Background

We provide some relevant background material in this section.

What is a shared resource?

For background information about shared resources, please refer to the abstract pattern CRITICAL SECTION [this paper].

The role of interrupts in TT systems

For background information about interrupts in TT systems, please see DISABLE TIMER INTERRUPTS [this paper].

Solution

The pattern is intended to help you answer the question: "How can you implement a resource lock for your embedded system?"

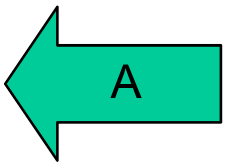
A lock appears, at first inspection, very easy to implement. Before entering the critical section of code, we 'lock' the associated resource; when we have finished with the resource we 'unlock' it. While locked, no other process may enter the critical section.

This is one way we might try to achieve this:

1. Task A checks the 'lock' for Port X it wishes to access.
2. If the section is locked, Task A waits.
3. When the port is unlocked, Task A sets the lock and then uses the port.
4. When Task A has finished with the port, it leaves the critical section and unlocks the port.

Implementing this algorithm in code also seems straightforward, as illustrated in Listing 1.

```
#define UNLOCKED   0
#define LOCKED     1

bit Lock;  // Global lock flag

// ...

// Ready to enter critical section
// - Wait for lock to become clear
// (FOR SIMPLICITY, NO TIMEOUT CAPABILITY IS SHOWN)
while(Lock == LOCKED);

// Lock is clear
// Enter critical section

// Set the lock
Lock = LOCKED;

// CRITICAL CODE HERE //

// Ready to leave critical section
// Release the lock
Lock = UNLOCKED;

// ...
```



Listing 1: Attempting to implement a simple locking mechanism in a pre-emptive scheduler. See text for details.

However, the above code cannot be guaranteed to work correctly under all circumstances.

Consider the part of the code labelled 'A' in Listing 1. If our system is fully pre-emptive, then our task can reach this point at the same time as the scheduler performs a context switch and allows (say) Task B access to the CPU.

If Task B also requires access the Port X, we can then have a situation as follows:

- Task A has check the lock for Port X and found that the port is not locked; Task A has, however, not yet changed the lock flag.
- Task B is then 'switched in'. Task B checks the lock flag and it is still clear. Task B sets the lock flag and begins to use Port X.
- Task A is 'switched in' again. As far as Task A is concerned, the port is not locked; this task therefore sets the flag, and starts to use the port, unaware that Task B is already doing so.
- …

As we can see, this simple lock code violates the principal of mutual exclusion: that is, it allows more than one task to access a critical code section. The problem arises because it is possible for the context switch to occur after a task has checked the lock flag but before the task changes the lock flag. **In other words, the lock 'check and set code' (designed to control access to a critical section of code), is itself a critical section.**

This problem can be solved. For example, because it takes little time to 'check and set' the lock code, we can disable timer interrupt for this period (see DISABLE TIMER INTERRUPT [this paper]).

Related patterns and alternative solutions

In situations where you have more than two levels of task priority and you use a lock, you will generally need to use an appropriate locking protocol to avoid problems with priority inversion, deadlock and chained blocking. CRITICAL SECTION [this paper] provides background information on these topics.

The patterns PRIORITY INHERITANCE PROTOCOL [this paper] and IMPROVED PRIORITY CEILING PROTOCOL [this paper] describe solutions to some of the problems caused by use of resource locks in systems with more than 2 levels of task priority.

Reliability and safety implications

As discussed in CRITICAL SECTION [this paper], use of a resource lock can give rise to problems of priority inversion. The patterns PRIORITY INHERITANCE PROTOCOL and IMPROVED PRIORITY CEILING PROTOCOL provide (partial) solutions to this problem.

Overall strengths and weaknesses

☺ Easy to implement

☹ May give rise to "priority inversion" if not implemented with care.

PRIORITY INHERITANCE PROTOCOL

{pattern}

Context

- You are developing an embedded system using a computer system with CPU and / or memory resources which are – compared with typical desktop designs – rather limited.
- Your system employs a single CPU.
- Your system employs a TT SCHEDULER [Pont et al., 2007: this conference].
- Your system supports task pre-emption.
- Predictable system behaviour is a key design requirement: in particular, predictable task timing is a concern.

Problem

How can you ensure that access to shared resources in your system is mutually exclusive and avoids priority inversion?

Background

For background information about shared resources, please refer to the abstract pattern CRITICAL SECTION [this paper].

Solution

The pattern is intended to help you answer the question: "How can you ensure that access to shared resources in your system is mutually exclusive and avoids priority inversion?"

To avoid unbounded priority inversion, Sha et al (1990) introduced the priority inheritance protocol.

In the priority inheritance protocol, a low priority task inherits the priority of a high priority task if the high priority task requires access to the shared resource owned by the low priority task. The high priority task is blocked and the low priority task can continue executing its critical section until it releases the resource. Then its priority returns to the original and the high priority task starts to run.

This process is illustrated in Figure 11. In this example, the priority of $Task_L$ is raised to the priority of $Task_H$ once the higher-priority task tries to access the critical section (at t_3).

If an medium-priority $Task_M$ pre-empts $Task_L$ while executing the critical section, due to the fact that the priority of $Task_L$ has been raised to the priority of $Task_H$, $Task_M$ has to wait until $Task_H$ completes and $Task_L$ finishes the critical section. Therefore, the highest-priority task $Task_H$ is not pre-empted by the medium-priority $Task_M$.

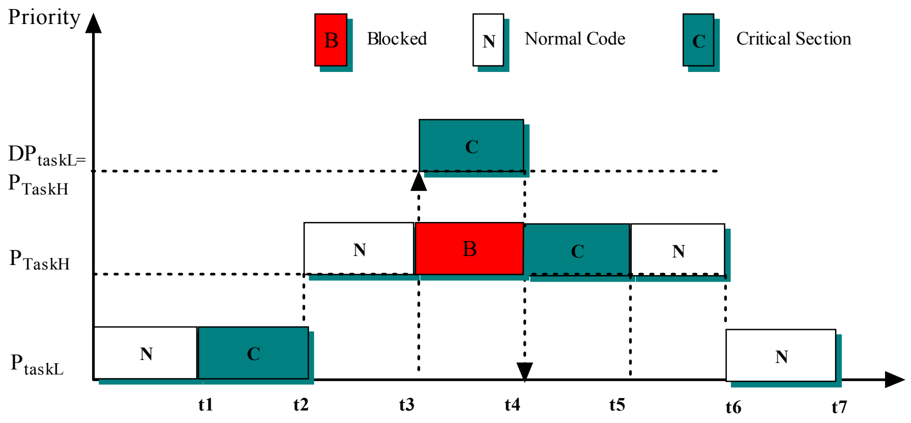


Figure 11 Priority Inheritance Protocol

Related patterns and alternative solutions

The pattern CRITICAL SECTION provides general background material on mechanisms for dealing with shared resources in TT systems which involve task pre-emption.

The following patterns describe some alternative ways of handling conflicts over shared resourcess:

- DISABLE TIMER INTERRUPT
- RESOURCE LOCK
- IMPROVED PRIORITY CEILING PROTOCOL

Reliability and safety implications

Use of priority inheritance protocol avoids priority inversion, increases the stability of a system. Most of commercial real time operating system support this feature, or as additional package, such as μC/OS-II, eCOS, FreeRTOS and RTLinux etc.

Although priority inheritance protocol is generally found to be an effective and powerful technique to prevent priority inversion, it is not without its critics (e.g. see Yodaiken, 2002).

Of particular concern is that this protocol cannot avoid deadlock and blocking chains when tasks have nested shared resources. Therefore, to use PIP safely, an appropriate software architecture design is needed that avoids unnecessary coupling between tasks through shared resources (Locke, 2002), and it is important to avoid nested resources in applications.

Overall strengths and weaknesses

☺ Prevents priority inversion

☺ Has better average –case performance than Priority Ceiling protocol. When a critical section is not contended, priorities do not change, there is not context switches and no additional overhead.(Lcoke,2002; Renwick,2004)

☹ Difficult to implement when compared with DISABLE TIMER INTERRUPT [this paper].

- ☹ Does not prevent deadlock and blocking chains (Sha, 1990).
- ☹ Wastes processor time if there are not immediate tasks ready to run during the time that a higher-priority task is blocked by a lower-priority task
- ☹ Worst-case performance is worse than the worst-case performance for priority ceiling protocol since nested resource locks increase the wait time (Lcoke,2002; Renwick,2004).

IMPROVED PRIORITY CEILING PROTOCOL

{pattern}

Context

- You are developing an embedded system using a computer system with CPU and / or memory resources which are – compared with typical desktop designs – rather limited.
- Your system employs a single CPU.
- Your system employs a TT SCHEDULER [Pont et al., 2007: this conference].
- Your system supports task pre-emption.
- Your tasks may have nested shared resources.
- Predictable system behaviour is a key design requirement: in particular, predictable task timing is a concern.

Problem

How can we ensure that the shared resources are mutually exclusive and that priority inversion, deadlock and blocking chains are avoided?

Background

For background information about shared resources, please refer to the abstract pattern CRITICAL SECTION [this paper].

Solution

The pattern is intended to help you answer the question: "How can we ensure that the shared resources are mutually exclusive and that priority inversion, deadlock and blocking chains are avoided?"

Nested resource locks are the underlying cause of deadlock and blocking chains. Therefore, the simplest solution is to avoid nested resource locks at the design stage (indeed, some operating systems do not allow use of nested locks).

Sha et al. (1990) presented an alternative solution to the priority inheritance protocol: this was the priority ceiling protocol (PCP). However, this original priority ceiling protocol is expensive to implement. A simplified version of the original PCP is widely used (Locke, 2002). In this pattern we refer to this as the "Improved Priority Ceiling Protocol" (IPCP)[4].

In IPCP, each task has an assigned static priority; each resource has also been assigned a priority which is the highest priority of tasks that need access it (i.e. its priority ceiling: Burns, 2001). When a task acquires a shared resource, the task is raised to its ceiling priority. Therefore, the task will not be pre-empted by any other tasks attempting to access the same

4 IPCP is of often incorrectly referred to as the priority ceiling protocol. What we refer to as IPCP here is also known as the Priority Ceiling Emulation in Real-Time Java, Priority Protect Protocol in POSIX and as the Immediate Ceiling Priority Protocol (Burns, 2001).

resource with the same priority. When the task releases the resource, the task is returned to its original priority.

The deadlock case shown in Figure 8 is illustrated in Figure 12 to explain how IPCP works. $Task_H$ and $Task_L$ access both resources C1 and C2. Thereby the ceiling priorities of C1 (P_{c1}) and C2 (P_{c2}) are the priority of $Task_H$ (P_{TaskH}). $Task_L$ runs first. At t_1, it needs to access C2, according to IPCP, it will be raised to the ceiling priority of C1, which equals to $P_{TaskH.}$ At t2, $Task_H$ is ready to run. However, its priority is the same as the dynamic priority of $Task_L$. It will not able to pre-empt $Task_L$ until $Task_L$ completes the critical sections and returns to the original priority. Therefore, the deadlock is prevented.

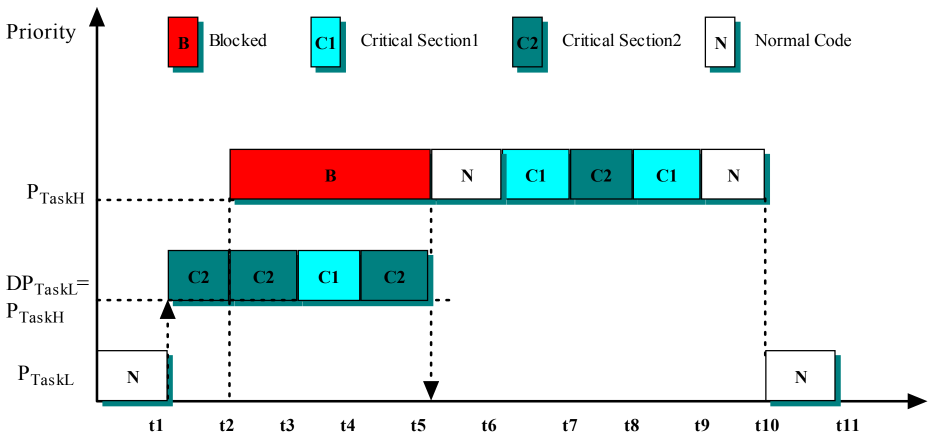


Figure 12 Operation Sequences of Tasks with IPCP

Related patterns and alternative solutions

Related patterns

The pattern CRITICAL SECTION provides general background material on mechanisms for dealing with shared resources in TT systems which involve task pre-emption.

The following patterns describe some alternative ways of handling conflicts over shared resources:

- DISABLE TIMER INTERRUPT
- RESOURCE LOCK
- PRIORITY INHERITANCE PROTOCOL

Alternative solutions

Sha et al (1990) originally presented the priority ceiling protocol (PCP).

In original priority ceiling protocol (OPCP), each task has an assigned static priority; each resource has also been assigned a priority which is the highest priority of tasks that need access it, i.e. its priority ceiling, which are the same as the IPCP. There are two differences. One is that each task's dynamic priority is the maximum of its own static priority and its inheritance priority due to it blocking higher-priority tasks. The second is that a task can only lock a resource if its dynamic priority is higher than the ceiling priority of any currently locked resource (Burns, 2001).

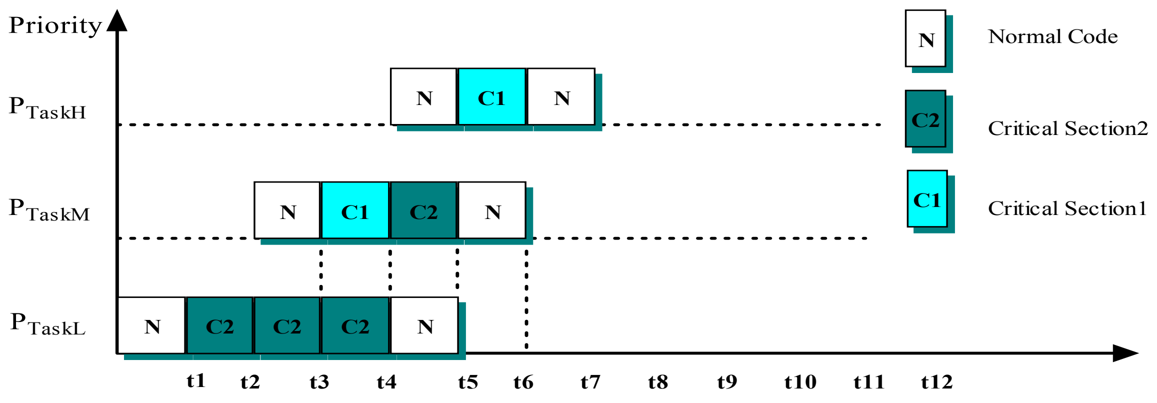


Figure 13 Operational sequences of three tasks with two shared resources

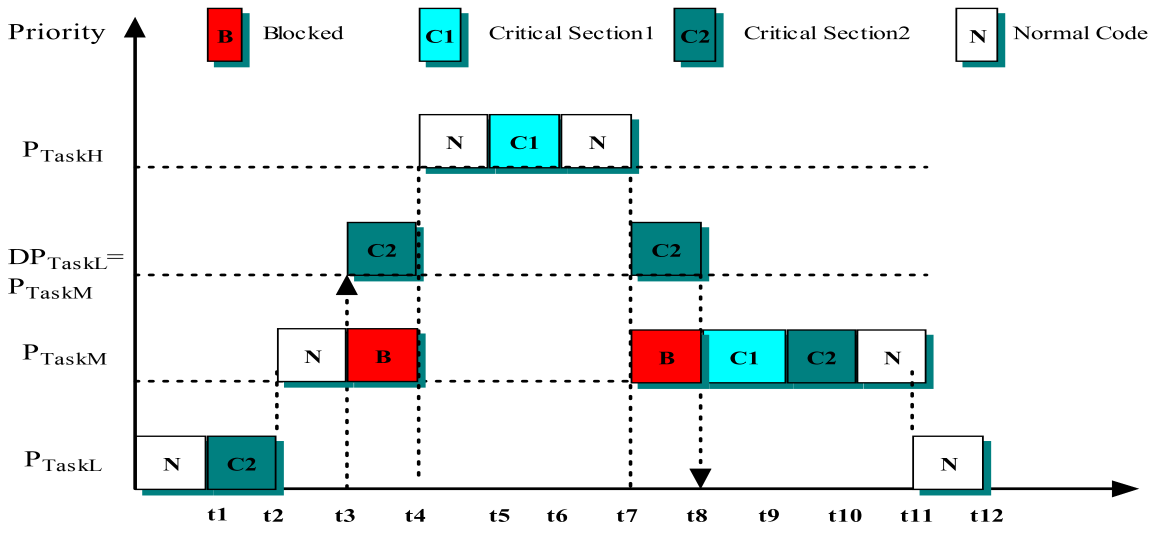


Figure 14 Operational sequences of three tasks with OPCP

Figure 14 explains how OPCP works. Three tasks intend to run shown in Figure 13. At t2, $Task_M$ pre-empts $Task_{L,}$ at t3 $Task_M$ is attempting to lock the resource C1. However, due that its dynamic priority P_{TaskM} is not higher than the ceiling priority of C2 ($P_{c2} = P_{TaskM}$) which is currently locked by $Task_L$, it cannot lock C1, and is blocked by $Task_L$. $Task_L$ inherits $Task_M$ priority due to it blocking $Task_M$ and continues running in a higher priority. At t4 $Task_H$ starts to run and at t5, it is attempting to lock C1. Because its priority is higher than Pc2, it successfully locked C1 and runs to completion. After TaskL releases C2 and returns to its original priority at t8, $Task_M$ locks C1 and runs to finish.

To compare with improved priority ceiling protocol, the solution using IPCP in this case is illustrated in Figure 15. From Figure 14 and Figure 15 it is seen that there are 6 times context switches using OPCP and 4 times context switches in IPCP. In addition, OPCP needs to check blocking information for tasks dynamic priority. Therefore, OPCP is more difficult to implement.

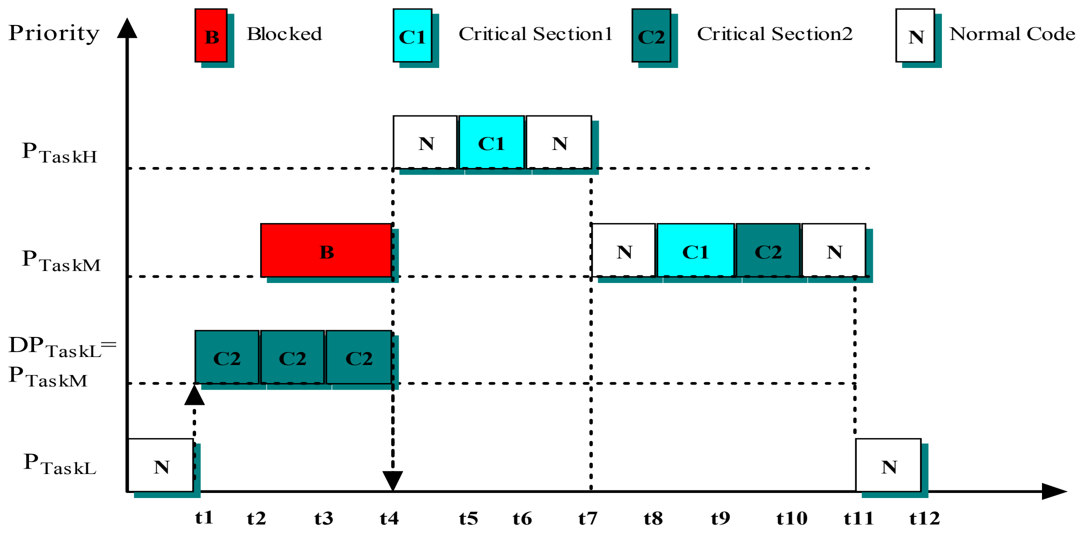


Figure 15 Operational sequences of three tasks with IPCP

Reliability and safety implications

IPCP is an attractive choice when there may be nested locks among tasks. Preventing deadlock and blocking chain increases the stability of a system.

Comparing with DISABLE TIMER INTERRUPT, IPCP is more difficult to implement (and test). An appropriate software architecture design is needed that avoids unnecessary coupling between tasks through shared resources (Locke, 2002), and avoids nested resources if possible.

If several tasks do use the same resource, designers should consider combining them into a single task (Renwick, 2004).

Overall strengths and weaknesses

☺ Prevents priority inversion

☺ Prevents deadlock and blocking chains

☺ Has better worst-case performance than PIP. The worst-case wait time for a high priority task waiting for a shared resource is limited to the longest critical section of any lower priority tasks that accesses the shared resource.

☹ Difficult to implement

☹ Requires static analysis of a system to find the priority ceiling of each critical section.

☹ Average –case performance is worse than PIP. IPCP changes a task's priority when it requires a resource, regardless of whether there is contention for the resource or not, resulting in higher overhead and many unnecessary context switches and blocking in unrelated tasks (Locke,2002)

References

Burns, A., Wellings, A. (2001) "Real-Time Systems and Programming Languages", Addison-Wesley / ACM Press. ISBN: 0-201-72988-1.

Jones, M. (1997) "What really happened on Mars?" *http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html*

Kalinsky, D., 2001. Context switch, Embedded Systems Programming, 14(1), 94-105.

Labrosse, J.J. (2002) "MicroC/OS-II: The Real-Time Kernel", CMP Books. ISBN:1-57820-103-9

Laplante, P.A. (2004) "Real-Time Systems Design and Analysis", Wiley-Interscience. ISBN: 0-471-22855-9.

Locke, D. (2002) "Priority Inheritance: The Real Story", *http://www.linuxdevices.com/articles/AT5698775833.html*

Maaita, A. and Pont, M.J. (2005) "Using 'planned pre-emption' to reduce levels of task jitter in a time-triggered hybrid scheduler". In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (Eds.), Proceedings of the Second UK Embedded Forum (Birmingham, UK, October 2005), pp.18-35. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0191-9].

Pont, M.J., Kurian, S., Wang, H. and Phatrapornnant, T. (2007) "Selecting an appropriate scheduler for use with time-triggered embedded systems" Paper to be presented at EuroPLoP 2007 (Paper C3).

Renwick, K., Renwick, B. (2004) "How to use priority inheritance", *Http://www.embedded.com/showArticle.jhtml?articleID=20600062*

Sha, L., Rajkumar, R., Lehoczky, J.P. (1990) "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", *IEEE Transactions on Computers*, **39**(9): 1175-1185.

Simon, D.E. (2001) "An Embedded Software Primer", Addison-Wesley / ACM Press. ISBN: 0-201-61569.

Xu , J. and Parnas, D.L. (1990) "Scheduling Processes with Release Times, Deadlines, Precedence and Exclusion Relations,'' IEEE Transactions on Software Engineering, 16(3), pp. 360-369.

Yodaiken, V. (2002) "Against priority inheritance", *http://www.linuxdevices.com/articles/AT7168794919.html*