A Formalisation of Priority Inheritance Protocol for Correct and Efficient Implementation

Xingyuan Zhang[1], Christian Urban[2], and Chunhan Wu[1]

[1] PLA University of Science and Technology, China
[2] King's College London, United Kingdom

Abstract. In real-time systems with support for resource locking and for processes with priorities, one faces the problem of priority inversion. This problem can make the behaviour of processes unpredictable and the resulting bugs can be hard to find. The Priority Inheritance Protocol is one solution implemented in many systems for solving this problem, but the correctness of this solution has never been formally verified in a theorem prover. As already pointed out in the literature, the original informal investigation of the Property Inheritance Protocol presents a correctness "proof" for an *incorrect* algorithm. In this paper we fix the problem of this proof by making all notions precise and implementing a variant of a solution proposed earlier. Our formalisation in Isabelle/HOL uncovered facts not mentioned in the literature, but also shows how to efficiently implement this protocol. Earlier correct implementations were criticised as too inefficient. Our formalisation is based on Paulson's inductive approach to verifying protocols.

Keywords: Priority Inheritance Protocol, formal connectness proof, real-time systems

1 Introduction

Many real-time systems need to support processes with priorities and locking of resources. Locking of resources ensures mutual exclusion when accessing shared data or devices that cannot be preempted. Priorities allow scheduling of processes that need to finish their work within deadlines. Unfortunately, both features can interact in subtle ways leading to a problem, called *Priority Inversion*. Suppose three processes having priorities H(igh), M(edium) and L(ow). We would expect that the process H blocks any other process with lower priority and itself cannot be blocked by any process with lower priority. Alas, in a naive implementation of resource looking and priorities this property can be violated. Even worse, H can be delayed indefinitely by processes with lower priorities. For this let L be in the possession of a lock for a resource that also H needs. H must therefore wait for L to exit the critical section and release this lock. The problem is that L might in turn be blocked by any process with priority M, and so H sits there potentially waiting indefinitely. Since H is blocked by processes with lower priorities, the problem is called Priority Inversion. It was first described in [5] in the context of the Mesa programming language designed for concurrent programming.

If the problem of Priority Inversion is ignored, real-time systems can become unpredictable and resulting bugs can be hard to diagnose. The classic example where this happened is the software that controlled the Mars Pathfinder mission in 1997 [8]. Once the spacecraft landed, the software shut down at irregular intervals leading to loss of project time as normal operation of the craft could only resume the next day (the mission and data already collected were fortunately not lost, because of a clever system design). The reason for the shutdowns was that the scheduling software fell victim of Priority Inversion: a low priority task locking a resource prevented a high priority process from running in time leading to a system reset. Once the problem was found, it was rectified by enabling the *Priority Inheritance Protocol* (PIP) [6] in the scheduling software.

The idea behind PIP is to let the process L temporarily inherit the high priority from H until L leaves the critical section unlocking the resource. This solves the problem of H having to wait indefinitely, because L cannot, for example, be blocked by processes having priority M. While a few other solutions exist for the Priority Inversion problem [5], PIP is one that is widely deployed and implemented. This includes VxWorks (a proprietary real-time OS used in the Mars Pathfinder mission, in Boeing's 787 Dreamliner, Honda's ASIMO robot, etc.), but also the POSIX 1003.1c Standard realised for example in libraries for FreeBSD, Solaris and Linux.

One advantage of PIP is that increasing the priority of a process can be dynamically calculated by the scheduler. This is in contrast to, for example, *Priority Ceiling*, another solution to the Priority Inversion problem, which requires static analysis of the program in order to be helpful. However, there has also been strong criticism against PIP. For instance, PIP cannot prevent deadlocks, and also blocking times can be substantial (more than just the duration of a critical section). Though, most criticism against PIP centres around unreliable implementations and around PIP being too complicated and too inefficient. For example, Yodaiken writes in [13]:

> *"Priority inheritance is neither efficient nor reliable. Implementations are either incomplete (and unreliable) or surprisingly complex and intrusive."*

He suggests to avoid PIP altogether by not allowing critical sections to be preempted. While this was a sensible solution in 2002, in our modern multiprocessor world, this seems out of date. However, there is clearly a need for investigating correct and efficient algorithms for PIP. A few specifications for PIP exist (in English) and also a few high-level descriptions of implementations (e.g. in the textbook [10, Section 5.6.5]), but they help little with actual implementations. That this is a problem in practise is proved by an email from Baker, who wrote on 13 July 2009 on the Linux Kernel mailing list:

> *"I observed in the kernel code (to my disgust), the Linux PIP implementation is a nightmare: extremely heavy weight, involving maintenance of a full wait-for graph, and requiring updates for a range of events, including priority changes and interruptions of wait operations."*

This however means it is useful to look at PIP again from a more abstract level (but still concrete enough to inform an implementation) and makes PIP an ideal candidate for a formal verification. One reason, of course, is that the original presentation of PIP, despite being informally "proved" correct, is flawed. Yodaiken [13] points to a subtlety that has been overlooked by Sha et al.

But this is too simplistic. Consider Priority Inversion problem has been known since 1980 [5], but Sha et al. give the first thorough analysis and present an informal correctness proof for PIP .[3]

POSIX.4: programming for the real world (Google eBook)

However, there are further subtleties: just lowering the priority of the process L to its low priority, as proposed in ???, is incorrect.

very little on implementations, not to mention implementations informed by formal correctness proofs.

Priority inversion referrers to the phenomena where tasks with higher priority are blocked by ones with lower priority. If priority inversion is not controlled, there will be no guarantee the urgent tasks will be processed in time. As reported in [8], priority inversion used to cause software system resets and data lose in JPL's Mars pathfinder project. Therefore, the avoiding, detecting and controlling of priority inversion is a key issue to attain predictability in priority based real-time systems.

The priority inversion phenomenon was first published in [5]. The two protocols widely used to eliminate priority inversion, namely PI (Priority Inheritance) and PCE (Priority Ceiling Emulation), were proposed in [6]. PCE is less convenient to use because it requires static analysis of programs. Therefore, PI is more commonly used in practice[7]. However, as pointed out in the literature, the analysis of priority inheritance protocol is quite subtle[**?**]. A formal analysis will certainly be helpful for us to understand and correctly implement PI. All existing formal analysis of PI [4,12,3] are based on the model checking technology. Because of the state explosion problem, model check is much like an exhaustive testing of finite models with limited size. The results obtained can not be safely generalized to models with arbitrarily large size. Worse still, since model checking is fully automatic, it give little insight on why the formal model is correct. It is therefore definitely desirable to analyze PI using theorem proving, which gives more general results as well as deeper insight. And this is the purpose of this paper which gives a formal analysis of PI in the interactive theorem prover Isabelle using Higher Order Logic (HOL). The formalization focuses on on two issues:

1. The correctness of the protocol model itself. A series of desirable properties is derived until we are fully convinced that the formal model of PI does eliminate priority inversion. And a better understanding of PI is so obtained in due course. For example, we find through formalization that the choice of next thread to take hold when a resource is released is irrelevant for the very basic property of PI to hold. A point never mentioned in literature.
2. The correctness of the implementation. A series of properties is derived the meaning of which can be used as guidelines on how PI can be implemented efficiently and correctly.

[3] Sha et al. call it the *Basic Priority Inheritance Protocol*.

The rest of the paper is organized as follows: Section 2 gives an overview of PI. Section 3 introduces the formal model of PI. Section 4 discusses a series of basic properties of PI. Section 5 shows formally how priority inversion is controlled by PI. Section 6 gives properties which can be used for guidelines of implementation. Section 7 discusses related works. Section 8 concludes the whole paper.

The basic priority inheritance protocol has two problems:

It does not prevent a deadlock from happening in a program with circular lock dependencies.

A chain of blocking may be formed; blocking duration can be substantial, though bounded.

Contributions

Despite the wide use of Priority Inheritance Protocol in real time operating system, it's correctness has never been formally proved and mechanically checked. All existing verification are based on model checking technology. Full automatic verification gives little help to understand why the protocol is correct. And results such obtained only apply to models of limited size. This paper presents a formal verification based on theorem proving. Machine checked formal proof does help to get deeper understanding. We found the fact which is not mentioned in the literature, that the choice of next thread to take over when an critical resource is release does not affect the correctness of the protocol. The paper also shows how formal proof can help to construct correct and efficient implementation.

2 An overview of priority inversion and priority inheritance

Priority inversion refers to the phenomenon when a thread with high priority is blocked by a thread with low priority. Priority happens when the high priority thread requests for some critical resource already taken by the low priority thread. Since the high priority thread has to wait for the low priority thread to complete, it is said to be blocked by the low priority thread. Priority inversion might prevent high priority thread from fulfill its task in time if the duration of priority inversion is indefinite and unpredictable. Indefinite priority inversion happens when indefinite number of threads with medium priorities is activated during the period when the high priority thread is blocked by the low priority thread. Although these medium priority threads can not preempt the high priority thread directly, they are able to preempt the low priority threads and cause it to stay in critical section for an indefinite long duration. In this way, the high priority thread may be blocked indefinitely.

Priority inheritance is one protocol proposed to avoid indefinite priority inversion. The basic idea is to let the high priority thread donate its priority to the low priority thread holding the critical resource, so that it will not be preempted by medium priority threads. The thread with highest priority will not be blocked unless it is requesting some critical resource already taken by other threads. Viewed from a different angle, any thread which is able to block the highest priority threads must already hold some critical resource. Further more, it must have hold some critical resource at the moment the highest priority is created, otherwise, it may never get change to run and get hold.

Since the number of such resource holding lower priority threads is finite, if every one of them finishes with its own critical section in a definite duration, the duration the highest priority thread is blocked is definite as well. The key to guarantee lower priority threads to finish in definite is to donate them the highest priority. In such cases, the lower priority threads is said to have inherited the highest priority. And this explains the name of the protocol: *Priority Inheritance* and how Priority Inheritance prevents indefinite delay.

The objectives of this paper are:

1. Build the above mentioned idea into formal model and prove a series of properties until we are convinced that the formal model does fulfill the original idea.
2. Show how formally derived properties can be used as guidelines for correct and efficient implementation.

The proof is totally formal in the sense that every detail is reduced to the very first principles of Higher Order Logic. The nature of interactive theorem proving is for the human user to persuade computer program to accept its arguments. A clear and simple understanding of the problem at hand is both a prerequisite and a byproduct of such an effort, because everything has finally be reduced to the very first principle to be checked mechanically. The former intuitive explanation of Priority Inheritance is just such a byproduct.

3 Formal model of Priority Inheritance

In this section, the formal model of Priority Inheritance is presented. The model is based on Paulson's inductive protocol verification method, where the state of the system is modelled as a list of events happened so far with the latest event put at the head.

To define events, the identifiers of *threads*, *priority* and *critical resources* (abbreviated as *cs*) need to be represented. All three are represetned using standard Isabelle/HOL type *nat*:

type-synonym *thread* = *nat* — Type for thread identifiers.
type-synonym *priority* = *nat* — Type for priorities.
type-synonym *cs* = *nat* — Type for critical sections (or critical resources).

Every event in the system corresponds to a system call, the formats of which are defined as follows:

datatype *event* =
Create thread priority | — Thread *thread* is created with priority *priority*.
Exit thread | — Thread *thread* finishing its execution.
P thread cs | — Thread *thread* requesting critical resource *cs*.
V thread cs | — Thread *thread* releasing critical resource *cs*.
Set thread priority — Thread *thread* resets its priority to *priority*.

Resource Allocation Graph (RAG for short) is used extensively in our formal analysis. The following type *node* is used to represent nodes in RAG.

datatype *node* =
  Th thread | — Node for thread.
  Cs cs — Node for critical resource.

In Paulson's inductive method, the states of system are represented as lists of events, which is defined by the following type *state*:

type-synonym *state* = *event list*

The following function *threads* is used to calculate the set of live threads (*threads s*) in state *s*.

fun *threads* :: *state* ⇒ *thread set*
  where
  — At the start of the system, the set of threads is empty:
  threads [] = {} |
  — New thread is added to the *threads*:
  threads (*Create thread prio#s*) = {*thread*} ∪ *threads s* |
  — Finished thread is removed:
  threads (*Exit thread # s*) = (*threads s*) − {*thread*} |
  — Other kind of events does not affect the value of *threads*:
  threads (*e#s*) = *threads s*

Functions such as *threads*, which extract information out of system states, are called *observing functions*. A series of observing functions will be defined in the sequel in order to model the protocol. Observing function *original_priority* calculates the *original priority* of thread *th* in state *s*, expressed as : *original_priority th s*. The *original priority* is the priority assigned to a thread when it is created or when it is reset by system call *Set thread priority*.

fun *original_priority* :: *thread* ⇒ *state* ⇒ *priority*
  where
  — *0* is assigned to threads which have never been created:
  original_priority thread [] = *0* |
  original_priority thread (*Create thread′ prio#s*) =
    (*if thread′ = thread then prio else original_priority thread s*) |
  original_priority thread (*Set thread′ prio#s*) =
    (*if thread′ = thread then prio else original_priority thread s*) |
  original_priority thread (*e#s*) = *original_priority thread s*

In the following, *birthtime th s* is the time when thread *th* is created, observed from state *s*. The time in the system is measured by the number of events happened so far since the very beginning.

fun *birthtime* :: *thread* ⇒ *state* ⇒ *nat*
  where
  birthtime thread [] = *0* |
  birthtime thread ((*Create thread′ prio*)#*s*) =
    (*if* (*thread = thread′*) *then length s else birthtime thread s*) |
  birthtime thread ((*Set thread′ prio*)#*s*) =

$(if\ (thread = thread')\ then\ length\ s\ else\ birthtime\ thread\ s)\ |$
$birthtime\ thread\ (e\#s) = birthtime\ thread\ s$

The *precedence* is a notion derived from *priority*, where the *precedence* of a thread is the combination of its *original priority* and *birth time*. The intention is to discriminate threads with the same priority by giving threads whose priority is assigned earlier higher precedences, becasue such threads are more urgent to finish. This explains the following definition:

definition $preced :: thread \Rightarrow state \Rightarrow precedence$
where $preced\ thread\ s = Prc\ (original\_priority\ thread\ s)\ (birthtime\ thread\ s)$

A number of important notions are defined here:

consts
$holding :: 'b \Rightarrow thread \Rightarrow cs \Rightarrow bool$
$waiting :: 'b \Rightarrow thread \Rightarrow cs \Rightarrow bool$
$depend :: 'b \Rightarrow (node \times node)\ set$
$dependents :: 'b \Rightarrow thread \Rightarrow thread\ set$

In the definition of the following several functions, it is supposed that the waiting queue of every critical resource is given by a waiting queue function *wq*, which servers as arguments of these functions.

defs (**overloaded**)

— We define that the thread which is at the head of waiting queue of resource *cs* is holding the resource. This definition is slightly different from tradition where all threads in the waiting queue are considered as waiting for the resource. This notion is reflected in the definition of *holding wq th cs* as follows:

cs_holding_def:
$holding\ wq\ thread\ cs \stackrel{def}{=} (thread \in set\ (wq\ cs) \land thread = hd\ (wq\ cs))$

— In accordance with the definition of *holding wq th cs*, a thread *th* is considered waiting for *cs* if it is in the *waiting queue* of critical resource *cs*, but not at the head. This is reflected in the definition of *waiting wq th cs* as follows:

cs_waiting_def:
$waiting\ wq\ thread\ cs \stackrel{def}{=} (thread \in set\ (wq\ cs) \land thread \neq hd\ (wq\ cs))$

— *depend wq* represents the Resource Allocation Graph of the system under the waiting queue function *wq*.

cs_depend_def:
$depend\ (wq{::}cs \Rightarrow thread\ list) \stackrel{def}{=}$
$\{(Th\ t, Cs\ c)\ |\ t\ c.\ waiting\ wq\ t\ c\} \cup \{(Cs\ c, Th\ t)\ |\ c\ t.\ holding\ wq\ t\ c\}$

— The following *dependents wq th* represents the set of threads which are depending on thread *th* in Resource Allocation Graph *depend wq*:

cs_dependents_def:
$dependents\ (wq{::}cs \Rightarrow thread\ list)\ th \stackrel{def}{=} \{th'\ .\ (Th\ th', Th\ th) \in (depend\ wq)\hat{\ }+\}$

The data structure used by the operating system for scheduling is referred to as *schedule state*. It is represented as a record consisting of a function assigning waiting queue to resources and a function assigning precedence to threads:

record *schedule_state* =
  waiting_queue :: *cs* ⇒ *thread list* — The function assigning waiting queue.
  cur_preced :: *thread* ⇒ *precedence* — The function assigning precedence.

The following *cpreced s th* gives the *current precedence* of thread *th* under state *s*. The definition of *cpreced* reflects the basic idea of Priority Inheritance that the *current precedence* of a thread is the precedence inherited from the maximum of all its dependents, i.e. the threads which are waiting directly or indirectly waiting for some resources from it. If no such thread exits, *th*'s *current precedence* equals its original precedence, i.e. *preced th s*.

definition *cpreced* :: *state* ⇒ (*cs* ⇒ *thread list*) ⇒ *thread* ⇒ *precedence*
 where *cpreced s wq* = (λ *th*. *Max* ((λ *th*. *preced th s*) ‘ ({*th*} ∪ *dependents wq th*)))

The following function *schs* is used to calculate the schedule state *schs s*. It is the key function to model Priority Inheritance:

fun *schs* :: *state* ⇒ *schedule_state*
 where *schs* [] = (|*waiting_queue* = λ *cs*. [], *cur_preced* = *cpreced* [] (λ *cs*. [])|) |

1. *ps* is the schedule state of last moment.
2. *pwq* is the waiting queue function of last moment.
3. *pcp* is the precedence function of last moment.
4. *nwq* is the new waiting queue function. It is calculated using a *case* statement:
   (a) If the happening event is *P thread cs*, *thread* is added to the end of *cs*'s waiting queue.
   (b) If the happening event is *V thread cs* and *s* is a legal state, *th′* must equal to *thread*, because *thread* is the one currently holding *cs*. The case [] ⟹ [] may never be executed in a legal state. the (*SOME q*. *distinct q* ∧ *set q* = *set qs*) is used to choose arbitrarily one thread in waiting to take over the released resource *cs*. In our representation, this amounts to rearrange elements in waiting queue, so that one of them is put at the head.
   (c) For other happening event, the schedule state just does not change.
5. *ncp* is new precedence function, it is calculated from the newly updated waiting queue function. The dependency of precedence function on waiting queue function is the reason to put them in the same record so that they can evolve together.

```
schs (e#s) = (let ps = schs s in
      let pwq = waiting_queue ps in
      let pcp = cur_preced ps in
      let nwq = case e of
            P thread cs ⇒  pwq(cs:=(pwq cs @ [thread])) |
            V thread cs ⇒ let nq = case (pwq cs) of
                            [] ⇒ [] |
                            (th′#qs) ⇒ (SOME q. distinct q ∧ set q = set qs)
                      in pwq(cs:=nq)              |
            _ ⇒ pwq
```

$\quad$ *in let ncp = cpreced (e#s) nwq in*
$\quad\quad$ $(\!|$*waiting_queue = nwq, cur_preced = ncp*$|\!)$
$\quad$)

The following *wq* is a shorthand for *waiting_queue*.

definition *wq* :: *state* ⇒ *cs* ⇒ *thread list*
where *wq s = waiting_queue (schs s)*

The following *cp* is a shorthand for *cur_preced*.

definition *cp* :: *state* ⇒ *thread* ⇒ *precedence*
where *cp s = cur_preced (schs s)*

Functions *holding*, *waiting*, *depend* and *dependents* still have the same meaning, but redefined so that they no longer depend on the fictitious *waiting queue function wq*, but on system state *s*.

defs (**overloaded**)
s_holding_def:
holding (s::state) thread cs $\stackrel{def}{=}$ (*thread* ∈ *set* (*wq s cs*) ∧ *thread* = *hd* (*wq s cs*))
s_waiting_def:
waiting (s::state) thread cs $\stackrel{def}{=}$ (*thread* ∈ *set* (*wq s cs*) ∧ *thread* ≠ *hd* (*wq s cs*))
s_depend_def:
depend (s::state) $\stackrel{def}{=}$
{(*Th t*, *Cs c*) | *t c*. *waiting* (*wq s*) *t c*} ∪ {(*Cs c*, *Th t*) | *c t*. *holding* (*wq s*) *t c*}
s_dependents_def:
dependents (s::state) th $\stackrel{def}{=}$ {*th'* . (*Th th'*, *Th th*) ∈ (*depend* (*wq s*))^+}

The following function *readys* calculates the set of ready threads. A thread is *ready* for running if it is a live thread and it is not waiting for any critical resource.

definition *readys* :: *state* ⇒ *thread set*
where *readys s* = {*thread* . *thread* ∈ *threads s* ∧ (∀ *cs*. ¬ *waiting s thread cs*)}

The following function *runing* calculates the set of running thread, which is the ready thread with the highest precedence.

definition *runing* :: *state* ⇒ *thread set*
where *runing s* = {*th* . *th* ∈ *readys s* ∧ *cp s th* = *Max* ((*cp s*) ‘ (*readys s*))}

The following function *holdents s th* returns the set of resources held by thread *th* in state *s*.

definition *holdents* :: *state* ⇒ *thread* ⇒ *cs set*
where *holdents s th* = {*cs* . (*Cs cs*, *Th th*) ∈ *depend s*}

cntCS s th returns the number of resources held by thread *th* in state *s*:

definition *cntCS* :: *state* ⇒ *thread* ⇒ *nat*
where *cntCS s th* = *card* (*holdents s th*)

The fact that event e is eligible to happen next in state s is expressed as *step s e*. The predicate *step* is inductively defined as follows:

inductive *step* :: *state* ⇒ *event* ⇒ *bool*
where
— A thread can be created if it is not a live thread:
thread_create: $[\![thread \notin threads\ s]\!] \Longrightarrow step\ s\ (Create\ thread\ prio)$ |
— A thread can exit if it no longer hold any resource:
thread_exit: $[\![thread \in runing\ s;\ holdents\ s\ thread = \{\}]\!] \Longrightarrow step\ s\ (Exit\ thread)$ |
— A thread can request for an critical resource *cs*, if it is running and the request does not form a loop in the current RAG. The latter condition is set up to avoid deadlock. The condition also reflects our assumption all threads are carefully programmed so that deadlock can not happen:
thread_P: $[\![thread \in runing\ s;\ (Cs\ cs, Th\ thread) \notin (depend\ s)\hat{}+]\!] \Longrightarrow$
$step\ s\ (P\ thread\ cs)$ |
— A thread can release a critical resource *cs* if it is running and holding that resource:
thread_V: $[\![thread \in runing\ s;\ holding\ s\ thread\ cs]\!] \Longrightarrow step\ s\ (V\ thread\ cs)$ |
— A thread can adjust its own priority as long as it is current running:
thread_set: $[\![thread \in runing\ s]\!] \Longrightarrow step\ s\ (Set\ thread\ prio)$

With predicate *step*, the fact that s is a legal state in Priority Inheritance protocol can be expressed as: *vt step s*, where the predicate *vt* can be defined as the following:

inductive *vt* :: (*state* ⇒ *event* ⇒ *bool*) ⇒ *state* ⇒ *bool*
for *cs* — *cs* is an argument representing any step predicate.
where
— Empty list [] is a legal state in any protocol:
vt_nil[*intro*]: *vt cs* [] |
— If s a legal state, and event e is eligible to happen in state s, then $e\#s$ is a legal state as well:
vt_cons[*intro*]: $[\![vt\ cs\ s;\ cs\ s\ e]\!] \Longrightarrow vt\ cs\ (e\#s)$

It is easy to see that the definition of *vt* is generic. It can be applied to any step predicate to get the set of legal states.

The following two functions *the_cs* and *the_th* are used to extract critical resource and thread respectively out of RAG nodes.

fun *the_cs* :: *node* ⇒ *cs*
where *the_cs* (*Cs cs*) = *cs*

fun *the_th* :: *node* ⇒ *thread*
where *the_th* (*Th th*) = *th*

The following predicate *next_th* describe the next thread to take over when a critical resource is released. In *next_th s th cs t*, *th* is the thread to release, *t* is the one to take over.

definition *next_th*:: *state* ⇒ *thread* ⇒ *cs* ⇒ *thread* ⇒ *bool*
where $next\_th\ s\ th\ cs\ t = (\exists\ rest.\ wq\ s\ cs = th\#rest \wedge rest \neq [] \wedge$

$$t = hd\ (SOME\ q.\ distinct\ q \wedge set\ q = set\ rest))$$

The function *count Q l* is used to count the occurrence of situation *Q* in list *l*:

definition $count :: ('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow nat$
where $count\ Q\ l = length\ (filter\ Q\ l)$

The following *cntP s* returns the number of operation *P* happened before reaching state *s*.

definition $cntP :: state \Rightarrow thread \Rightarrow nat$
where $cntP\ s\ th = count\ (\lambda\ e.\ \exists\ cs.\ e = P\ th\ cs)\ s$

The following *cntV s* returns the number of operation *V* happened before reaching state *s*.

definition $cntV :: state \Rightarrow thread \Rightarrow nat$
where $cntV\ s\ th = count\ (\lambda\ e.\ \exists\ cs.\ e = V\ th\ cs)\ s$

4 General properties of Priority Inheritance

The following are several very basic prioprites:

1. All runing threads must be ready (*runing_ready*):

   $runing\ s \subseteq readys\ s$

2. All ready threads must be living (*readys_threads*):

   $readys\ s \subseteq threads\ s$

3. There are finite many living threads at any moment (*finite_threads*):

   $vt\ step\ s \Longrightarrow finite\ (threads\ s)$

4. Every waiting queue does not contain duplcated elements (*wq_distinct*):

   $vt\ step\ s \Longrightarrow distinct\ (wq\ s\ cs)$

5. All threads in waiting queues are living threads (*wq_threads*):

   $[\![vt\ step\ s;\ th \in set\ (wq\ s\ cs)]\!] \Longrightarrow th \in threads\ s$

6. The event which can get a thread into waiting queue must be *P*-events (*block_pre*):

   $[\![vt\ step\ (e \cdot s);\ thread \notin set\ (wq\ s\ cs);\ thread \in set\ (wq\ (e \cdot s)\ cs)]\!]$
   $\Longrightarrow e = P\ thread\ cs$

7. A thread may never wait for two different critical resources (*waiting_unique*):

   $[\![vt\ step\ s;\ waiting\ s\ th\ cs_1;\ waiting\ s\ th\ cs_2]\!] \Longrightarrow cs_1 = cs_2$

8. Every resource can only be held by one thread (*held_unique*):

$[\![vt\ step\ s;\ holding\ s\ th_1\ cs;\ holding\ s\ th_2\ cs]\!] \Longrightarrow th_1 = th_2$

9. Every living thread has an unique precedence (*preced_unique*):

   $[\![preced\ th_1\ s = preced\ th_2\ s;\ th_1 \in threads\ s;\ th_2 \in threads\ s]\!] \Longrightarrow th_1 = th_2$

The following lemmas show how RAG is changed with the execution of events:

1. Execution of *Set* does not change RAG (*depend_set_unchanged*):

   $depend\ (Set\ th\ prio \cdot s) = depend\ s$

2. Execution of *Create* does not change RAG (*depend_create_unchanged*):

   $depend\ (Create\ th\ prio \cdot s) = depend\ s$

3. Execution of *Exit* does not change RAG (*depend_exit_unchanged*):

   $depend\ (Exit\ th \cdot s) = depend\ s$

4. Execution of *P* (*step_depend_p*):

   $vt\ step\ (P\ th\ cs \cdot s) \Longrightarrow$
   $depend\ (P\ th\ cs \cdot s) =$
   $(\textsf{if}\ wq\ s\ cs = []\ \textsf{then}\ depend\ s \cup \{(Cs\ cs,\ Th\ th)\}$
   $\ \textsf{else}\ depend\ s \cup \{(Th\ th,\ Cs\ cs)\})$

5. Execution of *V* (*step_depend_v*):

   $vt\ step\ (V\ th\ cs \cdot s) \Longrightarrow$
   $depend\ (V\ th\ cs \cdot s) =$
   $depend\ s - \{(Cs\ cs,\ Th\ th)\} - \{(Th\ th',\ Cs\ cs) \mid next\_th\ s\ th\ cs\ th'\} \cup$
   $\{(Cs\ cs,\ Th\ th') \mid next\_th\ s\ th\ cs\ th'\}$

These properties are used to derive the following important results about RAG:

1. RAG is loop free (*acyclic_depend*):

   $vt\ step\ s \Longrightarrow acyclic\ (depend\ s)$

2. RAGs are finite (*finite_depend*):

   $vt\ step\ s \Longrightarrow finite\ (depend\ s)$

3. Reverse paths in RAG are well founded (*wf_dep_converse*):

   $vt\ step\ s \Longrightarrow wf\ ((depend\ s)^{-1})$

4. The dependence relation represented by RAG has a tree structure (*unique_depend*):

   $[\![vt\ step\ s;\ (n,\ n_1) \in depend\ s;\ (n,\ n_2) \in depend\ s]\!] \Longrightarrow n_1 = n_2$

5. All threads in RAG are living threads (*dm_depend_threads* and *range_in*):

$[\![vt\ step\ s;\ Th\ th \in Domain\ (depend\ s)]\!] \Longrightarrow th \in threads\ s$
$[\![vt\ step\ s;\ Th\ th \in Range\ (depend\ s)]\!] \Longrightarrow th \in threads\ s$

The following lemmas show how every node in RAG can be chased to ready threads:

1. Every node in RAG can be chased to a ready thread (*chain_building*):

   $[\![vt\ step\ s;\ node \in Domain\ (depend\ s)]\!]$
   $\Longrightarrow \exists\, th'.\ th' \in readys\ s \wedge (node, Th\ th') \in (depend\ s)^+$

2. The ready thread chased to is unique (*dchain_unique*):

   $[\![vt\ step\ s;\ (n, Th\ th_1) \in (depend\ s)^+;\ th_1 \in readys\ s;$
   $(n, Th\ th_2) \in (depend\ s)^+;\ th_2 \in readys\ s]\!]$
   $\Longrightarrow th_1 = th_2$

Properties about *next_th*:

1. The thread taking over is different from the thread which is releasing (*next_th_neq*):

   $[\![vt\ step\ s;\ next\_th\ s\ th\ cs\ th']\!] \Longrightarrow th' \neq th$

2. The thread taking over is unique (*next_th_unique*):

   $[\![next\_th\ s\ th\ cs\ th_1;\ next\_th\ s\ th\ cs\ th_2]\!] \Longrightarrow th_1 = th_2$

Some deeper results about the system:

1. There can only be one running thread (*runing_unique*):

   $[\![vt\ step\ s;\ th_1 \in runing\ s;\ th_2 \in runing\ s]\!] \Longrightarrow th_1 = th_2$

2. The maximum of *cp* and *preced* are equal (*max_cp_eq*):

   $vt\ step\ s \Longrightarrow Max\ (cp\ s\ `\ threads\ s) = Max\ ((\lambda th.\ preced\ th\ s)\ `\ threads\ s)$

3. There must be one ready thread having the max *cp*-value (*max_cp_readys_threads*):

   $vt\ step\ s \Longrightarrow Max\ (cp\ s\ `\ readys\ s) = Max\ (cp\ s\ `\ threads\ s)$

The relationship between the count of *P* and *V* and the number of critical resources held by a thread is given as follows:

1. The *V*-operation decreases the number of critical resources one thread holds (*cntCS_v_dec*)

   $vt\ step\ (V\ thread\ cs{\cdot}s) \Longrightarrow cntCS\ (V\ thread\ cs{\cdot}s)\ thread + 1 = cntCS\ s\ thread$

2. The number of *V* never exceeds the number of *P* (*cnp_cnv_cncs*):

   $vt\ step\ s \Longrightarrow$
   $cntP\ s\ th =$
   $cntV\ s\ th +$
   $(\textsf{if}\ th \in readys\ s \vee th \notin threads\ s\ \textsf{then}\ cntCS\ s\ th\ \textsf{else}\ cntCS\ s\ th + 1)$

3. The number of V equals the number of P when the relevant thread is not living: (*cnp_cnv_eq*):

   $[\![vt\ step\ s;\ th \notin threads\ s]\!] \Longrightarrow cntP\ s\ th = cntV\ s\ th$

4. When a thread is not living, it does not hold any critical resource (*not_thread_holdents*):

   $[\![vt\ step\ s;\ th \notin threads\ s]\!] \Longrightarrow holdents\ s\ th = \varnothing$

5. When the number of P equals the number of V, the relevant thread does not hold any critical resource, therefore no thread can depend on it (*count_eq_dependents*):

   $[\![vt\ step\ s;\ cntP\ s\ th = cntV\ s\ th]\!] \Longrightarrow dependents\ (wq\ s)\ th = \varnothing$

5 Key properties

The essential of *Priority Inheritance* is to avoid indefinite priority inversion. For this purpose, we need to investigate what happens after one thread takes the highest precedence. A locale is used to describe such a situation, which assumes:

1. s is a valid state (*vt_s*): $vt\ step\ s$.
2. th is a living thread in s (*threads_s*): $th \in threads\ s$.
3. th has the highest precedence in s (*highest*): $preced\ th\ s = Max\ (cp\ s\ `\ threads\ s)$.
4. The precedence of th is $Prc\ prio\ tm$ (*preced_th*): $preced\ th\ s = Prc\ prio\ tm$.

Under these assumptions, some basic priority can be derived for th:

1. The current precedence of th equals its own precedence (*eq_cp_s_th*):

   $cp\ s\ th = preced\ th\ s$

2. The current precedence of th is the highest precedence in the system (*highest_cp_preced*):

   $cp\ s\ th = Max\ ((\lambda th'.\ preced\ th'\ s)\ `\ threads\ s)$

3. The precedence of th is the highest precedence in the system (*highest_preced_thread*):

   $preced\ th\ s = Max\ ((\lambda th'.\ preced\ th'\ s)\ `\ threads\ s)$

4. The current precedence of th is the highest current precedence in the system (*highest'*):

   $cp\ s\ th = Max\ (cp\ s\ `\ threads\ s)$

To analysis what happens after state s a sub-locale is defined, which assumes:

1. t is a valid extension of s (*vt_t*): $vt\ step\ (t\ @\ s)$.
2. Any thread created in t has priority no higher than *prio*, therefore its precedence can not be higher than th, therefore th remain to be the one with the highest precedence (*create_low*):

$Create\ th'\ prio' \in set\ t \Longrightarrow prio' \leq prio$

3. Any adjustment of priority in *t* does not happen to *th* and the priority set is no higher than *prio*, therefore *th* remain to be the one with the highest precedence (*set_diff_low*):

   $Set\ th'\ prio' \in set\ t \Longrightarrow th' \neq th \wedge prio' \leq prio$

4. Since we are investigating what happens to *th*, it is assumed *th* does not exit during *t* (*exit_diff*):

   $Exit\ th' \in set\ t \Longrightarrow th' \neq th$

All these assumptions are put into a predicate *extend_highest_gen*. It can be proved that *extend_highest_gen* holds for any moment *i* in it *t* (*red_moment*):

$extend\_highest\_gen\ s\ th\ prio\ tm\ (moment\ i\ t)$

From this, an induction principle can be derived for *t*, so that properties already derived for *t* can be applied to any prefix of *t* in the proof of new properties about *t* (*ind*):

$$\begin{array}{l} [\![R\ [\,];\\ \bigwedge e\ t.\ [\![vt\ step\ (t\ @\ s);\ step\ (t\ @\ s)\ e;\ extend\_highest\_gen\ s\ th\ prio\ tm\ t;\\ \quad extend\_highest\_gen\ s\ th\ prio\ tm\ (e{\cdot}t);\ R\ t]\!]\\ \quad \Longrightarrow R\ (e{\cdot}t)]\!]\\ \Longrightarrow R\ t \end{array}$$

The following properties can be proved about *th* in *t*:

1. In *t*, thread *th* is kept live and its precedence is preserved as well (*th_kept*):

   $th \in threads\ (t\ @\ s) \wedge preced\ th\ (t\ @\ s) = preced\ th\ s$

2. In *t*, thread *th*'s precedence is always the maximum among all living threads (*max_preced*):

   $preced\ th\ (t\ @\ s) = Max\ ((\lambda th'.\ preced\ th'\ (t\ @\ s))\ `\ threads\ (t\ @\ s))$

3. In *t*, thread *th*'s current precedence is always the maximum precedence among all living threads (*th_cp_max_preced*):

   $cp\ (t\ @\ s)\ th = Max\ ((\lambda th'.\ preced\ th'\ (t\ @\ s))\ `\ threads\ (t\ @\ s))$

4. In *t*, thread *th*'s current precedence is always the maximum current precedence among all living threads (*th_cp_max*):

   $cp\ (t\ @\ s)\ th = Max\ (cp\ (t\ @\ s)\ `\ threads\ (t\ @\ s))$

5. In *t*, thread *th*'s current precedence equals its precedence at moment *s* (*th_cp_preced*):

   $cp\ (t\ @\ s)\ th = preced\ th\ s$

The main theorem of this part is to characterizing the running thread during *t* (*runing_inversion_2*):

$th' \in runing\ (t\ @\ s) \Longrightarrow$
$th' = th \vee th' \neq th \wedge th' \in threads\ s \wedge cntV\ s\ th' < cntP\ s\ th'$

According to this, if a thread is running, it is either *th* or was already live and held some resource at moment *s* (expressed by: $cntV\ s\ th' < cntP\ s\ th'$).

Since there are only finite many threads live and holding some resource at any moment, if every such thread can release all its resources in finite duration, then after finite duration, none of them may block *th* anymore. So, no priority inversion may happen then.

6 Properties to guide implementation

The properties (especially *runing_inversion_2*) convinced us that the model defined in Section 3 does prevent indefinite priority inversion and therefore fulfills the fundamental requirement of Priority Inheritance protocol. Another purpose of this paper is to show how this model can be used to guide a concrete implementation. As discussed in Section 5.6.5 of [9], the implementation of Priority Inheritance in Solaris uses sophisticated linking data structure. Except discussing two scenarios to show how the data structure should be manipulated, a lot of details of the implementation are missing. In [3,4,12] the protocol is described formally using different notations, but little information is given on how this protocol can be implemented efficiently, especially there is no information on how these data structure should be manipulated.

Because the scheduling of threads is based on current precedence, the central issue in implementation of Priority Inheritance is how to compute the precedence correctly and efficiently. As long as the precedence is correct, it is very easy to modify the scheduling algorithm to select the correct thread to execute.

First, it can be proved that the computation of current precedence *cp* of a threads only involves its children (*cp_rec*):

$$vt\ step\ s \Longrightarrow cp\ s\ th = Max\ (\{preced\ th\ s\} \cup cp\ s\ `\ children\ s\ th)$$

where *children s th* represents the set of children of *th* in the current RAG:

$$children\ s\ th \stackrel{def}{=} \{th' \mid (Th\ th', Th\ th) \in child\ s\}$$

where the definition of *child* is:

$$child\ s \stackrel{def}{=} \{(Th\ th', Th\ th) \mid \exists cs.\ (Th\ th', Cs\ cs) \in depend\ s \wedge (Cs\ cs, Th\ th) \in depend\ s\}$$

The aim of this section is to fill the missing details of how current precedence should be changed with the happening of events, with each event type treated by one subsection, where the computation of *cp* uses lemma *cp_rec*.

6.1 Event *Set th prio*

The context under which event *Set th prio* happens is formalized as follows:

1. The formation of s (s_def): $s \stackrel{def}{=} Set\ th\ prio \cdot s'$.
2. State s is a valid state (vt_s): $vt\ step\ s$. This implies event $Set\ th\ prio$ is eligible to happen under state s' and state s' is a valid state.

Under such a context, we investigated how the current precedence cp of threads change from state s' to s and obtained the following conclusions:

1. All threads with no dependence relation with thread th have their cp-value unchanged (eq_cp):

   $[\![th' \neq th; th \notin dependents\ s\ th']\!] \Longrightarrow cp\ s\ th' = cp\ s'\ th'$

   This lemma implies the cp-value of th and those threads which have a dependence relation with th might need to be recomputed. The way to do this is to start from th and follow the *depend*-chain to recompute the cp-value of every encountered thread using lemma cp_rec. Since the *depend*-relation is loop free, this procedure can always stop. The the following lemma shows this procedure actually could stop earlier.
2. The following two lemma shows, if a thread the re-computation of which gives an unchanged cp-value, the procedure described above can stop.
   (a) Lemma eq_up_self shows if the re-computation of th's cp gives the same result, the procedure can stop:
   $[\![th \in dependents\ s\ th''; cp\ s\ th = cp\ s'\ th]\!] \Longrightarrow cp\ s\ th'' = cp\ s'\ th''$
   (b) Lemma eq_up) shows if the re-computation at intermediate threads gives unchanged result, the procedure can stop:
   $[\![th \in dependents\ s\ th'; th' \in dependents\ s\ th''; cp\ s\ th' = cp\ s'\ th']\!]$
   $\Longrightarrow cp\ s\ th'' = cp\ s'\ th''$

6.2 Event $V\ th\ cs$

The context under which event $V\ th\ cs$ happens is formalized as follows:

1. The formation of s (s_def): $s \stackrel{def}{=} V\ th\ cs \cdot s'$.
2. State s is a valid state (vt_s): $vt\ step\ s$. This implies event $V\ th\ cs$ is eligible to happen under state s' and state s' is a valid state.

Under such a context, we investigated how the current precedence cp of threads change from state s' to s.

Two subcases are considerted, where the first is that there exits th' such that

$next\_th\ s'\ th\ cs\ th'$

holds, which means there exists a thread th' to take over the resource release by thread th. In this sub-case, the following results are obtained:

1. The change of RAG is given by lemma $depend\_s$:

$depend\ s =$
$depend\ s' - \{(Cs\ cs, Th\ th)\} - \{(Th\ th', Cs\ cs)\} \cup \{(Cs\ cs, Th\ th')\}$

which shows two edges are removed while one is added. These changes imply how the current precedences should be re-computed.

2. First all threads different from *th* and th' have their *cp*-value kept, therefore do not need a re-computation (*cp_kept*):

   $[\![th'' \neq th; th'' \neq th']\!] \Longrightarrow cp\ s\ th'' = cp\ s'\ th''$

   This lemma also implies, only the *cp*-values of *th* and th' need to be recomputed.

The other sub-case is when for all th'

$\neg\ next\_th\ s'\ th\ cs\ th'$

holds, no such thread exists. The following results can be obtained for this sub-case:

1. The change of RAG is given by lemma *depend_s*:

   $depend\ s = depend\ s' - \{(Cs\ cs, Th\ th)\}$

   which means only one edge is removed.
2. In this case, no re-computation is needed (*eq_cp*):

   $cp\ s\ th' = cp\ s'\ th'$

6.3 Event *P th cs*

The context under which event *P th cs* happens is formalized as follows:

1. The formation of *s* (*s_def*): $s \stackrel{def}{=} P\ th\ cs \cdot s'$.
2. State *s* is a valid state (*vt_s*): *vt step s*. This implies event *P th cs* is eligible to happen under state s' and state s' is a valid state.

This case is further divided into two sub-cases. The first is when $wq\ s'\ cs = []$ holds. The following results can be obtained:

1. One edge is added to the RAG (*depend_s*):

   $depend\ s = depend\ s' \cup \{(Cs\ cs, Th\ th)\}$

2. No re-computation is needed (*eq_cp*):

   $cp\ s\ th' = cp\ s'\ th'$

The second is when $wq\ s'\ cs \neq []$ holds. The following results can be obtained:

1. One edge is added to the RAG (*depend_s*):

   $depend\ s = depend\ s' \cup \{(Th\ th, Cs\ cs)\}$

2. Threads with no dependence relation with *th* do not need a re-computation of their *cp*-values (*eq_cp*):

   $th \notin dependents\ s\ th' \Longrightarrow cp\ s\ th' = cp\ s'\ th'$

   This lemma implies all threads with a dependence relation with *th* may need re-computation.
3. Similar to the case of *Set*, the computation procedure could stop earlier (*eq_up*):

   $[\![th \in dependents\ s\ th';\ th' \in dependents\ s\ th'';\ cp\ s\ th' = cp\ s'\ th']\!]$
   $\Longrightarrow cp\ s\ th'' = cp\ s'\ th''$

6.4 Event *Create th prio*

The context under which event *Create th prio* happens is formalized as follows:

1. The formation of *s* (*s_def*): $s \stackrel{def}{=} Create\ th\ prio{\cdot}s'$.
2. State *s* is a valid state (*vt_s*): *vt step s*. This implies event *Create th prio* is eligible to happen under state s' and state s' is a valid state.

The following results can be obtained under this context:

1. The RAG does not change (*eq_dep*):

   $depend\ s = depend\ s'$

2. All threads other than *th* do not need re-computation (*eq_cp*):

   $th' \neq th \Longrightarrow cp\ s\ th' = cp\ s'\ th'$

3. The *cp*-value of *th* equals its precedence (*eq_cp_th*):

   $cp\ s\ th = preced\ th\ s$

6.5 Event *Exit th*

The context under which event *Exit th* happens is formalized as follows:

1. The formation of *s* (*s_def*): $s \stackrel{def}{=} Exit\ th{\cdot}s'$.
2. State *s* is a valid state (*vt_s*): *vt step s*. This implies event *Exit th* is eligible to happen under state s' and state s' is a valid state.

The following results can be obtained under this context:

1. The RAG does not change (*eq_dep*):

   $depend\ s = depend\ s'$

2. All threads other than *th* do not need re-computation (*eq_cp*):

   $th' \neq th \Longrightarrow cp\ s\ th' = cp\ s'\ th'$

Since *th* does not live in state *s*, there is no need to compute its *cp*-value.

7 Related works

1. *Integrating Priority Inheritance Algorithms in the Real-Time Specification for Java* [12] models and verifies the combination of Priority Inheritance (PI) and Priority Ceiling Emulation (PCE) protocols in the setting of Java virtual machine using extended Timed Automata(TA) formalism of the UPPAAL tool. Although a detailed formal model of combined PI and PCE is given, the number of properties is quite small and the focus is put on the harmonious working of PI and PCE. Most key features of PI (as well as PCE) are not shown. Because of the limitation of the model checking technique used there, properties are shown only for a small number of scenarios. Therefore, the verification does not show the correctness of the formal model itself in a convincing way.
2. *Formal Development of Solutions for Real-Time Operating Systems with TLA+/TLC* [3]. A formal model of PI is given in TLA+. Only 3 properties are shown for PI using model checking. The limitation of model checking is intrinsic to the work.
3. *Synchronous modeling and validation of priority inheritance schedulers* [4]. Gives a formal model of PI and PCE in AADL (Architecture Analysis & Design Language) and checked several properties using model checking. The number of properties shown there is less than here and the scale is also limited by the model checking technique.
4. *The Priority Ceiling Protocol: Formalization and Analysis Using PVS* [2]. Formalized another protocol for Priority Inversion in the interactive theorem proving system PVS.

There are several works on inversion avoidance:

1. *Solving the group priority inversion problem in a timed asynchronous system* [11]. The notion of Group Priority Inversion is introduced. The main strategy is still inversion avoidance. The method is by reordering requests in the setting of Client-Server.
2. *A Formalization of Priority Inversion* [1]. Formalized the notion of Priority Inversion and proposes methods to avoid it.

Examples of inaccurate specification of the protocol ???.

8 Conclusions

References

1. Ö Babaoglu, K. Marzullo, and F. B. Schneider. A formalization of priority inversion. *Real-Time Systems*, 5(4):285–303, 1993.
2. B. Dutertre. The Priority Ceiling Protocol: Formalization and analysis using PVS. Technical report, System Design Laboratory, SRI International, Menlo Park, CA, October 1999. Available at http://www.sdl.sri.com/dsa/publis/prio-ceiling.html.
3. J. M. S. Faria. Formal development of solutions for real-time operating systems with tla+/tlc. http://repositorio-aberto.up.pt/bitstream/10216/11466/2/Texto%20integral.pdf, 2008.

4. E. Jahier, B. Halbwachs, and P. Raymond. Synchronous modeling and validation of priority inheritance schedulers. In Marsha Chechik and Martin Wirsing, editors, *FASE*, volume 5503 of *Lecture Notes in Computer Science*, pages 140–154. Springer, 2009.
5. B. Lampson and D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.
6. S. Liu, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers*, 39(9):1175–1185, 1990.
7. D. Locke. Priority inheritance: The real story. http://www.math.unipd.it/~tullio/SCD/2007/Materiale/Locke.pdf, 2002.
8. G. Reeves. Re: What really happened on mars? *Risks-Forum Digest*, 19(58), January 1998.
9. U. Vahalia. *UNIX Internals, The New Frontiers*. Prentice-Hall, 1996.
10. Uresh Vahalia. *UNIX Internals: the New Frontier*. Prentice-Hall, 1996.
11. Y. Wang, E. Anceaume, F. Brasileiro, F. Greve, and M. Hurfin. Solving the group priority inversion problem in a timed asynchronous system. *IEEE Transactions on Computers*, 51(8):900–915, August 2002.
12. A. J. Wellings, A. Burns, O. M. Santos, and B. M. Brosgol. Integrating priority inheritance algorithms in the real-time specification for java. In *Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 115–123. IEEE Computer Society, 2007.
13. V. Yodaiken. Against Priority Inheritance. http://www.linuxfordevices.com/files/misc/yodaiken-july02.pdf, 2002.