Priority Inheritance Protocol Proved Correct

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 threads, resource locking and priority scheduling, one faces the problem of Priority Inversion. This problem can make the behaviour of threads 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 uncovers 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, Isabelle/HOL

1 Introduction

Many real-time systems need to support threads involving 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 threads 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 threads having priorities H(igh), M(edium) and L(ow). We would expect that the thread H blocks any other thread with lower priority and itself cannot be blocked by any thread 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 threads 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 thread with priority M, and so H sits there potentially waiting indefinitely. Since H is blocked by threads with lower priorities, the problem is called Priority Inversion. It was first described in [4] 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 [6]. 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 thread locking a resource prevented a high priority thread from running in time leading to a system reset. Once the problem was found, it was rectified by enabling the *Priority Inheritance Protocol* (PIP) [7][3] in the scheduling software.

The idea behind PIP is to let the thread 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 be blocked by threads having priority M. While a few other solutions exist for the Priority Inversion problem, 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 thread can be dynamically calculated by the scheduler. This is in contrast to, for example, *Priority Ceiling* [7], another solution to the Priority Inversion problem, which requires static analysis of the program in order to prevent Priority Inversion. However, there has also been strong criticism against PIP. For instance, PIP cannot prevent deadlocks when lock dependencies are circular, 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 PIP being too complicated and too inefficient. For example, Yodaiken writes in [12]:

> *"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. Unfortunately, this solution does not help in real-time systems with hard deadlines for high-priority threads.

In our opinion, there is clearly a need for investigating correct 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 [9, 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."*

[3] Sha et al. call it the *Basic Priority Inheritance Protocol* [7] and others sometimes also call it *Priority Boosting*.

The criticism by Yodaiken, Baker and others suggests to us to look again at PIP 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 [7], despite being informally "proved" correct, is actually *flawed*.

Yodaiken [12] points to a subtlety that had been overlooked in the informal proof by Sha et al. They specify in [7] that after the thread (whose priority has been raised) completes its critical section and releases the lock, it "returns to its original priority level." This leads them to believe that an implementation of PIP is "rather straightforward" [7]. Unfortunately, as Yodaiken points out, this behaviour is too simplistic. Consider the case where the low priority thread L locks *two* resources, and two high-priority threads H and H' each wait for one of them. If L releases one resource so that H, say, can proceed, then we still have Priority Inversion with H' (which waits for the other resource). The correct behaviour for L is to revert to the highest remaining priority of the threads that it blocks. The advantage of formalising the correctness of a high-level specification of PIP in a theorem prover is that such issues clearly show up and cannot be overlooked as in informal reasoning (since we have to analyse all possible behaviours of threads, i.e. *traces*, that could possibly happen).

Contributions: There have been earlier formal investigations into PIP [2,3,11], but they employ model checking techniques. This paper presents a formalised and mechanically checked proof for the correctness of PIP (to our knowledge the first one; the earlier informal proof by Sha et al. [7] is flawed). In contrast to model checking, our formalisation provides insight into why PIP is correct and allows us to prove stronger properties that, as we will show, can inform an efficient implementation. For example, we found by "playing" with the formalisation that the choice of the next thread to take over a lock when a resource is released is irrelevant for PIP being correct. Something which has not been mentioned in the relevant literature.

2 Formal Model of the Priority Inheritance Protocol

The Priority Inheritance Protocol, short PIP, is a scheduling algorithm for a single-processor system.[4] Our model of PIP is based on Paulson's inductive approach to protocol verification [5], where the *state* of a system is given by a list of events that happened so far. *Events* of PIP fall into five categories defined as the datatype:

datatype *event*	=	*Create thread priority*	
	|	*Exit thread*	
	|	*Set thread priority*	reset of the priority for *thread*
	|	*P thread cs*	request of resource *cs* by *thread*
	|	*V thread cs*	release of resource *cs* by *thread*

whereby threads, priorities and (critical) resources are represented as natural numbers. The event *Set* models the situation that a thread obtains a new priority given by the

[4] We shall come back later to the case of PIP on multi-processor systems.

programmer or user (for example via the `nice` utility under UNIX). As in Paulson's work, we need to define functions that allow us to make some observations about states. One, called *threads*, calculates the set of "live" threads that we have seen so far:

$$\begin{array}{ll}
\textit{threads}\ [] & \overset{\textit{def}}{=}\ \varnothing \\
\textit{threads}\ (\textit{Create th prio}{::}s) & \overset{\textit{def}}{=}\ \{th\} \cup \textit{threads}\ s \\
\textit{threads}\ (\textit{Exit th}{::}s) & \overset{\textit{def}}{=}\ \textit{threads}\ s - \{th\} \\
\textit{threads}\ (\_{::}s) & \overset{\textit{def}}{=}\ \textit{threads}\ s
\end{array}$$

In this definition $\_{::}\_$ stands for list-cons. Another function calculates the priority for a thread *th*, which is defined as

$$\begin{array}{ll}
\textit{priority th}\ [] & \overset{\textit{def}}{=}\ \textit{0} \\
\textit{priority th}\ (\textit{Create th}'\ \textit{prio}{::}s) & \overset{\textit{def}}{=}\ \textit{if th}' = \textit{th then prio else priority th s} \\
\textit{priority th}\ (\textit{Set th}'\ \textit{prio}{::}s) & \overset{\textit{def}}{=}\ \textit{if th}' = \textit{th then prio else priority th s} \\
\textit{priority th}\ (\_{::}s) & \overset{\textit{def}}{=}\ \textit{priority th s}
\end{array}$$

In this definition we set *0* as the default priority for threads that have not (yet) been created. The last function we need calculates the "time", or index, at which time a process had its priority last set.

$$\begin{array}{ll}
\textit{last\_set th}\ [] & \overset{\textit{def}}{=}\ \textit{0} \\
\textit{last\_set th}\ (\textit{Create th}'\ \textit{prio}{::}s) & \overset{\textit{def}}{=}\ \textit{if th} = \textit{th}'\ \textit{then}\ |s|\ \textit{else last\_set th s} \\
\textit{last\_set th}\ (\textit{Set th}'\ \textit{prio}{::}s) & \overset{\textit{def}}{=}\ \textit{if th} = \textit{th}'\ \textit{then}\ |s|\ \textit{else last\_set th s} \\
\textit{last\_set th}\ (\_{::}s) & \overset{\textit{def}}{=}\ \textit{last\_set th s}
\end{array}$$

In this definition $|s|$ stands for the length of the list of events s. Again the default value in this function is *0* for threads that have not been created yet. A *precedence* of a thread *th* in a state s is the pair of natural numbers defined as

$$\textit{prec th s}\ \overset{\textit{def}}{=}\ (\textit{priority th s}, \textit{last\_set th s})$$

The point of precedences is to schedule threads not according to priorities (because what should we do in case two threads have the same priority), but according to precedences. Precedences allow us to always discriminate between two threads with equal priority by taking into account the time when the priority was last set. We order precedences so that threads with the same priority get a higher precedence if their priority has been set earlier, since for such threads it is more urgent to finish their work. In an implementation this choice would translate to a quite natural FIFO-scheduling of processes with the same priority.

Next, we introduce the concept of *waiting queues*. They are lists of threads associated with every resource. The first thread in this list (i.e. the head, or short *hd*) is chosen to be the one that is in possession of the "lock" of the corresponding resource. We model waiting queues as functions, below abbreviated as *wq*. They take a resource as argument and return a list of threads. This allows us to define when a thread *holds*, respectively *waits* for, a resource *cs* given a waiting queue function *wq*.

$$holds\ wq\ th\ cs \overset{def}{=} th \in set\ (wq\ cs) \wedge th = hd\ (wq\ cs)$$
$$waits\ wq\ th\ cs \overset{def}{=} th \in set\ (wq\ cs) \wedge th \neq hd\ (wq\ cs)$$

In this definition we assume *set* converts a list into a set. At the beginning, that is in the state where no thread is created yet, the waiting queue function will be the function that returns the empty list for every resource.

$$all\_unlocked \overset{def}{=} \lambda\_.\ [] \qquad (1)$$

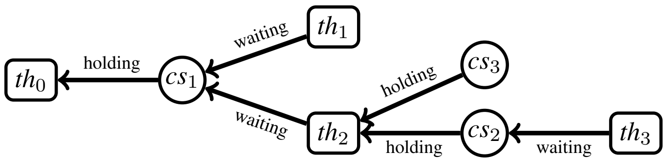
Using *holds* and *waits*, we can introduce *Resource Allocation Graphs* (RAG), which represent the dependencies between threads and resources. We represent RAGs as relations using pairs of the form

$$(T\ th, C\ cs) \quad \text{and} \quad (C\ cs, T\ th)$$

where the first stands for a *waiting edge* and the second for a *holding edge* (*C* and *T* are constructors of a datatype for vertices). Given a waiting queue function, a RAG is defined as the union of the sets of waiting and holding edges, namely

$$RAG\ wq \overset{def}{=} \{(T\ th, C\ cs) \mid waits\ wq\ th\ cs\} \cup \{(C\ cs, T\ th) \mid holds\ wq\ th\ cs\}$$

Given three threads and three resources, an instance of a RAG can be pictured as follows:



The use of relations for representing RAGs allows us to conveniently define the notion of the *dependants* of a thread using the transitive closure operation for relations. This gives

$$dependants\ wq\ th \overset{def}{=} \{th' \mid (T\ th', T\ th) \in (RAG\ wq)^+\}$$

This definition needs to account for all threads that wait for a thread to release a resource. This means we need to include threads that transitively wait for a resource being released (in the picture above this means the dependants of th_0 are th_1 and th_2, which wait for resource cs_1, but also th_3, which cannot make any progress unless th_2 makes progress, which in turn needs to wait for th_0 to finish). If there is a circle in a RAG, then clearly we have a deadlock. Therefore when a thread requests a resource, we must ensure that the resulting RAG is not circular.

Next we introduce the notion of the *current precedence* of a thread *th* in a state *s*. It is defined as

$$cprec\ wq\ s\ th \overset{def}{=} Max\ (\{prec\ th\ s\} \cup \{prec\ th'\ s \mid th' \in dependants\ wq\ th\}) \qquad (2)$$

where the dependants of *th* are given by the waiting queue function. While the precedence *prec* of a thread is determined by the programmer (for example when the thread is created), the point of the current precedence is to let the scheduler increase this precedence, if needed according to PIP. Therefore the current precedence of *th* is given as the maximum of the precedence *th* has in state *s and* all threads that are dependants of *th*. Since the notion *dependants* is defined as the transitive closure of all dependent threads, we deal correctly with the problem in the informal algorithm by Sha et al. [7] where a priority of a thread is lowered prematurely.

The next function, called *schs*, defines the behaviour of the scheduler. It will be defined by recursion on the state (a list of events); this function returns a *schedule state*, which we represent as a record consisting of two functions:

$$(\!|\mathit{wq\_fun},\ \mathit{cprec\_fun}|\!)$$

The first function is a waiting queue function (that is, it takes a resource *cs* and returns the corresponding list of threads that lock, respectively wait for, it); the second is a function that takes a thread and returns its current precedence (see (2)). We assume the usual getter and setter methods for such records.

In the initial state, the scheduler starts with all resources unlocked (the corresponding function is defined in (1)) and the current precedence of every thread is initialised with $(\mathit{0},\ \mathit{0})$; that means $\mathit{initial\_cprec} \overset{\mathit{def}}{=} \lambda\_.\ (\mathit{0},\ \mathit{0})$. Therefore we have for the initial state

$$\begin{aligned}&\mathit{schs}\ []\ \overset{\mathit{def}}{=}\\ &\quad (\!|\mathit{wq\_fun} = \mathit{all\_unlocked},\ \mathit{cprec\_fun} = \mathit{initial\_cprec}|\!)\end{aligned}$$

The cases for *Create*, *Exit* and *Set* are also straightforward: we calculate the waiting queue function of the (previous) state *s*; this waiting queue function *wq* is unchanged in the next schedule state—because none of these events lock or release any resource; for calculating the next *cprec_fun*, we use *wq* and *cprec*. This gives the following three clauses for *schs*:

$$\begin{aligned}&\mathit{schs}\ (\mathit{Create\ th\ prio}{::}s)\ \overset{\mathit{def}}{=}\\ &\quad \mathit{let\ wq} = \mathit{wq\_fun}\ (\mathit{schs}\ s)\ \mathit{in}\\ &\qquad (\!|\mathit{wq\_fun} = \mathit{wq},\ \mathit{cprec\_fun} = \mathit{cprec\ wq}\ (\mathit{Create\ th\ prio}{::}s)|\!)\\ &\mathit{schs}\ (\mathit{Exit\ th}{::}s)\ \overset{\mathit{def}}{=}\\ &\quad \mathit{let\ wq} = \mathit{wq\_fun}\ (\mathit{schs}\ s)\ \mathit{in}\\ &\qquad (\!|\mathit{wq\_fun} = \mathit{wq},\ \mathit{cprec\_fun} = \mathit{cprec\ wq}\ (\mathit{Exit\ th}{::}s)|\!)\\ &\mathit{schs}\ (\mathit{Set\ th\ prio}{::}s)\ \overset{\mathit{def}}{=}\\ &\quad \mathit{let\ wq} = \mathit{wq\_fun}\ (\mathit{schs}\ s)\ \mathit{in}\\ &\qquad (\!|\mathit{wq\_fun} = \mathit{wq},\ \mathit{cprec\_fun} = \mathit{cprec\ wq}\ (\mathit{Set\ th\ prio}{::}s)|\!)\end{aligned}$$

More interesting are the cases where a resource, say *cs*, is locked or released. In these cases we need to calculate a new waiting queue function. For the event *P th cs*, we have to update the function so that the new thread list for *cs* is the old thread list plus the thread *th* appended to the end of that list (remember the head of this list is assigned to be in the possession of this resource). This gives the clause

$$\begin{array}{l} \mathit{schs}\ (P\ \mathit{th}\ \mathit{cs}{::}s) \overset{\mathit{def}}{=} \\ \quad \mathit{let}\ \mathit{wq} = \mathit{wq\_fun}\ (\mathit{schs}\ s)\ \mathit{in} \\ \quad \mathit{let}\ \mathit{new\_wq} = \mathit{wq}(\mathit{cs} := (\mathit{wq}\ \mathit{cs}\ @\ [\mathit{th}]))\ \mathit{in} \\ \quad\ (\!|\mathit{wq\_fun} = \mathit{new\_wq}, \mathit{cprec\_fun} = \mathit{cprec}\ \mathit{new\_wq}\ (P\ \mathit{th}\ \mathit{cs}{::}s)|\!) \end{array}$$

The clause for event *V th cs* is similar, except that we need to update the waiting queue function so that the thread that possessed the lock is deleted from the corresponding thread list. For this list transformation, we use the auxiliary function *release*. A simple version of *release* would just delete this thread and return the remaining threads, namely

$$\begin{array}{ll} \mathit{release}\ [] & \overset{\mathit{def}}{=} [] \\ \mathit{release}\ (\_{::}\mathit{qs}) & \overset{\mathit{def}}{=} \mathit{qs} \end{array}$$

In practice, however, often the thread with the highest precedence in the list will get the lock next. We have implemented this choice, but later found out that the choice of which thread is chosen next is actually irrelevant for the correctness of PIP. Therefore we prove the stronger result where *release* is defined as

$$\begin{array}{ll} \mathit{release}\ [] & \overset{\mathit{def}}{=} [] \\ \mathit{release}\ (\_{::}\mathit{qs}) & \overset{\mathit{def}}{=} \mathit{SOME}\ \mathit{qs}'.\ \mathit{distinct}\ \mathit{qs}' \wedge \mathit{set}\ \mathit{qs}' = \mathit{set}\ \mathit{qs} \end{array}$$

where *SOME* stands for Hilbert's epsilon and implements an arbitrary choice for the next waiting list. It just has to be a list of distinctive threads and contain the same elements as *qs*. This gives for *V* the clause:

$$\begin{array}{l} \mathit{schs}\ (V\ \mathit{th}\ \mathit{cs}{::}s) \overset{\mathit{def}}{=} \\ \quad \mathit{let}\ \mathit{wq} = \mathit{wq\_fun}\ (\mathit{schs}\ s)\ \mathit{in} \\ \quad \mathit{let}\ \mathit{new\_wq} = \mathit{release}\ (\mathit{wq}\ \mathit{cs})\ \mathit{in} \\ \quad\ (\!|\mathit{wq\_fun} = \mathit{new\_wq}, \mathit{cprec\_fun} = \mathit{cprec}\ \mathit{new\_wq}\ (V\ \mathit{th}\ \mathit{cs}{::}s)|\!) \end{array}$$

Having the scheduler function *schs* at our disposal, we can "lift", or overload, the notions *waits*, *holds*, *RAG* and *cprec* to operate on states only.

$$\begin{array}{l} \mathit{holds}\ s \overset{\mathit{def}}{=} \mathit{holds}\ (\mathit{wq\_fun}\ (\mathit{schs}\ s)) \\ \mathit{waits}\ s \overset{\mathit{def}}{=} \mathit{waits}\ (\mathit{wq\_fun}\ (\mathit{schs}\ s)) \\ \mathit{RAG}\ s \overset{\mathit{def}}{=} \mathit{RAG}\ (\mathit{wq\_fun}\ (\mathit{schs}\ s)) \\ \mathit{cprec}\ s \overset{\mathit{def}}{=} \mathit{cprec\_fun}\ (\mathit{schs}\ s) \end{array}$$

With these abbreviations we can introduce the notion of threads being *ready* in a state (i.e. threads that do not wait for any resource) and the running thread.

$$\begin{array}{l} \mathit{ready}\ s \overset{\mathit{def}}{=} \{\mathit{th} \in \mathit{threads}\ s \mid \forall \mathit{cs}.\ \neg\ \mathit{waits}\ s\ \mathit{th}\ \mathit{cs}\} \\ \mathit{running}\ s \overset{\mathit{def}}{=} \{\mathit{th} \in \mathit{ready}\ s \mid \mathit{cprec}\ s\ \mathit{th} = \mathit{Max}\ (\mathit{cprec}\ s\ `\ \mathit{ready}\ s)\} \end{array}$$

In this definition _ ' _ stands for the image of a set under a function. Note that in the initial state, that is where the list of events is empty, the set *threads* is empty and therefore there is neither a thread ready nor running. If there is one or more threads ready, then there can only be *one* thread running, namely the one whose current precedence is equal to the maximum of all ready threads. We use sets to capture both possibilities. We can now also conveniently define the set of resources that are locked by a thread in a given state.

$$\textit{resources}\ s\ \textit{th} \overset{\textit{def}}{=} \{cs \mid (C\ cs, T\ \textit{th}) \in \textit{RAG}\ s\}$$

Finally we can define what a *valid state* is in our model of PIP. For example we cannot expect to be able to exit a thread, if it was not created yet. These validity constraints on states are characterised by the inductive predicate *step* and *valid_state*. We first give five inference rules for *step* relating a state and an event that can happen next.

$$\frac{\textit{th} \notin \textit{threads}\ s}{\textit{step}\ s\ (\textit{Create}\ \textit{th}\ \textit{prio})} \qquad \frac{\textit{th} \in \textit{running}\ s \qquad \textit{resources}\ s\ \textit{th} = \varnothing}{\textit{step}\ s\ (\textit{Exit}\ \textit{th})}$$

The first rule states that a thread can only be created, if it does not yet exists. Similarly, the second rule states that a thread can only be terminated if it was running and does not lock any resources anymore (this simplifies slightly our model; in practice we would expect the operating system releases all locks held by a thread that is about to exit). The event *Set* can happen if the corresponding thread is running.

$$\frac{\textit{th} \in \textit{running}\ s}{\textit{step}\ s\ (\textit{Set}\ \textit{th}\ \textit{prio})}$$

If a thread wants to lock a resource, then the thread needs to be running and also we have to make sure that the resource lock does not lead to a cycle in the RAG. In practice, ensuring the latter is the responsibility of the programmer. In our formal model we brush aside these problematic cases in order to be able to make some meaningful statements about PIP.[5]

$$\frac{\textit{th} \in \textit{running}\ s \qquad (C\ cs, T\ \textit{th}) \notin (\textit{RAG}\ s)^{+}}{\textit{step}\ s\ (P\ \textit{th}\ cs)}$$

Similarly, if a thread wants to release a lock on a resource, then it must be running and in the possession of that lock. This is formally given by the last inference rule of *step*.

$$\frac{\textit{th} \in \textit{running}\ s \qquad \textit{holds}\ s\ \textit{th}\ cs}{\textit{step}\ s\ (V\ \textit{th}\ cs)}$$

A valid state of PIP can then be conveniently be defined as follows:

[5] This situation is similar to the infamous occurs check in Prolog: In order to say anything meaningful about unification, one needs to perform an occurs check. But in practice the occurs check is ommited and the responsibility for avoiding problems rests with the programmer.

$$\frac{}{\textit{valid\_state}\ []} \qquad \frac{\textit{valid\_state}\ s \qquad \textit{step}\ s\ e}{\textit{valid\_state}\ (e{::}s)}$$

This completes our formal model of PIP. In the next section we present properties that show our model of PIP is correct.

3 The Correctness Proof

Sha et al. [7, Theorem 6] state their correctness criterion for PIP in terms of the number of critical resources: if there are *m* critical resources, then a blocked job with high priority can only be blocked *m* times—that is a bounded number of times. This, strictly speaking, does *not* prevent Priority Inversion, because if one low-priority thread does not give up its resource the high-priority thread is waiting for, then the high-priority thread can never run. The argument of Sha et al. is that *if* threads release locked resources in a finite amount of time, then Priority Inversion cannot occur—the high-priority thread is guaranteed to run eventually. The assumption is that programmers always ensure that this is the case. However, even taking this assumption into account, this property is *not* true for their version of PIP (as pointed out by Yodaiken [12]): A high-priority thread can be blocked an unbounded number of times by creating medium-priority threads that block a thread, which in turn locks a critical resource and has too low priority to make progress. In the way we have set up our formal model of PIP, their proof idea, even when fixed, does not seem to go through.

The idea behind our correctness criterion of PIP is as follows: for all states *s*, we know the corresponding thread *th* with the highest precedence; we show that in every future state (denoted by $s' @ s$) in which *th* is still alive, either *th* is running or it is blocked by a thread that was alive in the state *s*. Since in *s*, as in every state, the set of alive threads is finite, *th* can only be blocked a finite number of times. We will actually prove a stricter bound below. However, this correctness criterion hinges upon a number of assumptions about the states *s* and $s' @ s$, the thread *th* and the events happening in s'. We list them next:

Assumptions on the states *s* and $s' @ s$: In order to make any meaningful statement, we need to require that *s* and $s' @ s$ are valid states, namely

$\textit{valid\_state}\ s$
$\textit{valid\_state}\ (s' @ s)$

Assumptions on the thread *th*: The thread *th* must be alive in *s* and has the highest precedence of all alive threads in *s*. Furthermore the priority of *th* is *prio* (we need this in the next assumptions).

$th \in \textit{threads}\ s$
$\textit{prec}\ th\ s = \textit{Max}\ (\textit{cprec}\ s\ \text{`}\ \textit{threads}\ s)$
$\textit{prec}\ th\ s = (\textit{prio},\ \_)$

Assumptions on the events in s': We want to prove that *th* cannot be blocked indefinitely. Of course this can happen if threads with higher priority than *th* are continously created in s'. Therefore we have to assume that events in s' can only create (respectively set) threads with equal or lower priority than *prio* of *th*. We also need to assume that the priority of *th* does not get reset and also that *th* does not get "exited" in s'. This can be ensured by assuming the following three implications.

$$\begin{array}{l}
\textit{If Create th}'\,\textit{prio}' \in \textit{set } s' \textit{ then } \textit{prio}' \leq \textit{prio} \\
\textit{If Set th}'\,\textit{prio}' \in \textit{set } s' \textit{ then } \textit{th}' \neq \textit{th} \textit{ and } \textit{prio}' \leq \textit{prio} \\
\textit{If Exit th}' \in \textit{set } s' \textit{ then } \textit{th}' \neq \textit{th}
\end{array}$$

Under these assumptions we will prove the following correctness property:

Theorem 1. *Given the assumptions about states s and s' @ s, the thread th and the events in s', if $th' \in$ running $(s'$ @ $s)$ and $th' \neq th$ then $th' \in$ threads s.*

This theorem ensures that the thread *th*, which has the highest precedence in the state *s*, can only be blocked in the state s' @ s by a thread th' that already existed in *s*. As we shall see shortly, that means by only finitely many threads. Consequently, indefinite wait of *th*—which would be Priority Inversion—cannot occur.

In what follows we will describe properties of PIP that allow us to prove Theorem 1. It is relatively easily to see that

$$\begin{array}{l}
\textit{running } s \subseteq \textit{ready } s \subseteq \textit{threads } s \\
\textit{If valid\_state } s \textit{ then finite } (\textit{threads } s).
\end{array}$$

where the second property is by induction of *valid_state*. The next three properties are

$$\begin{array}{l}
\textit{If valid\_state } s \textit{ and waits } s\ \textit{th}\ cs_1 \textit{ and waits } s\ \textit{th}\ cs_2 \textit{ then } cs_1 = cs_2. \\
\textit{If holds } s\ \textit{th}_1\ cs \textit{ and holds } s\ \textit{th}_2\ cs \textit{ then } \textit{th}_1 = \textit{th}_2. \\
\textit{If valid\_state } s \textit{ and } \textit{th}_1 \in \textit{running } s \textit{ and } \textit{th}_2 \in \textit{running } s \textit{ then } \textit{th}_1 = \textit{th}_2.
\end{array}$$

The first one states that every waiting thread can only wait for a single resource (because it gets suspended after requesting that resource and having to wait for it); the second that every resource can only be held by a single thread; the third property establishes that in every given valid state, there is at most one running thread. We can also show the following properties about the RAG in *s*.

$$\begin{array}{l}
\textit{If valid\_state } s \textit{ then:} \\
\quad \textit{acyclic } (\textit{RAG } s), \textit{finite } (\textit{RAG } s) \textit{ and wf } ((\textit{RAG } s)^{-1}), \\
\quad \textit{if } T\ \textit{th} \in \textit{Domain } (\textit{RAG } s) \textit{ then } \textit{th} \in \textit{threads } s \\
\quad \textit{if } T\ \textit{th} \in \textit{Range } (\textit{RAG } s) \textit{ then } \textit{th} \in \textit{threads } s
\end{array}$$

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

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

   $\textit{RAG } (\textit{Set th prio}::s) = \textit{RAG } s$

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

   $RAG\ (Create\ th\ prio{::}s) = RAG\ s$

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

   $RAG\ (Exit\ th{::}s) = RAG\ s$

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

   $valid\_state\ (P\ th\ cs{::}s) \Longrightarrow$
   $RAG\ (P\ th\ cs{::}s) =$
   $(if\ wq\ s\ cs = []\ then\ RAG\ s \cup \{(C\ cs, T\ th)\}\ else\ RAG\ s \cup \{(T\ th, C\ cs)\})$

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

   $valid\_state\ (V\ th\ cs{::}s) \Longrightarrow$
   $RAG\ (V\ th\ cs{::}s) =$
   $RAG\ s - \{(C\ cs, T\ th)\} - \{(T\ th', C\ cs) \mid next\_th\ s\ th\ cs\ th'\} \cup$
   $\{(C\ cs, T\ th') \mid next\_th\ s\ th\ cs\ th'\}$

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

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

   $valid\_state\ s \Longrightarrow acyclic\ (RAG\ s)$

2. RAGs are finite (*finite_depend*):

   $valid\_state\ s \Longrightarrow finite\ (RAG\ s)$

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

   $valid\_state\ s \Longrightarrow wf\ ((RAG\ s)^{-1})$

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

   $[\![valid\_state\ s;\ (n, n_1) \in RAG\ s;\ (n, n_2) \in RAG\ s]\!] \Longrightarrow n_1 = n_2$

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

   $[\![valid\_state\ s;\ T\ th \in Domain\ (RAG\ s)]\!] \Longrightarrow th \in threads\ s$
   $[\![valid\_state\ s;\ T\ th \in Range\ (RAG\ s)]\!] \Longrightarrow th \in threads\ s$

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*):

   $[\![valid\_state\ s;\ node \in Domain\ (RAG\ s)]\!]$
   $\Longrightarrow \exists\, th'.\ th' \in ready\ s \wedge (node, T\ th') \in (RAG\ s)^{+}$

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

$$\llbracket \mathit{valid\_state}\ s;\ (n,\ T\ th_1) \in (\mathit{RAG}\ s)^+;\ th_1 \in \mathit{ready}\ s;\ (n,\ T\ th_2) \in (\mathit{RAG}\ s)^+;\\ th_2 \in \mathit{ready}\ s\rrbracket\\ \Longrightarrow th_1 = th_2$$

Properties about *next_th*:

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

   $\llbracket \mathit{valid\_state}\ s;\ \mathit{next\_th}\ s\ th\ cs\ th'\rrbracket \Longrightarrow th' \neq th$

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

   $\llbracket \mathit{next\_th}\ s\ th\ cs\ th_1;\ \mathit{next\_th}\ s\ th\ cs\ th_2\rrbracket \Longrightarrow th_1 = th_2$

Some deeper results about the system:

1. The maximum of *cprec* and *prec* are equal (*max_cp_eq*):

   $\mathit{valid\_state}\ s \Longrightarrow$
   $\mathit{Max}\ (\mathit{cprec}\ s\ \text{‘}\ \mathit{threads}\ s) = \mathit{Max}\ ((\lambda th.\ \mathit{prec}\ th\ s)\ \text{‘}\ \mathit{threads}\ s)$

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

   $\mathit{valid\_state}\ s \Longrightarrow \mathit{Max}\ (\mathit{cprec}\ s\ \text{‘}\ \mathit{ready}\ s) = \mathit{Max}\ (\mathit{cprec}\ s\ \text{‘}\ \mathit{threads}\ s)$

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*)

   $\mathit{valid\_state}\ (V\ \mathit{thread}\ cs{::}s) \Longrightarrow$
   $\mathit{cntCS}\ (V\ \mathit{thread}\ cs{::}s)\ \mathit{thread} + 1 = \mathit{cntCS}\ s\ \mathit{thread}$

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

   $\mathit{valid\_state}\ s \Longrightarrow$
   $\mathit{cntP}\ s\ th =$
   $\mathit{cntV}\ s\ th +$
   $(\mathit{if}\ th \in \mathit{ready}\ s \vee th \notin \mathit{threads}\ s\ \mathit{then}\ \mathit{cntCS}\ s\ th\ \mathit{else}\ \mathit{cntCS}\ s\ th + 1)$

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

   $\llbracket \mathit{valid\_state}\ s;\ th \notin \mathit{threads}\ s\rrbracket \Longrightarrow \mathit{cntP}\ s\ th = \mathit{cntV}\ s\ th$

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

   $\llbracket \mathit{valid\_state}\ s;\ th \notin \mathit{threads}\ s\rrbracket \Longrightarrow \mathit{resources}\ s\ th = \varnothing$

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*):

   $\llbracket \mathit{valid\_state}\ s;\ \mathit{cntP}\ s\ th = \mathit{cntV}\ s\ th\rrbracket \Longrightarrow \mathit{dependants}\ (\mathit{wq}\ s)\ th = \varnothing$

3.1 Proof idea

The reason that only threads which already held some resoures can be runing and block *th* is that if , otherwise, one thread does not hold any resource, it may never have its prioirty raised and will not get a chance to run. This fact is supported by lemma *moment_blocked*:

$$\llbracket th' \neq th;\ th' \in threads\ (moment\ i\ t\ @\ s);\ cntP\ (moment\ i\ t\ @\ s)\ th' = cntV\ (moment\ i\ t\ @\ s)\ th';\ i \leq j \rrbracket \\ \Longrightarrow cntP\ (moment\ j\ t\ @\ s)\ th' = cntV\ (moment\ j\ t\ @\ s)\ th' \wedge \\ th' \in threads\ (moment\ j\ t\ @\ s) \wedge th' \notin running\ (moment\ j\ t\ @\ s)$$

When instantiating *i* to *0*, the lemma means threads which did not hold any resource in state *s* will not have a change to run latter. Rephrased, it means any thread which is running after *th* became the highest must have already held some resource at state *s*.

When instantiating *i* to a number larger than *0*, the lemma means if a thread releases all its resources at some moment in *t*, after that, it may never get a change to run. If every thread releases its resource in finite duration, then after a while, only thread *th* is left running. This shows how indefinite priority inversion can be avoided.

So, the key of the proof is to establish the correctness of *moment_blocked*. We are going to show how this lemma is proved. At the heart of this proof, is lemma *pv_blocked*:

$$\llbracket th' \in threads\ (t\ @\ s);\ th' \neq th;\ cntP\ (t\ @\ s)\ th' = cntV\ (t\ @\ s)\ th' \rrbracket \\ \Longrightarrow th' \notin running\ (t\ @\ s)$$

This lemma says: for any *s*-extension text ”t”, if thread *th′* does not hold any resource, it can not be running at *t*@*s*.
Proof:

1. Since thread *th′* does not hold any resource, no thread may depend on it, so its current precedence $cp\ (t@s)\ th'$ equals to its own precedence $preced\ th'\ (t@s)$.
2. Since *th* has the highest precedence in the system and precedences are distinct among threads, we have $preced\ th'\ (t@s) < preced\ th\ (t@s)$. From this and item 1, we have $cp\ (t@s)\ th' < preced\ th\ (t@s)$.
3. Since $preced\ th\ (t@s)$ is already the highest in the system, $cp\ (t@s)\ th$ can not be higher than this and can not be lower neither (by the definition of *cp*), we have $preced\ th\ (t@s) = cp\ (t@s)\ th$.
4. Finally we have $cp\ (t@s)\ th' < cp\ (t@s)\ th$.
5. By defintion of *running*, *th′* can not be runing at *t*@*s*.

Since *th′* is not able to run at state *t*@*s*, it is not able to make either text ”P” or *V* action, so if *t*@*s* is extended one step further, *th′* still does not hold any resource. The situation will not unchanged in further extensions as long as *th* holds the highest precedence. Since this *t* is arbitarily chosen except being constrained by predicate *extend_highest_gen* and this predicate has the property that if it holds for *t*, it also holds for any moment *i* inside *t*, as shown by lemma *red_moment*:

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

so *pv_blocked* can be applied to any *moment i t*. From this, lemma *moment_blocked* follows.

4 Properties for an Implementation

While a formal correctness proof for our model of PIP is certainly attractive (especially in light of the flawed proof by Sha et al. [7]), we found that the formalisation can even help us with efficiently implementing PIP.

For example Baker complained that calculating the current precedence in PIP is quite "heavy weight" in Linux (see the Introduction). In our model of PIP the current precedence of a thread in a state s depends on all its dependants—a "global" transitive notion, which is indeed heavy weight (see Def. shown in (2)). We can however improve upon this. For this let us define the notion of *children* of a thread *th* in a state *s* as

$$\mathit{children}\ s\ \mathit{th} \overset{def}{=} \{\mathit{th}' \mid \exists\, cs.\ (T\ \mathit{th}', C\ cs) \in \mathit{RAG}\ s \wedge (C\ cs, T\ \mathit{th}) \in \mathit{RAG}\ s\}$$

where a child is a thread that is one "hop" away from the tread *th* in the *RAG* (and waiting for *th* to release a resource). We can prove that

Lemma 1. *If valid_state s then*

$$\mathit{cprec}\ s\ \mathit{th} = \mathit{Max}\ (\{\mathit{prec}\ \mathit{th}\ s\} \cup \mathit{cprec}\ s\ \text{‘}\ \mathit{children}\ s\ \mathit{th}).$$

That means the current precedence of a thread *th* can be computed locally by considering only the children of *th*. In effect, it only needs to be recomputed for *th* when one of its children changes its current precedence. Once the current precedence is computed in this more efficient manner, the selection of the thread with highest precedence from a set of ready threads is a standard scheduling operation implemented in most operating systems.

Of course the main implementation work for PIP involves the scheduler and coding how it should react to events. Below we outline how our formalisation guides this implementation for each kind of event.

Create th prio: We assume that the current state s' and the next state $s \overset{def}{=} \mathit{Create}\ \mathit{th}\ \mathit{prio}{::}s'$ are both valid (meaning the event is allowed to occur). In this situation we can show that

$\mathit{RAG}\ s = \mathit{RAG}\ s'$,
$\mathit{cprec}\ s\ \mathit{th} = \mathit{prec}\ \mathit{th}\ s$, *and*
If $\mathit{th}' \neq \mathit{th}$ *then* $\mathit{cprec}\ s\ \mathit{th}' = \mathit{cprec}\ s'\ \mathit{th}'$.

This means we do not have recalculate the *RAG* and also none of the current precedences of the other threads. The current precedence of the created thread *th* is just its precedence, namely the pair $(\mathit{prio}, |s|)$.

Exit th: We again assume that the current state s' and the next state $s \overset{def}{=} \mathit{Exit}\ \mathit{th}{::}s'$ are both valid. We can show that

$RAG\ s = RAG\ s'$, *and*
If $th' \neq th$ *then* $cprec\ s\ th' = cprec\ s'\ th'$.

This means again we do not have to recalculate the *RAG* and also not the current precedences for the other threads. Since *th* is not alive anymore in state *s*, there is no need to calculate its current precedence.

Set th prio: We assume that s' and $s \overset{def}{=} Set\ th\ prio{::}s'$ are both valid. We can show that

$RAG\ s = RAG\ s'$, *and*
If $th' \neq th$ *and* $th \notin dependants\ s\ th'$ *then* $cprec\ s\ th' = cprec\ s'\ th'$.

The first property is again telling us we do not need to change the *RAG*. The second however states that only threads that are *not* dependants of *th* have their current precedence unchanged. For the others we have to recalculate the current precedence. To do this we can start from *th* and follow the *RAG*-chains to recompute the *cprec* of every thread encountered on the way using Lemma 1. Since the *RAG* is loop free, this procedure will always stop. The following two lemmas show, however, that this procedure can actually stop often earlier without having to consider all dependants.

If $th \in dependants\ s\ th''$ *and* $cprec\ s\ th = cprec\ s'\ th$ *then* $cprec\ s\ th'' = cprec\ s'\ th''$.
If $th \in dependants\ s\ th'$, $th' \in dependants\ s\ th''$ *and* $cprec\ s\ th' = cprec\ s'\ th'$
then $cprec\ s\ th'' = cprec\ s'\ th''$.

The first states that if the current precedence of *th* is unchanged, then the procedure can stop immediately (all dependent threads have their *cprec*-value unchanged). The second states that if an intermediate *cprec*-value does not change, then the procedure can also stop, because none of its dependent threads will have their current precedence changed.

V th cs: We assume that s' and $s \overset{def}{=} V\ th\ cs{::}s'$ are both valid. We have to consider two subcases: one where there is a thread to "take over" the released resource *cs*, and one where there is not. Let us consider them in turn. Suppose in state *s*, the thread th' takes over resource *cs* from thread *th*. We can show

$$RAG\ s = RAG\ s' - \{(C\ cs, T\ th), (T\ th', C\ cs)\} \cup \{(C\ cs, T\ th')\}$$

which shows how the *RAG* needs to be changed. This also suggests how the current precedences need to be recalculated. For threads that are not *th* and th' nothing needs to be changed, since we can show

If $th'' \neq th$ *and* $th'' \neq th'$ *then* $cprec\ s\ th'' = cprec\ s'\ th''$.

For *th* and th' we need to use Lemma 1 to recalculate their current prcedence since their children have changed.

In the other case where there is no thread that takes over *cs*, we can show how to recalculate the *RAG* and also show that no current precedence needs to be recalculated.

$RAG\ s = RAG\ s' - \{(C\ cs,\ T\ th)\}$
$cprec\ s\ th' = cprec\ s'\ th'$

$P\ th\ cs$: We assume that s' and $s \overset{def}{=} P\ th\ cs{::}s'$ are both valid. We again have to analyse two subcases, namely the one where cs is locked, and where it is not. We treat the second case first by showing that

$RAG\ s = RAG\ s' \cup \{(C\ cs,\ T\ th)\}$
$cprec\ s\ th' = cprec\ s'\ th'$

This means we do not need to add a holding edge to the *RAG* and no current precedence needs to be recalculated.

In the second case we know that resouce cs is locked. We can show that

$RAG\ s = RAG\ s' \cup \{(T\ th,\ C\ cs)\}$
If $th \notin dependants\ s\ th'$ *then* $cprec\ s\ th' = cprec\ s'\ th'$.

That means we have to add a waiting edge to the *RAG*. Furthermore the current precedence for all threads that are not dependants of *th* are unchanged. For the others we need to follow the edges in the *RAG* and recompute the *cprec*. However, like in the @case of text Set, this operation can stop often earlier, namely when intermediate values do not change.

A pleasing result of our formalisation is that the properties in this section closely inform an implementation of PIP: Whether the *RAG* needs to be reconfigured or current precedences need to recalculated for an event is given by a lemma we proved.

5 Conclusion

The Priority Inheritance Protocol (PIP) is a classic textbook algorithm used in real-time operating systems in order to avoid the problem of Priority Inversion. Although classic and widely used, PIP does have its faults: for example it does not prevent deadlocks in cases where threads have circular lock dependencies.

We had two goals in mind with our formalisation of PIP: One is to make the notions in the correctness proof by Sha et al. [7] precise so that they can be processed by a theorem prover. The reason is that a mechanically checked proof avoids the flaws that crept into their informal reasoning. We achieved this goal: The correctness of PIP now only hinges on the assumptions behind our formal model. The reasoning, which is sometimes quite intricate and tedious, has been checked beyond any reasonable doubt by Isabelle/HOL. We can also confirm that Paulson's inductive method for protocol verification [5] is quite suitable for our formal model and proof. The traditional application area of this method is security protocols. The only other application of Paulson's method we know of outside this area is [10].

The second goal of our formalisation is to provide a specification for actually implementing PIP. Textbooks, for example [9, Section 5.6.5], explain how to use various implementations of PIP and abstractly discuss their properties, but surprisingly lack most details for a programmer who wants to implement PIP. That this is an issue in practice is illustrated by the email from Baker we cited in the Introduction. We achieved also this goal: The formalisation gives the first author enough data to enable his undergraduate students to implement PIP (as part of their OS course) on top of PINTOS, a small operating system for teaching purposes. A byproduct of our formalisation effort is that nearly all design choices for the PIP scheduler are backed up with a proved lemma. We were also able to establish the property that the choice of the next thread which takes over a lock is irrelevant for the correctness of PIP. Earlier model checking approaches which verified implementations of PIP [2,3,11] cannot provide this kind of "deep understanding" about the principles behind PIP and its correctness.

PIP is a scheduling algorithm for single-processor systems. We are now living in a multi-processor world. So the question naturally arises whether PIP has any relevance in such a world beyond teaching. Priority Inversion certainly occurs also in multiprocessor systems. However, the surprising answer, according to [8], is that except for one unsatisfactory proposal nobody has a good idea for how PIP should be modified to work correctly on multi-processor systems. The difficulties become clear when considering that locking and releasing a resource always requires a small amount of time. If processes work independently, then a low priority process can "steal" in such an unguarded moment a lock for a resource that was supposed allow a high-priority process to run next. Thus the problem of Priority Inversion is not really prevented. It seems difficult to design a PIP-algorithm with a meaningful correctness property on a multiprocessor systems where processes work independently. We can imagine PIP to be of use in situations where processes are *not* independent, but coordinated via a master process that distributes work over some slave processes. However, a formal investigation of this is beyond the scope of this paper. We are not aware of any proofs in this area, not even informal ones.

The most closely related work to ours is the formal verification in PVS for Priority Ceiling done by Dutertre [1]. His formalisation consists of 407 lemmas and 2500 lines of "specification" (we do not know whether this includes also code for proofs). Our formalisation consists of around 210 lemmas and overall 6950 lines of readable Isabelle/Isar code with a few apply-scripts interspersed. The formal model of PIP is 385 lines long; the formal correctness proof 3800 lines. Some auxiliary definitions and proofs took 770 lines of code. The properties relevant for an implementation took 2000 lines. Our code can be downloaded from ...

References

1. 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.
2. J. M. S. Faria. *Formal Development of Solutions for Real-Time Operating Systems with TLA+/TLC*. PhD thesis, University of Porto, 2008.
3. E. Jahier, B. Halbwachs, and P. Raymond. Synchronous Modeling and Validation of Priority Inheritance Schedulers. In *Proc. of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 5503 of *LNCS*, pages 140–154, 2009.
4. B. W. Lampson and D. D. Redell. Experiences with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):105–117, 1980.

5. L. C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *Journal of Computer Security*, 6(1–2):85–128, 1998.
6. G. E. Reeves. Re: What Really Happened on Mars? *Risks Forum*, 19(54), 1998.
7. L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.
8. U. Steinberg, A. Böttcher, and B. Kauer. Timeslice Donation in Component-Based Systems. In *Proc. of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, pages 16–23, 2010.
9. U. Vahalia. *UNIX Internals: The New Frontiers*. Prentice-Hall, 1996.
10. J. Wang, H. Yang, and X. Zhang. Liveness Reasoning with Isabelle/HOL. In *Proc. of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *LNCS*, pages 485–499, 2009.
11. A. Wellings, A. Burns, O. M. Santos, and B. M. Brosgol. Integrating Priority Inheritance Algorithms in the Real-Time Specification for Java. In *Proc. of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 115–123. IEEE Computer Society, 2007.
12. V. Yodaiken. Against Priority Inheritance. Technical report, Finite State Machine Labs (FSMLabs), 2004.

6 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*): *valid_state s*.
2. th is a living thread in s (*threads_s*): $th \in threads\ s$.
3. th has the highest precedence in s (*highest*): $prec\ th\ s = Max\ (cprec\ s\ `\ threads\ s)$.
4. The precedence of th is $(prio, tm)$ (*preced_th*): $prec\ th\ s = (prio, tm)$.

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*):

   $cprec\ s\ th = prec\ th\ s$

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

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

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

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

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

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

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

1. t is a valid extension of s (*vt_t*): $valid\_state\ (t\ @\ s)$.
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*):

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

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*):

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

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

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

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

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

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

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

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

   $cprec\ (t\ @\ s)\ th = prec\ th\ s$

The main theorem of this part is to characterizing the running thread during t ($runing\_inversion\_2$):

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

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.