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 [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 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) [9][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* [9], 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 [15]:

> *"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 [11, 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* [9] 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 [9], despite being informally "proved" correct, is actually *flawed*.

Yodaiken [15] points to a subtlety that had been overlooked in the informal proof by Sha et al. They specify in [9] 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" [9]. 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 [3,4,14], 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. [9] 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 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 [7], 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:

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

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

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

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.

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

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

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

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*.

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

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.

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

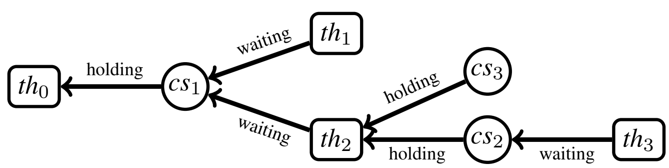
Using *holds* and *waits*, we can introduce *Resource Allocation Graphs* (RAG), which represent the dependencies between threads and resources. We represent RAGs as relations using pairs of the form

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

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

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

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

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

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

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

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. [9] 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 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 $(0,\ 0)$; that means $\mathit{initial\_cprec} \stackrel{def}{=} \lambda\_.\ (0,\ 0)$. Therefore we have for the initial state

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

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*:

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

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 seen to be in the possession of this resource). This gives the clause

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

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

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

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

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

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:

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

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.

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

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.

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

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 the set-comprehension to capture both possibilities. We can now also conveniently define the set of resources that are locked by a thread in a given state.

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

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.

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

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 held lock of a thread that is about to exit). The event *Set* can happen if the corresponding thread is running.

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

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 of course the responsibility of the programmer. In our formal model we just exclude such problematic cases in order to be able to make some meaningful statements about PIP.[5]

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

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*.

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

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. [9, 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 can only be blocked *m* times—that is a bounded number of times. For their version of PIP, this property is *not* true (as pointed out by Yodaiken [15]) as 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.

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

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

running s $\subseteq$ *ready s* $\subseteq$ *threads s*
If valid_state s then finite (*threads s*).

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

If valid_state s and waits s th cs_1 *and waits s th* cs_2 *then* $cs_1 = cs_2$.
If holds s th_1 *cs and holds s* th_2 *cs then* $th_1 = th_2$.
If valid_state s and $th_1 \in$ *running s and* $th_2 \in$ *running s then* $th_1 = th_2$.

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*.

If valid_state s then:
acyclic (*RAG s*), *finite* (*RAG s*) *and wf* $((\textit{RAG } s)^{-1})$,
if T th $\in$ *Domain* (*RAG s*) *then th* $\in$ *threads s*
if T th $\in$ *Range* (*RAG s*) *then th* $\in$ *threads s*

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*):

   RAG (*Set th prio*::*s*) = *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*):

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

5. Execution of V (*step_depend_v*):

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

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

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

$$\mathit{valid\_state}\ s \Longrightarrow \mathit{acyclic}\ (\mathit{RAG}\ s)$$

2. RAGs are finite (*finite_depend*):

$$\mathit{valid\_state}\ s \Longrightarrow \mathit{finite}\ (\mathit{RAG}\ s)$$

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

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

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

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

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

$$
\begin{array}{l}
[\![\mathit{valid\_state}\ s;\ T\ th \in \mathit{Domain}\ (\mathit{RAG}\ s)]\!] \Longrightarrow th \in \mathit{threads}\ s \\
[\![\mathit{valid\_state}\ s;\ T\ th \in \mathit{Range}\ (\mathit{RAG}\ s)]\!] \Longrightarrow th \in \mathit{threads}\ s
\end{array}
$$

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*):

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

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

$$
\begin{array}{l}
[\![\mathit{valid\_state}\ s;\ (n,\ T\ th_1) \in (\mathit{RAG}\ s)^{+};\ th_1 \in \mathit{ready}\ s;\ (n,\ T\ th_2) \in (\mathit{RAG}\ s)^{+}; \\
\ th_2 \in \mathit{ready}\ s]\!] \\
\Longrightarrow th_1 = th_2
\end{array}
$$

Properties about *next_th*:

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

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

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

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

Some deeper results about the system:

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

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

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

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

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*)

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

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

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

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

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

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

$\llbracket valid\_state\ s;\ th \notin threads\ s \rrbracket \Longrightarrow resources\ s\ th = \emptyset$

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*):

$\llbracket valid\_state\ s;\ cntP\ s\ th = cntV\ s\ th \rrbracket \Longrightarrow dependants\ (wq\ s)\ th = \emptyset$

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. [9]), 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 our 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 prove how to improve upon this. For this let us define the notion of *children* of a thread as

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

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

Lemma 1. *If valid_state s then*

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

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 change their 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 the events, for example which datastructures need to be modified (mainly *RAG* and *cprec*). Below we outline how our formalisation guides this implementation for each event.

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

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

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 is just its precedence, that is the pair $(\mathit{prio}, |s|)$.

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

$$\mathit{RAG}\ s = \mathit{RAG}\ s'$$
$$\mathit{If}\ th' \neq th\ \mathit{then}\ \mathit{cprec}\ s\ th' = \mathit{cprec}\ s'\ th'.$$

This means also we do not have to recalculate the *RAG* and 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 $s \stackrel{def}{=} \textit{Set th prio}::s'$ are both valid. We can show that

$$\textit{RAG } s = \textit{RAG } s'$$
$$\textit{If } th' \neq th \textit{ and } th \notin \textit{dependants } s\ th' \textit{ then } \textit{cprec } s\ th' = \textit{cprec } s'\ th'.$$

The first is again telling us we do not need to change the *RAG*. The second however states that only threads that are *not* dependent on *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 always stop. The the following two lemmas show this procedure can actually stop often earlier.

$$\textit{If } th \in \textit{dependants } s\ th'' \textit{ and } \textit{cprec } s\ th = \textit{cprec } s'\ th \textit{ then } \textit{cprec } s\ th'' = \textit{cprec } s'\ th''.$$
$$\textit{If } th \in \textit{dependants } s\ th',\ th' \in \textit{dependants } s\ th'' \textit{ and cprec } s\ th' = \textit{cprec } s'\ th' \textit{ then cprec } s\ th'' = \textit{cprec } s'\ th''.$$

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 $s \stackrel{def}{=} V\ th\ cs::s'$ are both valid. We have to consider two subcases: one where there is a thread to “take over” the released resource *cs*, and 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

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

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

$$\textit{If } th'' \neq th \textit{ and } th'' \neq th' \textit{ then } \textit{cprec } s\ th'' = \textit{cprec } s'\ th''.$$

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, except for *th* (like in the case above).

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

P th cs: We assume that s' and $s \stackrel{def}{=} P\ th\ cs{::}s'$ 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 must be recalculated (including that for *th*).

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

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

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

TO DO a few sentences summarising what has been achieved.

5 Conclusion

The Priority Inheritance Protocol is a classic textbook algorithm used in real-time systems in order to avoid the problem of Priority Inversion.

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.

Our formalisation and the one presented in [12] are the only ones that employ Paulson's method for verifying protocols which are *not* security related.

TO DO

no clue about multi-processor case according to [10]

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 [9]. PCE is less convenient to use because it requires static analysis of programs. Therefore, PI is more commonly used in practice[6]. 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 [4,14,3] 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.

The rest of the paper is organized as follows: Section 6 gives an overview of PI. Section **??** introduces the formal model of PI. Section 7 discusses a series of basic properties of PI. Section 8 shows formally how priority inversion is controlled by PI. Section 4 gives properties which can be used for guidelines of implementation. Section 9 discusses related works. Section 10 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.

6 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.

7 General properties of Priority Inheritance

8 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*):

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

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*):

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

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*):

$$\begin{array}{l} th' \in running\ (t\ @\ s) \Longrightarrow\\ th' = th \vee th' \neq th \wedge th' \in threads\ s \wedge cntV\ s\ th' < cntP\ s\ th' \end{array}$$

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.

9 Related works

1. *Integrating Priority Inheritance Algorithms in the Real-Time Specification for Java* [14] 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* [13]. 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 ???.

10 Conclusions

The work in this paper only deals with single CPU configurations. The ”one CPU” assumption is essential for our formalisation, because the main lemma fails in multi-CPU configuration. The lemma says that any runing thead must be the one with the highest prioirty or already held some resource when the highest priority thread was initiated. When there are multiple CPUs, it may well be the case that a threads did not hold any resource when the highest priority thread was initiated, but that thread still runs after that moment on a separate CPU. In this way, the main lemma does not hold anymore.

There are some works deals with priority inversion in multi-CPU configurations[???], but none of them have given a formal correctness proof. The extension of our formal proof to deal with multi-CPU configurations is not obvious. One possibility, as suggested in paper [???], is change our formal model (the defiintion of ”schs”) to give the released resource to the thread with the highest prioirty. In this way, indefinite prioirty inversion can be avoided, but for a quite different reason from the one formalized in this paper (because the ”mail lemma” will be different). This means a formal correctness proof for milt-CPU configuration would be quite different from the one given in this paper. The solution of prioirty inversion problem in mult-CPU configurations is a different problem which needs different solutions which is outside the scope of this paper.

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*. PhD thesis, University of Porto, 2008.
4. 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.
5. B. W. Lampson and D. D. Redell. Experiences with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):105–117, 1980.
6. D. Locke. Priority inheritance: The real story. http://www.math.unipd.it/~tullio/SCD/2007/Materiale/Locke.pdf, 2002.
7. L. C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *Journal of Computer Security*, 6(1–2):85–128, 1998.
8. G. E. Reeves. Re: What Really Happened on Mars? *Risks Forum*, 19(54), 1998.
9. 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.
10. U. Steinberg, A. Bötcher, 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.
11. U. Vahalia. *UNIX Internals: The New Frontiers*. Prentice-Hall, 1996.
12. 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.
13. 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.
14. 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.
15. V. Yodaiken. Against Priority Inheritance. Technical report, Finite State Machine Labs (FSMLabs), 2004.