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, 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:

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

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

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

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.

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

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

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

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*.

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

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.

$$all\_unlocked \stackrel{def}{=} \lambda\_.\ [] \tag{1}$$

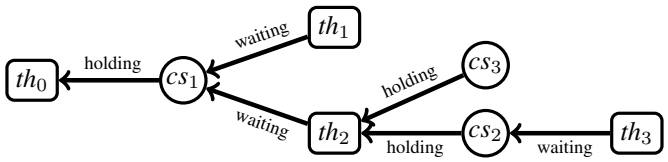
Using *holds* and *waits*, we can introduce *Resource Allocation Graphs* (RAG), which represent the dependencies between threads and resources. We represent RAGs as relations using pairs of the form

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

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

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

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

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

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

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

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:

$$(\!|wq\_fun, cprec\_fun|\!)$$

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 $initial\_cprec \stackrel{def}{=} \lambda\_.\ (0, 0)$. Therefore we have for the initial state

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

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*:

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

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

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

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

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

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

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

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:

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

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.

$$\begin{array}{l}
holds\ s \stackrel{def}{=} holds\ (wq_fun\ (schs\ s)) \\
waits\ s \stackrel{def}{=} waits\ (wq_fun\ (schs\ s)) \\
RAG\ s \stackrel{def}{=} RAG\ (wq_fun\ (schs\ s)) \\
cprec\ s \stackrel{def}{=} cprec_fun\ (schs\ s)
\end{array}$$

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.

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

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 no thread ready nor a 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 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.

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

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

3 Correctness Proof

Sha et al. [9] state their correctness criterion of 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 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 locking a critical resource and having a too low priority. 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 active, either th is running or it is blocked by a thread that was active in the state s. Since in s, as in every state, the set of active threads is finite, th can only blocked a finite number of time. We will actually prove a stricter bound. However, this correctness criterion depends on a number of assumptions about the states s and $s' @ s$, the thread th and the events happening in s'.

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

$$valid\_state\ s$$
$$valid\_state\ (s' @ s)$$

Assumptions on the thread th: The thread th must be active in s and has the highest precedence of all active threads in s. Furthermore the priority of th is $prio$.

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

Assumptions on the events in s': We want to prove that th cannot be blocked indefinitely. Of course this can by violated if threads with higher priority than th are created in s'. Therefore we have to assume that events in s' can only create (respectively set) threads with lower or equal priority than $prio$ of th. We also have to assume that th does not get "exited" in s'.

$$If\ \ Create\ \_\ prio' \in set\ s'\ \ then\ \ prio' \leq prio$$
$$If\ \ Set\ th'\ prio' \in set\ s'\ \ then\ \ th' \neq th\ \ and\ \ prio' \leq prio$$
$$If\ \ Exit\ th' \in set\ s'\ \ then\ \ th' \neq th$$

Under these assumption we will prove the following property:

Theorem 1. *Given the assumptions about states s and $s' @ s$, the thread th and the events in s'. If $th' \in running\ (t @ s)$ and $th' \neq th$ then $th' \in threads\ s$.*

Theorem 2. *If $th' \in running\ (t @ s)$ then $th' = th \vee th' \neq th \wedge th' \in threads\ s \wedge cntV\ s\ th' < cntP\ s\ th'$.*

TO DO

4 Properties for an Implementation

TO DO

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 7 introduces the formal model of PI. Section 8 discusses a series of basic properties of PI. Section 9 shows formally how priority inversion is controlled by PI. Section 10 gives properties which can be used for guidelines of implementation. Section 11 discusses related works. Section 12 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 Formal model of Priority Inheritance

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

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

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

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

datatype *event* =

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

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

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

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

type-synonym *state* = *event list*

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

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

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

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

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

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

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

definition *preced* :: *thread* ⇒ *state* ⇒ *precedence*
where *preced thread s* $\stackrel{def}{=}$ *Prc* (*original_priority thread s*) (*birthtime thread s*)

A number of important notions are defined here:

consts
holding :: *'b* ⇒ *thread* ⇒ *cs* ⇒ *bool*
waiting :: *'b* ⇒ *thread* ⇒ *cs* ⇒ *bool*
depend :: *'b* ⇒ (*node* × *node*) *set*
dependents :: *'b* ⇒ *thread* ⇒ *thread set*

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

defs (**overloaded**)
— We define that the thread which is at the head of waiting queue of resource *cs* is holding the resource. This definition is slightly different from tradition where all threads in the waiting queue are considered as waiting for the resource. This notion is reflected in the definition of *holding wq th cs* as follows:
cs_holding_def:
holding wq thread cs $\stackrel{def}{=}$ (*thread* ∈ *set* (*wq cs*) ∧ *thread* = *hd* (*wq cs*))
— In accordance with the definition of *holding wq th cs*, a thread *th* is considered waiting for *cs* if it is in the *waiting queue* of critical resource *cs*, but not at the head. This is reflected in the definition of *waiting wq th cs* as follows:
cs_waiting_def:
waiting wq thread cs $\stackrel{def}{=}$ (*thread* ∈ *set* (*wq cs*) ∧ *thread* ≠ *hd* (*wq cs*))
— *depend wq* represents the Resource Allocation Graph of the system under the waiting queue function *wq*.
cs_depend_def:

depend (*wq*::*cs* ⇒ *thread list*) $\stackrel{def}{=}$
$\{(Th\ th, Cs\ cs) \mid th\ cs.\ waiting\ wq\ th\ cs\} \cup \{(Cs\ cs, Th\ th) \mid cs\ th.\ holding\ wq\ th\ cs\}$
— The following *dependents wq th* represents the set of threads which are depending on thread *th* in Resource Allocation Graph *depend wq*:
cs_dependents_def:
dependents (*wq*::*cs* ⇒ *thread list*) *th* $\stackrel{def}{=}$ $\{th'.\ (Th\ th', Th\ th) \in (depend\ wq)\hat{}+\}$

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

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

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

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

abbreviation
all_unlocked $\stackrel{def}{=}$ λ_::*cs*. ([]::*thread list*)

abbreviation
initial_cprec $\stackrel{def}{=}$ λ_::*thread*. *Prc 0 0*

abbreviation
release qs $\stackrel{def}{=}$ *case qs of*
[] => []
| (_#*qs*) => (*SOME q*. *distinct q* ∧ *set q* = *set qs*)

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

fun *schs* :: *state* ⇒ *schedule_state*
where
schs [] = (| *wq_fun* = λ *cs*. [], *cprec_fun* = (λ_. *Prc 0 0*) |) |

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

```
 schs (Create th prio # s) =
    (let wq = wq_fun (schs s) in
      (|wq_fun = wq, cprec_fun = cpreced wq (Create th prio # s)|))
| schs (Exit th # s) =
    (let wq = wq_fun (schs s) in
      (|wq_fun = wq, cprec_fun = cpreced wq (Exit th # s)|))
| schs (Set th prio # s) =
    (let wq = wq_fun (schs s) in
      (|wq_fun = wq, cprec_fun = cpreced wq (Set th prio # s)|))
| schs (P th cs # s) =
    (let wq = wq_fun (schs s) in
     let new_wq = wq(cs := (wq cs @ [th])) in
      (|wq_fun = new_wq, cprec_fun = cpreced new_wq (P th cs # s)|))
| schs (V th cs # s) =
    (let wq = wq_fun (schs s) in
     let new_wq = wq(cs := release (wq cs)) in
      (|wq_fun = new_wq, cprec_fun = cpreced new_wq (V th cs # s)|))
```

lemma *cpreced_initial*:
cpreced (λ *cs*. []) [] = (λ_. (*Prc 0 0*))
apply(*simp add*: *cpreced_def*)
apply(*simp add*: *cs_dependents_def cs_depend_def cs_waiting_def cs_holding_def*)
apply(*simp add*: *preced_def*)
done

lemma *sch_old_def*:
schs (*e*#*s*) = (*let ps* = *schs s in*

```
        let pwq = wq_fun ps in
        let nwq = case e of
                P th cs ⇒ pwq(cs:=(pwq cs @ [th])) |
                V th cs ⇒ let nq = case (pwq cs) of
                                  [] ⇒ [] |
                                  (_#qs) ⇒ (SOME q. distinct q ∧ set q = set qs)
                          in pwq(cs:=nq)              |
                _ ⇒ pwq
        in let ncp = cpreced nwq (e#s) in
          (|wq_fun = nwq, cprec_fun = ncp|)
       )
apply(cases e)
apply(simp_all)
done
```

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

```
definition wq :: state ⇒ cs ⇒ thread list
  where wq s = wq_fun (schs s)
```

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

```
definition cp :: state ⇒ thread ⇒ precedence
  where cp s ≡ cprec_fun (schs s)
```

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

```
defs (overloaded)
  s_holding_abv:
  holding (s::state) ≡ holding (wq_fun (schs s))
  s_waiting_abv:
  waiting (s::state) ≡ waiting (wq_fun (schs s))
  s_depend_abv:
  depend (s::state) ≡ depend (wq_fun (schs s))
  s_dependents_abv:
  dependents (s::state) ≡ dependents (wq_fun (schs s))
```

The following lemma can be proved easily:

```
lemma
  s_holding_def:
  holding (s::state) th cs ≡ (th ∈ set (wq_fun (schs s) cs) ∧ th = hd (wq_fun (schs s)
cs))
  by (auto simp:s_holding_abv wq_def cs_holding_def)

lemma s_waiting_def:
```

waiting (*s*::*state*) *th cs* $\stackrel{def}{=}$ (*th* ∈ *set* (*wq_fun* (*schs s*) *cs*) ∧ *th* ≠ *hd* (*wq_fun* (*schs s*) *cs*))
by (*auto simp*:*s_waiting_abv wq_def cs_waiting_def*)

lemma *s_depend_def*:
depend (*s*::*state*) =
{(*Th th*, *Cs cs*) | *th cs*. *waiting* (*wq s*) *th cs*} ∪ {(*Cs cs*, *Th th*) | *cs th*. *holding* (*wq s*) *th cs*}
by (*auto simp*:*s_depend_abv wq_def cs_depend_def*)

lemma
s_dependents_def:
dependents (*s*::*state*) *th* $\stackrel{def}{=}$ {*th'* . (*Th th'*, *Th th*) ∈ (*depend* (*wq s*))^+}
by (*auto simp*:*s_dependents_abv wq_def cs_dependents_def*)

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

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

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

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

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

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

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

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

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

inductive *step* :: *state* ⇒ *event* ⇒ *bool*
where
— A thread can be created if it is not a live thread:
thread_create: ⟦*thread* ∉ *threads s*⟧ ⟹ *step s* (*Create thread prio*) |
— A thread can exit if it no longer hold any resource:
thread_exit: ⟦*thread* ∈ *runing s*; *holdents s thread* = {}⟧ ⟹ *step s* (*Exit thread*) |
— A thread can request for an critical resource *cs*, if it is running and the request does not form a loop in the current RAG. The latter condition is set up to avoid deadlock. The condition also reflects our assumption all threads are carefully programmed so that deadlock can not happen:

thread_P: $[\![thread \in runing\ s;\ (Cs\ cs, Th\ thread)\ \notin (depend\ s)\hat{}+]\!] \Longrightarrow$
$step\ s\ (P\ thread\ cs)$ |
— A thread can release a critical resource *cs* if it is running and holding that resource:
thread_V: $[\![thread \in runing\ s;\ holding\ s\ thread\ cs]\!] \Longrightarrow step\ s\ (V\ thread\ cs)$ |
— A thread can adjust its own priority as long as it is current running:
thread_set: $[\![thread \in runing\ s]\!] \Longrightarrow step\ s\ (Set\ thread\ prio)$

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

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

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

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

fun *the_cs* :: *node* ⇒ *cs*
where $the\_cs\ (Cs\ cs) = cs$

fun *the_th* :: *node* ⇒ *thread*
where $the\_th\ (Th\ th) = th$

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

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

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

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

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

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

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

definition *cntV* :: *state* ⇒ *thread* ⇒ *nat*

where *cntV s th* = *count* $(\lambda\, e.\ \exists\ cs.\ e = V\ th\ cs)\ s$

8 General properties of Priority Inheritance

The following are several very basic prioprites:

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

   running s $\subseteq$ *ready s*

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

   ready s $\subseteq$ *threads s*

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

   valid_state s $\Longrightarrow$ *finite* (*threads s*)

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

   valid_state s $\Longrightarrow$ *distinct* (*wq s cs*)

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

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

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

   $[\![$*valid_state* (*e*::*s*); *thread* $\notin$ *set* (*wq s cs*); *thread* $\in$ *set* (*wq* (*e*::*s*) *cs*)$]\!]$
   $\Longrightarrow$ *e* = *P thread cs*

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

   $[\![$*valid_state s*; *waits s th* cs_1; *waits s th* $cs_2]\!] \Longrightarrow cs_1 = cs_2$

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

   $[\![$*valid_state s*; *holds s* th_1 *cs*; *holds s* th_2 *cs*$]\!] \Longrightarrow th_1 = th_2$

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

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

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

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

   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*):

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

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*):

   $[\![valid\_state\ s; (n, T\ th_1) \in (RAG\ s)^+; th_1 \in ready\ s; (n, T\ th_2) \in (RAG\ s)^+;$
   $th_2 \in ready\ s]\!]$
   $\Longrightarrow th_1 = th_2$

Properties about *next_th*:

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

   $[\![valid\_state\ s; next\_th\ s\ th\ cs\ th']\!] \Longrightarrow th' \neq th$

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

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

Some deeper results about the system:

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

   $[\![valid\_state\ s; th_1 \in running\ s; th_2 \in running\ s]\!] \Longrightarrow th_1 = th_2$

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

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

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

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

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

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

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

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

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

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

   $[\![valid\_state\ s; th \notin threads\ s]\!] \Longrightarrow cntP\ s\ th = cntV\ s\ th$

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

   $[\![valid\_state\ s; th \notin threads\ s]\!] \Longrightarrow resources\ s\ th = \varnothing$

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*):

   $[\![valid\_state\ s; cntP\ s\ th = cntV\ s\ th]\!] \Longrightarrow dependants\ (wq\ s)\ th = \varnothing$

9 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*):

$$\begin{array}{l}
[\![R\ []; \\
\bigwedge e\ t.\ [\![valid_state\ (t\ @\ s);\ step\ (t\ @\ s)\ e; \\
\quad\quad extend_highest_gen\ s\ th\ prio\ tm\ t; \\
\quad\quad extend_highest_gen\ s\ th\ prio\ tm\ (e::t);\ R\ t]\!] \\
\quad\quad \Longrightarrow R\ (e::t)]\!] \\
\Longrightarrow R\ t
\end{array}$$

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.

10 Properties to guide implementation

The properties (especially *runing_inversion_2*) convinced us that the model defined in Section 7 does prevent indefinite priority inversion and therefore fulfills the fundamental requirement of Priority Inheritance protocol. Another purpose of this paper is to show how this model can be used to guide a concrete implementation. As discussed in Section 5.6.5 of [11], the implementation of Priority Inheritance in Solaris uses sophisticated linking data structure. Except discussing two scenarios to show how the data structure should be manipulated, a lot of details of the implementation are missing. In [3,4,14] the protocol is described formally using different notations, but little information is given on how this protocol can be implemented efficiently, especially there is no information on how these data structure should be manipulated.

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

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

$$valid\_state\ s \Longrightarrow cprec\ s\ th = Max\ (\{prec\ th\ s\} \cup cprec\ s\ `\ children\ s\ th)$$

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

$$children\ s\ th \stackrel{def}{=} \{th' \mid (T\ th', T\ th) \in child\ s\}$$

where the definition of *child* is:

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

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

10.1 Event *Set th prio*

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

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

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

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

$[\![th' \neq th; th \notin dependants\ s\ th']\!] \Longrightarrow cprec\ s\ th' = cprec\ s'\ th'$

This lemma implies the *cprec*-value of *th* and those threads which have a dependence relation with *th* might need to be recomputed. The way to do this is to start from *th* and follow the *RAG*-chain to recompute the *cprec*-value of every encountered thread using lemma *cp_rec*. Since the *RAG*-relation is loop free, this procedure can always stop. The the following lemma shows this procedure actually could stop earlier.

2. The following two lemma shows, if a thread the re-computation of which gives an unchanged *cprec*-value, the procedure described above can stop.
   (a) Lemma *eq_up_self* shows if the re-computation of *th*'s *cprec* gives the same result, the procedure can stop:
   $[\![th \in dependants\ s\ th''; cprec\ s\ th = cprec\ s'\ th]\!]$
   $\Longrightarrow cprec\ s\ th'' = cprec\ s'\ th''$
   (b) Lemma *eq_up*) shows if the re-computation at intermediate threads gives unchanged result, the procedure can stop:
   $[\![th \in dependants\ s\ th'; th' \in dependants\ s\ th''; cprec\ s\ th' = cprec\ s'\ th']\!]$
   $\Longrightarrow cprec\ s\ th'' = cprec\ s'\ th''$

10.2 Event *V th cs*

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

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

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

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

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

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

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

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

   which shows two edges are removed while one is added. These changes imply how the current precedences should be re-computed.
2. First all threads different from *th* and th' have their *cprec*-value kept, therefore do not need a re-computation (*cp_kept*):

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

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

The other sub-case is when for all th'

$\neg$ *next_th s' th cs th'*

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

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

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

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

   $cprec\ s\ th' = cprec\ s'\ th'$

10.3 Event *P th cs*

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

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

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

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

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

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

   $cprec\ s\ th' = cprec\ s'\ th'$

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

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

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

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

   $th \notin dependants\ s\ th' \Longrightarrow cprec\ s\ th' = cprec\ s'\ th'$

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

   $[\![th \in dependants\ s\ th';\ th' \in dependants\ s\ th'';\ cprec\ s\ th' = cprec\ s'\ th']\!]$
   $\Longrightarrow cprec\ s\ th'' = cprec\ s'\ th''$

10.4 Event *Create th prio*

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

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

The following results can be obtained under this context:

1. The RAG does not change (eq_dep):

   $RAG\ s = RAG\ s'$

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

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

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

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

10.5 Event *Exit th*

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

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

The following results can be obtained under this context:

1. The RAG does not change (eq_dep):

   $RAG\ s = RAG\ s'$

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

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

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

11 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 ???.

12 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.