Priority Inheritance Protocol Proved Correct

Xingyuan Zhang, Christian Urban and Chunhan Wu

December 21, 2012

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. We also generalise the original informal proof to the practically relevant case where critical sections can overlap. Our formalisation in Isabelle/HOL not just 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; our implementation builds on top of the small PINTOS operating system.

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 the thread itself cannot be blocked indefinitely by threads with lower priority. Alas, in a naive implementation of resource locking and priorities this property can be violated. For this let L be in the possession of a lock for a resource that H also 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

[*] This is a revised and expanded version of [16].

is the software that controlled the Mars Pathfinder mission in 1997 [9]. 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 to 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) [11][1] 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 performed dynamically by the scheduler. This is in contrast to, for example, *Priority Ceiling* [11], 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 avoiding PIP altogether by designing the system so that no priority inversion may happen in the first place. However, such ideal designs may not always be achievable in practice.

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 [12, Section 5.6.5]), but they help little with actual implementations. That this is a problem in practice is proved by an email by 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."*

The criticism by Yodaiken, Baker and others suggests another look at PIP from a more abstract level (but still concrete enough to inform an implementation), and makes PIP a good candidate for a formal verification. An additional reason is that the original presentation of PIP [11], 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 [11] that after the thread (whose priority has been raised) completes its

[1] Sha et al. call it the *Basic Priority Inheritance Protocol* [11] and others sometimes also call it *Priority Boosting*, *Priority Donation* or *Priority Lending*.

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” [11]. 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 switch to the highest remaining priority of the threads that it blocks. The advantage of formalising the correctness of a high-level specification of PIP in a theorem prover is that such issues clearly show up and cannot be overlooked as in informal reasoning (since we have to analyse all possible behaviours of threads, i.e. *traces*, that could possibly happen).

Contributions: There have been earlier formal investigations into PIP [2, 4, 14], but they employ model checking techniques. This paper presents a formalised and mechanically checked proof for the correctness of PIP. For this we needed to design a new correctness criterion for PIP. 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 help with an efficient implementation of PIP in the educational PINTOS operating system [8]. 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—a fact that has not been mentioned in the literature and not been used in the reference implementation of PIP in PINTOS. This fact, however, is important for an efficient implementation of PIP, because we can give the lock to the thread with the highest priority so that it terminates more quickly. We were also able to generalise the scheduler of Sha et al. [11] to the practically relevant case where critical sections can overlap.

2 Formal Model of the Priority Inheritance Protocol

The Priority Inheritance Protocol, short PIP, is a scheduling algorithm for a single-processor system.[2] Following good experience in earlier work [13], our model of PIP is based on Paulson’s inductive approach to protocol verification [7]. In this approach a *state* of a system is given by a list of events that happened so far (with new events prepended to the list). *Events* of PIP fall into five categories defined as the datatype:

$$\begin{array}{lcll}
\textbf{datatype}\ \textit{event} & = & \textit{Create thread priority} & \\
 & | & \textit{Exit thread} & \\
 & | & \textit{Set thread priority} & \text{reset of the priority for } \textit{thread} \\
 & | & \textit{P thread cs} & \text{request of resource } \textit{cs} \text{ by } \textit{thread} \\
 & | & \textit{V thread cs} & \text{release of resource } \textit{cs} \text{ by } \textit{thread}
\end{array}$$

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 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:

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

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

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

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

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.

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

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

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

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*.

$$\begin{aligned}
&\mathit{holds\ wq\ th\ cs} \overset{\mathit{def}}{=} \mathit{th} \in \mathit{set}\ (\mathit{wq\ cs}) \wedge \mathit{th} = \mathit{hd}\ (\mathit{wq\ cs})\\
&\mathit{waits\ wq\ th\ cs} \overset{\mathit{def}}{=} \mathit{th} \in \mathit{set}\ (\mathit{wq\ cs}) \wedge \mathit{th} \neq \mathit{hd}\ (\mathit{wq\ cs})
\end{aligned}$$

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.

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

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

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

If there is no cycle, then every RAG can be pictured as a forrest of trees, for example as follows:

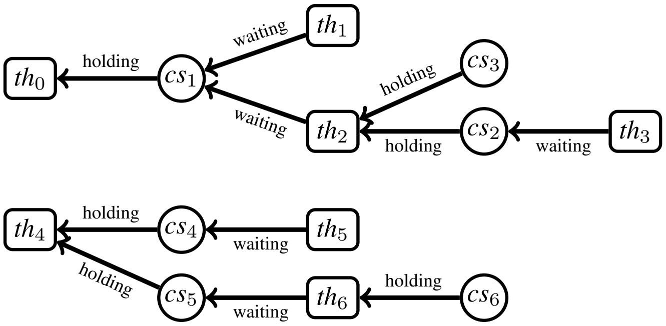

Figure 1: An instance of a Resource Allocation Graph (RAG).

We will design our scheduler so that every thread can be in the possession of several resources, that is it has potentially several incoming holding edges in the RAG, but has at most one outgoing waiting edge. The reason is that when a thread asks for resource that is locked already, then the process is blocked and cannot ask for another resource. Clearly, also every resource can only have at most one outgoing holding edge—indicating that the resource is locked. In this way we can always start at a thread waiting for a resource and "chase" outgoing arrows leading to a single root of a tree.

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

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

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 of dependencies 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. In practice, the programmer has to ensure this.

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

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

where the dependants of *th* are given by the waiting queue function. While the precedence *prec* of a thread is determined statically (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. [11] where a priority of a thread is lowered prematurely.

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

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

The first function is a waiting queue function (that is, it takes a resource *cs* and returns the corresponding list of threads that lock, respectively wait for, it); the second is a function that takes a thread and returns its current precedence (see the definition in (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 $\mathit{initial\_cprec} \overset{\mathit{def}}{=} \lambda\_.\ (\mathit{0},\ \mathit{0})$. Therefore we have for the initial shedule state

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

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*:

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

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

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

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

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

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

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

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

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

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.

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

With these abbreviations in place we can introduce the notion of a thread being *ready* in a state (i.e. threads that do not wait for any resource, which are the roots of the trees in the RAG, see Figure 1). The *running* thread is then the thread with the highest current precedence of all ready threads.

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

In the second definition _ ‘ _ stands for the image of a set under a function. Note that in the initial state, that is where the list of events is empty, the set *threads* is empty and therefore there is neither a thread ready nor running. If there is one or more threads ready, then there can only be *one* thread running, namely the one whose current precedence is equal to the maximum of all ready threads. We use sets to capture both possibilities. We can now also conveniently define the set of resources that are locked by a thread in a given state and also when a thread is detached in a state (meaning the thread neither holds nor waits for a resource—in the RAG this would correspond to an isolated node without any incoming and outgoing edges, see Figure 1):

$$
\begin{array}{l}
\mathit{resources}\ s\ \mathit{th} \overset{\mathit{def}}{=} \{\mathit{cs} \mid \mathit{holds}\ s\ \mathit{th}\ \mathit{cs}\} \\
\mathit{detached}\ s\ \mathit{th} \overset{\mathit{def}}{=} (\nexists\, \mathit{cs}.\ \mathit{holds}\ s\ \mathit{th}\ \mathit{cs}) \wedge (\nexists\, \mathit{cs}.\ \mathit{waits}\ s\ \mathit{th}\ \mathit{cs})
\end{array}
$$

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.

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

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

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

If a thread wants to lock a resource, then the thread needs to be running and also we have to make sure that the resource lock does not lead to a cycle in the RAG. In practice, ensuring the latter is the responsibility of the programmer. In our formal model we brush aside these problematic cases in order to be able to make some meaningful statements about PIP.[3]

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

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*.

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

[3] 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 omitted and the responsibility for avoiding problems rests with the programmer.

Note, however, that apart from the circularity condition, we do not make any assumption on how different resources can locked and released relative to each other. In our model it is possible that critical sections overlap. This is in contrast to Sha et al [11] who require that critical sections are properly nested.

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

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

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. state their first correctness criterion for PIP in terms of the number of low-priority threads [11, Theorem 3]: if there are n low-priority threads, then a blocked job with high priority can only be blocked a maximum of n times. Their second correctness criterion is given in terms of the number of critical resources [11, Theorem 6]: if there are m critical resources, then a blocked job with high priority can only be blocked a maximum of m times. Both results on their own, strictly speaking, do *not* prevent indefinite, or unbounded, Priority Inversion, because if a low-priority thread does not give up its critical resource (the one the high-priority thread is waiting for), then the high-priority thread can never run. The argument of Sha et al. is that *if* threads release locked resources in a finite amount of time, then indefinite Priority Inversion cannot occur—the high-priority thread is guaranteed to run eventually. The assumption is that programmers must ensure that threads are programmed in this way. However, even taking this assumption into account, the correctness properties of Sha et al. are *not* true for their version of PIP—despite being "proved". As Yodaiken [15] pointed out: If a low-priority thread possesses locks to two resources for which two high-priority threads are waiting for, then lowering the priority prematurely after giving up only one lock, can cause indefinite Priority Inversion for one of the high-priority threads, invalidating their two bounds.

Even when fixed, their proof idea does not seem to go through for us, because of the way we have set up our formal model of PIP. One reason is that we allow critical sections, which start with a P-event and finish with a corresponding V-event, to arbitrarily overlap (something Sha et al. explicitly exclude). Therefore we have designed a different correctness criterion for PIP. The idea behind our criterion 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 and was waiting for or in the possession of a lock in 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. This is independent of how many threads of lower priority are created in s'. We will actually prove a stronger statement where we also provide the current precedence of the blocking thread. 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:** We need to require that s and s' @ s are valid states:

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

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).

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

Assumptions on the events in s'**:** We want to prove that *th* cannot be blocked indefinitely. Of course this can happen if threads with higher priority than *th* are continuously 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.

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

The locale mechanism of Isabelle helps us to manage conveniently such assumptions [3]. Under these assumptions we shall 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* $th' \in threads\ s$, $\neg\ detached\ s\ th'$ *and* $cprec\ (s'\ @\ s)\ th' = prec\ th\ s$.

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* and requested or had a lock on at least one resource—that means the thread was not *detached* in *s*. As we shall see shortly, that means there are only finitely many threads that can block *th* in this way and then they need to run with the same precedence as *th*.

Like in the argument by Sha et al. our finite bound does not guarantee absence of indefinite Priority Inversion. For this we further have to assume that every thread gives up its resources after a finite amount of time. We found that this assumption is awkward to formalise in our model. Therefore we leave it out and let the programmer assume the responsibility to program threads in such a benign manner (in addition to causing no circularity in the RAG). In this detail, we do not make any progress in comparison with the work by Sha et al. However, we are able to combine their two separate bounds into a single theorem improving their bound.

In what follows we will describe properties of PIP that allow us to prove Theorem 1 and, when instructive, briefly describe our argument. It is relatively easy to see that

$running\ s \subseteq ready\ s \subseteq threads\ s$
$If\ \ valid\_state\ s\ \ then\ \ finite\ (threads\ s).$

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 property states that every waiting thread can only wait for a single resource (because it gets suspended after requesting that resource); 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* $wf\ ((RAG\ s)^{-1})$,
if $T\ th \in Domain\ (RAG\ s)$ *then* $th \in threads\ s$ *and*
if $T\ th \in Range\ (RAG\ s)$ *then* $th \in threads\ s$.

The acyclicity property follows from how we restricted the events in *step*; similarly the finiteness and well-foundedness property. The last two properties establish that every thread in a *RAG* (either holding or waiting for a resource) is a live thread.

The key lemma in our proof of Theorem 1 is as follows:

Lemma 2 *Given the assumptions about states* s *and* $s' @ s$*, the thread* th *and the events in* s'*, if* $th' \in threads\ (s' @ s)$, $th' \neq th$ *and* $detached\ (s' @ s)\ th'$
then $th' \notin running\ (s' @ s)$.

The point of this lemma is that a thread different from *th* (which has the highest precedence in *s*) and not holding any resource, cannot be running in the state $s' @ s$.

Proof *Since thread* th' *does not hold any resource, no thread can depend on it. Therefore its current precedence* $cprec\ (s' @ s)\ th'$ *equals its own precedence* $prec\ th'\ (s' @ s)$*. Since* th *has the highest precedence in the state* $(s' @ s)$ *and precedences are distinct among threads, we have* $prec\ th'\ (s' @ s) < prec\ th\ (s' @ s)$*. From this we have* $cprec\ (s' @ s)\ th' < prec\ th\ (s' @ s)$*. Since* $prec\ th\ (s' @ s)$ *is already the highest* $cprec\ (s' @ s)\ th$ *can not be higher than this and can not be lower either (by definition of cprec). Consequently, we have* $prec\ th\ (s' @ s) = cprec\ (s' @ s)\ th$*. Finally we have* $cprec\ (s' @ s)\ th' < cprec\ (s' @ s)\ th$*. By defintion of running,* th' *can not be running in state* $s' @ s$*, as we had to show.* ■

Since th' is not able to run in state $s' @ s$, it is not able to issue a *P* or *V* event. Therefore if $s' @ s$ is extended one step further, th' still cannot hold any resource. The situation will not change in further extensions as long as *th* holds the highest precedence.

From this lemma we can deduce Theorem 1: that *th* can only be blocked by a thread th' that held some resource in state *s* (that is not *detached*). And furthermore that the current precedence of th' in state $(s' @ s)$ must be equal to the precedence of *th* in *s*. We show this theorem by induction on s' using Lemma 2. This theorem gives a stricter bound on the threads that can block *th* than the one obtained by Sha et al. [11]: only threads that were alive in state *s* and moreover held a resource. This means our bound is in terms of both—alive threads in state *s* and number of critical resources. Finally, the theorem establishes that the blocking threads have the current precedence raised to the precedence of *th*.

We can furthermore prove that under our assumptions no deadlock exists in the state $s' @ s$ by showing that $running\ (s' @ s)$ is not empty.

Lemma 3 *Given the assumptions about states* s *and* $s' @ s$*, the thread* th *and the events in* s'*,* $running\ (s' @ s) \neq \varnothing$.

Proof *If th is blocked, then by following its dependants graph, we can always reach a ready thread th', and that thread must have inherited the precedence of th.* ■

4 Properties for an Implementation

While our formalised proof gives us confidence about the correctness of our model of PIP, we found that the formalisation can even help us with efficiently implementing it.

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

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

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

Lemma 4 *If valid_state s then*

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

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 changes its current precedence. Once the current precedence is computed in this more efficient manner, the selection of the thread with highest precedence from a set of ready threads is a standard scheduling operation implemented in most operating systems.

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

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

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

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

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

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

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

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

$$\begin{array}{l} \mathit{RAG}\ s = \mathit{RAG}\ s',\ \textit{and} \\ \textit{If}\ \mathit{th}' \neq \mathit{th}\ \textit{ and }\ \mathit{th} \notin \mathit{dependants}\ s\ \mathit{th}'\ \textit{ then }\ \mathit{cprec}\ s\ \mathit{th}' = \mathit{cprec}\ s'\ \mathit{th}'. \end{array}$$

The first property is again telling us we do not need to change the *RAG*. The second shows that the *cprec*-values of all threads other than *th* are unchanged. The reason is that *th* is running; therefore it is not in the *dependants* relation of any other thread. This in turn means that the change of its priority cannot affect other threads.

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

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

which shows how the *RAG* needs to be changed. The next lemma 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

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

For *th* and th' we need to use Lemma 4 to recalculate their current precedence 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.

$$\begin{array}{l} \mathit{RAG}\ s = \mathit{RAG}\ s' - \{(C\ \mathit{cs}, T\ \mathit{th})\} \\ \mathit{cprec}\ s\ \mathit{th}' = \mathit{cprec}\ s'\ \mathit{th}' \end{array}$$

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

$$\begin{array}{l} \mathit{RAG}\ s = \mathit{RAG}\ s' \cup \{(C\ \mathit{cs}, T\ \mathit{th})\} \\ \mathit{cprec}\ s\ \mathit{th}' = \mathit{cprec}\ s'\ \mathit{th}' \end{array}$$

This means we need to add a holding edge to the *RAG* and no current precedence needs to be recalculated.

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

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

That means we have to add a waiting edge to the *RAG*. Furthermore the current precedence for all threads that are not dependants of th' are unchanged. For the others we need to follow the edges in the *RAG* and recompute the *cprec*. To do this we can start from *th* and follow the *RAG*-edges to recompute using Lemma 4 the *cprec* of every thread encountered on the way. Since the *RAG* is loop free, this procedure will always stop. The following lemma shows, however, that this procedure can actually stop often earlier without having to consider all dependants.

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

This lemma 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.

As can be seen, a pleasing byproduct of our formalisation is that the properties in this section closely inform an implementation of PIP, namely whether the RAG needs to be reconfigured or current precedences need to be recalculated for an event. This information is provided by the lemmas we proved. We confirmed that our observations translate into practice by implementing our version of PIP on top of PINTOS, a small operating system written in C and used for teaching at Stanford University [8]. To implement PIP, we only need to modify the kernel functions corresponding to the events in our formal model. The events translate to the following function interface in PINTOS:

Event	**PINTOS function**
Create	`thread_create`
Exit	`thread_exit`
Set	`thread_set_priority`
P	`lock_acquire`
V	`lock_release`

Our implicit assumption that every event is an atomic operation is ensured by the architecture of PINTOS (which allows to disable interrupts when some operations are performed). The case where an unlocked resource is given next to the waiting thread with the highest precedence is realised in our implementation by priority queues. We implemented them as *Braun trees* [6], which provide efficient $O(log\ n)$-operations for accessing and updating. Apart from having to implement relatively complex datastructures in C using pointers, our experience with the implementation has been very positive: our specification and formalisation of PIP translates smoothly to an efficent implementation in PINTOS. Let us illustrate this with the C-code of the function `lock_acquire`, shown in Figure 2. This function implements the operation of requesting and, if free, locking of a resource by the currently running thread. The convention in the PINTOS code is use the terminology *locks* rather than resources. A lock is represented as a pointer to the structure `lock` (Line 1).

Lines 2 to 4 of the function `lock_acquire` contain diagnostic code: first, we check that the lock is a "valid" lock by testing whether it is not `NULL`; second, we check that the code is not called as part of an interrupt—acquiring a lock should only be initiated by a request from a (user) thread, not an interrupt; third, we make sure the current thread does not ask twice for a lock. These assertions are supposed to be satisfied because of the assumptions in PINTOS about how this code is called. If not, then the assertions indicate a bug in PINTOS and the result will be a *kernel panic*. We took these three lines from the original code of `lock_acquire` in PINTOS.

```
void lock_acquire (struct lock *lock)
{ ASSERT (lock != NULL);
  ASSERT (!intr_context());
  ASSERT (!lock_held_by_current_thread (lock));

  enum intr_level old_level;
  old_level = intr_disable();
  if (lock->value == 0) {
    queue_insert(thread_cprec, &lock->wq, &thread_current()->helem);
    thread_current()->waiting = lock;
    struct thread *pt;
    pt = lock->holder;
    while (pt) {
      queue_update(lock_cprec, &pt->held, &lock->helem);
      if (!(update_cprec(pt)))
        break;
      lock = pt->waiting;
      if (!lock) {
        queue_update(higher_cprec, &ready_queue, &pt->helem);
        break;
      };
      queue_update(thread_cprec, &lock->wq, &pt->helem);
      pt = lock->holder;
    };
    thread_block();
  } else {
    lock->value--;
    lock->holder = thread_current();
    queue_insert(lock_prec, &thread_current()->held, &lock->helem);
  };
  intr_set_level(old_level);
}
```

Figure 2: Our version of the `lock_release` function in PINTOS. It implements the operation corresponding to a *P*-event.

Line 6 and 7 of `lock_acquire` make the operation of acquiring a lock atomic by disabling all interrupts, but saving them for resumption at the end of the function (Line 31). In Line 8, the interesting code with respect to scheduling starts: we first check whether the lock is already taken (its value is then 0 indicating "already taken", or 1 for being "free"). In case the lock is taken, we enter the if-branch inserting the current thread into the waiting queue of this lock (Line 9). The waiting queue is referenced in the usual C-way as `&lock->wq`. Next we record that the current thread is waiting for the lock (Line 10). Thus we established two pointers: one in the waiting queue of the lock pointing to the current thread, and the other from the currend thread pointing to the lock. According to our specification in Section 2 and the properties we were able to prove for *P*, we need to "chase" all the dependants in the RAG (Resource Allocation Graph) and update their current precedence, however we only have to do this as long as there is change in the current precedence from one thread at hand to another.

The "chase" is implemented in the while-loop in Lines 13 to 24. To initialise the loop, we assign in Lines 11 and 12 the variable `pt` to the owner of the lock. Inside the loop, we first update the precedence of the lock held by `pt` (Line 14). Next, we check whether there is a change in the current precedence of `pt`. If not, then we leave the loop, since nothing else needs to be updated (Lines 15 and 16). If there is a change, then we have to continue our "chase". We check what lock the thread `pt` is waiting for (Lines 17 and 18). If there is none, then the thread `pt` is ready (the "chase" is finished with finding a root). In this case we update the ready-queue accordingly (Lines 19 and 20). If there is a lock `pt` is waiting for, we update the waiting queue for this lock and we continue the loop with the holder of that lock (Lines 22 and 23). After all current precedences have been updated, we finally need to block the current thread, because the lock it asked for was taken (Line 25).

If the lock the current thread asked for is *not* taken, we proceed with the else-branch (Lines 26 to 30). We first decrease the value of the lock to 0, meaning it is taken now (Line 27). Second, we update the reference of the holder of the lock (Line 28), and finally update the queue of locks the current thread already possesses (Line 29). The very last is to enable interrupts again thus leaving the protected section.

Similar operations need to be implementated for the `lock_release` function, which we however do not show. The reader should note that we did not verify our C-code. The verification of the specification however provided us with the justification for designing the C-code in this way. It gave us confidence for leaving the "chase" early whenever there is no change in the calculated current precedence.

5 Conclusion

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

We had two goals in mind with our formalisation of PIP: One is to make the notions in the correctness proof by Sha et al. [11] precise so that they can be processed by a theorem prover. The reason is that a mechanically checked proof avoids the flaws that crept into their informal reasoning. We achieved this goal: The correctness of PIP now only hinges on the assumptions behind our formal model. The reasoning, which is sometimes quite intricate and tedious, has been checked by Isabelle/HOL. We can also confirm that Paulson's inductive method for protocol verification [7] is quite suitable for our formal model and proof. The traditional application area of this method is security protocols.

The second goal of our formalisation is to provide a specification for actually implementing PIP. Textbooks, for example [12, Section 5.6.5], explain how to use various implementations of PIP and abstractly discuss their properties, but surprisingly lack most details important for a programmer who wants to implement PIP (similarly Sha et al. [11]). That this is an issue in practice is illustrated by the email from Baker we cited in the Introduction. We achieved also this goal: The formalisation allowed us to efficently implement our version of PIP on top of PINTOS [8], a simple instructional operating system for the x86 architecture. It also gives the first author enough data to enable his undergraduate students to implement PIP (as part of their OS course). A byproduct of our formalisation effort is that nearly all design choices for the implementation of PIP scheduler are backed up with a proved lemma. We were also able to establish the property that the choice of the next thread which takes over a lock is irrelevant for the correctness of PIP. Moreover, we eliminated a crucial restriction present in the proof of Sha et al.: they require that critical sections nest properly, whereas our scheduler allows critical sections to overlap.

PIP is a scheduling algorithm for single-processor systems. We are now living in a multi-processor world. Priority Inversion certainly occurs also there. However, there is very little "foundational" work about PIP-algorithms on multi-processor systems. We are not aware of any correctness proofs, not even informal ones. There is an implementation of a PIP-algorithm for multi-processors as part of the "real-time" effort in Linux, including an informal description of the implemented scheduling algorithm given in [10]. We estimate that the formal verification of this algorithm, involving more fine-grained events, is a magnitude harder than the one we presented here, but still within reach of current theorem proving technology. We leave this for future work.

The most closely related work to ours is the formal verification in PVS of the Priority Ceiling Protocol done by Dutertre [1]—another solution to the Priority Inversion problem, which however needs static analysis of programs in order to avoid it. There have been earlier formal investigations into PIP [2, 4, 14], but they employ model checking techniques. The results obtained by them apply, however, only to systems with a fixed size, such as a fixed number of events and threads. In contrast, our result applies to systems of arbitrary size. Moreover, our result is a good witness for one of the major reasons to be interested in machine checked reasoning: gaining deeper understanding of the subject matter.

Our formalisation consists of around 210 lemmas and overall 6950 lines of readable Isabelle/Isar code with a few apply-scripts interspersed. The formal model of PIP is 385 lines long; the formal correctness proof 3800 lines. Some auxiliary definitions and proofs span over 770 lines of code. The properties relevant for an implementation require 2000 lines.

Acknowledgements: We are grateful for the comments we received from anonymous referees.

References

[1] B. Dutertre. The Priority Ceiling Protocol: Formalization and Analysis Using PVS. In *Proc. of the 21st IEEE Conference on Real-Time Systems Symposium (RTSS)*, pages 151–160. IEEE Computer Society, 2000.

[2] J. M. S. Faria. *Formal Development of Solutions for Real-Time Operating Systems with TLA+/TLC*. PhD thesis, University of Porto, 2008.

[3] F. Haftmann and M. Wenzel. Local Theory Specifications in Isabelle/Isar. In *Proc. of the International Conference on Types, Proofs and Programs (TYPES)*, volume 5497 of *LNCS*, pages 153–168, 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] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996.

[7] L. C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *Journal of Computer Security*, 6(1–2):85–128, 1998.

[8] B. Pfaff. PINTOS. http://www.stanford.edu/class/cs140/projects/.

[9] G. E. Reeves. Re: What Really Happened on Mars? *Risks Forum*, 19(54), 1998.

[10] S. Rostedt. *RT-Mutex Implementation Design*. Linux Kernel Distribution at, www.kernel.org/doc/Documentation/rt-mutex-design.txt.

[11] 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.

[12] U. Vahalia. *UNIX Internals: The New Frontiers*. Prentice-Hall, 1996.

[13] 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.

[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.

[16] X. Zhang, C. Urban, and C. Wu. Priority Inheritance Protocol Proved Correct. In *Proc. of the 3rd Conference on Interactive Theorem Proving*, volume 7406 of *LNCS*, pages 217–232, 2012.