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. We also generalise the scheduling problem 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 helps with implementing efficiently 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 used for teaching.

Keywords: Priority Inheritance Protocol, formal correctness 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 the thread itself cannot be blocked indefinitely by threads with lower priority. Alas, in a naive implementation of resource locking and priorities

⋆ This paper is a revised, corrected and expanded version of [31]. Compared with that paper we give an actual implementation of our formalised scheduling algorithm in C and the operating system PINTOS. Our implementation follows closely all results we proved about optimisations of the Priority Inheritance Protocol. We are giving in this paper more details about the proof and also surveying the existing literature in more depth.

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 [12] 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 [21]. On Earth the software run mostly without any problem, but once the spacecraft landed on Mars, it shut down at irregular, but frequent, 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) [24][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.) and ThreadX (another proprietary real-time OS used in nearly all HP inkjet printers [28]), but also the POSIX 1003.1c Standard realised for example in libraries for FreeBSD, Solaris and Linux.

Two advantages of PIP are that it is deterministic and that increasing the priority of a thread can be performed dynamically by the scheduler. This is in contrast to *Priority Ceiling* [24], another solution to the Priority Inversion problem, which requires static analysis of the program in order to prevent Priority Inversion, and also in contrast to the approach taken in the Windows NT scheduler, which avoids this problem by randomly boosting the priority of ready low-priority threads (see for instance [2]). 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 [30]:

> *"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.

[3] Sha et al. call it the *Basic Priority Inheritance Protocol* [24] and others sometimes also call it *Priority Boosting*, *Priority Donation* or *Priority Lending*.

In our opinion, there is clearly a need for investigating correct algorithms for PIP. A few specifications for PIP exist (in informal English) and also a few high-level descriptions of implementations (e.g. in the textbooks [15, Section 12.3.1] and [26, 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 specification of PIP [24], despite being informally "proved" correct, is actually *flawed*.

Yodaiken [30] and also Moylan et al. [16] point to a subtlety that had been overlooked in the informal proof by Sha et al. They specify PIP in [24, Section III] so 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*" [24]. Unfortunately, as Yodaiken and Moylan et al. point out, this behaviour is too simplistic. Moylan et al. write that there are "*some hidden traps*" [16]. 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. A similar error is made in the textbook [20, Section 2.3.1] which specifies for a process that inherited a higher priority and exits a critical section "*it resumes the priority it had at the point of entry into the critical section*". This error can also be found in the textbook [14, Section 16.4.1] where the authors write about this process: "*its priority is immediately lowered to the level originally assigned*"; and also in the more recent textbook [13, Page 119] where the authors state: "*when [the task] exits the critical section that caused the block, it reverts to the priority it had when it entered that section*". The textbook [15, Page 286] contains a simlar flawed specification and even goes on to develop pseudo-code based on this flawed specification. Accordingly, the operating system primitives for inheritance and restoration of priorities in [15] depend on maintaining a data structure called *inheritance log*. This log is maintained for every thread and broadly specified as containing "*[h]istorical information on how the thread inherited its current priority*" [15, Page 527]. Unfortunately, the important information about actually computing the priority to be restored solely from this log is not explained in [15] but left as an "*excercise*" to the reader. As we shall see, a correct version of PIP does not need to maintain this (potentially expensive) data structure at all. Surprisingly also the widely read and frequently updated textbook [25] gives the wrong specification. For example on Page 254 the authors write: "*Upon releasing the lock, the [low-priority] thread will revert to its original priority.*" The same error is also repeated later in this popular textbook.

While [13,14,15,20,24,25] are the only formal publications we have found that specify the incorrect behaviour, it seems also many informal descriptions of PIP overlook the possibility that another high-priority might wait for a low-priority process to finish. A notable exception is the texbook [3], which gives the correct behaviour of resetting the priority of a thread to the highest remaining priority of the threads it blocks. This textbook also gives an informal proof for the correctness of PIP in the style of Sha et al. Unfortunately, this informal proof is too vague to be useful for formalising the correctness of PIP and the specification leaves out nearly all details in order to implement PIP efficiently.

Contributions: There have been earlier formal investigations into PIP [8,10,29], 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. We illustrate this with an implementation of PIP in the educational PINTOS operating system [19]. 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 are also being able to generalise the scheduler of Sha et al. [24] to the practically relevant case where critical sections can overlap; see Figure 1 *a)* below for an example of this restriction. In the existing literature there is no proof and also no proving method that cover this generalised case.

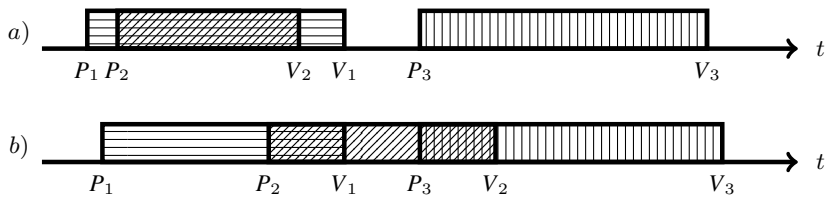

Fig. 1. Assume a process is over time locking and unlocking, say, three resources. The locking requests are labelled P_1, P_2, and P_3 respectively, and the corresponding unlocking operations are labelled V_1, V_2, and V_3. Then graph $a)$ shows *properly nested* critical sections as required by Sha et al. [24] in their proof—the sections must either be contained within each other (the section P_2–V_2 is contained in P_1–V_1) or be independent (P_3–V_3 is independent from the other two). Graph $b)$ shows the general case where the locking and unlocking of different 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.[4] Following good experience in earlier work [27], our model of PIP is based on Paulson's inductive approach for protocol verification [18]. 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:

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

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). For states we define the following type-synonym:

$$\textbf{type\_synonym}\ \mathit{state} = \mathit{event\ list}$$

As in Paulson's work, we need to define functions that allow us to make some observations about states. One function, called *threads*, calculates the set of "live" threads that we have seen so far in a state:

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

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

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

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

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

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

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. An *actor* of an event is defined as

$$\begin{array}{ll} actor\ (Create\ th\ prio) & \stackrel{def}{=} th \\ actor\ (Exit\ th) & \stackrel{def}{=} th \\ actor\ (Set\ th\ pty) & \stackrel{def}{=} th \\ actor\ (P\ th\ cs) & \stackrel{def}{=} th \\ actor\ (V\ th\ cs) & \stackrel{def}{=} th \end{array}$$

This allows us to define what actions a set of threads *ths* might perform in a list of events s, namely

$$actions\_of\ ths\ s \stackrel{def}{=} [e \leftarrow s\ .\ actor\ e \in ths].$$

where we use Isabelle's notation for list-comprehensions. This notation is very similar to notation used in Haskell for list comprehensions. A *precedence* of a thread *th* in a state s is the pair of natural numbers defined as

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

We also use the abbreviation

$$precs\ ths\ s \stackrel{def}{=} \{prec\ th\ s \mid th \in ths\}$$

for the set of precedences of threads *ths* in state 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 threads with the same priority.

Moylan et al. [16] considered the alternative of "time-slicing" threads with equal priority, but found that it does not lead to advantages in practice. On the contrary, according to their work having a policy like our FIFO-scheduling of threads with equal priority reduces the number of tasks involved in the inheritance process and thus minimises the number of potentially expensive thread-switches.

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*.

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

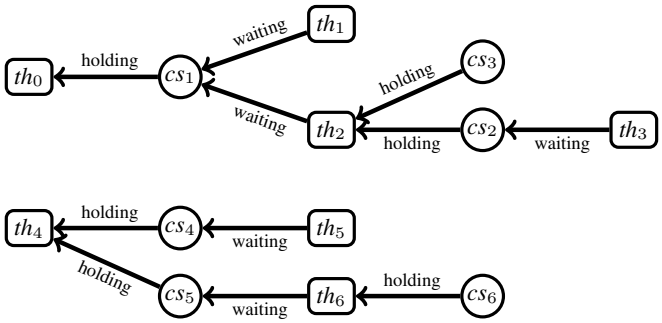

Fig. 2. An instance of a Resource Allocation Graph (RAG).

In this definition we assume *set* converts a list into a set. Note that in the first definition the condition about $th \in set\ (wq\ cs)$ does not follow from $th = hd\ (set\ (wq\ cs))$, since the head of an empty list is undefined in Isabelle/HOL. 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 choose to represent RAGs as relations using pairs of the form

$$(T\ th, C\ cs) \qquad \text{and} \qquad (C\ cs, T\ th) \tag{2}$$

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

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

If there is no cycle, then every RAG can be pictured as a forrest of trees, as for example in Figure 2.

Because of the RAGs, we will need to formalise some results about graphs. While there are few formalisations for graphs already implemented in Isabelle, we choose to introduce our own library of graphs for PIP. The justification for this is that we wanted to be able to reason about potentially infinite graphs (in the sense of infinitely branching and infinite size): the property that our RAGs are actually forrests of finitely branching trees having only an finite depth should be something we can *prove* for our model of PIP—it should not be an assumption we build already into our model. It seemed for our purposes the most convenient represeantation of graphs are binary relations given by sets of pairs shown in (2). The pairs stand for the edges in graphs. This relation-based representation is convenient since we can use the notions of transitive closure operations

$\_^+$ and $\_^*$, as well as relation composition. A *forrest* is defined as the relation *rel* that is *single valued* and *acyclic*:

$$\begin{array}{l} \mathit{single\_valued\ rel} \stackrel{\mathit{def}}{=} \forall x\ y.\ (x, y) \in \mathit{rel} \longrightarrow (\forall z.\ (x, z) \in \mathit{rel} \longrightarrow y = z) \\ \mathit{acyclic\ rel} \stackrel{\mathit{def}}{=} \forall x.\ (x, x) \notin \mathit{rel}^+ \end{array}$$

The *children*, *subtree* and *ancestors* of a node in a graph can be easily defined relationally as

$$\begin{array}{l} \mathit{children\ rel\ node} \stackrel{\mathit{def}}{=} \{y \mid (y, \mathit{node}) \in \mathit{rel}\} \\ \mathit{subtree\ rel\ node} \stackrel{\mathit{def}}{=} \{y \mid (y, \mathit{node}) \in \mathit{rel}^*\} \\ \mathit{ancestors\ rel\ node} \stackrel{\mathit{def}}{=} \{y \mid (\mathit{node}, y) \in \mathit{rel}^+\} \end{array}$$

Note that forrests can have trees with infinte depth and containing nodes with infinitely many children. A *finite forrest* is a forrest which is well-founded and every node has finitely many children (is only finitely branching).

The locking mechanism of PIP ensures that for each thread node, there can be many incoming holding edges in the RAG, but at most one out going waiting edge. The reason is that when a thread asks for resource that is locked already, then the thread 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. So if the *RAG* is well-founded and finite, 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

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

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 to be 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. Our model will enforce that critical resources can only be requested provided no circularity can arise.

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

$$\mathit{cprec\ wq\ s\ th} \stackrel{\mathit{def}}{=} \mathit{Max}\ (\mathit{precs}\ (\{th\} \cup \mathit{dependants\ wq\ th})\ s) \tag{3}$$

where the dependants of *th* are given by the waiting queue function. While the precedence *prec* of any thread is determined statically (for example when the thread is created), the point of the current precedence is to dynamically increase this precedence, if

needed according to PIP. Therefore the current precedence of *th* is given as the maximum of the precedence of *th and* all threads that are dependants of *th* in the state *s*. 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. [24] where a priority of a thread is lowered prematurely (see Introduction). We again introduce an abbreviation for current precedeces of a set of threads, written *cprecs wq s ths*.

$$cprecs\ wq\ s\ ths \stackrel{def}{=} \{cprec\ wq\ s\ th \mid th \in ths\}$$

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 lock, respectively wait for, it); the second is a function that takes a thread and returns its current precedence (see the definition in (3)). 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 shedule 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 requested 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

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

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

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

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

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

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

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

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

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

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 2). The *running* thread is then the thread with the highest current precedence of all ready threads.

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

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 2):

$$resources\ s\ th \stackrel{def}{=} \{cs \mid holds\ s\ th\ cs\}$$
$$detached\ s\ th \stackrel{def}{=} (\nexists\, cs.\ holds\ s\ th\ cs) \wedge (\nexists\, cs.\ waits\ s\ th\ cs)$$

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 *PIP* and *valid_state*. We first give five inference rules for *PIP* relating a state and an event that can happen next.

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

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.

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

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 (the prurpose of the second premise in the rule below). In practice, ensuring the latter is the responsibility of the programmer. In our formal model we brush aside these problematic cases in order to be able to make some meaningful statements about PIP.[5]

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

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 *PIP*.

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

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

Note, however, that apart from the circularity condition, we do not make any assumption on how different resources can be 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 [24] who require that critical sections are properly nested (recall Fig. 1).

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

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

This completes our formal model of PIP. In the next section we present a series of desirable properties derived from our model of PIP. This can be regarded as a validation of the correctness of our model.

3 The Correctness Proof

Sha et al. state their first correctness criterion for PIP in terms of the number of low-priority threads [24, 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 [24, 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 [30] and Moylan et al. [16] 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 (recall the counter example described in the Introduction).

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 threads. We will actually prove a stronger statement where we also provide the current precedence of the blocking thread.

However, the theorem we are going to prove hinges upon a number of natural 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).

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

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 all other reset priorities are either less or equal. Moreover, we assume 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 [9]. 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', then either*

- $th \in running\ (s' @ s)$ *or*
- *there exists a thread th' with $th' \neq th$ and $th' \in running\ (s' @ s)$ such that $th' \in threads\ s$, $\neg\ detached\ s\ th'$ and $cprec\ (s' @ s)\ th' = prec\ th\ s$.*

This theorem ensures that the thread th, which has the highest precedence in the state s, is either running in state $s' @ s$, or can only be blocked in the state $s' @ s$ by a thread th' that already existed in s and is waiting for a resource 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.

Given our assumptions (on th), the first property we show that a running thread th' must either wait for or hold a resource in state s.

Lemma 1. *If* $th' \in running\ (s' @ s)$ *and* $th \neq th'$ *then* $\neg\ detached\ s\ th'$.

Proof. Let us assume th' is detached in state s, then, according to the definition of detached, th does not hold or wait for any resource. Hence the cp-value of th' in s is not boosted, that is $cprec\ s\ th' = prec\ th'\ s$, and is therefore lower than the precedence (as well as the cp-value) of th. This means th' will not run as long as th is a live thread. In turn this means th' cannot take any action in state $s' @ s$ to change its current status; therefore th' is still detached in state $s' @ s$. Consequently th' is also not boosted in state $s' @ s$ and would not run. This contradicts our assumption. □

Proof (of Theorem 1). If $th \in running\ (s' @ s)$, then there is nothing to show. So let us assume otherwise. Since the *RAG* is well-founded, we know there exists an ancestor of th that is the root of the corrsponding subtree and therefore is ready (it does not request any resources). Let us call this thread th'. Since in PIP the $cprec$-value of any thread equals the maximum precedence of all threads in its *RAG*-subtree, and th is in the subtree of th', the $cprec$-value of th' cannot be lower than the precedence of th. But, it can also not be higher, because the precedence of th is the maximum among all threads. Therefore we know that the $cprec$-value of th' is the same as the precedence of th. The result is that th' must be running. This is because $cprec$-value of th' is the highest of all ready threads. This follows from the fact that the $cprec$-value of any ready thread is the maximum of the precedences of all threads in its subtrees (with th having the highest of all threads and being in the subtree of th'). We also have that $th \neq th'$ since we assumed th is not running. By Lem. 1 we have that $\neg\ detached\ s\ th'$. If th' is not detached in s, that is either holding or waiting for a resource, it must be that $th' \in threads\ s$.

This concludes the proof of Theorem 1. □

4 A Finite Bound on Priority Inversion

Like in the work by Sha et al. our result in Thm 1 does not yet guarantee the absence of indefinite Priority Inversion. For this we further need the property that every thread gives up its resources after a finite amount of time. We found that this property is not so straightforward to formalise in our model. There are mainly two reasons for this: First, we do not specify what "running" the code of a thread means, for example by giving an operational semantics for machine instructions. Therefore we cannot characterise what are "good" programs that contain for every locking request for a resource also a corresponding unlocking request. Second, we need to distinghish between a thread that "just" locks a resource for a finite amount of time (even if it is very long) and one that locks it forever (there might be an unbounded loop in between the locking and unlocking requests).

Because of these problems, we decided in our earlier paper [31] to leave out this property and let the programmer take on the responsibility to program threads in such a benign manner (in addition to causing no circularity in the RAG). This leave-it-to-the-programmer was also the approach taken by Sha et al. in their paper. However, in this paper we can make an improvement by establishing a finite bound on the duration

of Priority Inversion measured by the number of events. The events can be seen as a *rough(!)* abstraction of the "runtime behaviour" of threads and also as an abstract notion of "time"—when a new event happens, some time must have passed.

What we will establish in this section is that there can only be a finite number of states after state s in which the thread th is blocked (recall for this that a state is a list of events). For this finiteness bound to exist, Sha et al. informally make two assumtions: first, there is a finite pool of threads (active or hibernating) and second, each of them giving up its resources after a finite amount of time. However, we do not have this concept of active or hibernating threads in our model. In fact we can dispence with the first assumption altogether and allow that in our model we can create new threads or exit existing threads arbitrarily. Consequently, the avoidance of indefinite priority inversion we are trying to establish is in our model not true, unless we stipulate an upper bound on the number of threads that have been created during the time leading to any future state after s. Otherwise our PIP scheduler could be "swamped" with *Create*-requests. So our first assumption states:

> **Assumption on the number of threads created after the state s:** Given the state s, in every "future" valid state $es \,@\, s$, we require that the number of created threads is less than a bound BC, that is
>
> $$len\ (filter\ isCreate\ es) < BC$$
>
> wherby es is a list of events.

Note that it is not enough to just to state that there are only finite number of threads created up until a single state $s' \,@\, s$ after s. Instead, we need to put this bound on the *Create* events for all valid states after s. This ensures that no matter which "future" state is reached, the number of *Create*-events is finite. We use $es \,@\, s$ to stand for *future states* after s—it is s extended with some list es of events.

For our second assumption about giving up resources after a finite amount of "time", let us introduce the following definition about threads that can potentially block th:

$$blockers \stackrel{def}{=} \{th' \mid \neg\ detached\ s\ th' \wedge th' \neq th\}$$

This set contains all treads that are not detached in state s. According to our definiton of *detached*, this means a thread in *blockers* either holds or waits for some resource in state s . Our Them 1 implies that any of those threads can all potentially block th after state s. We need to make the following assumption about the threads in the *blockers*-set:

> **Assumptions on the threads $th' \in blockers$:** For each such th' there exists a finite bound $BND(th')$ such that for all future valid states $es \,@\, s$, we have that if $\neg\ detached\ (es \,@\, s)\ th'$, then
>
> $$len\ (actions\_of\ \{th'\}\ es) < BND(th')$$

By this assumption we enforce that any thread potentially blocking th must become detached (that is lock no resource anymore) after a finite number of events in $es \,@\, s$.

Again we have to state this bound to hold in all valid states after s. The bound reflects how each thread th' is programmed: Though we cannot express what instructions a thread is executing, the events in our model correspond to the system calls made by a thread. Our $BND(th')$ binds the number of these "calls".

The main reason for these two assumptions is that we can prove the following: The number of states after s in which the thread th is not running (that is where Priority Inversion occurs) can be bounded by the number of actions the threads in *blockers* perform (i.e. events) and how many threads are newly created. To state our bound formally, we need to make a definition of what we mean by intermediate states between a state s and a future state after s; they will be the list of states starting from s upto the state es @ s. For example, suppose $es = [e_n, e_{n-1}, \ldots, e_2, e_1]$, then the intermediate states from s upto es @ s are

$$\begin{array}{l} s \\ e_1 :: s \\ e_2 :: e_1 :: s \\ \ldots \\ e_{n-1} :: \ldots :: e_2 :: e_1 :: s \end{array}$$

This list of *intermediate states* can be defined by the following recursive function

$$\begin{array}{lcl} s\ upto\ [] & \stackrel{\text{def}}{=} & [] \\ s\ upto\ (\_::es) & \stackrel{\text{def}}{=} & (es\ @\ s) :: s\ upto\ es \end{array}$$

Our theorem can then be stated as follows:

Theorem 2. *Given our assumptions about bounds, we have that*

$$len\,[s' \leftarrow s\ upto\ es.\ \ th \notin running\ s'] \ \leq\ BC + \sum th' \in blockers.\ \ BND(th')\ .$$

This theorem uses Isabelle's list-comprehension notation, which lists all intermediate states between s and es @ s, and then filters this list according to states in which th is not running. By calculating the number of elements in the filtered list using the function *len*, we have the number of intermediate states in which th is not running and which by the theorem is bounded by the term on the right-hand side.

Proof. There are two characterisations for the number of events in es: First, in each state in $s\ upto\ es$, clearly either th is running or not running. Together with $len\ es = len\ (s\ upto\ es)$, that implies

$$\begin{array}{rcl} len\ es & = & len\,[s' \leftarrow s\ upto\ es.\ \ th \in running\ s'] \\ & & + len\,[s' \leftarrow s\ upto\ es.\ \ th \notin running\ s'] \end{array} \tag{4}$$

Second by Thm 1, the events are either the actions of th or *Create*-events or actions of the threads in blockers. That is

$$\begin{array}{rcl} len\ es & = & len\ (actions\_of\ \{th\}\ es) \\ & & + len\ (filter\ isCreate\ es) \\ & & + len\ (actions\_of\ blockers\ es) \end{array} \tag{5}$$

Furthermore we know that an action of *th* in the intermediate states *s upto es* can only be taken when *th* is running. Therefore

$$len\ (actions\_of\ \{th\}\ es)\ \leq\ len\ [s' \leftarrow s\ upto\ es.\ \ th \in running\ s']$$

holds. Substituting this into (4) gives

$$len\ [s' \leftarrow s\ upto\ es.\ \ th \notin running\ s']\ \leq\ len\ es - len\ (actions\_of\ \{th\}\ es)$$

into which we can substitute (5) yielding

$$\begin{array}{lll} len[s' \leftarrow s\ upto\ es.\ \ th \notin running\ s'] & \leq & len\ (filter\ isCreate\ es) \\ & & +len\ (actions\_of\ blockers\ es) \end{array}$$

By our first assumption we know that the number of *Create*-events are bounded by the bound *BC*. By our second assumption we can prove that the actions of all blockers is bounded by the sum of bounds of the individual blocking threads, that is

$$len\ (actions\_of\ blockers\ es)\ \leq\ \sum th' \in blockers.\ \ BND(th')$$

With this in place we can conclude our theorem. □

This theorem is the main conclusion we obtain for the Priority Inheritance Protocol. It is based on the fact that the set of *blockers* is fixed at state *s* when *th* becomes the thread with highest priority. Then no additional blocker of *th* can appear after the state *s*. And in this way we can bound the number of states where the thread *th* with the highest priority is prevented from running. Our bound does not depend on the restriction of well-nested critical sections in the Priority Inheritance Protocol as imposed by Sha et al.

5 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 Definition shown in (3)). We can however improve upon this. For this let us define the notion of *children* of a thread *th* in a state *s* as

$$??\ children\ r\ x = \{y \mid (y, x) \in r\}$$

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 2. *HERE*

That means the current precedence of a thread *th* can be computed locally by considering only the current precedences of 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 $s \stackrel{def}{=} \mathit{Create}\ th\ \mathit{prio}::s'$ are both valid (meaning the event is allowed to occur). In this situation we can show that

$$\mathit{HERE}\ ??\ \mathit{valid\_trace\_create}\ s\ e\ th\ \mathit{prio} \Longrightarrow \mathit{cprec}\ (e::s)\ th = \mathit{prec}\ th\ (e::s),\ \mathit{and}$$
$$\mathit{If}\ \mathit{valid\_trace\_create}\ s\ e\ th\ \mathit{prio}\ \ \mathit{and}\ \ th' \neq th\ \ \mathit{then}\ \ \mathit{cprec}\ (e::s)\ th' = \mathit{cprec}\ s\ th'.$$

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

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

$$\mathit{HERE}\ \mathit{If}\ \mathit{valid\_trace\_create}\ s\ e\ th\ \mathit{prio}\ \ \mathit{and}\ \ th' \neq th\ \ \mathit{then}\ \ \mathit{cprec}\ (e::s)\ th' = \mathit{cprec}\ s\ th'.$$

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

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

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

For *th* and *th'* we need to use Lemma 2 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.

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

HERE

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

HERE

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 *depend*-edges to recompute using Lemma 2 the *cprec* of every thread encountered on the way. Since the *depend* 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.

HERE

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 [19]. An alternative would have been the small Xv6 operating system used for teaching at MIT

[4,5]. However this operating system implements a simple round robin scheduler that lacks stubs for dealing with priorities. This is inconvenient for our purposes.

To implement PIP in PINTOS, 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 disabling of 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* [17], which provide efficient $O(log\ n)$-operations for accessing and updating. In the code we shall describe below, we use the function `queue_insert`, for inserting a new element into a priority queue, and the function `queue_update`, for updating the position of an element that is already in a queue. Both functions take an extra argument that specifies the comparison function used for organising the priority queue.

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 for the function `lock_acquire`, shown in Figure 3. This function implements the operation of requesting and, if free, locking of a resource by the current running thread. The convention in the PINTOS code is to use the terminology *locks* rather than resources. A lock is represented as a pointer to the structure `lock` (Line 1). Lines 2 to 4 are taken from the original code of `lock_acquire` in PINTOS. They contain diagnostic code: first, there is a check that the lock is a "valid" lock by testing whether it is not `NULL`; second, a 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 from an interrupt; third, it is ensured that 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".

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

```
1  void lock_acquire (struct lock *lock)
2  { ASSERT (lock != NULL);
3    ASSERT (!intr_context());
4    ASSERT (!lock_held_by_current_thread (lock));
5
6    enum intr_level old_level;
7    old_level = intr_disable();
8    if (lock->value == 0) {
9      queue_insert(thread_cprec, &lock->wq, &thread_current()->helem);
10     thread_current()->waiting = lock;
11     struct thread *pt;
12     pt = lock->holder;
13     while (pt) {
14       queue_update(lock_cprec, &pt->held, &lock->helem);
15       if (!(update_cprec(pt)))
16         break;
17       lock = pt->waiting;
18       if (!lock) {
19         queue_update(higher_cprec, &ready_queue, &pt->helem);
20         break;
21       };
22       queue_update(thread_cprec, &lock->wq, &pt->helem);
23       pt = lock->holder;
24     };
25     thread_block();
26   } else {
27     lock->value--;
28     lock->holder = thread_current();
29     queue_insert(lock_prec, &thread_current()->held, &lock->helem);
30   };
31   intr_set_level(old_level);
32 }
```

Fig. 3. Our version of the `lock_acquire` function for the small operating system PINTOS. It implements the operation corresponding to a *P*-event.

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.

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 the RAG). 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 step 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 though that we did *not* verify our C-code. This is in contrast, for example, to the work on seL4, which actually verified in Isabelle/HOL that their C-code satisfies its specification, thought this specification does not contain anything about PIP [11]. Our verification of PIP however provided us with the justification for designing the C-code. It gave us confidence that leaving the "chase" early, whenever there is no change in the calculated current precedence, does not break the correctness of the algorithm.

6 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. [24] 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 [18] 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 [26, 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. [24]). 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 [19], 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. What we are not able to do is to mechanically "synthesise" an actual implementation from our formalisation. To do so for C-code seems quite hard and is beyond current technology available for Isabelle. Also our proof-method based on events is not "computational" in the sense of having a concrete algorithm behind it: our formalisation is really more about the specification of PIP and ensuring that it has the desired properties (the informal specification by Sha et al. did not).

PIP is a scheduling algorithm for single-processor systems. We are now living in a multi-processor world. Priority Inversion certainly occurs also there, see for example [1,6]. 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 [23]. 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.

To us, it seems sound reasoning about scheduling algorithms is fiendishly difficult if done informally by "pencil-and-paper". We infer this from the flawed proof in the paper by Sha et al. [24] and also from [22] where Regehr points out an error in a paper about Preemption Threshold Scheduling [28]. The use of a theorem prover was invaluable to us in order to be confident about the correctness of our reasoning (for example no corner case can be overlooked). The most closely related work to ours is the formal verification in PVS of the Priority Ceiling Protocol done by Dutertre [7]—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 [8,10,29], 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. The code of our formalisation can be downloaded from the Mercurial repository at http://www.dcs.kcl.ac.uk/staff/urbanc/cgi-bin/repos.cgi/pip.

References

1. B. B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.
2. L. Budin and L. Jelenkovic. Time-Constrained Programming in Windows NT Environment. In *Proc. of the IEEE International Symposium on Industrial Electronics (ISIE)*, volume 1, pages 90–94, 1999.
3. G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 3rd edition, 2011.
4. R. Cox, F. Kaashoek, and R. Morris. Xv6. http://pdos.csail.mit.edu/6.828/2012/xv6.html.
5. R. Cox, F. Kaashoek, and R. Morris. Xv6: A Simple, Unix-like Teaching Operating System. Technical report, MIT, 2012.
6. R. I. Davis and A. Burns. A Survey of Hard Real-Time Scheduling for Multiprocessor Systems. *ACM Computing Surveys*, 43(4):35:1–35:44, 2011.
7. 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.
8. J. M. S. Faria. *Formal Development of Solutions for Real-Time Operating Systems with TLA+/TLC*. PhD thesis, University of Porto, 2008.
9. 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.
10. 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.
11. G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. *Communications of the ACM*, 53(6):107–115, 2010.
12. B. W. Lampson and D. D. Redell. Experiences with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):105–117, 1980.
13. P. A. Laplante and S. J. Ovaska. *Real-Time Systems Design and Analysis: Tools for the Practitioner*. Wiley-IEEE Press, 4th edition, 2011.
14. Q. Li and C. Yao. *Real-Time Concepts for Embedded Systems*. CRC Press, 2003.
15. J. W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000.
16. P. J. Moylan, R. E. Betz, and R. H. Middleton. The Priority Disinheritance Problem. Technical Report EE9345, University of Newcastle, 1993.
17. L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996.
18. L. C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *Journal of Computer Security*, 6(1–2):85–128, 1998.
19. B. Pfaff. PINTOS. http://www.stanford.edu/class/cs140/projects/.
20. R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer, 1991.

21. G. E. Reeves. Re: What Really Happened on Mars? *Risks Forum*, 19(54), 1998.
22. J. Regehr. Scheduling Tasks with Mixed Preemption Relations for Robustness to Timing Faults. In *Proc. of the 23rd IEEE Real-Time Systems Symposium (RTSS)*, pages 315–326, 2002.
23. S. Rostedt. *RT-Mutex Implementation Design*. Linux Kernel Distribution at, www.kernel.org/doc/Documentation/rt-mutex-design.txt.
24. 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.
25. A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. John Wiley & Sons, 9th edition, 2013.
26. U. Vahalia. *UNIX Internals: The New Frontiers*. Prentice-Hall, 1996.
27. 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.
28. Y. Wang and M. Saksena. Scheduling Fixed-Priority Tasks with Preemption Threshold. In *Proc. of the 6th Workshop on Real-Time Computing Systems and Applications (RTCSA)*, pages 328–337, 1999.
29. 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.
30. V. Yodaiken. Against Priority Inheritance. Technical report, Finite State Machine Labs (FSMLabs), 2004.
31. X. Zhang, C. Urban, and C. Wu. Priority Inheritance Protocol Proved Correct. In *Proc. of the 3rd Conference on Interactive Theorem Proving (ITP)*, volume 7406 of *LNCS*, pages 217–232, 2012.

Notes

[1] $th \notin running\ (t\ @\ s) \Longrightarrow \exists\, th'.\ T\ th' \in ancestors\ (RAG\ (t\ @\ s))\ (T\ th) \wedge th' \in running\ (t\ @\ s)$ – thm-blockedE??

th_kept shows that th is a thread in s'-s