A Formalisation of Priority Inheritance Protocol for Correct and Efficient Implementation

Xingyuan Zhang[1], Christian Urban[2], and Chunhan Wu[1]

[1] PLA University of Science and Technology, China
[2] King's College London, U.K.

Abstract. Despite the wide use of Priority Inheritance Protocol in real time operating system, it's correctness has never been formally proved and mechanically checked. All existing verification are based on model checking technology. Full automatic verification gives little help to understand why the protocol is correct. And results such obtained only apply to models of limited size. This paper presents a formal verification based on theorem proving. Machine checked formal proof does help to get deeper understanding. We found the fact which is not mentioned in the literature, that the choice of next thread to take over when an critical resource is release does not affect the correctness of the protocol. The paper also shows how formal proof can help to construct correct and efficient implementation.

1 Introduction

Priority inversion referrers to the phenomena where tasks with higher priority are blocked by ones with lower priority. If priority inversion is not controlled, there will be no guarantee the urgent tasks will be processed in time. As reported in [8], priority inversion used to cause software system resets and data lose in JPL's Mars pathfinder project. Therefore, the avoiding, detecting and controlling of priority inversion is a key issue to attain predictability in priority based real-time systems.

The priority inversion phenomenon was first published in [5]. The two protocols widely used to eliminate priority inversion, namely PI (Priority Inheritance) and PCE (Priority Ceiling Emulation), were proposed in [6]. PCE is less convenient to use because it requires static analysis of programs. Therefore, PI is more commonly used in practice[7]. However, as pointed out in the literature, the analysis of priority inheritance protocol is quite subtle[11]. A formal analysis will certainly be helpful for us to understand and correctly implement PI. All existing formal analysis of PI [4,10,3] are based on the model checking technology. Because of the state explosion problem, model check is much like an exhaustive testing of finite models with limited size. The results obtained can not be safely generalized to models with arbitrarily large size. Worse still, since model checking is fully automatic, it give little insight on why the formal model is correct. It is therefore definitely desirable to analyze PI using theorem proving, which gives more general results as well as deeper insight. And this is the purpose of this paper which gives a formal analysis of PI in the interactive theorem prover Isabelle using Higher Order Logic (HOL). The formalization focuses on on two issues:

1. The correctness of the protocol model itself. A series of desirable properties is derived until we are fully convinced that the formal model of PI does eliminate priority inversion. And a better understanding of PI is so obtained in due course. For example, we find through formalization that the choice of next thread to take hold when a resource is released is irrelevant for the very basic property of PI to hold. A point never mentioned in literature.
2. The correctness of the implementation. A series of properties is derived the meaning of which can be used as guidelines on how PI can be implemented efficiently and correctly.

The rest of the paper is organized as follows: Section 2 gives an overview of PI. Section 3 introduces the formal model of PI. Section 4 discusses a series of basic properties of PI. Section 5 shows formally how priority inversion is controlled by PI. Section 6 gives properties which can be used for guidelines of implementation. Section 7 discusses related works. Section 8 concludes the whole paper.

2 An overview of priority inversion and priority inheritance

Priority inversion refers to the phenomenon when a thread with high priority is blocked by a thread with low priority. Priority happens when the high priority thread requests for some critical resource already taken by the low priority thread. Since the high priority thread has to wait for the low priority thread to complete, it is said to be blocked by the low priority thread. Priority inversion might prevent high priority thread from fulfill its task in time if the duration of priority inversion is indefinite and unpredictable. Indefinite priority inversion happens when indefinite number of threads with medium priorities is activated during the period when the high priority thread is blocked by the low priority thread. Although these medium priority threads can not preempt the high priority thread directly, they are able to preempt the low priority threads and cause it to stay in critical section for an indefinite long duration. In this way, the high priority thread may be blocked indefinitely.

Priority inheritance is one protocol proposed to avoid indefinite priority inversion. The basic idea is to let the high priority thread donate its priority to the low priority thread holding the critical resource, so that it will not be preempted by medium priority threads. The thread with highest priority will not be blocked unless it is requesting some critical resource already taken by other threads. Viewed from a different angle, any thread which is able to block the highest priority threads must already hold some critical resource. Further more, it must have hold some critical resource at the moment the highest priority is created, otherwise, it may never get change to run and get hold. Since the number of such resource holding lower priority threads is finite, if every one of them finishes with its own critical section in a definite duration, the duration the highest priority thread is blocked is definite as well. The key to guarantee lower priority threads to finish in definite is to donate them the highest priority. In such cases, the lower priority threads is said to have inherited the highest priority. And this explains the name of the protocol: *Priority Inheritance* and how Priority Inheritance prevents indefinite delay.

The objectives of this paper are:

1. Build the above mentioned idea into formal model and prove a series of properties until we are convinced that the formal model does fulfill the original idea.
2. Show how formally derived properties can be used as guidelines for correct and efficient implementation.

The proof is totally formal in the sense that every detail is reduced to the very first principles of Higher Order Logic. The nature of interactive theorem proving is for the human user to persuade computer program to accept its arguments. A clear and simple understanding of the problem at hand is both a prerequisite and a byproduct of such an effort, because everything has finally be reduced to the very first principle to be checked mechanically. The former intuitive explanation of Priority Inheritance is just such a byproduct.

3 Formal model of Priority Inheritance

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

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

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

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

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

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

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

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

type-synonym *state* = *event list*

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

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

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

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

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

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

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

definition *preced* :: *thread* ⇒ *state* ⇒ *precedence*
where $preced\ thread\ s = Prc\ (original\_priority\ thread\ s)\ (birthtime\ thread\ s)$

A number of important notions are defined here:

consts
holding :: $'b \Rightarrow thread \Rightarrow cs \Rightarrow bool$
waiting :: $'b \Rightarrow thread \Rightarrow cs \Rightarrow bool$
depend :: $'b \Rightarrow (node \times node)\ set$
dependents :: $'b \Rightarrow thread \Rightarrow thread\ set$

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

defs (**overloaded**)
— We define that the thread which is at the head of waiting queue of resource *cs* is holding the resource. This definition is slightly different from tradition where all threads in the waiting queue are considered as waiting for the resource. This notion is reflected in the definition of *holding wq th cs* as follows:
cs_holding_def:
$holding\ wq\ thread\ cs \stackrel{def}{=} (thread \in set\ (wq\ cs) \wedge thread = hd\ (wq\ cs))$
— In accordance with the definition of *holding wq th cs*, a thread *th* is considered waiting for *cs* if it is in the *waiting queue* of critical resource *cs*, but not at the head. This is reflected in the definition of *waiting wq th cs* as follows:
cs_waiting_def:
$waiting\ wq\ thread\ cs \stackrel{def}{=} (thread \in set\ (wq\ cs) \wedge thread \neq hd\ (wq\ cs))$
— *depend wq* represents the Resource Allocation Graph of the system under the waiting queue function *wq*.
cs_depend_def:
$depend\ (wq{::}cs \Rightarrow thread\ list) \stackrel{def}{=}$
$\{(Th\ t, Cs\ c) \mid t\ c.\ waiting\ wq\ t\ c\} \cup \{(Cs\ c, Th\ t) \mid c\ t.\ holding\ wq\ t\ c\}$
— The following *dependents wq th* represents the set of threads which are depending on thread *th* in Resource Allocation Graph *depend wq*:
cs_dependents_def:
$dependents\ (wq{::}cs \Rightarrow thread\ list)\ th \stackrel{def}{=} \{th'\ .\ (Th\ th', Th\ th) \in (depend\ wq)\hat{}{+}\}$

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

record *schedule_state* =
waiting_queue :: $cs \Rightarrow thread\ list$ — The function assigning waiting queue.
cur_preced :: $thread \Rightarrow precedence$ — The function assigning precedence.

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

definition *cpreced* :: $state \Rightarrow (cs \Rightarrow thread\ list) \Rightarrow thread \Rightarrow precedence$

where $cpreced\ s\ wq = (\lambda\ th.\ Max\ ((\lambda\ th.\ preced\ th\ s)\ `\ (\{th\} \cup dependents\ wq\ th)))$

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

fun $schs :: state \Rightarrow schedule\_state$
where $schs\ [] = (\!|waiting\_queue = \lambda\ cs.\ [],\ cur\_preced = cpreced\ []\ (\lambda\ cs.\ [])|\!)\ |$

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

—

$schs\ (e\#s) = (let\ ps = schs\ s\ in$
$\quad let\ pwq = waiting\_queue\ ps\ in$
$\quad let\ pcp = cur\_preced\ ps\ in$
$\quad let\ nwq = case\ e\ of$
$\qquad P\ thread\ cs \Rightarrow\ pwq(cs{:=}(pwq\ cs\ @\ [thread]))\ |$
$\qquad V\ thread\ cs \Rightarrow let\ nq = case\ (pwq\ cs)\ of$
$\qquad\qquad [] \Rightarrow []\ |$
$\qquad\qquad (th'\#qs) \Rightarrow (SOME\ q.\ distinct\ q \wedge set\ q = set\ qs)$
$\qquad\quad in\ pwq(cs{:=}nq)\qquad |$
$\qquad \_ \Rightarrow pwq$
$\quad in\ let\ ncp = cpreced\ (e\#s)\ nwq\ in$
$\quad (\!|waiting\_queue = nwq,\ cur\_preced = ncp|\!)$
$\quad)$

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

definition $wq :: state \Rightarrow cs \Rightarrow thread\ list$
where $wq\ s = waiting\_queue\ (schs\ s)$

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

definition $cp :: state \Rightarrow thread \Rightarrow precedence$
where $cp\ s = cur\_preced\ (schs\ s)$

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

defs (**overloaded**)
s_holding_def:
$holding\ (s{::}state)\ thread\ cs \stackrel{def}{=} (thread \in set\ (wq\ s\ cs) \wedge thread = hd\ (wq\ s\ cs))$
s_waiting_def:
$waiting\ (s{::}state)\ thread\ cs \stackrel{def}{=} (thread \in set\ (wq\ s\ cs) \wedge thread \neq hd\ (wq\ s\ cs))$
s_depend_def:
$depend\ (s{::}state) \stackrel{def}{=}$
$\{(Th\ t, Cs\ c) \mid t\ c.\ waiting\ (wq\ s)\ t\ c\} \cup \{(Cs\ c, Th\ t) \mid c\ t.\ holding\ (wq\ s)\ t\ c\}$
s_dependents_def:
$dependents\ (s{::}state)\ th \stackrel{def}{=} \{th'\ .\ (Th\ th', Th\ th) \in (depend\ (wq\ s))\hat{}+\}$

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

definition $readys :: state \Rightarrow thread\ set$
where $readys\ s = \{thread\ .\ thread \in threads\ s \wedge (\forall\ cs.\ \neg\ waiting\ s\ thread\ cs)\}$

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

definition $runing :: state \Rightarrow thread\ set$
where $runing\ s = \{th\ .\ th \in readys\ s \wedge cp\ s\ th = Max\ ((cp\ s)\ `\ (readys\ s))\}$

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

definition $holdents :: state \Rightarrow thread \Rightarrow cs\ set$
where $holdents\ s\ th = \{cs\ .\ (Cs\ cs, Th\ th) \in depend\ s\}$

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

definition $cntCS :: state \Rightarrow thread \Rightarrow nat$
where $cntCS\ s\ th = card\ (holdents\ s\ th)$

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

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

thread_P: ⟦*thread* ∈ *runing s*; (*Cs cs*, *Th thread*) ∉ (*depend s*)ˆ+⟧ ⟹ *step s* (*P thread cs*) |
— A thread can release a critical resource *cs* if it is running and holding that resource:
thread_V: ⟦*thread* ∈ *runing s*; *holding s thread cs*⟧ ⟹ *step s* (*V thread cs*) |
— A thread can adjust its own priority as long as it is current running:
thread_set: ⟦*thread* ∈ *runing s*⟧ ⟹ *step s* (*Set thread prio*)

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

inductive *vt* :: (*state* ⇒ *event* ⇒ *bool*) ⇒ *state* ⇒ *bool*
for *cs* — *cs* is an argument representing any step predicate.
where
— Empty list [] is a legal state in any protocol:
vt_nil[*intro*]: *vt cs* [] |
— If *s* a legal state, and event *e* is eligible to happen in state *s*, then *e#s* is a legal state as well:
vt_cons[*intro*]: ⟦*vt cs s*; *cs s e*⟧ ⟹ *vt cs* (*e#s*)

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

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

fun *the_cs* :: *node* ⇒ *cs*
where *the_cs* (*Cs cs*) = *cs*

fun *the_th* :: *node* ⇒ *thread*
where *the_th* (*Th th*) = *th*

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

definition *next_th*:: *state* ⇒ *thread* ⇒ *cs* ⇒ *thread* ⇒ *bool*
where *next_th s th cs t* = (∃ *rest*. *wq s cs* = *th#rest* ∧ *rest* ≠ [] ∧ *t* = *hd* (*SOME q*. *distinct q* ∧ *set q* = *set rest*))

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

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

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

definition *cntP* :: *state* ⇒ *thread* ⇒ *nat*
where *cntP s th* = *count* (λ *e*. ∃ *cs*. *e* = *P th cs*) *s*

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

definition $cntV :: state \Rightarrow thread \Rightarrow nat$
where $cntV\ s\ th = count\ (\lambda\ e.\ \exists\ cs.\ e = V\ th\ cs)\ s$

4 General properties of Priority Inheritance

5 Key properties

6 Properties to guide implementation

7 Related works

1. *Integrating Priority Inheritance Algorithms in the Real-Time Specification for Java* [10] models and verifies the combination of Priority Inheritance (PI) and Priority Ceiling Emulation (PCE) protocols in the setting of Java virtual machine using extended Timed Automata(TA) formalism of the UPPAAL tool. Although a detailed formal model of combined PI and PCE is given, the number of properties is quite small and the focus is put on the harmonious working of PI and PCE. Most key features of PI (as well as PCE) are not shown. Because of the limitation of the model checking technique used there, properties are shown only for a small number of scenarios. Therefore, the verification does not show the correctness of the formal model itself in a convincing way.
2. *Formal Development of Solutions for Real-Time Operating Systems with TLA+/TLC* [3]. A formal model of PI is given in TLA+. Only 3 properties are shown for PI using model checking. The limitation of model checking is intrinsic to the work.
3. *Synchronous modeling and validation of priority inheritance schedulers* [4]. Gives a formal model of PI and PCE in AADL (Architecture Analysis & Design Language) and checked several properties using model checking. The number of properties shown there is less than here and the scale is also limited by the model checking technique.
4. *The Priority Ceiling Protocol: Formalization and Analysis Using PVS* [2]. Formalized another protocol for Priority Inversion in the interactive theorem proving system PVS.

There are several works on inversion avoidance:

1. *Solving the group priority inversion problem in a timed asynchronous system* [9]. The notion of Group Priority Inversion is introduced. The main strategy is still inversion avoidance. The method is by reordering requests in the setting of Client-Server.
2. *A Formalization of Priority Inversion* [1]. Formalized the notion of Priority Inversion and proposes methods to avoid it.

Examples of inaccurate specification of the protocol ???.

8 Conclusions

References

1. Ö Babaoglu, K. Marzullo, and F. B. Schneider. A formalization of priority inversion. *Real-Time Systems*, 5(4):285–303, 1993.
2. B. Dutertre. The Priority Ceiling Protocol: Formalization and analysis using PVS. Technical report, System Design Laboratory, SRI International, Menlo Park, CA, October 1999. Available at http://www.sdl.sri.com/dsa/publis/prio-ceiling.html.
3. J. M. S. Faria. Formal development of solutions for real-time operating systems with tla+/tlc. http://repositorio-aberto.up.pt/bitstream/10216/11466/2/Texto%20integral.pdf, 2008.
4. E. Jahier, B. Halbwachs, and P. Raymond. Synchronous modeling and validation of priority inheritance schedulers. In Marsha Chechik and Martin Wirsing, editors, *FASE*, volume 5503 of *Lecture Notes in Computer Science*, pages 140–154. Springer, 2009.
5. B. Lampson and D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.
6. S. Liu, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers*, 39(9):1175–1185, 1990.
7. D. Locke. Priority inheritance: The real story. http://www.math.unipd.it/~tullio/SCD/2007/Materiale/Locke.pdf, 2002.
8. G. Reeves. Re: What really happened on mars? *Risks-Forum Digest*, 19(58), January 1998.
9. Y. Wang, E. Anceaume, F. Brasileiro, F. Greve, and M. Hurfin. Solving the group priority inversion problem in a timed asynchronous system. *IEEE Transactions on Computers*, 51(8):900–915, August 2002.
10. A. J. Wellings, A. Burns, O. M. Santos, and B. M. Brosgol. Integrating priority inheritance algorithms in the real-time specification for java. In *Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 115–123. IEEE Computer Society, 2007.
11. V. Yodaiken. Against priority inheritance. http://www.linuxfordevices.com/files/misc/yodaiken-july02.pdf, 2002.