A Provably Correct Implementation of the Priority Inheritance Protocol

Christian Urban

joint work with Xingyuan Zhang and Chunhan Wu

from the PLA University of Science and Technology in Nanjing

Isabelle Theorem Prover

My background:

- mechanical reasoning about languages with binders (Nominal)
- Barendregt's variable convention can lead to false
- found a bug in a proof by Bob Harper and Frank Pfenning (CMU) on LF (ACM TOCL, 2005)

Real-Time OSes

- Processes have priorities
- Resources can be locked and unlocked

Problem

High-priority process

Low-priority process

Problem

High-priority process

Medium-priority process

Low-priority process

Problem

High-priority process

Medium-priority process

Low-priority process

- Priority Inversion $\stackrel{\text{def}}{=}$ $H < L$

Problem

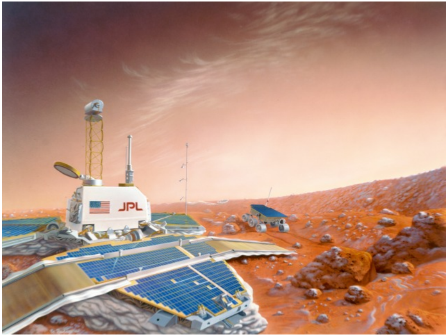
High-priority process

Medium-priority process

Low-priority process

- Priority Inversion $\overset{\text{def}}{=}$ $H < L$
- avoid indefinite priority inversion

Mars Pathfinder Mission 1997



Solution

Priority Inheritance Protocol (PIP):

High-priority process

Medium-priority process

Low-priority process

(temporarily raise its priority)

"Priority inheritance is neither efficient nor reliable. Implementations are either incomplete (and unreliable) or surprisingly complex and intrusive."

"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."

A Correctness “Proof” in 1990

- a paper* in 1990 “proved” the correctness of an algorithm for PIP

...after the thread (whose priority has been raised) completes its critical section and releases the lock, it “returns to its original priority level”.

* in IEEE Transactions on Computers

High-priority process 1

High-priority process 2

Low-priority process

High-priority process 1

High-priority process 2

Low-priority process

- Solution:
  Return to highest remaining priority

Events

Create thread priority
Exit thread
Set thread priority
Lock thread cs
Unlock thread cs

Events

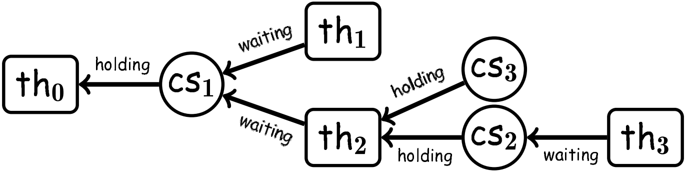
Create thread priority
Exit thread
Set thread priority
Lock thread cs
Unlock thread cs

A state is a list of events (that happened so far).

Precedences

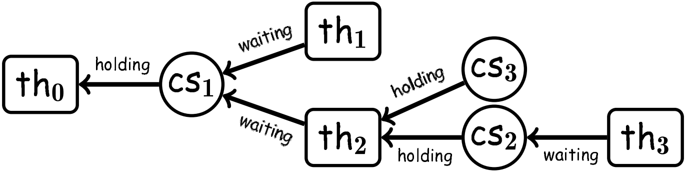
$$\text{prec th s} \stackrel{\text{def}}{=} (\text{priority th s}, \text{last\_set th s})$$

RAGs



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

RAGs



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

Good Next Events

$$\frac{\text{th} \notin \text{threads } s}{\text{step } s \text{ (Create th prio)}}$$

$$\frac{\text{th} \in \text{running } s \qquad \text{resources } s \text{ th} = \varnothing}{\text{step } s \text{ (Exit th)}}$$

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

Good Next Events

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

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

Theorem

"No indefinite priority inversion"

Theorem:* If th is the thread with the highest precedence in state s, then in every future state s′ in which th is still alive

- th is blocked by a thread th′ that was alive in s
- th′ held a resource in s, and
- th′ is running with the precedence of th.

* modulo some further assumptions

Theorem

"No indefinite priority inversion"

Theorem:* If th is the thread with the highest precedence in state s, then in every future state s′ in which th is still alive

- th is blocked by a thread th′ that was alive in s
- th′ held a resource in s, and
- th′ is running with the precedence of th.

* modulo some further assumptions

It does not matter which process gets a released lock.

Implementation

s = current state; s' = next state

Create th prio, Exit th

- RAG s' = RAG s
- precedences of descendants stay all the same

Implementation

s = current state; s' = next state

Set th prio

- RAG s' = RAG s
- we have to recalculate the precedence of the direct descendants

Implementation

s = current state; s' = next state

Unlock th cs where there is a thread to take over

- RAG s' = RAG s - {(C cs, T th), (T th', C cs)} ∪ {(C cs, T th')}
- we have to recalculate the precedence of the direct descendants

Unlock th cs where no thread takes over

- RAG s' = RAG s - {(C cs, T th)}
- no recalculation of precedences

Implementation

s = current state; s' = next state

Lock th cs where cs is not locked

- RAG s' = RAG $s \cup \{(C\ cs, T\ th')\}$
- no recalculation of precedences

Lock th cs where cs is locked

- RAG s' = RAG s - $\{(T\ th, C\ cs)\}$
- we have to recalculate the precedence of the descendants

PINTOS

- . . . small operating system developed at Stanford for teaching; written in C

Event	PINTOS function
Create	thread_create
Exit	thread_exit
Set	thread_set_priority
Lock	lock_acquire
Unlock	lock_release

Conclusion

- surprised how pleasant the experience was
- no real specification existed for PIP
- general technique (a "hammer"):
  events, separation of good and bad configurations
- scheduler in RT-Linux
- multiprocessor case
- other "nails" ? (networks, . . .)