# Priority inheritance in the kernel

[Posted April 3, 2006 by corbet]

Imagine a system with two processes running, one at high priority and the other at a much lower priority. These processes share resources which are protected by locks. At some point, the low-priority process manages to run and obtains a lock for one of those resources. If the high-priority process then attempts to obtain the same lock, it will have to wait. Essentially, the low-priority process has trumped the high-priority process, at least for as long as it holds the contended lock.

Now imagine a third process, one which uses a lot of processor time, and which has a priority between the other two. If that process starts to crank, it will push the low-priority process out of the CPU indefinitely. As a result, the third process can keep the highest-priority process out of the CPU indefinitely.

This situation, called "priority inversion," tends to be followed by system failure, upset users, and unemployed engineers. There are a number of approaches to avoiding priority inversion, including lockless designs, carefully thought-out locking scenarios, and a technique known as priority inheritance. The priority inheritance method is simple in concept: when a process holds a lock, it should run at (at least) the priority of the highest-priority process waiting for the lock. When a lock is taken by a low-priority process, the priority of that process might need to be boosted until the lock is released.

There are a number of approaches to priority inheritance. In effect, the kernel performs a very simple form of it by not allowing kernel code to be preempted while holding a spinlock. In some systems, each lock has a priority associated with it; whenever a process takes a lock, its priority is raised to the lock's priority. In others, a high-priority process will have its priority "inherited" by another process which holds a needed lock. Most priority inheritance schemes have shown a tendency to complicate and slow down the locking code, and they can be used to paper over poor application designs. So they are unpopular in many circles. Linus was reasonably clear about how he felt on the subject last December:

> "Friends don't let friends use priority inheritance".
>
> Just don't do it. If you really need it, your system is broken anyway.

Faced with this sort of opposition, many developers would quietly shelve their priority inheritance designs and go back to working on accounting code. The kernel development community, however, happens to have a member who has a track record of getting code merged in spite of this sort of objection: Ingo Molnar. History may well repeat itself, as Ingo (working with Thomas Gleixner) has posted a priority-inheriting futex implementation with a request that it be merged into the mainline. This approach, says Ingo, provides a useful functionality to user space (it is not meant to provide priority-inheriting kernel mutual exclusion primitives) while avoiding the pitfalls which have hit other implementations.

The PI-futex patch adds a couple of new operations to the `futex()` system call: `FUTEX_LOCK_PI` and `FUTEX_UNLOCK_PI`. In the uncontended case, a PI-futex can be taken without involving the kernel at all, just like an ordinary futex. When there is contention, instead, the `FUTEX_LOCK_PI` operation is requested from the kernel. The requesting process is put into a special queue, and, if necessary, that process lends its priority to the process actually holding the contended futex. The priority inheritance is chained, so that, if the holding process is blocked on a second futex, the boosted priority will propagate to the holder of that second futex. As soon as a futex is released, any associated priority boost is removed.

As with regular futexes, the kernel only needs to know about a PI-futex while it is being contended. So the number of futexes in the system can become quite large without serious overhead on the kernel side.

Within the kernel, the PI-futex type is implemented by way of a new primitive called an `rt_mutex`. The `rt_mutex` is superficially similar to regular mutexes, with the addition of the priority inheritance capability. They are, however, an entirely different type, with no code shared with the mutex implementation. The API

will be familiar to mutex users, however; in brief, it is:

```
#include <linux/rtmutex.h>

void rt_mutex_init(struct rt_mutex *lock);
void rt_mutex_destroy(struct rt_mutex *lock);

void rt_mutex_lock(struct rt_mutex *lock);
int rt_mutex_lock_interruptible(struct rt_mutex *lock,
                                int detect_deadlock);
int rt_mutex_timed_lock(struct rt_mutex *lock,
                        struct hrtimer_sleeper *timeout,
                        int detect_deadlock);
int rt_mutex_trylock(struct rt_mutex *lock);
void rt_mutex_unlock(struct rt_mutex *lock);
int rt_mutex_is_locked(struct rt_mutex *lock);
```

The alert reader may have noticed that this looks much like the realtime mutex type found in the realtime preemption patch. Ingo once said that the realtime patches would slowly trickle into the mainline, and that is what appears to be happening here. With this patch set, the PI-futex code is the only user of the new `rt_mutex` type, but that could certainly change over time.

The PI-futex patch also includes a new, priority-sorted list type which could find users elsewhere in the kernel.

There has been relatively little discussion of this patch so far; it has been included in recent -mm trees. It is too late for 2.6.17, but, if no real opposition develops, the PI-futex code might just find its way into a subsequent kernel.

**Index entries for this article**

Kernel Futex
Kernel Locking mechanisms/Mutexes
Kernel Priority inheritance
Kernel Realtime

---

(Log in to post comments)

## Priority inheritance in the kernel

Posted Apr 6, 2006 4:48 UTC (Thu) by **kirkengaard** (guest, #15022) [Link]

"Faced with this sort of opposition, many developers would quietly shelve their priority inheritance designs and go back to working on accounting code. The kernel development community, however, happens to have a member who has a track record of getting code merged in spite of this sort of objection: Ingo Molnar."

This is generally because Ingo writes sane patches and doesn't irritate people needlessly. "Plays well with others." His process with the realtime patches especially has been very open and release-early-and-often, he seems to work with anyone who has a problem in the patch, he gives good advice and accepts help graciously.

And, this stuff is coming from a patch people actually use, and that meets a defined need. It isn't "Oh, I think I'll introduce PI into the kernel today." Makes it easier to ack the patches that come from it.

In short, it's hard not to like him, and it's hard to fault his patches.

Reply to this comment

## Priority inheritance in the kernel

Posted Apr 6, 2006 11:02 UTC (Thu) by **nix** (subscriber, #2304) [Link]

He also describes the patches very, very well.

(Compare to, for instance. H. J. Lu, who also writes good patches much of the time but rarely actually describes what they do, and often what they do is thoroughly unobvious...)

Reply to this comment

## Priority inheritance in the kernel

Posted Apr 18, 2006 17:22 UTC (Tue) by **efexis** (guest, #26355) [Link]

But it doesn't support priority inversion, it supports code that could /otherwise/ result in priority inversion. I don't think removing the change of something occuring quite constitutes as "supporting" :-p

Reply to this comment

## Priority inversion caused problems for Mars pathfinder

Posted Apr 6, 2006 14:32 UTC (Thu) by **dwheeler** (guest, #1216) [Link]

The Mars Pathfinder had problems due to priority inversion. Frankly, it seems like a good idea to include support for priority inversion in the kernel. It's very hard to be SURE that priority inversion can't happen, and in some circumstances if the problem occurs you are in deep trouble. At the least, support for them in user space sounds like a good thing.

Reply to this comment

## Priority inversion caused problems for Mars pathfinder

Posted Apr 6, 2006 19:12 UTC (Thu) by **Los__D** (guest, #15263) [Link]

"Frankly, it seems like a good idea to include support for priority inversion in the kernel."

Hmmm.. I'm sure that's not what you really meant, eh? I'd quite prefer priority inversion avoided, using priority inheritance... ;p

Reply to this comment

## priority inversion support

Posted Apr 8, 2006 0:48 UTC (Sat) by **giraffedata** (guest, #1954) [Link]

The posting uses the word "support," which can mean lots of different things. To some, "support for priority inversion" would mean code that effects priority version. For others, it means code that recognizes and/or handles priority inversion.

I always encourage people to avoid the term "support" altogether, because there is always a plainer, more informative word you can use. In this case, I would say "handling of priority inversion" or "priority inheritance function."

Remember that support *really* means to hold up or assist, and priority inversion doesn't require any support; it does fine all by itself.

Reply to this comment