**PI-futex: -V1**

From <https://lwn.net/Articles/177111/>

|  |  |  |
| --- | --- | --- |
| **From**: |  | Ingo Molnar <mingo@elte.hu> |
| **To**: |  | linux-kernel@vger.kernel.org |
| **Subject**: |  | [patch 00/10] PI-futex: -V1 |
| **Date**: |  | Sat, 25 Mar 2006 19:45:28 +0100 |
| **Cc**: |  | Thomas Gleixner <tglx@linutronix.de>, Linus Torvalds <torvalds@osdl.org>, Andrew Morton <akpm@osdl.org>, Arjan van de Ven <arjan@infradead.org> |
| **Archive-link**: |  | [Article](http://article.gmane.org/gmane.linux.kernel/393224), [Thread](http://thread.gmane.org/gmane.linux.kernel/393224) |

We are pleased to announce "lightweight userspace priority inheritance"

(PI) support for futexes. The following patchset and glibc patch

implements it, ontop of the robust-futexes patchset which is included in

2.6.16-mm1.

We are calling it lightweight for 3 reasons:

- in the user-space fastpath a PI-enabled futex involves no kernel work

(or any other PI complexity) at all. No registration, no extra kernel

calls - just pure fast atomic ops in userspace.

- in the slowpath (in the lock-contention case), the system call and

scheduling pattern is in fact better than that of normal futexes,

due to the 'integrated' nature of FUTEX\_LOCK\_PI. [more about that

further down]

- the in-kernel PI implementation is streamlined around the mutex

abstraction, with strict rules that keep the implementation

relatively simple: only a single owner may own a lock (i.e. no

read-write lock support), only the owner may unlock a lock, no

recursive locking, etc.

Priority Inheritance - why, oh why???

-------------------------------------

Many of you heard the horror stories about the evil PI code circling

Linux for years, which makes no real sense at all and is only used by

buggy applications and which has horrible overhead. Some of you have

dreaded this very moment, when someone actually submits working PI code

;-)

So why would we like to see PI support for futexes?

We'd like to see it done purely for technological reasons. We dont think

it's a buggy concept, we think it's useful functionality to offer to

applications, which functionality cannot be achieved in other ways. We

also think it's the right thing to do, and we think we've got the right

arguments and the right numbers to prove that. We also believe that we

can address all the counter-arguments as well. For these reasons (and

the reasons outlined below) we are submitting this patch-set for

upstream kernel inclusion.

What are the benefits of PI?

The short reply:

----------------

User-space PI helps achieving/improving determinism for user-space

applications. In the best-case, it can help achieve determinism and

well-bound latencies. Even in the worst-case, PI will improve the

statistical distribution of locking related application delays.

The longer reply:

-----------------

Firstly, sharing locks between multiple tasks is a common programming

technique that often cannot be replaced with lockless algorithms. As we

can see it in the kernel [which is a quite complex program in itself],

lockless structures are rather the exception than the norm - the current

ratio of lockless vs. locky code for shared data structures is somewhere

between 1:10 and 1:100. Lockless is hard, and the complexity of lockless

algorithms often endangers to ability to do robust reviews of said code.

I.e. critical RT apps often choose lock structures to protect critical

data structures, instead of lockless algorithms. Furthermore, there are

cases (like shared hardware, or other resource limits) where lockless

access is mathematically impossible.

Media players (such as Jack) are an example of reasonable application

design with multiple tasks (with multiple priority levels) sharing

short-held locks: for example, a highprio audio playback thread is

combined with medium-prio construct-audio-data threads and low-prio

display-colory-stuff threads. Add video and decoding to the mix and

we've got even more priority levels.

So once we accept that synchronization objects (locks) are an

unavoidable fact of life, and once we accept that multi-task userspace

apps have a very fair expectation of being able to use locks, we've got

to think about how to offer the option of a deterministic locking

implementation to user-space.

Most of the technical counter-arguments against doing priority

inheritance only apply to kernel-space locks. But user-space locks are

different, there we cannot disable interrupts or make the task

non-preemptible in a critical section, so the 'use spinlocks' argument

does not apply (user-space spinlocks have the same priority inversion

problems as other user-space locking constructs). Fact is, pretty much

the only technique that currently enables good determinism for userspace

locks (such as futex-based pthread mutexes) is priority inheritance:

Currently (without PI), if a high-prio and a low-prio task shares a lock

[this is a quite common scenario for most non-trivial RT applications],

even if all critical sections are coded carefully to be deterministic

(i.e. all critical sections are short in duration and only execute a

limited number of instructions), the kernel cannot guarantee any

deterministic execution of the high-prio task: any medium-priority task

could preempt the low-prio task while it holds the shared lock and

executes the critical section, and could delay it indefinitely.

Implementation:

---------------

As mentioned before, the userspace fastpath of PI-enabled pthread

mutexes involves no kernel work at all - they behave quite similarly to

normal futex-based locks: a 0 value means unlocked, and a value==TID

means locked. (This is the same method as used by list-based robust

futexes.) Userspace uses atomic ops to lock/unlock these mutexes without

entering the kernel.

To handle the slowpath, we have added two new futex ops:

FUTEX\_LOCK\_PI

FUTEX\_UNLOCK\_PI

If the lock-acquire fastpath fails, [i.e. an atomic transition from 0 to

TID fails], then FUTEX\_LOCK\_PI is called. The kernel does all the

remaining work: if there is no futex-queue attached to the futex address

yet then the code looks up the task that owns the futex [it has put its

own TID into the futex value], and attaches a 'PI state' structure to

the futex-queue. The pi\_state includes an rt-mutex, which is a PI-aware,

kernel-based synchronization object. The 'other' task is made the owner

of the rt-mutex, and the FUTEX\_WAITERS bit is atomically set in the

futex value. Then this task tries to lock the rt-mutex, on which it

blocks. Once it returns, it has the mutex acquired, and it sets the

futex value to its own TID and returns. Userspace has no other work to

perform - it now owns the lock, and futex value contains

FUTEX\_WAITERS|TID.

If the unlock side fastpath succeeds, [i.e. userspace manages to do a

TID -> 0 atomic transition of the futex value], then no kernel work is

triggered.

If the unlock fastpath fails (because the FUTEX\_WAITERS bit is set),

then FUTEX\_UNLOCK\_PI is called, and the kernel unlocks the futex on the

behalf of userspace - and it also unlocks the attached

pi\_state->rt\_mutex and thus wakes up any potential waiters.

Note that under this approach, contrary to other PI-futex approaches,

there is no prior 'registration' of a PI-futex. [which is not quite

possible anyway, due to existing ABI properties of pthread mutexes.]

Also, under this scheme, 'robustness' and 'PI' are two orthogonal

properties of futexes, and all four combinations are possible: futex,

robust-futex, PI-futex, robust+PI-futex.

glibc support:

--------------

Ulrich Drepper and Jakub Jelinek have written glibc support for

PI-futexes (and robust futexes), enabling robust and PI

(PTHREAD\_PRIO\_INHERIT) POSIX mutexes. (PTHREAD\_PRIO\_PROTECT support will

be added later on too, no additional kernel changes are needed for

that). [NOTE: The glibc patch is obviously inofficial and unsupported

without matching upstream kernel functionality.]

the patch-queue and the glibc patch can also be downloaded from:

<http://redhat.com/~mingo/PI-futex-patches/>

a diffstat is attached below. The patch-queue is against 2.6.16-mm1,

plus the following small updates to -mm1:

lightweight-robust-futexes-updates.patch

lightweight-robust-futexes-updates-2.patch

itimer-validate-uservalue.patch

hrtimer-generic-sleeper.patch

futex-timeval-check.patch

all have been sent to Andrew and are independent of PI-futexes.

many thanks go to the people who helped us create this kernel feature:

Steven Rostedt, Esben Nielsen, Benedikt Spranger, Daniel Walker, John

Cooper, Arjan van de Ven, Oleg Nesterov and others. Credits for related

prior projects goes to Dirk Grambow, Inaky Perez-Gonzalez, Bill Huey and

many others.

Ingo Molnar, Thomas Gleixner

--

Documentation/rtmutex.txt | 60 +

arch/i386/mm/pageattr.c | 4

include/linux/futex.h | 11

include/linux/init\_task.h | 2

include/linux/mm.h | 11

include/linux/plist.h | 226 ++++++

include/linux/rtmutex.h | 119 +++

include/linux/rtmutex\_internal.h | 187 +++++

include/linux/sched.h | 34

include/linux/syscalls.h | 4

init/Kconfig | 5

kernel/Makefile | 3

kernel/exit.c | 9

kernel/fork.c | 6

kernel/futex.c | 929 +++++++++++++++++++++----

kernel/futex\_compat.c | 11

kernel/rtmutex-debug.c | 511 +++++++++++++

kernel/rtmutex-debug.h | 32

kernel/rtmutex-tester.c | 436 +++++++++++

kernel/rtmutex.c | 997 +++++++++++++++++++++++++++

kernel/rtmutex.h | 28

kernel/sched.c | 136 +++

lib/Kconfig | 6

lib/Kconfig.debug | 20

lib/Makefile | 1

lib/plist.c | 72 +

mm/page\_alloc.c | 4

mm/slab.c | 3

scripts/rt-tester/check-all.sh | 21

scripts/rt-tester/rt-tester.py | 222 ++++++

scripts/rt-tester/t2-l1-2rt-sameprio.tst | 101 ++

scripts/rt-tester/t2-l1-pi.tst | 84 ++

scripts/rt-tester/t2-l1-signal.tst | 79 ++

scripts/rt-tester/t2-l2-2rt-deadlock.tst | 91 ++

scripts/rt-tester/t3-l1-pi-1rt.tst | 95 ++

scripts/rt-tester/t3-l1-pi-2rt.tst | 96 ++

scripts/rt-tester/t3-l1-pi-3rt.tst | 95 ++

scripts/rt-tester/t3-l1-pi-signal.tst | 98 ++

scripts/rt-tester/t3-l1-pi-steal.tst | 99 ++

scripts/rt-tester/t3-l2-pi.tst | 95 ++

scripts/rt-tester/t4-l2-pi-deboost.tst | 127 +++

scripts/rt-tester/t5-l4-pi-boost-deboost.tst | 148 ++++

42 files changed, 5138 insertions(+), 180 deletions(-)

Copyright © 2006, Eklektix, Inc.  
Comments and public postings are copyrighted by their creators.  
Linux is a registered trademark of Linus Torval