Lightweight PI-futexes

We are calling them 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.

- even in the slowpath, the system call and scheduling pattern is very similar to normal futexes.

- 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?

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 previous 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.

More details about priority inheritance can be found in
Documentation/rt-mutex.txt.