

Futex Requeue PI

Requeueing of tasks from a non-PI futex to a PI futex requires special handling in order to ensure the underlying `rt_mutex` is never left without an owner if it has waiters; doing so would break the PI boosting logic [see `rt-mutex-design.txt`] For the purposes of brevity, this action will be referred to as "requeue_pi" throughout this document. Priority inheritance is abbreviated throughout as "PI".

Motivation

Without `requeue_pi`, the glibc implementation of `pthread_cond_broadcast()` must resort to waking all the tasks waiting on a `pthread_condvar` and letting them try to sort out which task gets to run first in classic thundering-herd formation. An ideal implementation would wake the highest-priority waiter, and leave the rest to the natural wakeup inherent in unlocking the mutex associated with the `condvar`.

Consider the simplified glibc calls:

```
/* caller must lock mutex */
pthread_cond_wait(cond, mutex)
{
    lock(cond->__data.__lock);
    unlock(mutex);
    do {
        unlock(cond->__data.__lock);
        futex_wait(cond->__data.__futex);
        lock(cond->__data.__lock);
    } while(...)
    unlock(cond->__data.__lock);
    lock(mutex);
}

pthread_cond_broadcast(cond)
{
    lock(cond->__data.__lock);
    unlock(cond->__data.__lock);
    futex_requeue(cond->data.__futex, cond->mutex);
}
```

Once `pthread_cond_broadcast()` requeues the tasks, the `cond->mutex` has waiters. Note that `pthread_cond_wait()` attempts to lock the mutex only after it has returned to user space. This will leave the underlying `rt_mutex` with waiters, and no owner, breaking the previously mentioned PI-boosting algorithms.

In order to support PI-aware `pthread_condvar`'s, the kernel needs to be able to requeue tasks to PI futexes. This support implies that upon a successful `futex_wait` system call, the caller would return to user space already holding the PI futex. The glibc implementation would be modified as follows:

```

/* caller must lock mutex */
pthread_cond_wait_pi(cond, mutex)
{
    lock(cond->__data.__lock);
    unlock(mutex);
    do {
        unlock(cond->__data.__lock);
        futex_wait_requeue_pi(cond->__data.__futex);
        lock(cond->__data.__lock);
    } while(...)
    unlock(cond->__data.__lock);
    /* the kernel acquired the the mutex for us */
}

pthread_cond_broadcast_pi(cond)
{
    lock(cond->__data.__lock);
    unlock(cond->__data.__lock);
    futex_requeue_pi(cond->data.__futex, cond->mutex);
}

```

The actual glibc implementation will likely test for PI and make the necessary changes inside the existing calls rather than creating new calls for the PI cases. Similar changes are needed for `pthread_cond_timedwait()` and `pthread_cond_signal()`.

Implementation

In order to ensure the `rt_mutex` has an owner if it has waiters, it is necessary for both the requeue code, as well as the waiting code, to be able to acquire the `rt_mutex` before returning to user space. The requeue code cannot simply wake the waiter and leave it to acquire the `rt_mutex` as it would open a race window between the requeue call returning to user space and the waiter waking and starting to run. This is especially true in the uncontended case.

The solution involves two new `rt_mutex` helper routines, `rt_mutex_start_proxy_lock()` and `rt_mutex_finish_proxy_lock()`, which allow the requeue code to acquire an uncontended `rt_mutex` on behalf of the waiter and to enqueue the waiter on a contended `rt_mutex`. Two new system calls provide the kernel<->user interface to requeue_pi: `FUTEX_WAIT_REQUEUE_PI` and `FUTEX_REQUEUE_CMP_PI`.

`FUTEX_WAIT_REQUEUE_PI` is called by the waiter (`pthread_cond_wait()` and `pthread_cond_timedwait()`) to block on the initial futex and wait to be requeued to a PI-aware futex. The implementation is the result of a high-speed collision between `futex_wait()` and `futex_lock_pi()`, with some extra logic to check for the additional wake-up scenarios.

`FUTEX_REQUEUE_CMP_PI` is called by the waker (`pthread_cond_broadcast()` and `pthread_cond_signal()`) to requeue and possibly wake the waiting tasks. Internally, this system call is

futex-requeue-pi.txt

still handled by `futex_requeue` (by passing `requeue_pi=1`). Before requeueing, `futex_requeue()` attempts to acquire the requeue target PI futex on behalf of the top waiter. If it can, this waiter is woken. `futex_requeue()` then proceeds to requeue the remaining `nr_wake+nr_requeue` tasks to the PI futex, calling `rt_mutex_start_proxy_lock()` prior to each requeue to prepare the task as a waiter on the underlying `rt_mutex`. It is possible that the lock can be acquired at this stage as well, if so, the next waiter is woken to finish the acquisition of the lock.

`FUTEX_REQUEUE_PI` accepts `nr_wake` and `nr_requeue` as arguments, but their sum is all that really matters. `futex_requeue()` will wake or requeue up to `nr_wake + nr_requeue` tasks. It will wake only as many tasks as it can acquire the lock for, which in the majority of cases should be 0 as good programming practice dictates that the caller of either `pthread_cond_broadcast()` or `pthread_cond_signal()` acquire the mutex prior to making the call. `FUTEX_REQUEUE_PI` requires that `nr_wake=1`. `nr_requeue` should be `INT_MAX` for broadcast and 0 for signal.