RT-mutex subsystem with PI support

RT-mutexes with priority inheritance are used to support PI-futexes, which enable pthread_mutex_t priority inheritance attributes (PTHREAD_PRIO_INHERIT). [See Documentation/locking/pi-futex.rst for more details about PI-futexes.]

This technology was developed in the -rt tree and streamlined for pthread_mutex support.

Basic principles:

RT-mutexes extend the semantics of simple mutexes by the priority inheritance protocol.

A low priority owner of a rt-mutex inherits the priority of a higher priority waiter until the rt-mutex is released. If the temporarily boosted owner blocks on a rt-mutex itself it propagates the priority boosting to the owner of the other rt_mutex it gets blocked on. The priority boosting is immediately removed once the rt_mutex has been unlocked.

This approach allows us to shorten the block of high-prio tasks on mutexes which protect shared resources. Priority inheritance is not a magic bullet for poorly designed applications, but it allows well-designed applications to use userspace locks in critical parts of an high priority thread, without losing determinism.

The enqueueing of the waiters into the rtmutex waiter tree is done in priority order. For same priorities FIFO order is chosen. For each rtmutex, only the top priority waiter is enqueued into the owner's priority waiters tree. This tree too queues in priority order. Whenever the top priority waiter of a task changes (for example it timed out or got a signal), the priority of the owner task is readjusted. The priority enqueueing is handled by "pi_waiters".

RT-mutexes are optimized for fastpath operations and have no internal locking overhead when locking an uncontended mutex or unlocking a mutex without waiters. The optimized fastpath operations require cmpxchg support. [If that is not available then the rt-mutex internal spinlock is used]

The state of the rt-mutex is tracked via the owner field of the rt-mutex structure:

lock->owner holds the task_struct pointer of the owner. Bit 0 is used to keep track of the "lock has waiters" state:

owner	bit0	Notes
NULL	0	lock is free (fast acquire possible)
NULL	1	lock is free and has waiters and the top waiter is going to take the lock [1]
taskpointer	0	lock is held (fast release possible)
taskpointer	1	lock is held and has waiters [2]

The fast atomic compare exchange based acquire and release is only possible when bit 0 of lock->owner is 0.

[1] It also can be a transitional state when grabbing the lock with ->wait_lock is held. To prevent any fast path cmpxchg to the lock, we need to set the bit0 before looking at the lock, and the owner may be NULL in this small time, hence this can be a transitional state.

[2] There is a small time when bit 0 is set but there are no waiters. This can happen when grabbing the lock in the slow path. To prevent a cmpxchg of the owner releasing the lock, we need to set this bit before looking at the lock.

BTW, there is still technically a "Pending Owner", it's just not called that anymore. The pending owner happens to be the top_waiter of a lock that has no owner and has been woken up to grab the lock.