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/pi-futex.txt 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 list 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 list. This list 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 "plists", see include/linux/plist.h for more details.]

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:

rt_mutex->owner holds the task_struct pointer of the owner. Bit 0 and 1 are used to keep track of the "owner is pending" and "rtmutex has waiters" state.

owner	bit1	bit0	
NULL	0	0	mutex is free (fast acquire possible)
NULL	0	1	invalid state
NULL	1	0	Transitional state*

NULL	1	1	invalid state
taskpointer	0	0	mutex is held (fast release possible)
taskpointer	0	1	task is pending owner
taskpointer	1	0	mutex is held and has waiters
taskpointer	1	1	task is pending owner and mutex has waiters

Pending-ownership handling is a performance optimization:
pending-ownership is assigned to the first (highest priority) waiter of
the mutex, when the mutex is released. The thread is woken up and once
it starts executing it can acquire the mutex. Until the mutex is taken
by it (bit 0 is cleared) a competing higher priority thread can "steal"
the mutex which puts the woken up thread back on the waiters list.

The pending-ownership optimization is especially important for the
uninterrupted workflow of high-prio tasks which repeatedly
takes/releases locks that have lower-prio waiters. Without this
optimization the higher-prio thread would ping-pong to the lower-prio
task [because at unlock time we always assign a new owner].

(*) The "mutex has waiters" bit gets set to take the lock. If the lock
doesn't already have an owner, this bit is quickly cleared if there are
no waiters. So this is a transitional state to synchronize with looking
at the owner field of the mutex and the mutex owner releasing the lock.