Generic Mutex Subsystem

started by Ingo Molnar <mingo@redhat.com>

updated by Davidlohr Bueso <davidlohr@hp.com>

What are mutexes?

In the Linux kernel, mutexes refer to a particular locking primitive that enforces serialization on shared memory systems, and not only to the generic term referring to 'mutual exclusion' found in academia or similar theoretical text books. Mutexes are sleeping locks which behave similarly to binary semaphores, and were introduced in 2006[1] as an alternative to these. This new data structure provided a number of advantages, including simpler interfaces, and at that time smaller code (see Disadvantages).

[1] https://lwn.net/Articles/164802/

Implementation

Mutexes are represented by 'struct mutex', defined in include/linux/mutex.h and implemented in kernel/locking/mutex.c. These locks use an atomic variable (->owner) to keep track of the lock state during its lifetime. Field owner actually contains *struct task_struct ** to the current lock owner and it is therefore NULL if not currently owned. Since task_struct pointers are aligned to at least L1_CACHE_BYTES, low bits (3) are used to store extra state (e.g., if waiter list is non-empty). In its most basic form it also includes a wait-queue and a spinlock that serializes access to it. Furthermore, CONFIG_MUTEX_SPIN_ON_OWNER=y systems use a spinner MCS lock (->osq), described below in (ii).

When acquiring a mutex, there are three possible paths that can be taken, depending on the state of the lock:

i. fastpath: tries to atomically acquire the lock by cmpxchg()ing the owner with the current task. This only works in the uncontended case (cmpxchg() checks against 0UL, so all 3 state bits above have to be 0). If the lock is contended it goes to the next possible path.

ii. midpath: aka optimistic spinning, tries to spin for acquisition while the lock owner is running and there are no other tasks ready to run that have higher priority (need_resched). The rationale is that if the lock owner is running, it is likely to release the lock soon. The mutex spinners are queued up using MCS lock so that only one spinner can compete for the mutex.

    The MCS lock (proposed by Mellor-Crummey and Scott) is a simple spinlock with the desirable properties of being fair and with each cpu trying to acquire the lock spinning on a local variable. It avoids expensive cacheline bouncing that common test-and-set spinlock implementations incur. An MCS-like lock is specially tailored for optimistic spinning for sleeping lock implementation. An important feature of the customized MCS lock is that it has the extra property that spinners are able to exit the MCS spinlock queue when they need to reschedule. This further helps avoid situations where MCS spinners that need to reschedule would continue waiting to spin on mutex owner, only to go directly to slowpath upon obtaining the MCS lock.

iii. slowpath: last resort, if the lock is still unable to be acquired, the task is added to the wait-queue and sleeps until woken up by the unlock path. Under normal circumstances it blocks as TASK_UNINTERRUPTIBLE.

While formally kernel mutexes are sleepable locks, it is path (ii) that makes them more practically a hybrid type. By simply not interrupting a task and busy-waiting for a few cycles instead of immediately sleeping, the performance of this lock has been seen to significantly improve a number of workloads. Note that this technique is also used for rw-semaphores.

Semantics

The mutex subsystem checks and enforces the following rules:

- Only one task can hold the mutex at a time.
- Only the owner can unlock the mutex.
- Multiple unlocks are not permitted.
- Recursive locking/unlocking is not permitted.
- A mutex must only be initialized via the API (see below).
- A task may not exit with a mutex held.
- Memory areas where held locks reside must not be freed.
- Held mutexes must not be reinitialized.
- Mutexes may not be used in hardware or software interrupt contexts such as tasklets and timers.

These semantics are fully enforced when CONFIG DEBUG_MUTEXES is enabled. In addition, the mutex debugging code also implements a number of other features that make lock debugging easier and faster:

- Uses symbolic names of mutexes, whenever they are printed in debug output.
- Point-of-acquire tracking, symbolic lookup of function names, list of all locks held in the system, printout of them.

- Owner tracking.
- Detects self-recursing locks and prints out all relevant info.
- Detects multi-task circular deadlocks and prints out all affected locks and tasks (and only those tasks).

Interfaces

Statically define the mutex:

```
DEFINE_MUTEX(name);
```

Dynamically initialize the mutex:

```
mutex_init(mutex);
```

Acquire the mutex, uninterruptible:

```
void mutex_lock(struct mutex *lock);
void mutex_lock_nested(struct mutex *lock, unsigned int subclass);
int  mutex_trylock(struct mutex *lock);
```

Acquire the mutex, interruptible:

```
int mutex_lock_interruptible_nested(struct mutex *lock,
                                    unsigned int subclass);
int mutex_lock_interruptible(struct mutex *lock);
```

Acquire the mutex, interruptible, if dec to 0:

```
int atomic_dec_and_mutex_lock(atomic_t *cnt, struct mutex *lock);
```

Unlock the mutex:

```
void mutex_unlock(struct mutex *lock);
```

Test if the mutex is taken:

```
int mutex_is_locked(struct mutex *lock);
```

Disadvantages

Unlike its original design and purpose, 'struct mutex' is among the largest locks in the kernel. E.g: on x86-64 it is 32 bytes, where 'struct semaphore' is 24 bytes and rw_semaphore is 40 bytes. Larger structure sizes mean more CPU cache and memory footprint.

When to use mutexes

Unless the strict semantics of mutexes are unsuitable and/or the critical region prevents the lock from being shared, always prefer them to any other locking primitive.