Generic Mutex Subsystem

started by Ingo Molnar <mingo@redhat.com>

"Why on earth do we need a new mutex subsystem, and what's wrong with semaphores?"

firstly, there's nothing wrong with semaphores. But if the simpler mutex semantics are sufficient for your code, then there are a couple of advantages of mutexes:

- 'struct mutex' is smaller on most architectures: .e.g on x86, 'struct semaphore' is 20 bytes, 'struct mutex' is 16 bytes. A smaller structure size means less RAM footprint, and better CPU-cache utilization.

- tighter code. On x86 i get the following .text sizes when switching all mutex-alike semaphores in the kernel to the mutex subsystem:

```
    text    data     bss     dec     hex filename
 3280380  868188  396860 4545428  455b94 vmlinux-semaphore
 3255329  865296  396732 4517357  44eded vmlinux-mutex
```

  that's 25051 bytes of code saved, or a 0.76% win - off the hottest codepaths of the kernel. (The .data savings are 2892 bytes, or 0.33%) Smaller code means better icache footprint, which is one of the major optimization goals in the Linux kernel currently.

- the mutex subsystem is slightly faster and has better scalability for contended workloads. On an 8-way x86 system, running a mutex-based kernel and testing creat+unlink+close (of separate, per-task files) in /tmp with 16 parallel tasks, the average number of ops/sec is:

```
 Semaphores:                        Mutexes:

 $ ./test-mutex V 16 10             $ ./test-mutex V 16 10
 8 CPUs, running 16 tasks.          8 CPUs, running 16 tasks.
 checking VFS performance.          checking VFS performance.
 avg loops/sec:      34713          avg loops/sec:      84153
 CPU utilization:    63%            CPU utilization:    22%
```

  i.e. in this workload, the mutex based kernel was 2.4 times faster than the semaphore based kernel, _and_ it also had 2.8 times less CPU utilization. (In terms of 'ops per CPU cycle', the semaphore kernel performed 551 ops/sec per 1% of CPU time used, while the mutex kernel performed 3825 ops/sec per 1% of CPU time used - it was 6.9 times more efficient.)

  the scalability difference is visible even on a 2-way P4 HT box:

```
 Semaphores:                        Mutexes:

 $ ./test-mutex V 16 10             $ ./test-mutex V 16 10
 4 CPUs, running 16 tasks.          8 CPUs, running 16 tasks.
 checking VFS performance.          checking VFS performance.
```

```
 avg loops/sec:      127659         avg loops/sec:      181082
 CPU utilization:    100%           CPU utilization:    34%
```

(the straight performance advantage of mutexes is 41%, the per-cycle efficiency of mutexes is 4.1 times better.)

- there are no fastpath tradeoffs, the mutex fastpath is just as tight as the semaphore fastpath. On x86, the locking fastpath is 2 instructions:

```
 c0377ccb <mutex_lock>:
 c0377ccb:       f0 ff 08                lock decl (%eax)
 c0377cce:       78 0e                   js     c0377cde <.text..lock.mutex>
 c0377cd0:       c3                      ret
```

  the unlocking fastpath is equally tight:

```
 c0377cd1 <mutex_unlock>:
 c0377cd1:       f0 ff 00                lock incl (%eax)
 c0377cd4:       7e 0f                   jle    c0377ce5
<.text..lock.mutex+0x7>
 c0377cd6:       c3                      ret
```

- 'struct mutex' semantics are well-defined and are enforced if CONFIG_DEBUG_MUTEXES is turned on. Semaphores on the other hand have virtually no debugging code or instrumentation. 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 is not permitted
* - a mutex object must be initialized via the API
* - a mutex object must not be initialized via memset or copying
* - task may not exit with 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
```

  furthermore, there are also convenience features in the debugging code:

```
* - 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)
```

Disadvantages

The stricter mutex API means you cannot use mutexes the same way you can use semaphores: e.g. they cannot be used from an interrupt context, nor can they be unlocked from a different context that which acquired it. [I'm not aware of any other (e.g. performance) disadvantages from using mutexes at the moment, please let me know if you find any.]

Implementation of mutexes

'struct mutex' is the new mutex type, defined in include/linux/mutex.h and implemented in kernel/mutex.c. It is a counter-based mutex with a spinlock and a wait-list. The counter has 3 states: 1 for "unlocked", 0 for "locked" and negative numbers (usually -1) for "locked, potential waiters queued".

the APIs of 'struct mutex' have been streamlined:

```
 DEFINE_MUTEX(name);

 mutex_init(mutex);

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