

[patch 6/8] mutex subsystem, core

Source: <http://linux.derkeiler.com/Mailing-Lists/Kernel/2005-12/msg06602.html>

- *From:* Ingo Molnar <mingo@xxxxxxx>
 - *Date:* Wed, 21 Dec 2005 23:37:30 +0100
-

mutex implementation, core files: just the basic subsystem, no users of it.

Signed-off-by: Ingo Molnar <mingo@xxxxxxx>

```
include/linux/mutex.h | 100 ++++++++
kernel/Makefile | 2
kernel/mutex.c | 453 +++++++++++++++++++++++++++++++++++++
3 files changed, 554 insertions(+), 1 deletion(-)
```

Index: linux/include/linux/mutex.h

```
=====
--- /dev/null
+++ linux/include/linux/mutex.h
@@ -0,0 +1,100 @@
+#ifndef __LINUX_MUTEX_H
+#define __LINUX_MUTEX_H
+
+/*
+ * Mutexes: blocking mutual exclusion locks
+ *
+ * started by Ingo Molnar:
+ *
+ * Copyright (C) 2004, 2005 Red Hat, Inc., Ingo Molnar <mingo@xxxxxxxxxx>
+ *
+ * This file contains the main data structure and API definitions.
+ */
+#include <linux/config.h>
+#include <asm/atomic.h>
+#include <linux/list.h>
+#include <linux/spinlock_types.h>
+
+/*
+ * Simple, straightforward mutexes with strict semantics:
+ *
+ * - only one task can hold the mutex at a time
+ * - only the owner can unlock the mutex
+ * - multiple unlocks are not permitted
```

[patch 6/8] mutex subsystem, core

```
+ * – 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 irq contexts
+ *
+ * These semantics are fully enforced when DEBUG_MUTEXES is
+ * enabled. Furthermore, besides enforcing the above rules, the mutex
+ * debugging code also implements a number of additional 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-recurring locks and prints out all relevant info
+ * – detects multi-task circular deadlocks and prints out all affected
+ * locks and tasks (and only those tasks)
+ */
+struct mutex {
+ // 1: unlocked, 0: locked, negative: locked, possible waiters
+ atomic_t count;
+ spinlock_t wait_lock;
+ struct list_head wait_list;
+#ifdef CONFIG_DEBUG_MUTEXES
+ struct thread_info *owner;
+ struct list_head held_list;
+ unsigned long acquire_ip;
+ const char *name;
+ void *magic;
+#endif
+};
+
+/*
+ * This is the control structure for tasks blocked on mutex,
+ * which resides on the blocked task's kernel stack:
+ */
+struct mutex_waiter {
+ struct list_head list;
+ struct thread_info *ti;
+#ifdef CONFIG_DEBUG_MUTEXES
+ struct mutex *lock;
+ void *magic;
+#endif
+};
+
+#ifdef CONFIG_DEBUG_MUTEXES
+# include <linux/mutex-debug.h>
+#else
```

[patch 6/8] mutex subsystem, core

```
+# define __DEBUG_MUTEX_INITIALIZER(lockname)
+# define mutex_init(sem) __mutex_init(sem, NULL)
+# define mutex_debug_show_all_locks() do { } while (0)
+# define mutex_debug_show_held_locks(p) do { } while (0)
+# define mutex_debug_check_no_locks_held(task) do { } while (0)
+# define mutex_debug_check_no_locks_freed(from, to) do { } while (0)
+#endif
+
+#define __MUTEX_INITIALIZER(lockname) \
+ { .count = ATOMIC_INIT(1) \
+ , .wait_lock = SPIN_LOCK_UNLOCKED \
+ , .wait_list = LIST_HEAD_INIT(lockname.wait_list) \
+ __DEBUG_MUTEX_INITIALIZER(lockname) }
+
+#define DEFINE_MUTEX(mutexname) \
+ struct mutex mutexname = __MUTEX_INITIALIZER(mutexname)
+
+extern void FASTCALL(__mutex_init(struct mutex *lock, const char *name));
+
+extern void FASTCALL(mutex_lock(struct mutex *lock));
+extern int FASTCALL(mutex_lock_interruptible(struct mutex *lock));
+extern int FASTCALL(mutex_trylock(struct mutex *lock));
+extern void FASTCALL(mutex_unlock(struct mutex *lock));
+extern int FASTCALL(mutex_is_locked(struct mutex *lock));
+
+#endif
```

Index: linux/kernel/Makefile

```
-----
--- linux.orig/kernel/Makefile
+++ linux/kernel/Makefile
@@ -7,7 +7,7 @@ obj-y = sched.o fork.o exec_domain.o
sysctl.o capability.o ptrace.o timer.o user.o \
signal.o sys.o kmod.o workqueue.o pid.o \
rcupdate.o intermodule.o extable.o params.o posix-timers.o \
- kthread.o wait.o kfifo.o sys_ni.o posix-cpu-timers.o
+ kthread.o wait.o kfifo.o sys_ni.o posix-cpu-timers.o mutex.o
```

```
obj-$(CONFIG_FUTEX) += futex.o
obj-$(CONFIG_GENERIC_ISA_DMA) += dma.o
Index: linux/kernel/mutex.c
```

```
-----
--- /dev/null
+++ linux/kernel/mutex.c
@@ -0,0 +1,453 @@
+/*
+ * kernel/mutex.c
+ *
+ * Mutexes: blocking mutual exclusion locks
+ *
+ * Started by Ingo Molnar:
+ *
```

[patch 6/8] mutex subsystem, core

```
+ * Copyright (C) 2004, 2005 Red Hat, Inc., Ingo Molnar <mingo@xxxxxxxxxx>
+ *
+ * Many thanks to Arjan van de Ven, Thomas Gleixner, Steven Rostedt and
+ * David Howells for suggestions and improvements.
+ */
+#include <linux/mutex.h>
+#include <linux/sched.h>
+#include <linux/delay.h>
+#include <linux/module.h>
+#include <linux/spinlock.h>
+#include <linux/kallsyms.h>
+#include <linux/syscalls.h>
+#include <linux/interrupt.h>
+
+/*
+ * Debugging constructs – they are NOPs in the !DEBUG case:
+ */
+#ifdef CONFIG_DEBUG_MUTEXES
+# include "mutex-debug.c"
+#else
+/*
+ * We can speed up the lock–acquire and lock–release codepath, if
+ * there's no debugging state to be set up (!DEBUG_MUTEXES).
+ */
+# define MUTEX_LOCKLESS_FASTPATH
+
+/*
+ * Return–address parameters/declarations that are NOPs in the !DEBUG case:
+ */
+# define __IP_DECL__
+# define __IP__
+# define __CALLER_IP__
+
+# define spin_lock_mutex(lock) spin_lock(lock)
+# define spin_unlock_mutex(lock) spin_unlock(lock)
+# define remove_waiter(lock, waiter, ti) \
+ __list_del((waiter)->list.prev, (waiter)->list.next)
+
+# define DEBUG_WARN_ON(c) do { } while (0)
+
+# define debug_set_owner(lock, new_owner) do { } while (0)
+# define debug_clear_owner(lock) do { } while (0)
+# define debug_init_waiter(waiter) do { } while (0)
+# define debug_wake_waiter(lock, waiter) do { } while (0)
+# define debug_free_waiter(waiter) do { } while (0)
+# define debug_add_waiter(lock, waiter, ti, ip) do { } while (0)
+# define debug_mutex_unlock(lock) do { } while (0)
+# define debug_mutex_init(lock, name) do { } while (0)
+#endif /* !CONFIG_DEBUG_MUTEXES */
+
+/*
```

[patch 6/8] mutex subsystem, core

```
+ * Block on a lock – add ourselves to the list of waiters.
+ * Called with lock->wait_lock held.
+ */
+static inline void
+add_waiter(struct mutex *lock, struct mutex_waiter *waiter,
+ struct thread_info *ti __IP_DECL__)
+{
+ debug_add_waiter(lock, waiter, ti, ip);
+
+ waiter->ti = ti;
+
+ /* Add waiting tasks to the end of the waitqueue (FIFO): */
+ list_add_tail(&waiter->list, &lock->wait_list);
+}
+
+/*
+ * Wake up a task and make it the new owner of the mutex:
+ */
+static inline void
+mutex_wakeup_waiter(struct mutex *lock __IP_DECL__)
+{
+ struct mutex_waiter *waiter;
+
+ /* get the first entry from the wait-list: */
+ waiter = list_entry(lock->wait_list.next, struct mutex_waiter, list);
+
+ debug_wake_waiter(lock, waiter);
+
+ wake_up_process(waiter->ti->task);
+}
+
+/*
+ * Lock a mutex, common slowpath. We just decremented the count,
+ * and it got negative as a result.
+ *
+ * We enter with the lock held, and return with it released.
+ */
+static inline int
+__mutex_lock_common(struct mutex *lock, struct mutex_waiter *waiter,
+ struct thread_info *ti,
+ unsigned long task_state __IP_DECL__)
+{
+ struct task_struct *task = ti->task;
+ unsigned int old_val;
+
+ /*
+ * Lets try to take the lock again – this is needed even if
+ * we get here for the first time (shortly after failing to
+ * acquire the lock), to make sure that we get a wakeup once
+ * it's unlocked. Later on this is the operation that gives
+ * us the lock. If there are other waiters we need to xchg it
```

[patch 6/8] mutex subsystem, core

```
+ * to -1, so that when we release the lock, we properly wake
+ * up the other waiters:
+ */
+ old_val = atomic_xchg(&lock->count, -1);
+
+ if (unlikely(old_val == 1)) {
+ /*
+ * Got the lock – rejoice! But there's one small
+ * detail to fix up: above we have set the lock to -1,
+ * unconditionally. But what if there are no waiters?
+ * While it would work with -1 too, 0 is a better value
+ * in that case, because we wont hit the slowpath when
+ * we release the lock. We can simply use atomic_set()
+ * for this, because we are the owners of the lock now,
+ * and are still holding the wait_lock:
+ */
+ if (likely(list_empty(&lock->wait_list)))
+ atomic_set(&lock->count, 0);
+ debug_set_owner(lock, ti __IP__);
+
+ spin_unlock_mutex(&lock->wait_lock);
+
+ debug_free_waiter(waiter);
+
+ DEBUG_WARN_ON(list_empty(&lock->held_list));
+ DEBUG_WARN_ON(lock->owner != ti);
+
+ return 1;
+ }
+
+ add_waiter(lock, waiter, ti __IP__);
+ __set_task_state(task, task_state);
+
+ /*
+ * Ok, didnt get the lock – we'll go to sleep after return:
+ */
+ spin_unlock_mutex(&lock->wait_lock);
+
+ return 0;
+ }
+
+ /*
+ * Lock the mutex:
+ */
+static inline void __mutex_lock_nonatomic(struct mutex *lock __IP_DECL__)
+{
+ struct thread_info *ti = current_thread_info();
+ struct mutex_waiter waiter;
+
+ debug_init_waiter(&waiter);
+
+ }
```

[patch 6/8] mutex subsystem, core

```
+ spin_lock_mutex(&lock->wait_lock);
+
+ /* releases the internal lock: */
+ while (!__mutex_lock_common(lock, &waiter, ti,
+ TASK_UNINTERRUPTIBLE __IP__)) {
+ /* wait to be woken up: */
+ schedule();
+
+ spin_lock_mutex(&lock->wait_lock);
+ remove_waiter(lock, &waiter, ti);
+ }
+ }
+
+ /*
+ * Lock a mutex interruptible:
+ */
+static int __sched
+__mutex_lock_interruptible_nonatomic(struct mutex *lock __IP_DECL__)
+{
+ struct thread_info *ti = current_thread_info();
+ struct mutex_waiter waiter;
+
+ debug_init_waiter(&waiter);
+
+ spin_lock_mutex(&lock->wait_lock);
+
+ for (;;) {
+ /* releases the internal lock: */
+ if (__mutex_lock_common(lock, &waiter, ti,
+ TASK_INTERRUPTIBLE __IP__))
+ return 0;
+
+ /* break out on a signal: */
+ if (unlikely(signal_pending(ti->task)))
+ break;
+
+ /* wait to be given the lock: */
+ schedule();
+
+ spin_lock_mutex(&lock->wait_lock);
+ remove_waiter(lock, &waiter, ti);
+ }
+ /*
+ * We got a signal. Remove ourselves from the wait list:
+ */
+ spin_lock_mutex(&lock->wait_lock);
+ remove_waiter(lock, &waiter, ti);
+ /*
+ * If there are other waiters then wake
+ * one up:
+ */
```

[patch 6/8] mutex subsystem, core

```
+ if (unlikely(!list_empty(&lock->wait_list)))
+ mutex_wakeup_waiter(lock __IP__);
+
+ spin_unlock_mutex(&lock->wait_lock);
+
+ __set_task_state(ti->task, TASK_RUNNING);
+
+ debug_free_waiter(&waiter);
+
+ return -EINTR;
+}
+
+/*
+ * Mutex trylock, returns 1 if successful, 0 if contention:
+ */
+#ifdef MUTEX_LOCKLESS_FASTPATH
+
+/*
+ * We have two fastpath variants. The cmpxchg based one is
+ * better (because it has no side-effect on mutex_unlock()),
+ * but cmpxchg is not available on every architecture, so we
+ * provide an atomic_dec_return based variant too:
+ */
+#ifdef __HAVE_ARCH_CMPXCHG
+static inline int __mutex_trylock(struct mutex *lock __IP_DECL__)
+{
+ if (atomic_cmpxchg(&lock->count, 1, 0) == 1)
+ return 1;
+ return 0;
+}
+#else
+static inline int __mutex_trylock(struct mutex *lock __IP_DECL__)
+{
+ /*
+ * If we do not get the lock then we have the counter
+ * negative, but that's not a big problem, it will
+ * force the next mutex_unlock() into the slowpath.
+ */
+ if (atomic_dec_return(&lock->count) < 0)
+ return 0;
+ return 1;
+}
+#endif
+
+#else /* !MUTEX_LOCKLESS_FASTPATH */
+
+static inline int __mutex_trylock(struct mutex *lock __IP_DECL__)
+{
+ struct thread_info *ti = current_thread_info();
+ int ret = 0;
+
+
```

[patch 6/8] mutex subsystem, core

```
+ spin_lock_mutex(&lock->wait_lock);
+
+ if (atomic_read(&lock->count) == 1) {
+ atomic_set(&lock->count, 0);
+ debug_set_owner(lock, ti __IP__);
+ ret = 1;
+ }
+
+ spin_unlock_mutex(&lock->wait_lock);
+
+ return ret;
+}
+
+ #endif /* !MUTEX_LOCKLESS_FASTPATH */
+
+ int fastcall mutex_is_locked(struct mutex *lock)
+ {
+ smp_mb();
+ return atomic_read(&lock->count) != 1;
+ }
+
+ EXPORT_SYMBOL_GPL(mutex_is_locked);
+
+ int fastcall mutex_trylock(struct mutex *lock)
+ {
+ return __mutex_trylock(lock __CALLER_IP__);
+ }
+
+ /*
+ * Release the lock:
+ */
+ static inline void __mutex_unlock_nonatomic(struct mutex *lock __IP_DECL__)
+ {
+ spin_lock_mutex(&lock->wait_lock);
+
+ debug_mutex_unlock(lock);
+
+ /*
+ * Set it back to 'unlocked'. We'll have a waiter in flight
+ * (if any), and if some other task comes around, let it
+ * steal the lock. Waiters take care of themselves and stay
+ * in flight until necessary.
+ */
+ atomic_set(&lock->count, 1);
+
+ if (!list_empty(&lock->wait_list))
+ mutex_wakeup_waiter(lock __IP__);
+
+ debug_clear_owner(lock);
+
+ spin_unlock_mutex(&lock->wait_lock);
+ }
```

[patch 6/8] mutex subsystem, core

```
+
+ #ifdef MUTEX_LOCKLESS_FASTPATH
+
+ /*
+ * We split it into a fastpath and a separate slowpath function,
+ * to reduce the register pressure on the fastpath:
+ *
+ * We want the atomic op come first, to make sure the
+ * branch is predicted as default-untaken:
+ */
+ static __sched void FASTCALL(__mutex_lock_noinline(atomic_t *lock_count));
+
+ /*
+ * The locking fastpath is the 1->0 transition from
+ * 'unlocked' into 'locked' state:
+ */
+ static inline void __mutex_lock_atomic(struct mutex *lock)
+ {
+     atomic_dec_call_if_negative(&lock->count, __mutex_lock_noinline);
+ }
+
+ static fastcall __sched void __mutex_lock_noinline(atomic_t *lock_count)
+ {
+     struct mutex *lock = container_of(lock_count, struct mutex, count);
+
+     __mutex_lock_nonatomic(lock);
+ }
+
+ static inline void __mutex_lock(struct mutex *lock)
+ {
+     __mutex_lock_atomic(lock);
+ }
+
+ static inline int __mutex_lock_interruptible(struct mutex *lock)
+ {
+     if (unlikely(atomic_dec_return(&lock->count) < 0))
+         return __mutex_lock_interruptible_nonatomic(lock);
+     return 0;
+ }
+
+ static void __sched FASTCALL(__mutex_unlock_noinline(atomic_t *lock_count));
+
+ /*
+ * The unlocking fastpath is the 0->1 transition from
+ * 'locked' into 'unlocked' state:
+ */
+ static inline void __mutex_unlock_atomic(struct mutex *lock)
+ {
+     atomic_inc_call_if_nonpositive(&lock->count, __mutex_unlock_noinline);
+ }
+
```

[patch 6/8] mutex subsystem, core

```
+static fastcall void __sched __mutex_unlock_noinline(atomic_t *lock_count)
+{
+ struct mutex *lock = container_of(lock_count, struct mutex, count);
+
+ __mutex_unlock_nonatomic(lock);
+}
+
+static inline void __mutex_unlock(struct mutex *lock)
+{
+ __mutex_unlock_atomic(lock);
+}
+
+##else
+
+static inline void __mutex_lock(struct mutex *lock __IP_DECL__)
+{
+ __mutex_lock_nonatomic(lock __IP__);
+}
+
+static inline void __mutex_unlock(struct mutex *lock __IP_DECL__)
+{
+ __mutex_unlock_nonatomic(lock __IP__);
+}
+
+static inline int __mutex_lock_interruptible(struct mutex *lock __IP_DECL__)
+{
+ return __mutex_lock_interruptible_nonatomic(lock __IP__);
+}
+
+##endif
+
+/*
+ * Some architectures provide hand-coded mutex_lock() functions,
+ * the will call the mutex_*_slowpath() generic functions:
+ */
+##ifdef ARCH_IMPLEMENTES_MUTEX_FASTPATH
+
+void __sched fastcall mutex_lock_slowpath(struct mutex *lock)
+{
+ __mutex_lock(lock);
+}
+
+void __sched fastcall mutex_unlock_slowpath(struct mutex *lock)
+{
+ __mutex_unlock(lock);
+}
+
+int __sched fastcall mutex_lock_interruptible_slowpath(struct mutex *lock)
+{
+ return __mutex_lock_interruptible(lock);
+}
+}
```

[patch 6/8] mutex subsystem, core

```
+
+##else
+
+void __sched fastcall mutex_lock(struct mutex *lock)
+{
+ __mutex_lock(lock __CALLER_IP__);
+}
+
+void __sched fastcall mutex_unlock(struct mutex *lock)
+{
+ DEBUG_WARN_ON(lock->owner != current_thread_info());
+ __mutex_unlock(lock __CALLER_IP__);
+}
+
+int __sched fastcall mutex_lock_interruptible(struct mutex *lock)
+{
+ return __mutex_lock_interruptible(lock __CALLER_IP__);
+}
+
+##endif
+
+EXPORT_SYMBOL_GPL(mutex_lock);
+EXPORT_SYMBOL_GPL(mutex_unlock);
+EXPORT_SYMBOL_GPL(mutex_lock_interruptible);
+
+/*
+ * Initialise the lock:
+ */
+void fastcall __mutex_init(struct mutex *lock, const char *name)
+{
+ atomic_set(&lock->count, 1);
+ spin_lock_init(&lock->wait_lock);
+ INIT_LIST_HEAD(&lock->wait_list);
+
+ debug_mutex_init(lock, name);
+}
+EXPORT_SYMBOL_GPL(__mutex_init);
+
-
```

To unsubscribe from this list: send the line "unsubscribe linux-kernel" in the body of a message to majordomo@xxxxxxxxxxxxxxxxxxx

More majordomo info at <http://vger.kernel.org/majordomo-info.html>

Please read the FAQ at <http://www.tux.org/lkml/>

-
- Prev by Date: [\[patch 1/8\] mutex subsystem, XFS namespace collision fixes](#)
 - Next by Date: [\[patch 7/8\] mutex subsystem, debugging code](#)
 - Previous by thread: [\[patch 6/8\] mutex subsystem, core](#)
 - Next by thread: [gla23xx driver in 2.6.x kernels](#)
 - Index(es):

[patch 6/8] mutex subsystem, core

- ◆ Date
- ◆ Thread