

[patch 05/26] pi-futex: Fix exit races and locking problems

Source: <http://linux.derkeiler.com/Mailing-Lists/Kernel/2007-07/msg13325.html>

- *From:* Greg KH <gregkh@xxxxxxx>
 - *Date:* Mon, 30 Jul 2007 21:31:36 -0700
-

–stable review patch. If anyone has any objections, please let us know.

From: Alexey Kuznetsov <kuznet@xxxxxxxxxxxxxx>

1. New entries can be added to `tsk->pi_state_list` after task completed `exit_pi_state_list()`. The result is memory leakage and deadlocks.

2. `handle_mm_fault()` is called under spinlock. The result is obvious.

3. results in self-inflicted deadlock inside `glibc`.

Sometimes `futex_lock_pi` returns `-ESRCH`, when it is not expected and `glibc` enters to `for(;;) sleep()` to simulate deadlock. This problem is quite obvious and I think the patch is right. Though it looks like each "if" in `futex_lock_pi()` got some stupid special case "else if". :-)

4. sometimes `futex_lock_pi()` returns `-EDEADLK`, when nobody has the lock. The reason is also obvious (see comment in the patch), but correct fix is far beyond my comprehension.

I guess someone already saw this, the chunk:

```
if (rt_mutex_trylock(&q.pi_state->pi_mutex))
    ret = 0;
```

is obviously from the same opera. But it does not work, because the `rtmutex` is really taken at this point: `wake_futex_pi()` of previous owner reassigned it to us. My fix works. But it looks very stupid. I would think about removal of shift of ownership in `wake_futex_pi()` and making all the work in context of process taking lock.

From: Thomas Gleixner <tglx@xxxxxxxxxxxxxx>

Fix 1) Avoid the tasklist lock variant of the exit race fix by adding an additional state transition to the exit code.

This fixes also the issue, when a task with recursive segfaults is not able to release the futexes.

[patch 05/26] pi-futex: Fix exit races and locking problems

Fix 2) Cleanup the lookup_pi_state() failure path and solve the -ESRCH problem finally.

Fix 3) Solve the fixup_pi_state_owner() problem which needs to do the fixup in the lock protected section by using the in_atomic userspace access functions.

This removes also the ugly lock drop / unqueue inside of fixup_pi_state()

Fix 4) Fix a stale lock in the error path of futex_wake_pi()

Added some error checks for verification.

The -EDEADLK problem is solved by the rtmutex fixups.

Cc: Alexey Kuznetsov <kuznet@xxxxxxxxxxxxxx>
Signed-off-by: Thomas Gleixner <tglx@xxxxxxxxxxxxxx>
Acked-by: Ingo Molnar <mingo@xxxxxxx>
Signed-off-by: Chris Wright <chrisw@xxxxxxxxxxxxxx>
Signed-off-by: Greg Kroah-Hartman <gregkh@xxxxxxx>

include/linux/sched.h | 1
kernel/exit.c | 22 +++++
kernel/futex.c | 191 +++-----
3 files changed, 150 insertions(+), 64 deletions(-)

--- linux-2.6.21.6.orig/kernel/futex.c
+++ linux-2.6.21.6/kernel/futex.c
@@ -396,10 +396,6 @@ static struct task_struct * futex_find_g
p = NULL;
goto out_unlock;
}
- if (p->exit_state != 0) {
- p = NULL;
- goto out_unlock;
- }
get_task_struct(p);
out_unlock:
rcu_read_unlock();
@@ -467,7 +463,7 @@ lookup_pi_state(u32 uval, struct futex_h
struct futex_q *this, *next;
struct list_head *head;
struct task_struct *p;
- pid_t pid;
+ pid_t pid = uval & FUTEX_TID_MASK;

head = &hb->chain;

@@ -485,6 +481,8 @@ lookup_pi_state(u32 uval, struct futex_h

[patch 05/26] pi-futex: Fix exit races and locking problems

```
return -EINVAL;

WARN_ON(!atomic_read(&pi_state->refcount));
+ WARN_ON(pid && pi_state->owner &&
+ pi_state->owner->pid != pid);

atomic_inc(&pi_state->refcount);
me->pi_state = pi_state;
@@ -495,15 +493,33 @@ lookup_pi_state(u32 uval, struct futex_h

/*
 * We are the first waiter - try to look up the real owner and attach
 * the new pi_state to it, but bail out when the owner died bit is set
 * and TID = 0:
+ * the new pi_state to it, but bail out when TID = 0
 */
- pid = uval & FUTEX_TID_MASK;
- if (!pid && (uval & FUTEX_OWNER_DIED))
+ if (!pid)
return -ESRCH;
p = futex_find_get_task(pid);
- if (!p)
- return -ESRCH;
+ if (IS_ERR(p))
+ return PTR_ERR(p);
+
+ /*
+ * We need to look at the task state flags to figure out,
+ * whether the task is exiting. To protect against the do_exit
+ * change of the task flags, we do this protected by
+ * p->pi_lock:
+ */
+ spin_lock_irq(&p->pi_lock);
+ if (unlikely(p->flags & PF_EXITING)) {
+ /*
+ * The task is on the way out. When PF_EXITPIDONE is
+ * set, we know that the task has finished the
+ * cleanup:
+ */
+ int ret = (p->flags & PF_EXITPIDONE) ? -ESRCH : -EAGAIN;
+
+ spin_unlock_irq(&p->pi_lock);
+ put_task_struct(p);
+ return ret;
+ }

pi_state = alloc_pi_state();

@@ -516,7 +532,6 @@ lookup_pi_state(u32 uval, struct futex_h
/* Store the key for possible exit cleanups: */
pi_state->key = me->key;
```

[patch 05/26] pi-futex: Fix exit races and locking problems

```
- spin_lock_irq(&p->pi_lock);
WARN_ON(!list_empty(&pi_state->list));
list_add(&pi_state->list, &p->pi_state_list);
pi_state->owner = p;
@@ -583,15 +598,22 @@ static int wake_futex_pi(u32 __user *uad
* preserve the owner died bit.)
*/
if (!(uval & FUTEX_OWNER_DIED)) {
+ int ret = 0;
+
newval = FUTEX_WAITERS | new_owner->pid;

pagefault_disable();
curval = futex_atomic_cmpxchg_inatomic(uaddr, uval, newval);
pagefault_enable();
+
if (curval == -EFAULT)
- return -EFAULT;
+ ret = -EFAULT;
if (curval != uval)
- return -EINVAL;
+ ret = -EINVAL;
+ if (ret) {
+ spin_unlock(&pi_state->pi_mutex.wait_lock);
+ return ret;
+ }
}

spin_lock_irq(&pi_state->owner->pi_lock);
@@ -1149,6 +1171,7 @@ static int futex_lock_pi(u32 __user *uad
if (unlikely(ret != 0))
goto out_release_sem;

+ retry_unlocked:
hb = queue_lock(&q, -1, NULL);

retry_locked:
@@ -1200,34 +1223,58 @@ static int futex_lock_pi(u32 __user *uad
ret = lookup_pi_state(uval, hb, &q);

if (unlikely(ret)) {
- /*
- * There were no waiters and the owner task lookup
- * failed. When the OWNER_DIED bit is set, then we
- * know that this is a robust futex and we actually
- * take the lock. This is safe as we are protected by
- * the hash bucket lock. We also set the waiters bit
- * unconditionally here, to simplify glibc handling of
- * multiple tasks racing to acquire the lock and
- * cleanup the problems which were left by the dead
```

[patch 05/26] pi-futex: Fix exit races and locking problems

```
- * owner.
- */
- if (curval & FUTEX_OWNER_DIED) {
- uval = newval;
- newval = current->pid |
- FUTEX_OWNER_DIED | FUTEX_WAITERS;
+ switch (ret) {

- pagefault_disable();
- curval = futex_atomic_cmpxchg_inatomic(uaddr,
- uval, newval);
- pagefault_enable();
+ case -EAGAIN:
+ /*
+ * Task is exiting and we just wait for the
+ * exit to complete.
+ */
+ queue_unlock(&q, hb);
+ up_read(&curr->mm->mmap_sem);
+ cond_resched();
+ goto retry;

- if (unlikely(curval == -EFAULT))
+ case -ESRCH:
+ /*
+ * No owner found for this futex. Check if the
+ * OWNER_DIED bit is set to figure out whether
+ * this is a robust futex or not.
+ */
+ if (get_futex_value_locked(&curval, uaddr))
goto uaddr_faulted;
- if (unlikely(curval != uval))
- goto retry_locked;
- ret = 0;
+
+ /*
+ * There were no waiters and the owner task lookup
+ * failed. When the OWNER_DIED bit is set, then we
+ * know that this is a robust futex and we actually
+ * take the lock. This is safe as we are protected by
+ * the hash bucket lock. We also set the waiters bit
+ * unconditionally here, to simplify glibc handling of
+ * multiple tasks racing to acquire the lock and
+ * cleanup the problems which were left by the dead
+ * owner.
+ */
+ if (curval & FUTEX_OWNER_DIED) {
+ uval = newval;
+ newval = current->pid |
+ FUTEX_OWNER_DIED | FUTEX_WAITERS;
+
+ }
```

[patch 05/26] pi-futex: Fix exit races and locking problems

```
+ pagefault_disable();
+ curval = futex_atomic_cmpxchg_inatomic(uaddr,
+ uval,
+ newval);
+ pagefault_enable();
+
+ if (unlikely(curval == -EFAULT))
+ goto uaddr_faulted;
+ if (unlikely(curval != uval))
+ goto retry_locked;
+ ret = 0;
+ }
+ default:
+ goto out_unlock_release_sem;
}
- goto out_unlock_release_sem;
}

/*
@@ -1279,39 +1326,52 @@ static int futex_lock_pi(u32 __user *uad
list_add(&q.pi_state->list, &current->pi_state_list);
spin_unlock_irq(&current->pi_lock);

- /* Unqueue and drop the lock */
- unqueue_me_pi(&q, hb);
- up_read(&curr->mm->mmap_sem);
/*
* We own it, so we have to replace the pending owner
- * TID. This must be atomic as we have preserve the
+ * TID. This must be atomic as we have to preserve the
* owner died bit here.
*/
- ret = get_user(uval, uaddr);
+ ret = get_futex_value_locked(&uval, uaddr);
while (!ret) {
newval = (uval & FUTEX_OWNER_DIED) | newtid;
+
+ pagefault_disable();
curval = futex_atomic_cmpxchg_inatomic(uaddr,
uval, newval);
+ pagefault_enable();
+
if (curval == -EFAULT)
ret = -EFAULT;
if (curval == uval)
break;
uval = curval;
}
- } else {
+ } else if (ret) {
/*
```

[patch 05/26] pi-futex: Fix exit races and locking problems

```
* Catch the rare case, where the lock was released
* when we were on the way back before we locked
* the hash bucket.
*/
- if (ret && q.pi_state->owner == curr) {
- if (rt_mutex_trylock(&q.pi_state->pi_mutex))
- ret = 0;
+ if (q.pi_state->owner == curr &&
+ rt_mutex_trylock(&q.pi_state->pi_mutex)) {
+ ret = 0;
+ } else {
+ /*
+ * Paranoia check. If we did not take the lock
+ * in the trylock above, then we should not be
+ * the owner of the rtmutex, neither the real
+ * nor the pending one:
+ */
+ if (rt_mutex_owner(&q.pi_state->pi_mutex) == curr)
+ printk(KERN_ERR "futex_lock_pi: ret = %d "
+ "pi-mutex: %p pi-state %p\n", ret,
+ q.pi_state->pi_mutex.owner,
+ q.pi_state->owner);
+ }
- /* Unqueue and drop the lock */
- unqueue_me_pi(&q, hb);
- up_read(&curr->mm->mmap_sem);
+ /* Unqueue and drop the lock */
+ unqueue_me_pi(&q, hb);
+ up_read(&curr->mm->mmap_sem);

if (!detect && ret == -EDEADLK && 0)
force_sig(SIGKILL, current);
@@ -1331,16 +1391,18 @@ static int futex_lock_pi(u32 __user *uad
* non-atomically. Therefore, if get_user below is not
* enough, we need to handle the fault ourselves, while
* still holding the mmap_sem.
+ *
+ * ... and hb->lock. :-)--ANK
*/
+ queue_unlock(&q, hb);
+
if (attempt++) {
- if (futex_handle_fault((unsigned long)uaddr, attempt)) {
- ret = -EFAULT;
- goto out_unlock_release_sem;
- }
- goto retry_locked;
+ ret = futex_handle_fault((unsigned long)uaddr, attempt);
+ if (ret)
+ goto out_release_sem;
```

[patch 05/26] pi-futex: Fix exit races and locking problems

```
+ goto retry_unlocked;
}

- queue_unlock(&q, hb);
up_read(&curr->mm->mmap_sem);

ret = get_user(uval, uaddr);
@@ -1382,9 +1444,9 @@ retry:
goto out;

hb = hash_futex(&key);
+retry_unlocked:
spin_lock(&hb->lock);

-retry_locked:
/*
 * To avoid races, try to do the TID -> 0 atomic transition
 * again. If it succeeds then we can return without waking
@@ -1446,16 +1508,17 @@ pi_faulted:
 * non-atomically. Therefore, if get_user below is not
 * enough, we need to handle the fault ourselves, while
 * still holding the mmap_sem.
+ *
+ * ... and hb->lock. :-) --ANK
*/
+ spin_unlock(&hb->lock);
+
if (attempt++) {
- if (futex_handle_fault((unsigned long)uaddr, attempt)) {
- ret = -EFAULT;
- goto out_unlock;
- }
- goto retry_locked;
+ ret = futex_handle_fault((unsigned long)uaddr, attempt);
+ if (ret)
+ goto out;
+ goto retry_unlocked;
}
-
- spin_unlock(&hb->lock);
up_read(&current->mm->mmap_sem);

ret = get_user(uval, uaddr);
--- linux-2.6.21.6.orig/include/linux/sched.h
+++ linux-2.6.21.6/include/linux/sched.h
@@ -1138,6 +1138,7 @@ static inline void put_task_struct(struc
/* Not implemented yet, only for 486*/
#define PF_STARTING 0x00000002 /* being created */
#define PF_EXITING 0x00000004 /* getting shut down */
+ #define PF_EXITPIDONE 0x00000008 /* pi exit done on shut down */
#define PF_FORKNOEXEC 0x00000040 /* forked but didn't exec */
```

[patch 05/26] pi-futex: Fix exit races and locking problems

```
#define PF_SUPERPRIV 0x00000100 /* used super-user privileges */
#define PF_DUMPCORE 0x00000200 /* dumped core */
--- linux-2.6.21.6.orig/kernel/exit.c
+++ linux-2.6.21.6/kernel/exit.c
@@ -884,13 +884,29 @@ fastcall NORET_TYPE void do_exit(long co
if (unlikely(tsk->flags & PF_EXITING)) {
printk(KERN_ALERT
"Fixing recursive fault but reboot is needed!\n");
+ /*
+ * We can do this unlocked here. The futex code uses
+ * this flag just to verify whether the pi state
+ * cleanup has been done or not. In the worst case it
+ * loops once more. We pretend that the cleanup was
+ * done as there is no way to return. Either the
+ * OWNER_DIED bit is set by now or we push the blocked
+ * task into the wait for ever nirwana as well.
+ */
+ tsk->flags |= PF_EXITPIDONE;
if (tsk->io_context)
exit_io_context();
set_current_state(TASK_UNINTERRUPTIBLE);
schedule();
}

+ /*
+ * tsk->flags are checked in the futex code to protect against
+ * an exiting task cleaning up the robust pi futexes.
+ */
+ spin_lock_irq(&tsk->pi_lock);
tsk->flags |= PF_EXITING;
+ spin_unlock_irq(&tsk->pi_lock);

if (unlikely(in_atomic()))
printk(KERN_INFO "note: %s[%d] exited with preempt_count %d\n",
@@ -957,6 +973,12 @@ fastcall NORET_TYPE void do_exit(long co
* Make sure we are holding no locks:
*/
debug_check_no_locks_held(tsk);
+ /*
+ * We can do this unlocked here. The futex code uses this flag
+ * just to verify whether the pi state cleanup has been done
+ * or not. In the worst case it loops once more.
+ */
+ tsk->flags |= PF_EXITPIDONE;

if (tsk->io_context)
exit_io_context();

--
-
```

To unsubscribe from this list: send the line "unsubscribe linux-kernel" in

[patch 05/26] pi-futex: Fix exit races and locking problems

the body of a message to majordomo@xxxxxxxxxxxxxxxx

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

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