The robust futex ABI

Author: Started by Paul Jackson <pj@sgi.com>

Robust_futexes provide a mechanism that is used in addition to normal futexes, for kernel assist of cleanup of held locks on task exit.

The interesting data as to what futexes a thread is holding is kept on a linked list in user space, where it can be updated efficiently as locks are taken and dropped, without kernel intervention. The only additional kernel intervention required for robust_futexes above and beyond what is required for futexes is:

1. a one time call, per thread, to tell the kernel where its list of held robust_futexes begins, and
2. internal kernel code at exit, to handle any listed locks held by the exiting thread.

The existing normal futexes already provide a "Fast Userspace Locking" mechanism, which handles uncontested locking without needing a system call, and handles contested locking by maintaining a list of waiting threads in the kernel. Options on the sys_futex(2) system call support waiting on a particular futex, and waking up the next waiter on a particular futex.

For robust_futexes to work, the user code (typically in a library such as glibc linked with the application) has to manage and place the necessary list elements exactly as the kernel expects them. If it fails to do so, then improperly listed locks will not be cleaned up on exit, probably causing deadlock or other such failure of the other threads waiting on the same locks.

A thread that anticipates possibly using robust_futexes should first issue the system call:

```
asmlinkage long
sys_set_robust_list(struct robust_list_head __user *head, size_t len);
```

The pointer 'head' points to a structure in the threads address space consisting of three words. Each word is 32 bits on 32 bit arch's, or 64 bits on 64 bit arch's, and local byte order. Each thread should have its own thread private 'head'.

If a thread is running in 32 bit compatibility mode on a 64 native arch kernel, then it can actually have two such structures - one using 32 bit words for 32 bit compatibility mode, and one using 64 bit words for 64 bit native mode. The kernel, if it is a 64 bit kernel supporting 32 bit compatibility mode, will attempt to process both lists on each task exit, if the corresponding sys_set_robust_list() call has been made to setup that list.

> The first word in the memory structure at 'head' contains a pointer to a single linked list of 'lock entries', one per lock, as described below. If the list is empty, the pointer will point to itself, 'head'. The last 'lock entry' points back to the 'head'.
>
> The second word, called 'offset', specifies the offset from the address of the associated 'lock entry', plus or minus, of what will be called the 'lock word', from that 'lock entry'. The 'lock word' is always a 32 bit word, unlike the other words above. The 'lock word' holds 2 flag bits in the upper 2 bits, and the thread id (TID) of the thread holding the lock in the bottom 30 bits. See further below for a description of the flag bits.
>
> The third word, called 'list_op_pending', contains transient copy of the address of the 'lock entry', during list insertion and removal, and is needed to correctly resolve races should a thread exit while in the middle of a locking or unlocking operation.

Each 'lock entry' on the single linked list starting at 'head' consists of just a single word, pointing to the next 'lock entry', or back to 'head' if there are no more entries. In addition, nearby to each 'lock entry', at an offset from the 'lock entry' specified by the 'offset' word, is one 'lock word'.

The 'lock word' is always 32 bits, and is intended to be the same 32 bit lock variable used by the futex mechanism, in conjunction with robust_futexes. The kernel will only be able to wakeup the next thread waiting for a lock on a threads exit if that next thread used the futex mechanism to register the address of that 'lock word' with the kernel.

For each futex lock currently held by a thread, if it wants this robust_futex support for exit cleanup of that lock, it should have one 'lock entry' on this list, with its associated 'lock word' at the specified 'offset'. Should a thread die while holding any such locks, the kernel will walk this list, mark any such locks with a bit indicating their holder died, and wakeup the next thread waiting for that lock using the futex mechanism.

When a thread has invoked the above system call to indicate it anticipates using robust_futexes, the kernel stores the passed in 'head' pointer for that task. The task may retrieve that value later on by using the system call:

```
asmlinkage long
sys_get_robust_list(int pid, struct robust_list_head __user **head_ptr,
                    size_t __user *len_ptr);
```

It is anticipated that threads will use robust_futexes embedded in larger, user level locking structures, one per lock. The kernel robust_futex mechanism doesn't care what else is in that structure, so long as the 'offset' to the 'lock word' is the same for all robust_futexes used by that thread. The thread should link those locks it currently holds using the 'lock entry' pointers. It may also have other links between the locks, such as the reverse side of a double linked list, but that doesn't matter to the kernel.

By keeping its locks linked this way, on a list starting with a 'head' pointer known to the kernel, the kernel can provide to a thread the essential service available for robust_futexes, which is to help clean up locks held at the time of (a perhaps unexpectedly) exit.

Actual locking and unlocking, during normal operations, is handled entirely by user level code in the contending threads, and by the existing futex mechanism to wait for, and wakeup, locks. The kernels only essential involvement in robust_futexes is to remember where the list 'head' is, and to walk the list on thread exit, handling locks still held by the departing thread, as described below.

There may exist thousands of futex lock structures in a threads shared memory, on various data structures, at a given point in time. Only those lock structures for locks currently held by that thread should be on that thread's robust_futex linked lock list a given time.

A given futex lock structure in a user shared memory region may be held at different times by any of the threads with access to that region. The thread currently holding such a lock, if any, is marked with the threads TID in the lower 30 bits of the 'lock word'.

When adding or removing a lock from its list of held locks, in order for the kernel to correctly handle lock cleanup regardless of when the task exits (perhaps it gets an unexpected signal 9 in the middle of manipulating this list), the user code must observe the following protocol on 'lock entry' insertion and removal:

On insertion:

1. set the 'list_op_pending' word to the address of the 'lock entry' to be inserted,
2. acquire the futex lock,
3. add the lock entry, with its thread id (TID) in the bottom 30 bits of the 'lock word', to the linked list starting at 'head', and
4. clear the 'list_op_pending' word.

On removal:

1. set the 'list_op_pending' word to the address of the 'lock entry' to be removed,
2. remove the lock entry for this lock from the 'head' list,
3. release the futex lock, and
4. clear the 'lock_op_pending' word.

On exit, the kernel will consider the address stored in 'list_op_pending' and the address of each 'lock word' found by walking the list starting at 'head'. For each such address, if the bottom 30 bits of the 'lock word' at offset 'offset' from that address equals the exiting threads TID, then the kernel will do two things:

1. if bit 31 (0x80000000) is set in that word, then attempt a futex wakeup on that address, which will waken the next thread that has used to the futex mechanism to wait on that address, and
2. atomically set bit 30 (0x40000000) in the 'lock word'.

In the above, bit 31 was set by futex waiters on that lock to indicate they were waiting, and bit 30 is set by the kernel to indicate that the lock owner died holding the lock.

The kernel exit code will silently stop scanning the list further if at any point:

1. the 'head' pointer or an subsequent linked list pointer is not a valid address of a user space word
2. the calculated location of the 'lock word' (address plus 'offset') is not the valid address of a 32 bit user space word
3. if the list contains more than 1 million (subject to future kernel configuration changes) elements.

When the kernel sees a list entry whose 'lock word' doesn't have the current threads TID in the lower 30 bits, it does nothing with that entry, and goes on to the next entry.