A description of what robust futexes are

Started by: Ingo Molnar <mingo@redhat.com>

Background

what are robust futexes? To answer that, we first need to understand what futexes are: normal futexes are special types of locks that in the noncontended case can be acquired/released from userspace without having to enter the kernel.

A futex is in essence a user-space address, e.g. a 32-bit lock variable field. If userspace notices contention (the lock is already owned and someone else wants to grab it too) then the lock is marked with a value that says "there's a waiter pending", and the sys_futex(FUTEX_WAIT) syscall is used to wait for the other guy to release it. The kernel creates a 'futex queue' internally, so that it can later on match up the waiter with the waker - without them having to know about each other. When the owner thread releases the futex, it notices (via the variable value) that there were waiter(s) pending, and does the sys_futex(FUTEX_WAKE) syscall to wake them up. Once all waiters have taken and released the lock, the futex is again back to 'uncontended' state, and there's no in-kernel state associated with it. The kernel completely forgets that there ever was a futex at that address. This method makes futexes very lightweight and scalable.

"Robustness" is about dealing with crashes while holding a lock: if a process exits prematurely while holding a pthread_mutex_t lock that is also shared with some other process (e.g. yum segfaults while holding a pthread_mutex_t, or yum is kill -9-ed), then waiters for that lock need to be notified that the last owner of the lock exited in some irregular way.

To solve such types of problems, "robust mutex" userspace APIs were created: pthread_mutex_lock() returns an error value if the owner exits prematurely - and the new owner can decide whether the data protected by the lock can be recovered safely.

There is a big conceptual problem with futex based mutexes though: it is the kernel that destroys the owner task (e.g. due to a SEGFAULT), but the kernel cannot help with the cleanup: if there is no 'futex queue' (and in most cases there is none, futexes being fast lightweight locks) then the kernel has no information to clean up after the held lock! Userspace has no chance to clean up after the lock either - userspace is the one that crashes, so it has no opportunity to clean up. Catch-22.

In practice, when e.g. yum is kill -9-ed (or segfaults), a system reboot is needed to release that futex based lock. This is one of the leading bugreports against yum.

To solve this problem, the traditional approach was to extend the vma (virtual memory area descriptor) concept to have a notion of 'pending robust futexes attached to this area'. This approach requires 3 new syscall variants to sys_futex(): FUTEX_REGISTER, FUTEX_DEREGISTER and FUTEX_RECOVER. At do_exit() time, all vmas are searched to see whether they have a robust_head set. This approach has two fundamental problems left:

- it has quite complex locking and race scenarios. The vma-based approach had been pending for years, but they are still not completely reliable.
- they have to scan _every_ vma at sys_exit() time, per thread!

The second disadvantage is a real killer: pthread_exit() takes around 1 microsecond on Linux, but with thousands (or tens of thousands) of vmas every pthread_exit() takes a millisecond or more, also totally destroying the CPU's L1 and L2 caches!

This is very much noticeable even for normal process sys_exit_group() calls: the kernel has to do the vma scanning unconditionally! (this is because the kernel has no knowledge about how many robust futexes there are to be cleaned up, because a robust futex might have been registered in another task, and the futex variable might have been simply mmap()-ed into this process's address space).

This huge overhead forced the creation of CONFIG_FUTEX_ROBUST so that normal kernels can turn it off, but worse than that: the overhead makes robust futexes impractical for any type of generic Linux distribution.

So something had to be done.

New approach to robust futexes

At the heart of this new approach there is a per-thread private list of robust locks that userspace is holding (maintained by glibc) - which userspace list is registered with the kernel via a new syscall [this registration happens at most once per thread lifetime]. At do_exit() time, the kernel checks this user-space list: are there any robust futex locks to be cleaned up?

In the common case, at do_exit() time, there is no list registered, so the cost of robust futexes is just a simple current->robust_list != NULL comparison. If the thread has registered a list, then normally the list is empty. If the thread/process crashed or terminated in some incorrect way then the list might be non-empty: in this case the kernel carefully walks the list [not trusting it], and marks all locks that are owned by this thread with the FUTEX_OWNER_DIED bit, and wakes up one waiter (if any).

The list is guaranteed to be private and per-thread at do_exit() time, so it can be accessed by the kernel in a lockless way.

There is one race possible though: since adding to and removing from the list is done after the futex is acquired by glibc, there is a few instructions window for the thread (or process) to die there, leaving the futex hung. To protect against this possibility, userspace (glibc) also maintains a simple per-thread 'list_op_pending' field, to allow the kernel to clean up if the thread dies after acquiring the

lock, but just before it could have added itself to the list. Glibc sets this list_op_pending field before it tries to acquire the futex, and clears it after the list-add (or list-remove) has finished.

That's all that is needed - all the rest of robust-futex cleanup is done in userspace [just like with the previous patches].

Ulrich Drepper has implemented the necessary glibc support for this new mechanism, which fully enables robust mutexes.

Key differences of this userspace-list based approach, compared to the vma based method:

- it's much, much faster: at thread exit time, there's no need to loop over every vma (!), which the VM-based method has to do. Only a very simple 'is the list empty' op is done.
- no VM changes are needed - 'struct address_space' is left alone.
- no registration of individual locks is needed: robust mutexes don't need any extra per-lock syscalls. Robust mutexes thus become a very lightweight primitive - so they don't force the application designer to do a hard choice between performance and robustness - robust mutexes are just as fast.
- no per-lock kernel allocation happens.
- no resource limits are needed.
- no kernel-space recovery call (FUTEX_RECOVER) is needed.
- the implementation and the locking is "obvious", and there are no interactions with the VM.

Performance

I have benchmarked the time needed for the kernel to process a list of 1 million (!) held locks, using the new method [on a 2GHz CPU]:

- with FUTEX_WAIT set [contended mutex]: 130 msecs
- without FUTEX_WAIT set [uncontended mutex]: 30 msecs

I have also measured an approach where glibc does the lock notification [which it currently does for !pshared robust mutexes], and that took 256 msecs - clearly slower, due to the 1 million FUTEX_WAKE syscalls userspace had to do.

(1 million held locks are unheard of - we expect at most a handful of locks to be held at a time. Nevertheless it's nice to know that this approach scales nicely.)

Implementation details

The patch adds two new syscalls: one to register the userspace list, and one to query the registered list pointer:

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

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

List registration is very fast: the pointer is simply stored in current->robust_list. [Note that in the future, if robust futexes become widespread, we could extend sys_clone() to register a robust-list head for new threads, without the need of another syscall.]

So there is virtually zero overhead for tasks not using robust futexes, and even for robust futex users, there is only one extra syscall per thread lifetime, and the cleanup operation, if it happens, is fast and straightforward. The kernel doesn't have any internal distinction between robust and normal futexes.

If a futex is found to be held at exit time, the kernel sets the following bit of the futex word:

```
#define FUTEX_OWNER_DIED        0x40000000
```

and wakes up the next futex waiter (if any). User-space does the rest of the cleanup.

Otherwise, robust futexes are acquired by glibc by putting the TID into the futex field atomically. Waiters set the FUTEX_WAITERS bit:

```
#define FUTEX_WAITERS           0x80000000
```

and the remaining bits are for the TID.

Testing, architecture support

I've tested the new syscalls on x86 and x86_64, and have made sure the parsing of the userspace list is robust [;-)] even if the list is deliberately corrupted.

i386 and x86_64 syscalls are wired up at the moment, and Ulrich has tested the new glibc code (on x86_64 and i386), and it works for his robust-mutex testcases.

All other architectures should build just fine too - but they won't have the new syscalls yet.

Architectures need to implement the new futex_atomic_cmpxchg_inatomic() inline function before writing up the syscalls.