Proper Locking Under a Preemptible Kernel: Keeping Kernel Code Preempt-Safe

Author: Robert Love <rml@tech9.net>

Introduction

A preemptible kernel creates new locking issues. The issues are the same as those under SMP: concurrency and reentrancy. Thankfully, the Linux preemptible kernel model leverages existing SMP locking mechanisms. Thus, the kernel requires explicit additional locking for very few additional situations.

This document is for all kernel hackers. Developing code in the kernel requires protecting these situations.

RULE #1: Per-CPU data structures need explicit protection

Two similar problems arise. An example code snippet:

```
struct this_needs_locking tux[NR_CPUS];
tux[smp_processor_id()] = some_value;
/* task is preempted here... */
something = tux[smp_processor_id()];
```

First, since the data is per-CPU, it may not have explicit SMP locking, but require it otherwise. Second, when a preempted task is finally rescheduled, the previous value of smp_processor_id may not equal the current. You must protect these situations by disabling preemption around them.

You can also use put_cpu() and get_cpu(), which will disable preemption.

RULE #2: CPU state must be protected.

Under preemption, the state of the CPU must be protected. This is arch- dependent, but includes CPU structures and state not preserved over a context switch. For example, on x86, entering and exiting FPU mode is now a critical section that must occur while preemption is disabled. Think what would happen if the kernel is executing a floating-point instruction and is then preempted. Remember, the kernel does not save FPU state except for user tasks. Therefore, upon preemption, the FPU registers will be sold to the lowest bidder. Thus, preemption must be disabled around such regions.

Note, some FPU functions are already explicitly preempt safe. For example, kernel_fpu_begin and kernel_fpu_end will disable and enable preemption.

RULE #3: Lock acquire and release must be performed by same task

A lock acquired in one task must be released by the same task. This means you can't do oddball things like acquire a lock and go off to play while another task releases it. If you want to do something like this, acquire and release the task in the same code path and have the caller wait on an event by the other task.

Solution

Data protection under preemption is achieved by disabling preemption for the duration of the critical region.

```
preempt_enable()              decrement the preempt counter
preempt_disable()             increment the preempt counter
preempt_enable_no_resched()   decrement, but do not immediately preempt
preempt_check_resched()       if needed, reschedule
preempt_count()               return the preempt counter
```

The functions are nestable. In other words, you can call preempt_disable n-times in a code path, and preemption will not be reenabled until the n-th call to preempt_enable. The preempt statements define to nothing if preemption is not enabled.

Note that you do not need to explicitly prevent preemption if you are holding any locks or interrupts are disabled, since preemption is implicitly disabled in those cases.

But keep in mind that 'irqs disabled' is a fundamentally unsafe way of disabling preemption - any cond_resched() or cond_resched_lock() might trigger a reschedule if the preempt count is 0. A simple printk() might trigger a reschedule. So use this implicit preemption-disabling property only if you know that the affected codepath does not do any of this. Best policy is to use this only for small, atomic code that you wrote and which calls no complex functions.

Example:

```
cpucache_t *cc; /* this is per-CPU */
preempt_disable();
cc = cc_data(searchp);
```

```
if (cc && cc->avail) {
        __free_block(searchp, cc_entry(cc), cc->avail);
        cc->avail = 0;
}
preempt_enable();
return 0;
```

Notice how the preemption statements must encompass every reference of the critical variables. Another example:

```
int buf[NR_CPUS];
set_cpu_val(buf);
if (buf[smp_processor_id()] == -1) printf(KERN_INFO "wee!\n");
spin_lock(&buf_lock);
/* ... */
```

This code is not preempt-safe, but see how easily we can fix it by simply moving the spin_lock up two lines.

Preventing preemption using interrupt disabling

It is possible to prevent a preemption event using local_irq_disable and local_irq_save. Note, when doing so, you must be very careful to not cause an event that would set need_resched and result in a preemption check. When in doubt, rely on locking or explicit preemption disabling.

Note in 2.5 interrupt disabling is now only per-CPU (e.g. local).

An additional concern is proper usage of local_irq_disable and local_irq_save. These may be used to protect from preemption, however, on exit, if preemption may be enabled, a test to see if preemption is required should be done. If these are called from the spin_lock and read/write lock macros, the right thing is done. They may also be called within a spin-lock protected region, however, if they are ever called outside of this context, a test for preemption should be made. Do note that calls from interrupt context or bottom half/ tasklets are also protected by preemption locks and so may use the versions which do not check preemption.