Lecture 20: Kernel Synchronizations

Fall 2018
Jason Tang

Slides based upon Linux Device Drivers, 3rd Edition
http://lwn.net/Kernel/LDD3/

Topics

- Mutexes
- Spinlocks
- Completions
- Delaying Work
- Creating Kthreads

Kernel Threads

- Linux kernel is threaded internally (kthreads)
  - Kernel creates kthreads to run periodic maintenance tasks
  - Drivers may also spawn kthreads for their needs
- In the process table, commands surrounded by brackets are kthreads

```
$ ps auwx
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0   4452  1656 ?        Ss   19:48   0:03 /sbin/init
root         2  0.0  0.0      0     0 ?        S    19:48   0:00 [kthreadd]
root         3  0.5  0.0      0     0 ?        S    19:48   0:48 [ksoftirqd/0]
root         5  0.0  0.0      0     0 ?        S<   19:48   0:00 [kworker/0:0H]
root         7  0.1  0.0      0     0 ?        S    19:48   0:11 [rcu_sched]
root         8  0.0  0.0      0     0 ?        S    19:48   0:00 [rcu_bh]
root         9  0.0  0.0      0     0 ?        S    19:48   0:00 [migration/0]
```

Kernel Concurrency

- Just as with user space threads, **kernel can have race conditions**, and its **threads can deadlock**
- Example:
  - A driver creates the device `/dev/foo` that user programs may write to
  - When a program writes a string to `/dev/foo`, that string is copied to a global buffer (a shared resource)
  - Race condition occurs if multiple user space programs and/or threads write to `/dev/foo` simultaneously

Mutexes

```
#include <linux/mutex.h>
static DEFINE_MUTEX(foo_mutex);
…
static void foo_func(void) {
    mutex_lock(&foo_mutex);
    /* critical section */
    mutex_unlock(&foo_mutex);
}
```

- Use `DEFINE_MUTEX()` macro to declare and initialize a mutex at compile time; otherwise call `mutex_init()` during runtime to initialize it
- Once a mutex has been initialized, call `mutex_lock()` to lock it
    - If kthread cannot immediately obtain lock, it will go to sleep

Mutexes

- The **same kthread that locked mutex** must eventually call `mutex_unlock()`
  - Kernel mutexes are non-recursive
- A kthread can query the state of a mutex by calling `mutex_is_locked()`
- Call `mutex_trylock()` to try obtaining mutex
  - If another kthread else already has lock, this function returns zero
  - If no kthread has lock, this function acquires lock and returns one

Spinlocks

- In most programs (and kernel drivers), code is rarely in critical sections
- High overhead when acquiring a mutex, even when lock is available
- Often, faster to busy-wait instead of sleeping, especially on multi-processor system
- In some places, a kthread **may not sleep** because preemption has been disabled
  - Spinlock must be used here

Spinlocks

- Defined in include/linux/spinlock.h
- Similar to mutex in that it enforces mutual exclusion
- Unlike mutex, if a kthread cannot obtain a spinlock it instead busy-waits until lock is obtained
  - Kthread will not immediately context switched away (at least, not until its time slice expires)
  - **May be used in places where kthreads may not sleep**
- Most kernel drivers use spinlocks instead of mutexes

Spinlocks

```
#include <linux/spinlock.h>
static DEFINE_SPINLOCK(foo_lock);
…
static void foo_func(void) {
    spin_lock(&foo_lock);
    /* critical section */
    spin_unlock(&foo_lock);
}
```

- Use `DEFINE_SPINLOCK()` macro to declare and initialize a spinlock at compile time; otherwise call `spin_lock_init()` to initialize it at runtime
- Call `spin_lock()` to acquire lock; spinlocks are also non-recursive
- Call `spin_unlock()` to release lock

Spinlocks and Preemption

- On a single-processor system, when a spinlock is obtained the kernel may disable preemption
    - While a spinlock is held, nothing else can be in critical section anyways, so the kernel might as well keep scheduling that kthread
- To forcibly disable preemption for that CPU, call `spin_lock_irqsave()`
    - Reenable preemption by calling `spin_unlock_irqrestore()`
- **Must** use `spin_lock_irqsave()` instead of `spin_lock()` when obtaining a lock that is also used in places that cannot sleep

Kthreads and Synchronizations

- Just as in user space programs, a kthread may need to be suspended until some condition occurs
  - Example: one kthread is a producer, while another kthread is a consumer
- Use a kernel completion variable to synchronize threads
  - Declared in include/linux/completion.h
  - Unlike Pthread condition variables, completion variables need not be associated with mutexes or spinlocks **(but almost are regardless)**

Completion Variables

```
#include <linux/completion.h>
static DECLARE_COMPLETION(foo_cv);
…
static void foo_producer(void) {
    /* produce work */
    complete(&foo_cv);
}

static void foo_consumer(void) {
    wait_for_completion(&foo_cv);
    /* consume work */
}
```

- Use `DECLARE_COMPLETION()` macro to declare and initialize a completion variable at compile time; otherwise call `init_completion()` to initialize it at runtime
- Consumer suspends itself with `wait_for_completion()`; producer awakens consumer with `complete()` or `complete_all()`

Completion Variables

```
/* 'work' is a shared resource */
static int work;
…
static void foo_producer(void) {
    spin_lock(&foo_lock);
    work++;
    complete_all(&foo_cv);
    spin_unlock(&foo_lock);
}
```

```
static void foo_consumer(void) {
    spin_lock(&foo_lock);
    while (work == 0) {
        spin_unlock(&foo_lock);
        wait_for_completion(&foo_cv);
        spin_lock(&foo_lock);
    }
    work--;
    spin_unlock(&foo_lock);
}
```

- Often, consuming kthread will call `wait_for_completion()` in a loop (especially if multiple kthreads are consuming)
- Producer calls `complete_all()` to awaken all kthreads currently waiting

Reusing Completion Variables

- Completion variables are designed to be one-shot
  - After producer has called `complete()` / `complete_all()`, no further kthreads are supposed to wait on it
- Completion variable may be reused by calling `reinit_completion()` after calling `complete()` / `complete_all()`

```
static void foo_consumer(void) {
    spin_lock(&foo_lock);
    while (work == 0) {
        spin_unlock(&foo_lock);
        wait_for_completion(&foo_cv);
        spin_lock(&foo_lock);
        reinit_completion(&foo_cv);
    }
    work--;
    spin_unlock(&foo_lock);
}
```

No chance of missing a completion because consumer kthread is holding the spinlock

Kernel Time

- Time since the kernel has been booted is stored in the global counter `jiffies`, declared in include/linux/jiffies.h
- Number of jiffies per second varies by system
  - Convert from `jiffies` to one second via `HZ` macro: `jiffies / HZ`
- Example: calculate time that is 5 seconds in the future

```
unsigned long now = jiffies;
unsigned long later = now + 5 * HZ;
```

Busy-Waiting

```
unsigned long later = jiffies + 5 * HZ;
while (time_before(jiffies, later))
    cpu_relax();
```

- Kthread can busy-wait by spinning until `jiffies` counter surpasses some target value
    - Use `time_before()` macro to avoid integer overflow during comparisons
- `cpu_relax()` is architecture-dependent function that performs a no-op
    - On multiprocessor system, it may yield to other kthreads

Scheduling

```
unsigned long later = 5 * HZ;
set_current_state(TASK_INTERRUPTIBLE);
/* suspend for at least 5 seconds */
schedule_timeout(5 * HZ)
```

- Instead of busy-waiting, kthread can suspend itself for a certain amount of time by calling `schedule_timeout()`
  - Kernel scheduler will resume kthread after requested delay; no guarantee as to exact amount of time
- Prior to calling `schedule_timeout()`, kthread must call `set_current_state()`

Interruptible vs. Uninterruptible

- A sleeping kthread can be set to be interruptible or uninterruptible:
  - Interruptible: kthread should awake prematurely upon signal (e.g., SIGKILL)
  - Uninterruptible: kthread should ignore all signals
- When calling `set_current_state()`, need to specify `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE`
- Likewise, `wait_for_completion()` is uninterruptible; use `wait_for_completion_interruptible()` to return upon either `complete()` / `complete_all()` or upon receiving a signal

Creating Kthreads

```
struct task_struct *kthread_run(int (*threadfn)(void *data),
                               void *data, const char namefmt[], …);
```

- Use `kthread_run()`, declared in include/linux/kthread.h, to create a kthread and start running it
- First parameter is kthread function to run
- Second parameter is pointer to data to pass into kthread function
- Final parameters are `printf()`-style name for kthread
- Return value is a pointer to newly created **and running** kthread

Linux Kernel Pointers and Error Codes

- Kernel stores error codes within pointers via special values
- Use `IS_ERR()`, defined in include/linux/err.h, to check if a pointer indicates an error or not
  - Use `PTR_ERR()` to convert an error pointer to a numeric error value

```
static int thread_func(void *data) {
    …
}
static int __init foo_init(void) {
    struct task_struct *my_kthread;
    my_kthread = kthread_run(thread_func, NULL, "my_kthread");
    if (IS_ERR(my_kthread))
        return PTR_ERR(my_kthread);
    …
    return 0;
}
```

Kthread Scheduling

- Kthread keeps running until its thread function returns
- Unlike Pthreads, no direct equivalent to `pthread_join()`
  - If kthread is still running, call `kthread_stop()` to wait for target kthread to terminate
  - When kthread exits, kernel will immediately reap that kthread
- Kthreads are not supposed to run continuously for long periods of time
  - If lots of work needed, that code should be moved into user space