More Fun With Locking

Advanced Embedded Linux Development

with **Dan Walkes**

University of Colorado **Boulder**

Learning objectives:

Introduce Spinlocks

Locking Strategies and Lock Ordering

Spinlocks

- Can be used in code which cannot sleep
  - For instance interrupt handlers
- Higher performance than semaphores when properly used
- Conceptually: Single bit in integer value + tight loop spin
  - atomic test and set of bit
  - Waiting processors is executing a tight loop

Linux Device Drivers 3rd Edition Chapter 5
https://www.kernel.org/doc/Documentation/scheduler/completion.txt

Spinlocks

```
spin_lock_init(&data.lock);
```

```
unsigned long flags;
spin_lock_irqsave(&data.lock,flags);
data.val++;
spin_unlock_irqrestore(&data.lock,flags);
```

```
spin_lock(&data.lock);
data.val++;
spin_unlock(&data.lock);
```

- irqsave/irqrestore versions are always safe in interrupt or non-interrupt context
  - If interrupts are enabled, disables.
  - Use flags value to store whether interrupts were previously enabled/need to be re-enabled.
  - If interrupts were enabled, irqrestore re-enables.

Linux Device Drivers 3rd Edition Chapter 5
https://www.kernel.org/doc/Documentation/locking/spinlocks.txt

Spinlocks

```
spin_lock_init(&data.lock);
```

```
spin_lock(&data.lock);
data.val++;
spin_unlock(&data.lock);
```

- Doesn’t disable/re-enable interrupts
- Only safe is lock is never used in interrupt or if you know interrupts are blocked.
- What happens if an interrupt on the same CPU attempts to access &data.lock when it’s held with spin_lock?
    - deadlock

Linux Device Drivers 3rd Edition Chapter 5
https://www.kernel.org/doc/Documentation/locking/spinlocks.txt

Spinlocks and Atomic Context

- Core rule: Any code must be atomic while holding the spinlock
  - Can’t sleep
  - Can’t relinquish the processor for anything other than interrupts
- How do I know a kernel function I call doesn’t sleep?
  - Difficult to know, but many functions can
  - Most functions which might allocate memory also might sleep
- Second Rule: Must be held for minimum time possible
  - Anyone waiting for the lock is “spinning” in tight CPU wait loops

Locking Rules

- Locking Rules should not be ambiguous
- Define a lock to control access to specific data
- Design locking in from the beginning
- Clearly enforce rules to ensure nested functions don’t try to acquire the same lock.
- Write functions which assume caller has allocated the lock and document assumptions explicitly.

Lock Ordering and Multiple Locks

Thread 1

Holds lock1

```
mutex_lock(&lock1);
mutex_lock(&lock2);  ← Instruction Pointer
do_some_operation_requiring_both_locks();
mutex_unlock(&lock2);
mutex_unlock(&lock1);
```

Thread 2

Holds no locks

```
mutex_lock(&lock2);  ← Instruction Pointer
mutex_lock(&lock1);
do_some_other_operation_requiring_both_locks();
mutex_unlock(&lock1);
mutex_unlock(&lock2)
```

- What happens when Thread 2 executes the current instruction just before Thread 1?

Lock Ordering and Multiple Locks

Thread 1

Holds lock1

```
mutex_lock(&lock1);
mutex_lock(&lock2);     Instruction Pointer
do_some_operation_requiring_both_locks();
mutex_unlock(&lock2);
mutex_unlock(&lock1);
```

Thread 2

Holds lock2

```
mutex_lock(&lock2);
mutex_lock(&lock1);   Instruction Pointer
do_some_other_operation_requiring_both_locks();
mutex_unlock(&lock1);
mutex_unlock(&lock2)
```

- What happens when Thread 2 executes the current instruction just before Thread 1?
  - Deadlock

Lock Ordering and Multiple Locks

- Locks should always be acquired in the same order.
- Unfortunately lock ordering rules are typically poorly documented
  - RTSL (Read the Source Luke)
- Rules of thumb:
  - Avoid the need for using multiple locks whenever possible.
  - Obtain your driver locks before locks used in other parts of the kernel
    - Why?
      - Minimize the chance you block when holding the most popular lock.
- Always hold semaphores before spinlocks
  - Why?
    - Semaphores lock steps may sleep and you can’t sleep when holding a spinlock

Linux Device Drivers 3rd Edition Chapter 5

Alternatives to Locking

- Atomic Variables and Bit Operations
  - Guaranteed atomic types on all architectures
- Lock-Free algorithms
  - Circular buffer with exactly 2 threads and atomic count values
  - Read-Copy-Update (RCU)
    - Old copies remain valid, cleanup happens when references are released.