

CS510 Concurrent Systems

Class 1a

Linux Kernel Locking Techniques

Intro to kernel locking techniques (Linux)

- ❑ Why do we need locking in the kernel?
 - Which problems are we trying to solve?
- ❑ What implementation choices do we have?
 - Is there a one-size-fits-all solution?

How does concurrency arise?

- ❑ True concurrency
 - Multiple processors execute instructions simultaneously
- ❑ Pseudo concurrency
 - Instructions of multiple execution sequences are interleaved

Sources of pseudo concurrency

- ❑ Software-based preemption
 - Voluntary preemption (sleep/yield)
 - Involuntary preemption (preemptible kernel)
 - *Scheduler switches threads regardless of whether they are running in user or kernel mode*
 - *The instructions of multiple threads running in kernel mode are interleaved*
- ❑ Hardware preemption
 - Interrupt/trap/fault/exception handlers can start executing at any time
- ❑ Reentrancy
 - A function calls itself

Critical sections

- ❑ Sections of code that are subject to concurrent execution in which at least one execution path modifies shared data
- ❑ Locking can be used to provide mutually exclusive access to critical sections
- ❑ Various locking primitives exist in Linux
 - Linux is a symmetric multiprocessing (SMP) preemptible kernel

Atomic operators

- ❑ Simplest synchronization primitives
 - Primitive operations that are indivisible
- ❑ Two types
 - methods that operate on integers
 - methods that operate on bits
- ❑ Implementation
 - Assembly language sequences that use the atomic read-modify-write instructions of the underlying CPU architecture

Atomic integer operators

```
atomic_t v;
atomic_set(&v, 5);           /* v = 5 (atomically) */
atomic_add(3, &v);          /* v = v + 3 (atomically) */
atomic_dec(&v);              /* v = v - 1 (atomically) */

printf("This will print 7: %d\n", atomic_read(&v));
```

Beware:

- Can only pass `atomic_t` to an atomic operator
- `atomic_add(3,&v);` and
{
 `atomic_add(1,&v);`
 `atomic_add1(2,&v);`
}
are not same!

Spin locks

- ❑ Mutual exclusion for larger (than one operator) critical sections requires additional support
- ❑ Spin locks
 - Single holder locks
 - When lock is unavailable, acquiring process keeps trying

Basic use of spin locks

```
spinlock_t mr_lock = SPIN_LOCK_UNLOCKED;  
spin_lock(&mr_lock);           /* critical section ... */  
spin_unlock(&mr_lock);
```

spin_lock()

- Acquires the spinlock using atomic instructions required for SMP

spin_unlock()

- Releases the spinlock

What if the spin lock holder is interrupted?

```
spinlock_t mr_lock = SPIN_LOCK_UNLOCKED;
unsigned long flags;
spin_lock_irqsave(&mr_lock, flags); /* critical section ... */
spin_unlock_irqrestore(&mr_lock, flags);
```

spin_lock_irqsave()

- disables interrupts locally
- acquires the spinlock using instructions required for SMP

spin_unlock_irqrestore()

- Restores interrupts to the state they were in when the lock was acquired

What if we're on a uniprocessor?

Previous code compiles to:

```
unsigned long flags;  
save_flags(flags);      /* save previous CPU state */  
cli();                  /* disable interrupts */  
...                     /* critical section ... */  
restore_flags(flags);   /* restore previous CPU state */
```

Hmm, why not just use:

```
cli();                  /* disable interrupts */  
...  
sti();                  /* enable interrupts */
```

Dealing with interrupt context

- ❑ Need to know if data being protected is accessed in interrupt context or just normal kernel context
 - Interrupt handlers should not block!
 - Interrupted spin lock holders should not be delayed
 - *Use appropriate primitives to manage hardware or software preemption*
- ❑ Need to know if interrupts were already enabled or disabled
 - Use appropriate primitives to save and restore CPU state

Bottom halves, softirqs and tasklets

- ❑ Softirqs, tasklets and BHs are deferrable functions
 - think of them as delayed interrupt handling that is scheduled
- ❑ Softirqs - the basic building block
 - Statically allocated
 - can not be interrupted by another softirq on the same CPU
 - *non-preemptive scheduling of softirqs*
 - softirqs of the same type can run concurrently on different CPUs
 - *synchronize with each other using spin-locks*
- ❑ Tasklets - built on softirqs
 - dynamically allocated
 - can not be interrupted by another tasklet on the same CPU
 - tasklets of the same type can not run concurrently on different CPUs
- ❑ BHs - built on softirqs (static, not concurrent)

Spin locks and deferred functions

- ❑ `spin_lock_bh()`
 - implements the standard spinlock
 - disables softirqs
 - needed for code outside a softirq that manipulates data also used inside a softirq
- ❑ `spin_unlock_bh()`
 - Releases the spinlock
 - Enables softirqs

Spin lock rules

- ❑ Do not try to re-acquire a spinlock you already hold!
 - it leads to self deadlock!
- ❑ Spinlocks should not be held for a long time
 - Excessive spinning wastes CPU cycles!
 - What is "a long time"?
- ❑ Do not sleep while holding a spinlock!
 - Someone spinning waiting for you will waste a lot of CPU
 - never call any function that touches user memory, allocates memory, calls a semaphore function or any of the schedule functions while holding a spinlock! All these can block.

Semaphores

- ❑ Semaphores are locks that are safe to hold for longer periods of time
 - contention for semaphores causes blocking not spinning
 - should not be used for short duration critical sections!
 - *Why?*
 - safe to sleep with!
 - *Can be used to synchronize with user contexts that might block or be preempted*
- ❑ Semaphores can allow concurrency for more than one process at a time, if necessary

Semaphore implementation

- Implemented as a wait queue and a usage count
 - wait queue: list of processes blocking on the semaphore
 - usage count: number of concurrently allowed holders
 - *if negative, the semaphore is unavailable, and*
 - *absolute value of usage count is the number of processes currently on the wait queue*
 - *if initialized to 1, the semaphore is a mutex*

Semaphore operations

- Down()
 - attempts to acquire the semaphore by decrementing the usage count and testing if its negative
 - *blocks if usage count is negative*
- Up()
 - releases the semaphore by incrementing the usage count and waking up one or more tasks blocked on it

Can you be interrupted when blocked?

- ❑ `down_interruptible()`
 - Returns `-EINTR` if signal received while blocked
 - Returns `0` on success
- ❑ `down_trylock()`
 - attempts to acquire the semaphore
 - on failure it returns nonzero instead of blocking

Reader/writer Locks

- No need to synchronize concurrent readers unless a writer is present
 - reader/writer locks allow multiple concurrent readers but only a single writer (with no concurrent readers)
- Both spin locks and semaphores have reader/writer variants

Reader/writer spin locks (rwlock)

```
rwlock_t mr_rwlock = RW_LOCK_UNLOCKED;

read_lock(&mr_rwlock);    /* critical section (read only) ... */
read_unlock(&mr_rwlock);

write_lock(&mr_rwlock);   /* critical section (read and write) ... */
write_unlock(&mr_rwlock);
```

Reader/writer semaphores (rw_semaphore)

```
struct rw_semaphore mr_rwsem;
init_rwsem(&mr_rwsem);

down_read(&mr_rwsem); /* critical region (read only) ... */
up_read(&mr_rwsem);

down_write(&mr_rwsem); /* critical region (read and write) ... */
up_write(&mr_rwsem);
```

Reader/writer lock warnings

- reader locks cannot be automatically upgraded to the writer variant
 - attempting to acquire exclusive access while holding reader access will deadlock!
 - if you know you will need to write eventually
 - *obtain the writer variant of the lock from the beginning*
 - *or, release the reader lock and re-acquire the lock as a writer*

Big reader locks (br_lock)

- ❑ Specialized form of reader/writer lock
 - very fast to acquire for reading
 - very slow to acquire for writing
 - good for read-mostly scenarios
- ❑ Implemented using per-CPU locks
 - readers acquire their own CPU's lock
 - writers must acquire all CPUs' locks

Big kernel lock (BKL)

- ❑ A global kernel lock - `kernel_flag`
 - used to be the only SMP lock
 - mostly replaced with fine-grain localized locks
- ❑ Implemented as a recursive spin lock
 - Reacquiring it when held will not deadlock
- ❑ Usage ... but don't ;)

```
lock_kernel();  
/* critical region ... */  
unlock_kernel();
```

Preemptible kernel issues

- Have to be careful of legacy code that assumes per-CPU data is implicitly protected from preemption
 - May need to use new `preempt_disable()` and `preempt_enable()` calls
 - Calls are nestable
 - for each n `preempt_disable()` calls, preemption will not be re-enabled until the n th `preempt_enable()` call

Conclusions

- Wow! Why does one system need so many different ways of doing synchronization?
 - Actually, there are more ways to do synchronization in Linux, this is just "locking"

Conclusions

- ❑ One size does not fit all:
 - need to be aware of different contexts in which code executes (user, kernel, interrupt etc) and the implications this has for whether hardware or software preemption or blocking can occur
 - the cost of synchronization is important, particularly its impact on scalability
 - *You only use more than one CPU because you hope to execute faster!*
 - *Each synchronization technique makes a different performance vs. complexity trade-off*
 - Real-time characteristics and I/O are becoming more and more important in general purpose systems