CS510 Concurrent Systems

Jonathan Walpole

Portland State
UNIVERSITY

Memory Invariance Examples

```
r1 = x;
r2 = x;
Assert (r1 == r2)
```

Memory Invariance Examples

```
lock (m)
r1 = x;
r2 = x;
unlock (m)
Assert (r1 == r2)
```

Memory Invariance Examples

```
lock (m)
r1 = x;
unlock (m)
lock (m)
r2 = x;
unlock (m)
Assert (r1 == r2)
```

Memory Invariance Examples

```
lock (m)
r1 = x;
r2 = x;
wait (c,m)
unlock (m)
Assert (r1 == r2)
```

Memory Invariance Examples

```
lock (m)
r1 = x;
wait (c,m)
r2 = x;
unlock (m)
Assert (r1 == r2)
```

Memory Invariance Examples

```
lock (m)
r1 = x;
signal(c,m)
r2 = x;
unlock (m)
Assert (r1 == r2)
```

Linux Kernel Locking Techniques

Locking In The Linux Kernel

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?

Concurrency in Linux

Linux is a symmetric multiprocessing (SMP) preemptible kernel

Its has true concurrency

- Multiple processors execute instructions simultaneously

And various forms of pseudo concurrency

- Instructions of multiple execution sequences are interleaved

Sources of Pseudo Concurrency

Software-based preemption

- Voluntary preemption (sleep/yield)
- Involuntary preemption (preemptable kernel)
  - *Scheduler switches threads regardless of whether they are running in user or kernel mode*
- Solutions: don't do the former, disable preemption to prevent the latter

Hardware preemption

- Interrupt/trap/fault/exception handlers can start executing at any time
- Solution: disable interrupts
  - *what about faults and traps?*

Uniprocessor Example

```
preempt disable;
r1 = x;
r2 = x;
preempt enable;
assert (r1 == r2)
```

Uniprocessor Example

```
preempt disable;
r1 = x;
yield();
r2 = x;
preempt enable;
assert (r1 == r2)
```

Uniprocessor Example

```
interrupt disable;
r1 = x;
r2 = x;
interrupt enable;
assert (r1 == r2)
```

True Concurrency

Solutions to pseudo-concurrency do not work in the presence of true concurrency

Alternatives include atomic operators, various forms of locking, RCU, and non-blocking synchronization

Locking can be used to provide mutually exclusive access to critical sections

- Locking can not be used everywhere, i.e., interrupt handlers can't block
- Locking primitives must support coexistence with various solutions for pseudo concurrency, i.e., we need hybrid primitives

Multiprocessor Example

```
interrupt disable;
r1 = x;
r2 = x;
interrupt enable;
assert (r1 == r2)
```

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

Memory Invariance Example

```
r1 = atomic read x;
r2 = atomic read x;
assert (r1 == r2)
```

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 the same! … Why?

Spin Locks

Mutual exclusion for larger (than one operator) critical sections requires additional support

Spin locks are one possibility

- Single holder locks
- When lock is unavailable, the 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

Spin Locks and Interrupts

Interrupting a spin lock holder may cause problems

- Spin lock holder is delayed, so is every thread spin waiting for the spin lock
  - Not a big problem if interrupt handlers are short
- Interrupt handler may access the data protected by the spin-lock
  - Should the interrupt handler use the lock?
  - Can it be delayed trying to acquire a spin lock?
  - What if the lock is already held by the thread it interrupted?

Solutions

If data is only accessed in interrupt context and is local to one specific CPU we can use interrupt disabling to synchronize

- A pseudo-concurrency solution like in the uniprocessor case

If data is accessed from other CPUs we need additional synchronization

- Spin locks

Normal code (kernel context) must disable interrupts and acquire spin lock

- interrupt context code can then safely acquire the spin lock!

Spin Locks & Interrupt Disabling

Non-interrupt code acquires spin lock to synchronize with other non-interrupt code

It also disables interrupts to synchronize with local invocations of the interrupt handler

Spin Locks & Interrupt Disabling

```
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

Memory Invariance Example

```
spin_lock_irqsave(m)
r1 = x;
r2 = x;
spin_unlock_irqrestore(m)
assert (r1 == r2)
```

Memory Invariance Example

```
spin_lock(m)
r1 = x;
r2 = x;
spin_unlock(m)
assert (r1 == r2)
```

Uniprocessor Optimization

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 */
```

Why not just use:

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

Bottom Halves and Softirqs

Softirqs, tasklets and BHs are deferrable functions
- delayed interrupt handling work that is scheduled
- they can wait for a spin lock without holding up devices
- they can access non-CPU local data

Softirqs – the basic building block
- statically allocated and non-preemptively scheduled
- can not be interrupted by another softirq on the same CPU
- can run concurrently on different CPUs, and synchronize with each other using spin-locks

Bottom Halves
- built on softirqs
- can not run concurrently on different CPUs

Spin Locks & Deferred Functions

spin_lock_bh()

- Implements the standard spinlock
- Disables softirqs
- Ineeded for code outside a softirq that manipulates data also used inside a softirq
- Allows the softirq to use non-preemption only

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!

Semaphores are 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

- i.e., initialize to a value greater than 1

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
  - initialize to 1 to use the semaphore as a mutex lock

Semaphore Operations

Down()

- Attempts to acquire the semaphore by decrementing the usage count and testing if it is 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

Unblocking Unsuccessfully

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_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

```
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);
```

Memory Invariance Example

```
read_lock(m)
r1 = x;
r2 = x;
read_unlock(m)
assert (r1 == r2)
```

Upgrading Read Locks?

read_lock(m)
write_lock(m)
write_unlock(m)
read_unlock(m)

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 it as a writer
  - But bear in mind that memory may have changed when you get in!

Big Reader Locks

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

Why does this work? How does it help?

Big Kernel Lock

A global kernel lock - kernel_flag

- Used to be the only SMP lock
- Mostly replaced with fine-grain localized lock

Implemented as a recursive spin lock

- Reacquiring it when held will not deadlock

Usage … but don't! ;)

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

Preemptibility Controls

Have to be careful of legacy code that assumes per-CPU data is implicitly protected from preemption

- Legacy code assumes “non-preemption in kernel mode”
- 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 nth 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
  - Generally, you only use more than one CPU because you hope to execute faster!
  - Each synchronization technique makes a different performance vs. complexity trade-off