CS510 Concurrent Systems

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 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?*
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
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

```c
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
What if the spin lock holder is interrupted?

- Interrupting a spin lock holder may cause several 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
    - Spin locks cannot be acquired in interrupt context because this might deadlock

- Normal code (kernel context) must disable interrupts and acquire spin lock
  - Interrupt context code need not acquire spin lock
  - Assumes data is not accessed by interrupt handlers on different CPUs, i.e., interrupts are CPU-local and this is CPU-local data
Combining spin locks and interrupt disabling

- Non-interrupt code acquires spin lock to synchronize with other non-interrupt code and disables interrupts to synchronize with local invocations of the interrupt handler
Combining spin locks and 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
What if we're on a uniprocessor?

Previous code compiles to:

```c
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:

```c
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 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
  - 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!
    - Why?
  - 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 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)

```c
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/reader semaphores (rw_semaphore)

struct rw_semaphore mr_rwsem;
init_rwlock(&mr_rwlock);

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

down_write(&mr_rwlock); /* critical region (read and write) ... */
up_write(&mr_rwlock);
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
      - But bear in mind that memory may have changed when you get in!
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! ;)

```c
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
  - 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