CS510 Concurrent Systems
Class 1

Linux Kernel Locking Techniques
Intro to kernel locking techniques

- Why do we need locking in the Linux kernel?
  - Which problems are we trying to solve?

- What causes concurrent execution of kernel code?

- What solutions can we apply?
  - 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
  - Might be the same instruction stream
  - Might be different instructions that access the same data

- Pseudo concurrency
  - Instructions of multiple execution sequences are interleaved on the same processor

- Why are both potentially problematic for kernel code in a shared memory multiprocessor?
Sources of pseudo concurrency

- Software-based preemption
- Voluntary preemption
  - code calls sleep or yield or makes some other blocking call
- Involuntary preemption (preemptible kernel)
  - Scheduler switches threads
    - even if they are currently in kernel mode
  - The instructions of multiple threads running in kernel mode are interleaved
- Solutions?
Sources of pseudo concurrency

- Hardware preemption
- Interrupt/trap/fault/exception handlers can start executing at any time
  - handler instructions are interleaved with normal kernel instructions
  - handler instructions may be interleaved with themselves

- Solutions?
Critical sections

- **Critical section**
  - Code that is subject to concurrent execution, and
  - At least one execution path modifies shared data
    - Why does this matter?

- **Correctness**
  - Reasoning about correctness involves reasoning about pre and post conditions and the code's effects
  - Certain data invariants are assumed to hold prior to and on completion of the critical section
  - Reasoning about sequential execution is easier than reasoning about concurrent execution (Why???)
Critical sections and Locks

- Locking can be used to:
  - provide mutually exclusive access to critical sections
  - i.e., prevent concurrent access to memory
  - Enable sequential reasoning to be applied to the critical section
  - How can we reason about the effects of code between critical sections?

- Various locking primitives exist in Linux
  - Linux is a symmetric multiprocessing (SMP) preemptible kernel
  - So which kinds of concurrency do we have to worry about?
Atomic operators

- **Simplest synchronization primitives**
  - Primitive operations that are indivisible
  - Degenerate case of a critical section - needs no lock!

- **Two types defined by Linux kernel**
  - methods that operate on integers
  - methods that operate on bits
  - Helps make kernel more portable

- **Implementation**
  - Assembly language sequences using the “atomic” read-modify-write instructions of the CPU’s instruction set
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
  ```c
  { 
    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
  - Single holder locks
  - When lock is unavailable, the acquiring process keeps trying to acquire it
  - Busy waiting, without releasing the CPU
  - Why is this a bad idea on a uniprocessor?
  - Why might it be a good idea on a multiprocessor?
Basic use of spin locks

```c
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
```
Interaction with interrupt handlers

- What happens if the data in the critical section is accessed by interrupt handlers?
  - How can normal kernel code synchronize with interrupt handlers?
  - How can interrupt handlers synchronize with other interrupt handlers?
  - How can normal kernel code synchronize with itself?

- A note about interrupts:
  - They are local to each CPU!
Interactions with interrupt handlers

```
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 */
  ```

- Why don’t we need a spin lock?
- Why not just use:

  ```c
  cli();                /* disable interrupts */
  ...
  sti();               /* enable interrupts */
  ```
Deferred interrupt handling

- Softirqs, tasklets and BHs are deferrable functions
  - think of them as delayed interrupt handling that is scheduled
  - Scheduling is non-preemptive
  - they use a cooperative multitasking approach for synchronization

- Softirqs - the basic building block
  - Statically allocated and non-preemptively scheduled
  - can not be interrupted by another softirq on the same CPU
  - softirqs of the same type can run concurrently on different CPUs
    - use per-CPU data or synchronize with each other using spin-locks

- Tasklets - built on softirqs
  - dynamically allocated and non-preemptively scheduled
  - can not be interrupted by another tasklet on the same CPU
  - tasklets of the same type can not run concurrently on different CPUs
Spin locks and deferred functions

- Interaction with deferred interrupt handling is similar to interaction with interrupt handlers

- 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!
  - Why not?
Spin lock rules

- Do not try to re-acquire a spinlock you already hold!
  - It leads to self deadlock!
Spin lock rules

- Spinlocks should not be held for a long time
  - Why not?
Spin lock rules

- Spinlocks should not be held for a long time
  - Excessive spinning wastes CPU cycles!
Spin lock rules

- Do not sleep while holding a spinlock!
  - Why not?
Spin lock rules

- 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
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
Interaction with software interrupts

- What happens if you are signaled while blocked on a semaphore?
  - `down_interruptible()`
    - Returns -EINTR if signal received while blocked
    - Returns 0 on success

- What if you don’t want to risk blocking
  - `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/writer semaphores (rw_semaphore)

struct rw_semaphore mr_rwsem;
init_rwlock(&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
      - 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 an array of locks, one per-CPU
  - Readers acquire their own CPU's lock only
  - Writers must acquire all CPUs' locks
  - Guarantees intersection between read and write lock sets
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

- Old kernels used to assume non-preemption in kernel mode:
  - On a uniprocessor this protects all kernel data not accessed by interrupt handlers
  - On a multiprocessor this protects all per-CPU data

- Newer kernels are preemptable
  - need to use 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