EC 440 – Introduction to Operating Systems

Orran Krieger (BU)
Larry Woodman (Red Hat)

Slides from Christopher Kruegel, UCSB

Today's lecture

- The real world - deadlock and synchronization in Linux
- Review of synchronization and Deadlock
- Fun programming (at least for Orran)

Today's lecture

- Short review of synchronization
- The real world - deadlock and synchronization in Linux
- Fun programming (at least for Orran)

Review: Deadlocks

When processes try to acquire resources concurrently they may end up "stuck"

Example:

1. Process A needs P, Q
2. Process B needs Q, P
3. Process A gets P
4. Process B gets Q
5. Process A tries to get Q and blocks
6. Process B tries to get P and blocks

Review: Livelock

- Sometimes processes can never be deadlocked, but still not make progress
- Consider this algorithm, they keep running... but, it is possible for neither to ever make progress
- Lots of real examples of this in OS, e.g. with two phased locking

```
void process_A(void) {
        acquire_lock(&resource_1);
        while (try_lock(&resource_2) == FAIL) {
                release_lock(&resource_1);
                wait_fixed_time();
                acquire_lock(&resource_1);
        }
        use_both_resources( );
        release_lock(&resource_2);
        release_lock(&resource_1);
}

void process_A(void) {
        acquire_lock(&resource_2);
        while (try_lock(&resource_1) == FAIL) {
                release_lock(&resource_2);
                wait_fixed_time();
                acquire_lock(&resource_2);
        }
        use_both_resources( );
        release_lock(&resource_1);
        release_lock(&resource_2);
}
```

Review: Lock Starvation

- A process never receives the lock it is waiting for, despite the resource (repeatedly) becoming free, the resource is always allocated to another waiting process or CPU.

- For blocking locks(semaphores and mutexes) scheduler is involved so is usually a priority issue.
  - Solution usually involves scheduling priority adjustment or finer granularity locks.

- For spinlocks non-uniform memory or cache placement is usually the issue.
  - Solution usually involves ticketed spinlocks, MCS locks, per-cpu or per numa node locks.
  - Even better is lockless code/RCU

Linux Locking and Synchronization

Linux user space deadlocks

- Linux uses the "Ostrich" approach to solving user deadlocks.
  - processes with user code deadlocks and livelocks get penalized by the scheduler.
  - starved processes will get rewarded by the scheduler.
  - Otherwise if user code is “stuck” dont do anything except support SIGKILL.
  - Resources are limited to user code:
    - excessive allocation will fail
    - worse case scenario is process gets killed by kernel.

Linux kernel approach to deadlocks

- Linux takes prevention approach to kernel deadlocks, lovelocks and starvation.
  - Primary focus is on prevention and lockless algorithms(RCU) in the Linux kernel.
  - Locks have a strict hierarchy and must be taken in correct order.
  - Kernel protects itself by throttling(failure to allocate more) and ultimately by process killing.
    - Resource issues arise on kernel data structures, e.g., task table, inode table.

Locking In the Linux kernel

- semaphores/counting semaphores
  - multiple count resources blocking
  - example: free objects on a list
- mutexes
  - can be reader/writer
  - single count resources blocking
  - example: file access
- spinlocks
  - can be reader/writer
  - reader: multiple access non-blocking
  - writer: single access non-blocking
  - can not block with spinlock held
- atomic operations
  - incrementing/decrementing counters
- RCU
  - read copy update

Linux Kernel examples

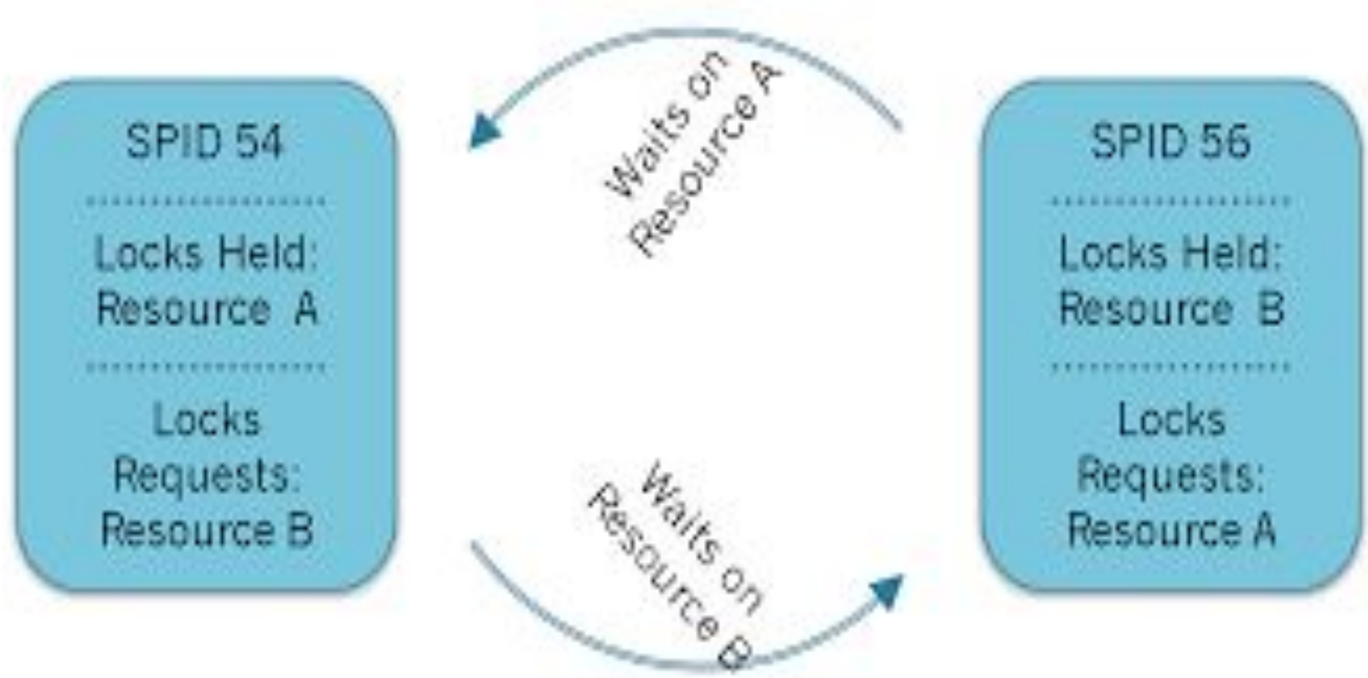
- spinlocks: non-blocking/busy-wait
  - simple spinlock: spinlock_t
    - spinlock(), spinlock_irq(), spin_trylock(), spin_unlock
  - nested spinlock: nested_spinlock_t
    - spinlock(), spinlock_irq(), spin_trylock(), spin_unlock
  - reader/writer spinlock: rw_spinlock_t
    - spin_readlock(), spin_writelock(), spinunlock()
- mutex & semaphores: blocking/waiting
  - simple mutex/semaphore: mutex_t/semaphore_t
    - down(), down_try(), up()
  - nested mutex: nested_mutex_t
    - down(), down_try(), up()
  - reader/writer mutex rw_mutex_t
    - downread(), downwrite(), up()

Detection in the Linux Kernel

- Deadlock
  - one or more processes permanently stuck on spinlock(set-and-test) without making progress.
  - default behavior: console message/panic if no reschedule in 30s and PC doesnt change.
- Livelock
  - one or more processes permanently stuck in a loop not making any progress.
  - default behavior: console message/panic if no reschedule in 60s but the PC does change.
- Starvation
  - One or more processes never getting CPU time even though they are runnable.
  - One or more CPUs stuck on a spinlock without ever acquiring it.
  - default behavior: SIGKILL if no reschedule in 120s and PC doesnt change but other CPUs get spinlock.

Deadlock

- One or more process permanently stuck is a “deadly embrace”



deadlocks In the Linux kernel

- Multi thread
  - thread A locks resource X
  - thread B lock resource Y
  - thread A attempts to lock resource Y
  - thread B attempts to lock resource X
- Single thread
  - thread A calls procedure M which locks resource X
  - procedure M calls procedure N which attempts to lock resource X
- Interrupt handling
  - thread A locks resource X
  - interrupt occurs
  - ISR attempts to lock resource X
- livelocks versus deadlocks
  - deadlock: stuck on a single lock
  - livelock: looping down and up a potentially long call chain.

Linux lock hierarchy

- Lock types in Linux kernel are ordered from 1 to N
- Low numbered/ordered locks are coarser than high numbered locks and must be acquired first

  example:
  - task_list_lock protect a list of struct_tasks
  - task_lock protects an individual struct_task
  - order of task_list_lock < order of task_lock
    a. lock(task_list_lock);
    b. lock_task(&task-being-locked);

- If you need lock A of order 1 and lock B of order 2 you must:
  – lock(A) before lock(B)
    or
  – lock(B) then trylock(A)

- You must unlock in reverse locking order
  – lock(A) lock(B) … unlock(B) unlock(A)
  – lock(B) trylock(A) … unlock(B) unlock(A)

- Well documented in ./GIT/linux/linux/Documentation/locking

Multi-thread deadlocks

- Spinlocks are ordered
- X>Y>Z
- kernel locks are ordered in terms of scope
  - X scope is a superset of Y scope
  - Y scope is a superset of Z scope
- X must be locked before Y which must be locked before Z
  - thread A locks resource X
  - thread B attempts to lock resource X
  - thread A locks resource Y
  - thread A releases locks Y then X
  - thread B is granted resource X
  - thread B locks resource Y
  - thread B releades locks Y then X

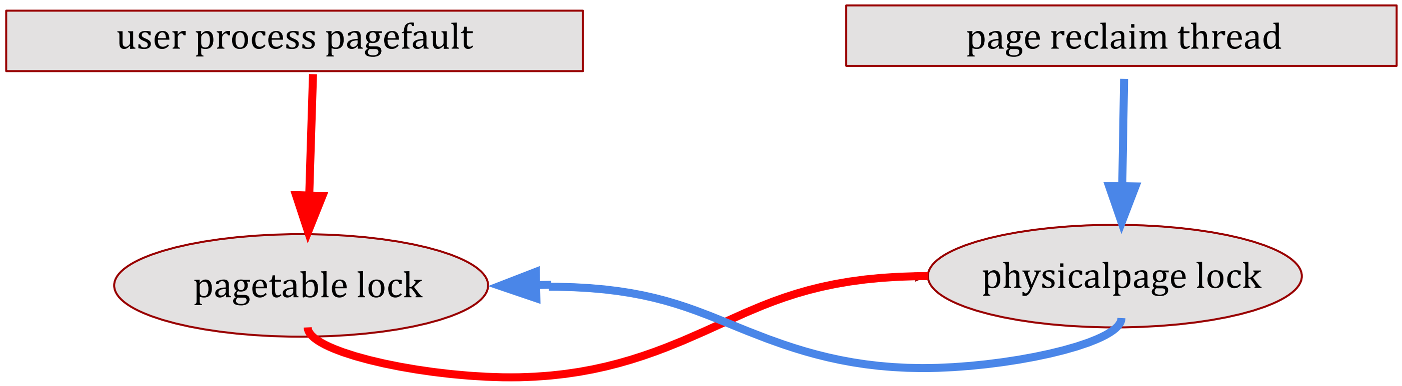
Multi-thread deadlocks

- user process 1 simple pagefault:
  1. Locate and acquire pagetable_lock
  2. acquire physical pagelock
  3. multi-threaded deadlock

- kernel page reclaim thread 2:
  1. locate physical page and acquire pagelock
  2. acquire owning process pagetable_lock
  3. multi-threaded deadlock

- Solution 1:
  - page reclaim thread does trylock(pagetable_lock)
- Solution 2:
  - Both processes/threads acquire in correct order
  1. pagetable_lock
  2. physical pagelock

Multi-thread deadlocks



Single threaded deadlocks

Thread A calls procedure M which locks resource X
procedure M calls procedure N which attempts to lock resource X

- Solution:
  - recursive locks:
    - Thread A is granted the same lock multiple times in a call chain
      - each lock attempt increments a depth count
      - each unlock decrements a depth count
    - Thread A calls M which locks recursive lock X
      - lock X is acquired, depth count incremented to 1
    - M calls N which locks recursive lock X
      - depth count is incremented to 2
    - N unlocks X
      - depth count is decremented to 1
    - N returns to M
      - depth count is decremented to 0, X is unlocked

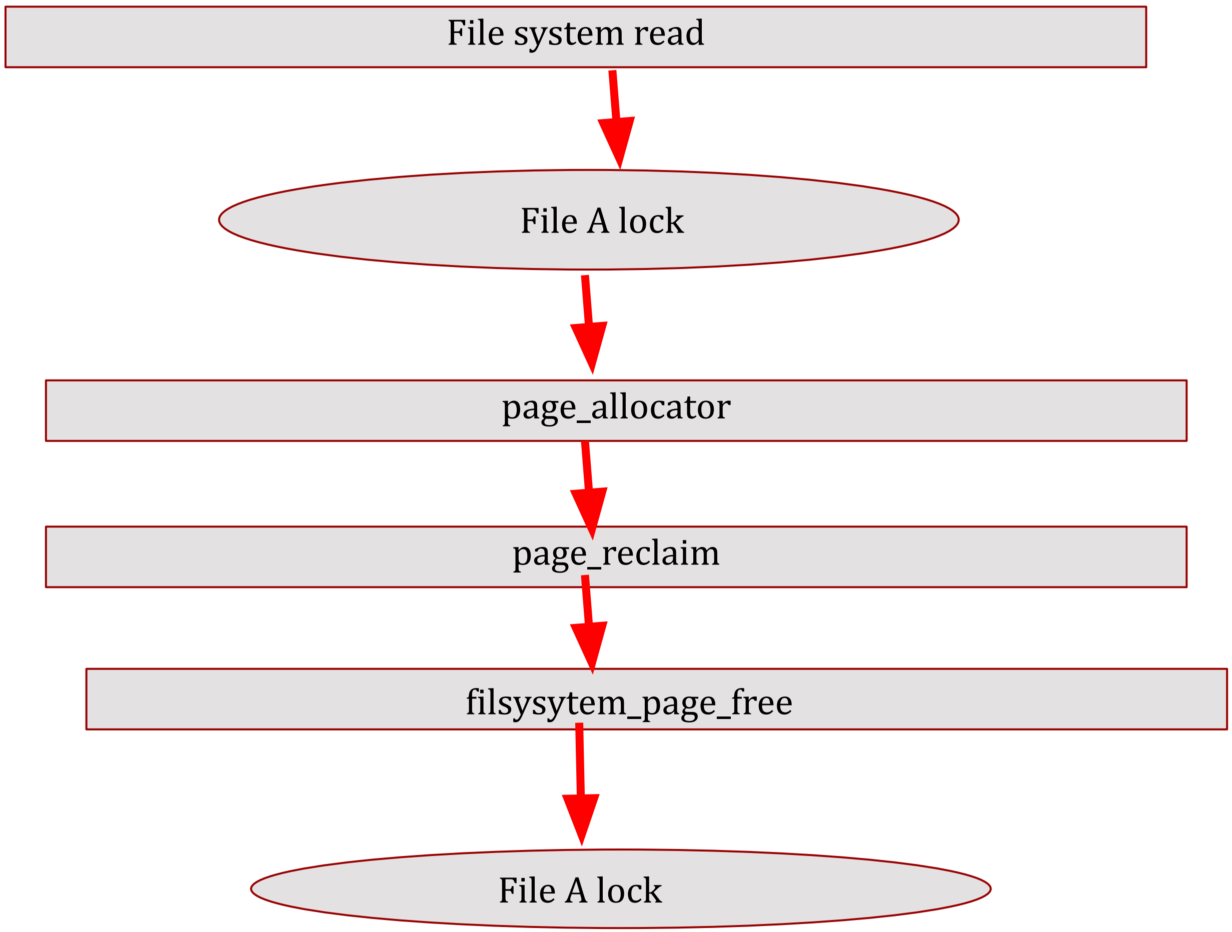
Single threaded deadlock example

1. Filesystem read operation locks file A
2. Filesystem read needs page, calls page allocator
3. Page allocator calls page reclaim code
4. Page reclaim code calls filesystem to free a page in file A
5. Filesystem freeing routine locks file A

6. Single threaded DEADLOCK

Solution: make filelock a recursive/nested lock.

Single threaded deadlock example



Interrupt deadlocks

Interrupt handling:

thread A locks resource X

interrupt occurs

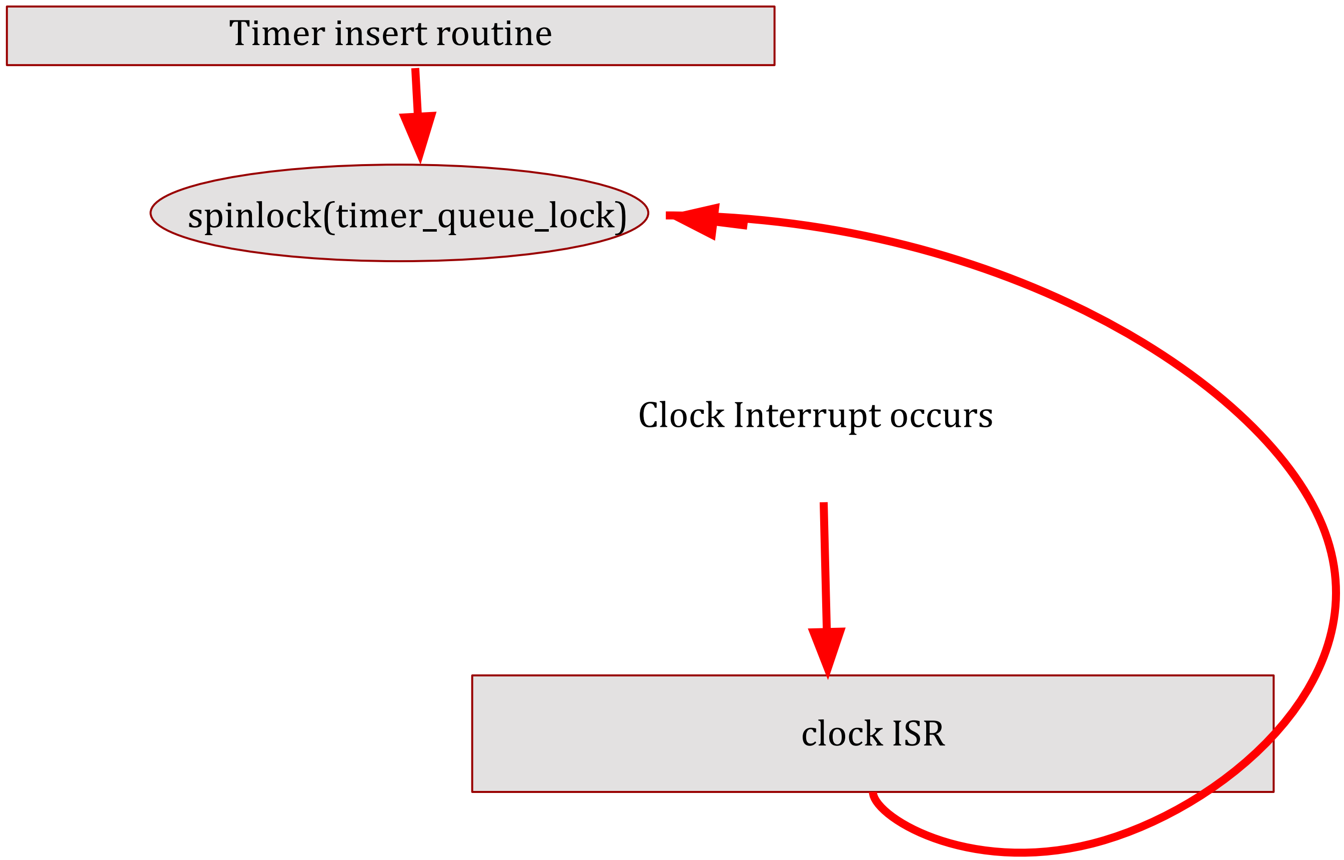
ISR attempts to lock resource X

- Solution:
  - spinlock_IRQ()/spinunlock_IRQ()
    - disable interrupts before acquiring lock X
    - unlock X then enable interrupts
      - thread A does spinlock_IRQ(X)
        - » disable interrupts then locks X
          - interrupts are blocked
      - thread A runs than does spinunlock_IRQ(X)
        - » unlock X then enables interrupts
      - interrupt occurs
        - » ISR locks X, runs then unlock X

Interrupt deadlocks

- Timer routine acquires timer queue lock
- Timer routine inserts task in timer queue
- Clock hardware interrupt occurs
- Clock ISR acquires timer queue lock
-
- DEADLOCK
-
- Solution: use spinlock_IRQ

Interrupt deadlocks



Today's lecture

- The real world - deadlock and synchronization in Linux
- Review of synchronization
- Fun programming (at least for Orran)

Race conditions

- Asynchronous events occur all the time:
  - Interrupt: e.g., timer, device, ...
  - Threads running asynchronously on different cores
- Race is when result depends on order of asynchronous operation
- Race is a bug if result is incorrect.

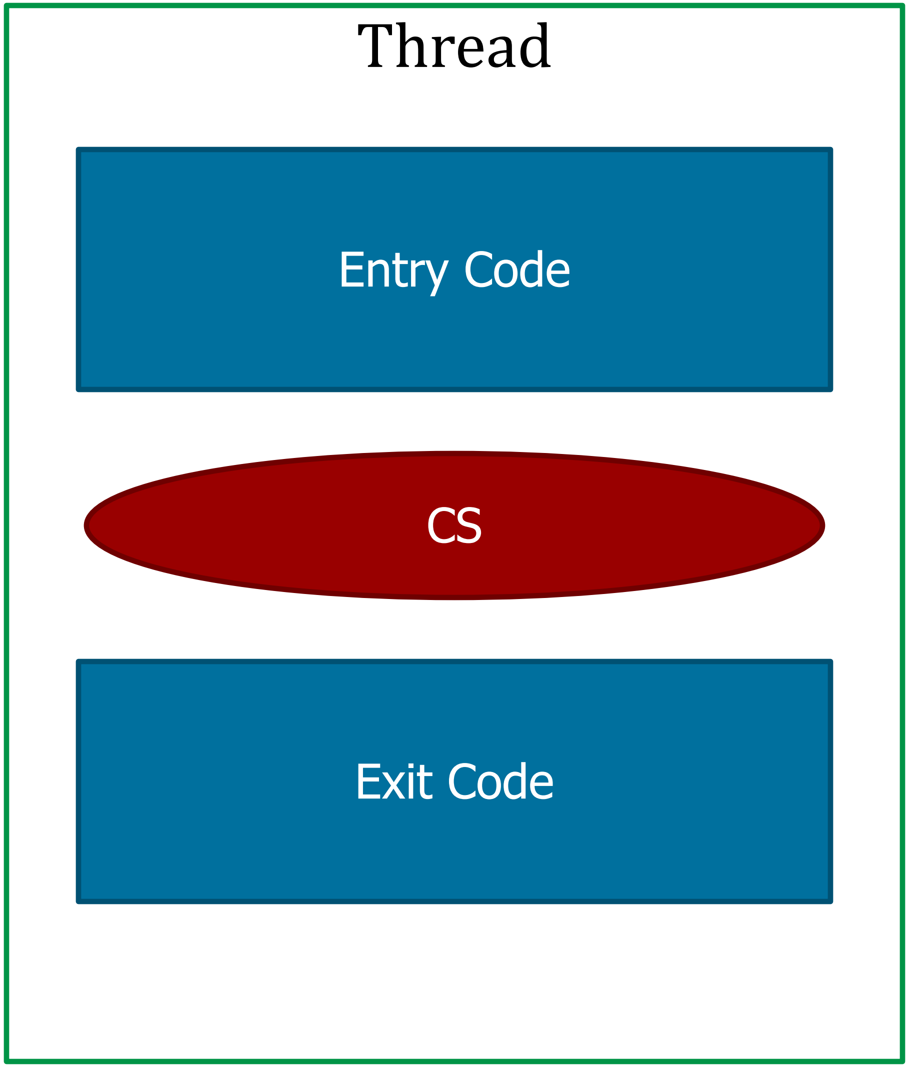
Introduced Critical Regions and Mutual Exclusion

- The part of the program where shared memory is accessed is called a *critical region* (or *critical section*)

- Critical regions should be accessed in *mutual exclusion*

- Solution: Synchronization
  1. No two processes may be simultaneously inside the same critical region
  2. No process running outside the critical region should block another process
  3. No process should wait forever to enter its critical region
  4. No assumptions can be made about speed/number of CPUs

Introduced Critical Regions



Locking

- Busy waiting on variable doesn't work... why?
  - if a thread reads lock, then writes, what if asynchronous event happens between
- Peterson's algorithm lets you take turns
- Hardware today supports atomic operations to let you read variable and write it atomically
- Now we can busy wait on lock... what's the problem with that?

Sleep, wakeup, semaphores

- Sleep and wakeup frees processors, what's the problem?
  - Sleep is not atomic...
- Semaphores:
  - P() or down() - decrement counter if > 0, else block atomically
  - V() or up() - increment counter and atomically wake process if was 0 and any blocked
- Examples, simple race and producer consumer

Monitors

- Programing language construct in Concurrent Pascal, Modula-2, Concurrent Euclid and Java:
  - Collection of code where only one thread can be active at a time.
  - Can wait and signal condition variables
- Hugely simplifies code, but:
  - Limits concurrency
  - Requires language level support

Examples

- Consumer producer implemented in semaphores and monitor
- Classic Dining philosophers problem
  - Lock per chopstick - can result in deadlock
  - Use trylock - can result in starvation
  - Can have one big lock - can limit concurrency
  - More complicated example:
    - no deadlock, no starvation, many philosophers can be picking up chopsticks at the same time

Fourth Solution

```
philosopher(i) {
  think();
  take_chopsticks(i);
  eat();
  put_chopsticks(i);
}
```

```
take_chopsticks(i) {
  mutex.down();
  state[i] = HUNGRY;
  test(i);
  mutex.up();
  philosopher[i].down();
}
```

```
put_chopsticks(i) {
  mutex.down();
  state[i] = THINKING;
  test((i + 1) % N);
  test((i + N - 1) % N);
  mutex.up();
}
```

```
test(i) {
  if (state[i] == HUNGRY && state[(i + 1) % N] != EATING &&
                        state[(i + N - 1) % N] != EATING) {
       state[i] = EATING;
       philosopher[i].up();
  }
}
```

The real world

- Processors have caches
- Moving cache lines between the cores expensive
- What we do:
  - Fine grained locks embedded in data
  - Scalable locks: ticketed spin locks, MCS locks
  - Read Copy Update - avoid locks all together

Deadlock?

Today's lecture

- The real world - deadlock and synchronization in Linux
- Review of synchronization and Deadlock
- Fun programming (at least for Orran)