CS 3204 Operating Systems

Lecture 8

Godmar Back

Announcements

- Project 1 Feb 20 (Tuesday) 11:59pm
- Monday Feb 12, 4:30-7:30 CS Career Fair in Torgersen Museum
  - Bring your resume
- Reading:
  - Read carefully 1.5, 3.1-3.3, 6.1-6.4

Project 1 Suggested Timeline

- End of last week: Feb 2:
  - Have read relevant project documentation, set up CVS, built and run your first kernel, designed your data structures for alarm clock
- Alarm clock by Feb 6
- Basic priority by Feb 8
- Priority Inheritance & Advanced Scheduler will take the most time to implement & debug, start them in parallel
  - Should have design for priority inheritance figured out by Feb 13
  - Develop & test fixed-point layer independently by Feb 13
- Due date Feb 20

Concurrency & Synchronization

Disabling Interrupts

- All asynchronous context switches start with interrupts
  - So disable interrupts to avoid them!

```
intr_level old = intr_disable();
/* modify shared data */
intr_set_level(old);
```

```
void intr_set_level(intr_level to)
{
  if (to == INTR_ON)
    intr_enable();
  else
    intr_disable();
}
```


Disabling Interrupts: Summary

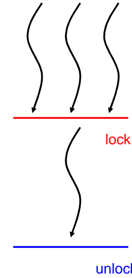
- (this applies to all variations)
- Sledgehammer solution
- Infinite loop means machine locks up
- Use this to protect data structures from concurrent access by interrupt handlers
  - Keep sections of code where irqs are disabled minimal (nothing else can happen until irqs are reenabled – latency penalty!)
  - If you block (give up CPU) mutual exclusion with other threads is not guaranteed
    - Any function that transitively calls thread_block() may block
- Want something more fine-grained
  - Key insight: don't exclude *everybody* else, only those contending for the same critical section

Critical Section Problem

- A solution for the CS Problem must
  1) Provide mutual exclusion: at most one thread can be inside CS
  2) Guarantee Progress: (no deadlock)
     - if more than one threads attempt to enter, one will succeed
     - ability to enter should not depend on activity of other threads not currently in CS
  3) Bounded Waiting: (no starvation)
     - A thread attempting to enter critical section eventually will (assuming no thread spends unbounded amount of time inside CS)
- A solution for CS problem should be
  - Fair (make sure waiting times are balanced)
  - Efficient (not waste resources)
  - Simple

Locks

- Thread that enters CS locks it
  - Others can't get in and have to wait
- Thread unlocks CS when leaving it
  - Lets in next thread
  - which one?
    - FIFO guarantees bounded waiting
    - Highest priority in Proj1
- Can view Lock as an abstract data type
  - Provides (at least) init, acquire, release



Implementing Locks

- Locks can be implemented directly, or – among other options - on top of semaphores
  - If implemented on top of semaphores, then semaphores must be implemented directly
  - Will explain this layered approach first to help in understanding project code
  - Issues in direct implementation of locks apply to direct implementation of semaphores as well

Semaphores

Source: inter.scoutnet.org

- Invented by Edsger Dijkstra in 1960s
- Counter S, initialized to some value, with two operations:
  - P(S) or “down” or “wait” – if counter greater than zero, decrement. Else wait until greater than zero, then decrement
  - V(S) or “up” or “signal” – increment counter, wake up any threads stuck in P.
- Semaphores don't go negative:
  - #V + InitialValue - #P >= 0
- Note: direct access to counter value after initialization is not allowed
- Counting vs Binary Semaphores
  - Binary: counter can only be 0 or 1
- Simple to implement, yet powerful
  - Can be used for many synchronization problems

Semaphores as Locks

- Semaphores can be used to build locks
  - Pintos does just that
- Must initialize semaphore with 1 to allow one thread to enter critical section

```
semaphore S(1); // allows initial down

lock_acquire()
{   // try to decrement, wait if 0
    sema_down(S);
}

lock_release()
{   // increment (wake up waiters if any)
    sema_up(S);
}
```

- Easily generalized to allow at most N simultaneous threads: multiplex pattern (i.e., a resource can be accessed by at most N threads)

Implementing Locks Directly

- NB: Same technique applies to implementing semaphores directly (as in done in Pintos)
  - Will see two applications of the same technique
- Different solutions exist to implement locks for uniprocessor and multiprocessors
- Will talk about how to implement locks for uniprocessors first – next slides all assume uniprocessor

Implementing Locks, Take 1

```
lock_acquire(struct lock *l)
{
    while (l->state == LOCKED)
            continue;
    l->state = LOCKED;
}
```

```
lock_release(struct lock *l)
{
    l->state = UNLOCKED;
}
```

- Does this work?

No – does not guarantee mutual exclusion property – more than one thread may see "state" in UNLOCKED state and break out of while loop. This implementation has itself a race condition.

Implementing Locks, Take 2

```
lock_acquire(struct lock *l)
{
    disable_preemption();
    while (l->state == LOCKED)
            continue;
    l->state = LOCKED;
    enable_preemption();
}
```

```
lock_release(struct lock *l)
{
    l->state = UNLOCKED;
}
```

- Does this work?

No – does not guarantee progress property. If one thread enters the while loop, no other thread will ever be scheduled since preemption is disabled – in particular, no thread that would call lock_release will ever be scheduled.

Implementing Locks, Take 3

```
lock_acquire(struct lock *l)
{
    while (true) {
        disable_preemption();
        if (l->state == UNLOCKED) {
                l->state = LOCKED;
                enable_preemption();
                return;
        }
        enable_preemption();
    }
}
```

```
lock_release(struct lock *l)
{
    l->state = UNLOCKED;
}
```

Yes, this works – but is grossly inefficient. A thread that encounters the lock in the LOCKED state will busy wait until it is unlocked, needlessly using up CPU time.

- Does this work?

Implementing Locks, Take 4

```
lock_acquire(struct lock *l)
{
  disable_preemption();
  while (l->state == LOCKED) {
     list_push_back(l->waiters,
                        &current->elem);
     thread_block(current);
  }
  l->state = LOCKED;
  enable_preemption();
}
```

```
lock_release(struct lock *l)
{
   disable_preemption();
   l->state = UNLOCKED;
   if (list_size(l->waiters) > 0)
      thread_unblock(
        list_entry(list_pop_front(l->waiters),
                 struct thread, elem));
   enable_preemption();
}
```

Correct & uses proper blocking.
Note that thread doing the unlock performs the work of unblocking the first waiting thread.