CS 3204 Operating Systems

Lecture 10

Godmar Back

Announcements

- Project 1 due Monday Sep 25, 11:59pm
  - Only 4 days left
  - Should be busy debugging priority donation & the advanced scheduler
  - gdb doesn't bite. Make sure you're on the same machine – type "hostname" to check
- Project 0 scores are forthcoming (!?!?)
- Reading assignments:
  - Read carefully Chapter 6.5-6.8

Concurrency & Synchronization

Recap: Synchronization

- Covered:
  - Critical Section Problem
  - Implementation uniprocessor vs multiprocessor
  - Using locks to express mutual exclusion constraint
- Now:
  - Higher-level synchronization constructs that can express precedence constraints

Infinite Buffer Problem

```
producer(item)
{
  lock_acquire(buffer);
  buffer[head++] = item;
  lock_release(buffer);
}
```

```
consumer()
{
   lock_acquire(buffer);
   while (buffer is empty) {
      lock_release(buffer);
      thread_yield();
      lock_acquire(buffer);
   }
   item = buffer[tail++];
   lock_release(buffer);
   return item
}
```

- Trying to implement infinite buffer problem with locks alone leads to a very inefficient solution (busy waiting!)
- Locks cannot express precedence constraint: A must happen before B.

Infinite Buffer Problem, Take 2

```
producer(item)
{
  lock_acquire(buffer);
  buffer[head++] = item;
  lock_release(buffer);
  if (#consumers > 0)
     for c in consumers {
       thread_unblock(c);
     }
}
```

```
consumer()
{
   lock_acquire(buffer);
   while (buffer is empty) {
      lock_release(buffer);
      consumers.add(current);
      thread_block(current);
      lock_acquire(buffer);
   }
```

Context switch here would cause *Lost Wakeup* problem: producer will put item in buffer, but won't unblock consumer thread (since consumer thread isn't in consumers yet)

Infinite Buffer Problem, Take 3

```
producer(item)
{
  lock_acquire(buffer);
  buffer[head++] = item;
  lock_release(buffer);
  if (#consumers > 0)
      for c in consumers {
        thread_unblock(c);
      }
}
```

```
consumer()
{
   lock_acquire(buffer);
   while (buffer is empty) {
      consumers.add(current);
      lock_release(buffer);
      thread_block(current);
      lock_acquire(buffer);
   }
   item = buffer[tail++];
   lock_release(buffer);
   return item
}
```

- What if consumers.add is done before lock is released?

Infinite Buffer Problem, Take 4

```
producer(item)
{
  lock_acquire(buffer);
  buffer[head++] = item;
  if (#consumers > 0)
      for c in consumers {
        thread_unblock(c);
      }
  lock_release(buffer);
}
```

```
consumer()
{
   lock_acquire(buffer);
   while (buffer is empty) {
      consumers.add(current);
      lock_release(buffer);
      thread_block(current);
      lock_acquire(buffer);
   }
   item = buffer[tail++];
   lock_release(buffer);
   return item
}
```

- This is correct, but complicated and very easy to get wrong.

Low-level vs. High-level Synchronization

- Low-level synchronization primitives:
  - Disabling preemption, (Blocking) Locks, Spinlocks
- Can implement mutual exclusion with them
- To implement precedence constraints, need direct access to thread_unblock/thread_block
  - Don't always have that or using them would violate encapsulation or be complicated
- We need well-understood higher-level constructs
  - Semaphores
  - Monitors

Semaphores

Source: inter.scoutnet.org

- Invented by Edsger Dijkstra in 1965s
- 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

Infinite Buffer w/ Semaphores (1)

```
semaphore items_avail(0);

producer()
{
  lock_acquire(buffer);
  buffer[head++] = item;
  lock_release(buffer);
  sema_up(items_avail);
}
```

```
consumer()
{
   sema_down(items_avail);
   lock_acquire(buffer);
   item = buffer[tail++];
   lock_release(buffer);
   return item;
}
```

- Semaphore "remembers" items put into queue (no updates are lost)

Implementing Semaphores

- Implementation is analogous to simple locks on uniprocessor
  - requires counter variable
  - requires disabling preemption
  - requires appropriate blocking/unblocking
- You already know the details from Pintos's synch.c

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)

Infinite Buffer w/ Semaphores (2)

```
semaphore items_avail(0);
semaphore buffer_access(1);

producer()
{
  sema_down(buffer_access);
  buffer[head++] = item;
  sema_up(buffer_access);
  sema_up(items_avail);
}
```

```
consumer()
{
   sema_down(items_avail);
   sema_down(buffer_access);
   item = buffer[tail++];
   sema_up(buffer_access);
   return item;
}
```

- Can use semaphore instead of lock to protect buffer access

Bounded Buffer w/ Semaphores

```
semaphore items_avail(0);
semaphore buffer_access(1);
semaphore slots_avail(CAPACITY);
producer()
{
  sema_down(slots_avail);
  sema_down(buffer_access);
  buffer[head++] = item;
  sema_up(buffer_access);
  sema_up(items_avail);
}
```

```
consumer()
{
   sema_down(items_avail);
   sema_down(buffer_access);
   item = buffer[tail++];
   sema_up(buffer_access);
   sema_up(slots_avail);
   return item;
}
```

- Semaphores allow for scheduling of resources

Rendezvous

- A needs to be sure B has advanced to point L, B needs to be sure A has advanced to L

```
semaphore A_madeit(0);

A_rendezvous_with_B()
{
  sema_up(A_madeit);
  sema_down(B_madeit);
}
```

```
semaphore B_madeit(0);

B_rendezvous_with_A()
{
  sema_up(B_madeit);
  sema_down(A_madeit);
}
```

Waiting for an activity to finish

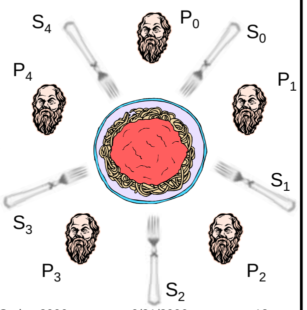
```
semaphore done_with_task(0);
thread_create(
      do_task,
      (void*)&done_with_task);

sema_down(done_with_task);
// safely access task's results
```

```
void
do_task(void *arg)
{
   semaphore *s = arg;
   /* do the task */
   sema_up(*s);
}
```

- Works no matter which thread is scheduled first after thread_create (parent or child)
- Elegant solution that avoids the need to share a "have done task" flag between parent & child
- Two applications of this technique in Pintos Project 2
  - signal successful process startup ("exec") to parent
  - signal process completion ("exit") to parent

Dining Philosophers (Dijkstra)

- A classic
- 5 Philosophers, 1 bowl of spaghetti
- Philosophers (threads) think & eat ad infinitum
  - Need left & right fork to eat (!?)
- Want solution that prevents starvation & does not delay hungry philosophers unnecessarily



Dining Philosophers (1)

```
semaphore fork[0..4](1);
philosopher(int i)                    // i is 0..4
{
  while (true) {
    /* think … finally */
    sema_down(fork[i]);               // get left fork
    sema_down(fork[(i+1)%5]);         // get right fork
     /* eat */
     sema_up(fork[i]);                // put down left fork
     sema_up(fork[(i+1)%5]);          // put down right fork
  }
}
```

- What is the problem with this solution?
- Deadlock if all pick up left fork

Dining Philosophers (2)

```
semaphore fork[0..4](1);
semaphore at_table(4);   // allow at most 4 to fight for forks
philosopher(int i)                    // i is 0..4
{
  while (true) {
    /* think … finally */
    sema_down(at_table);              // sit down at table
    sema_down(fork[i]);               // get left fork
    sema_down(fork[(i+1)%5]);         // get right fork
     /* eat … finally */
     sema_up(fork[i]);                // put down left fork
     sema_up(fork[(i+1)%5]);          // put down right fork
     sema_up(at_table);               // get up
  }
}
```

High vs Low Level Synchronization

- As we've seen, bounded buffer can be solved with higher-level synchronization primitives
  - semaphores (and monitors, as we'll see shortly)
- In Pintos kernel, one could also use thread_block/unblock directly
  - this is not always efficiently possible in other concurrent environments
- Q.: when should you use low-level synchronization (a la thread_block/thread_unblock) and when should you prefer higher-level synchronization?
- A.: Except for the simplest scenarios, higher-level synchronization abstractions are always preferable
  - They're well understood; make it possible to reason about code.