CS 3204
Operating Systems

Lecture 8

Godmar Back

Announcements

- Project 1 due on Sep 29, 11:59pm
- Reading:
  - Read carefully 1.5, 3.1-3.3, 6.1-6.4
- Dr. Back has no office hours this week

Project 1 Suggested Timeline

- By now, you have:
  - Have read relevant project documentation, set up CVS, built and run your first kernel, designed your data structures for alarm clock
- And should be finishing:
  - Alarm clock by Sep 16
- Pass all basic priority tests by Sep 18
- 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 Sep 23
  - Develop & test fixed-point layer independently by Sep 23
- Due date Sep 29

Concurrency & Synchronization

Semaphores

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;
  if (#consumers > 0)
     for c in consumers {
       thread_unblock(c);
     }
  lock_release(buffer);
}
```

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

- Q: Why does this not work?

Infinite Buffer Problem, Take 2

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

Problem 1:
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)

Problem 2: consumers is accessed without lock

Infinite Buffer Problem, Take 3

```
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
}
```

- Idea: move consumers.add before lock_release

Infinite Buffer Problem, Take 3

```
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
}
```

Context switch here would allow producer to see a consumer in the queue that is not yet blocked – thread_unblock() will panic. (Or, if thread_unblock() were written to ignore attempts at unblocking threads that aren't blocked, the wakeup would still be lost.)

- Idea: move consumers.add before lock_release

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);
      disable_preemption();
      lock_release(buffer);
      thread_block(current);
      enable_preemption();
      lock_acquire(buffer);
   }
   item = buffer[tail++];
   lock_release(buffer);
   return item
}
```

- Must ensure that releasing the lock and blocking happens atomically

Infinite Buffer Problem, Take 4

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

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

      = buffer[tail++];
      _release(buffer);
      rn item
}
```

Final problem: producer always wakes up *all* consumers, even though at most one will be able to consume the item produced. This is known as a *thundering herd* problem.

- M
  th
  happens atomically

Infinite Buffer Problem, Take 5

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

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

- This is correct, but complicated and very easy to get wrong
  - Want abstraction that does not require direct block/unblock call

Low-level vs. High-level Synchronization

- Low-level synchronization primitives:
  - Disabling preemption, (Blocking) Locks, Spinlocks
  - implement mutual exclusion
- Implementing precedence constraints directly via thread_unblock/thread_block is problematic because
  - It's complicated (see last slides)
  - It may violate encapsulation from a software engineering perspective
  - You may not have that access at all (unprivileged code!)
- We need well-understood higher-level constructs that have support for waiting/signaling "built-in"
  - 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)

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

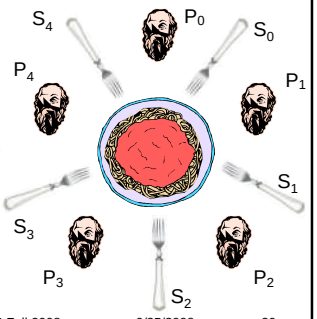
```
semaphore done_with_task(0);
thread_create(
      do_task,
      (void*)&done_with_task);

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
  }
}
```