CS 3204
Operating Systems

Lecture 13

Godmar Back

Virginia Tech

Announcements

- Project 1 is due Feb 27, 11:59pm
  - Should have finished alarm clock by now
  - Basic priority no later than Friday (Feb 17)
    - priority-change, -preempt, -fifo, -sema, -condvar
  - Priority donation & advanced scheduler will likely take more time than alarm clock & priority scheduling
- My office hours today 3-4pm

Rendezvous (revisited)

```
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_down(A_madeit);
  sema_up(B_madeit);
}
```

- Q.: does order of B's sema_up/sema_down matter?
  - Not for correctness. Correctness is paramount.
- Performance? Consider possibility of context switches:
  - sema_down if S == 0 will block
  - sema_down if S > 0 or sema_up with no thread waiting will not switch gratuitously
  - sema_up with thread waiting might or might not switch

Infinite Buffer Problem (revisited)

```
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 (revisited)

```
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
}
```

- Make sure "consumer" queue is protected, too

Infinite Buffer Problem (revisited)

- This is a correct solution; in fact, we've just reinvented "monitors" – topic of this lecture.
- Q1: What if we hadn't had direct access to thread_block/thread_unblock?
- Q2: Even if we have, should we use them?

Infinite Buffers w/o thread_block

```
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
}
```

- Very inefficient solution (repeated polling)

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.

Monitors

- A monitor combines a set of shared variables & operations to access them
  - Think of an enhanced C++ class with no public fields
- A monitor provides implicit synchronization (only one thread can access private variables simultaneously)
  - Single lock is used to ensure all code associated with monitor is within critical section
- A monitor provides a general signaling facility
  - Wait/Signal pattern (similar to, but different from semaphores)
  - May declare & maintain multiple signaling queues

Monitors (cont'd)

- Classic monitors are embedded in programming language
  - Invented by Hoare & Brinch-Hansen 1972/73
  - First used in Mesa/Cedar System @ Xerox PARC 1978
  - Limited version available in Java/C#
- (Classic) Monitors are safer than semaphores
  - can't forget to lock data – compiler checks this
- In contemporary C, monitors are a *synchronization pattern* that is achieved using locks & condition variables
  - Must understand monitor abstraction to use it

Infinite Buffer w/ Monitor

```
monitor buffer {
    /* implied: struct lock mlock;*/
private:
    char buffer[];
    int head, tail;
public:
    produce(item);
    item consume();
}
```

```
buffer::produce(item i)
{  /* try { lock_acquire(&mlock); */
   buffer[head++] = i;
   /* } finally {lock_release(&mlock);} */
}

buffer::consume()
{  /* try { lock_acquire(&mlock); */
   return buffer[tail++];
    /* } finally {lock_release(&mlock);} */
}
```

- Monitors provide implicit protection for their internal variables
  - Still need to add the signaling part

Condition Variables

- Variables used by a monitor for signaling a condition
  - a general (programmer-defined) condition, not just integer increment as with semaphores
- Monitor can have more than one condition variable
- Three operations:
  - Wait(): leave monitor, wait for condition to be signaled, reenter monitor
  - Signal(): signal one thread waiting on condition
  - Broadcast(): signal all threads waiting on condition

Bounded Buffer w/ Monitor

```
monitor buffer {
    condition items_avail;
    condition slots_avail;
private:
    char buffer[];
    int head, tail;
public:
    produce(item);
    item consume();
}
```

```
buffer::produce(item i)
{
  while ((tail+1–head)%CAPACITY==0)
    slots_avail.wait();
  buffer[head++] = i;
  items_avail.signal();
}
buffer::consume()
{
  while (head == tail)
    items_avail.wait();
  item i = buffer[tail++];
  slots_avail.signal();
  return i;
}
```


Bounded Buffer w/ Monitor

```
monitor buffer {
    condition items_avail;
    condition slots_avail;
private:
    char buffer[];
    int head, tail;
public:
    produce(item);
    item consume();
}
```

Q1.: How is lost update problem avoided?

Q2.: Why *while()* and not *if()*?

```
buffer::produce(item i)
{
  while ((tail+1–head)%CAPACITY==0)
    slots_avail.wait();
  buffer[head++] = i;
  items_avail.signal();
}
buffer::consume()
{
  while (head == tail)
    items_avail.wait();
  item i = buffer[t
  slots_avail.
  return i;
}
```

```
lock_release(&mlock);
block_on(items_avail);
lock_acquire(&mlock);
```


Implementing Condition Variables

- State is just a queue of waiters:
  - Wait(): adds current thread to (end of queue) & block
  - Signal(): pick one thread from queue & unblock it
    - Hoare-style Monitors: gives lock directly to waiter
    - Mesa-style monitors (C, Pintos, Java): signaler keeps lock – waiter gets READY, but can't enter until signaler gives up lock
  - Broadcast(): unblock all threads
- Compare to semaphores:
  - Condition variable signals are lost if nobody's on the queue (semaphore's V() are remembered)
  - Condition variable wait() always blocks (semaphore's P() may or may not block)

Monitors in C

- POSIX Threads & Pintos
- No compiler support, must do it manually
  - must declare locks & condition vars
  - must call lock_acquire/lock_release when entering&leaving the monitor
  - must use cond_wait/cond_signal to wait for/signal condition
- Note: cond_wait(&c, &m) takes monitor lock as parameter
  - Necessary so monitor can be left & reentered without losing signals
- Pintos cond_signal() takes lock as well
  - only as debugging help/assertion to check lock is held when signaling
  - pthread_cond_signal() does not

Summary

- Semaphores & Monitors are both higher-level constructs
- Monitors in C is just a programming pattern that involves mutex+condition variables
- When should you use which?