Rules for Easy Locking

Every shared variable must be protected by a lock

- One lock may protect more than one variable, but not too many
- Acquire lock before touching (reading or writing) variable
- Release when done, on all paths

If manipulating multiple variables, acquire locks protecting each

- Acquire locks always in same order (doesn't matter which order, but must be same)
- Release in opposite order
- Don't mix acquires & release (two-phase locking)

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
```

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;
  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
}
```

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

- 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

- Semaphores
- Monitors