CS161: Operating Systems

Matt Welsh
mdw@eecs.harvard.edu

Lecture 6: Semaphores, Monitors, and Condition Variables
February 20, 2007

Higher-level synchronization primitives

We have looked at one synchronization primitive: *locks*

Locks are useful for many things, but sometimes programs have different requirements.

Examples?

- Say we had a shared variable where we wanted *any number of threads* to **read** the variable, but only *one thread* to **write** it.
- How would you do this with locks?

```
Reader() {
  lock.acquire();
  mycopy = shared_var;
  lock.release();
  return mycopy;
}
```

```
Writer() {
  lock.acquire();
  shared_var = NEW_VALUE;
  lock.release();
}
```

What's wrong with this code?

Semaphores

Higher-level synchronization construct

- Designed by Edsger Dijkstra in the 1960's, part of the THE operating system (classic stuff!)

Semaphore is a *shared counter*

Two operations on semaphores:

P() or wait() or down()

- From Dutch "*proeberen*", meaning "test"
- Atomic action:
  - Wait for semaphore value to become > 0, then decrement it

V() or signal() or up()

- From Dutch "*verhogen*", meaning "increment"
- Atomic action:
  - Increments semaphore value by 1.

Semaphore Example

Semaphores can be used to implement locks:

```
Semaphore my_semaphore = 1; // Initialize to nonzero

int withdraw(account, amount) {
  P(my_semaphore);
  balance = get_balance(account);  }
  balance -= amount;               }  critical
  put_balance(account, balance);   }  section
  V(my_semaphore);                 }
  return balance;
}
```

A semaphore where the counter value is only 0 or 1 is called a *binary semaphore*.

Simple Semaphore Implementation

```
struct semaphore {
    int val;
    thread_list waiting;  // List of threads waiting for semaphore
}

P(semaphore Sem):    // Wait until > 0 then decrement
     while (Sem.val <= 0) {
          add this thread to Sem.waiting;
          block(this thread);  // What does this do??
     }
     Sem.val = Sem.val -1;
     return;

V(semaphore Sem):     // Increment value and wake up next thread
    Sem.val = Sem.val + 1;
    if (Sem.waiting is nonempty) {
          remove a thread T from Sem.waiting;
          wakeup(T);
    }
```

Why is this a while loop and not just an if statement?

What's wrong with this code???

Simple Semaphore Implementation

```
struct semaphore {
    int val;
    thread_list waiting;  // List of threads waiting for semaphore
}

P(semaphore Sem):    // Wait until > 0 then decrement
    while (Sem.val <= 0) {
        add this thread to Sem.waiting;
        block(this thread);  // What does this do??
    }
    Sem.val = Sem.val -1;
    return;

V(semaphore Sem):    // Increment value and wake up next thread
    Sem.val = Sem.val + 1;
    if (Sem.waiting is nonempty) {
        remove a thread T from Sem.waiting;
        wakeup(T);
    }
```

P() and V() must be atomic actions!

Semaphore Implementation

How do we ensure that the semaphore implementation is atomic?

Semaphore Implementation

How do we ensure that the semaphore implementation is atomic?

One approach: Make them system calls, and ensure only one P() or V() operation can be executed by any process at a time.

- This effectively puts a **lock** around the P() and V() operations themselves!
- Easy to do by disabling interrupts in the P() and V() calls.

Another approach: Use hardware support

- Say your CPU had atomic P and V instructions
- That would be sweet.

OK, but why are semaphores useful?

A binary semaphore (counter is always 0 or 1) is basically a lock.

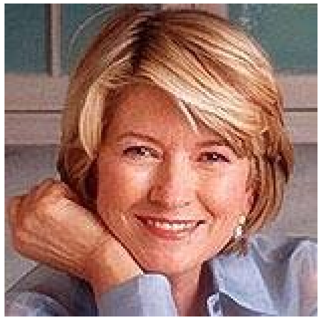
The real value of semaphores becomes apparent when the counter can be initialized to a value *other than 0 or 1.*

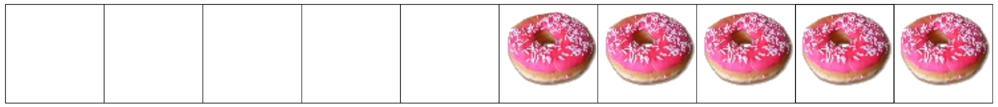
Say we initialize a semaphore's counter to 50.

- What does this mean about P() and V() operations?

The Producer/Consumer Problem

Also called the Bounded Buffer problem.





Mmmm... donuts

Producer pushes items into the buffer.

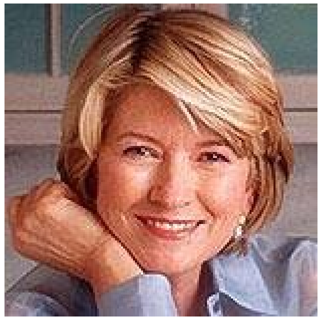
Consumer pulls items from the buffer.

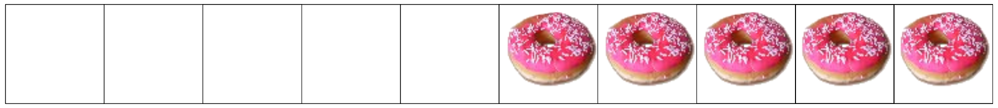
Producer needs to wait when buffer is full.

Consumer needs to wait when the buffer is empty.

The Producer/Consumer Problem

Also called the Bounded Buffer problem.





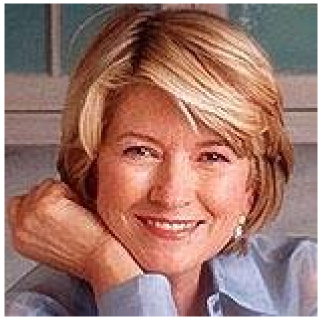
Producer pushes items into the buffer.

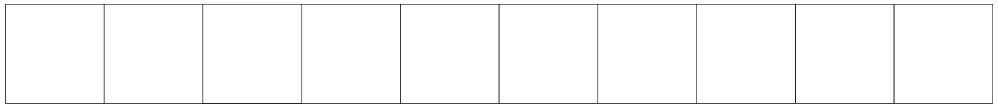
Consumer pulls items from the buffer.

Producer needs to wait when buffer is full.

Consumer needs to wait when the buffer is empty.

One implementation...





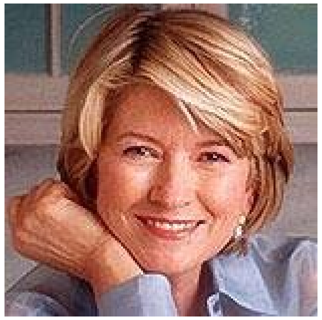
```
int count = 0;

Producer() {
    int item;
    while (TRUE) {
        item = bake();
        if (count == N) sleep();
        insert_item(item);
        count = count + 1;
        if (count == 1)
            wakeup(consumer);
    }
}
```

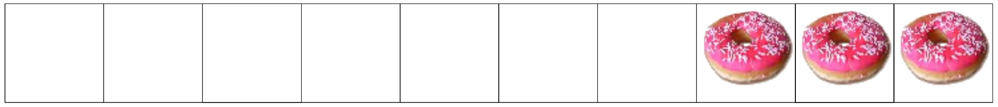
```
Consumer() {
    int item;
    while (TRUE) {
        if (count == 0) sleep();
        item = remove_item();
        count = count - 1;
        if (count == N-1)
            wakeup(producer);
        eat(item);
    }
}
```

What's wrong with this code?

What if we context switch right here??

A fix using semaphores





```
Semaphore mutex = 1;
Semaphore empty = N;
Semaphore full = 0;

Producer() {
    int item;
    while (TRUE) {
        item = bake();
        P(empty);
        P(mutex);
        insert_item(item);
        V(mutex);
        V(full);
    }
}
```

```
Consumer() {
    int item;
    while (TRUE) {
        P(full);
        P(mutex);
        item = remove_item();
        V(mutex);
        V(empty);
        eat(item);
    }
}
```

Reader/Writers

Let's go back to the problem at the beginning of lecture.

- Single shared object
- Want to allow any number of threads to read simultaneously
- But, only one thread should be able to write to the object at a time
  - *(And, not interfere with any readers...)*

```
Semaphore wrt = 1;
int readcount = 0;

Writer() {
    P(wrt);
    do_write();
    V(wrt);
}
```

Why the test here??

```
Reader() {

    readcount++;
    if (readcount == 1) {
        P(wrt);
    }

    do_read();

    readcount--;
    if (readcount == 0) {
        V(wrt);
    }
```

Where's the race condition?

Reader/Writers fixed

Problem: Multiple readers are accessing 'readcount' !

```
Semaphore mutex = 1;
Semaphore wrt = 1;
int readcount = 0;

Writer() {
    P(wrt);
    do_write();
    V(wrt);
}
```

```
Reader() {
    P(mutex);
    readcount++;
    if (readcount == 1) {
        P(wrt);
    }
    V(mutex);
    do_read();
    P(mutex);
    readcount--;
    if (readcount == 0) {
        V(wrt);
    }
    V(mutex);
```

Issues with Semaphores

Much of the power of semaphores derives from calls to P() and V() that are **unmatched**

- See previous example!

Unlike locks, acquire() and release() are not always paired.

This means it is a lot easier to get into trouble with semaphores.

- Semaphores are a lot of rope to hang yourself with...

Would be nice if we had some clean, well-defined *language support* for synchronization...

- Hey. Java does!

Java Synchronization Support: Mutexes

Every Java object can be used as a mutex!

```
Object foo;
synchronized (foo) {
  /* Do some stuff with 'foo' locked... */
  foo.counter++;
}
```

Compiler ensures that lock is released before leaving the `synchronized` block

- Even if there is an exception!!

```
try {
  synchronized(foo) {
      if (foo.doSomething() == false) {
        throw new Exception(“Bad!!”);
      }
  }
catch (Exception e) {
  /* Lock was released before getting here! */
  System.err.println(“Something bad happened!”);
}
```

Java Condition Variables

All Java objects can also act as condition variables.

A *condition variable* represents some condition that a thread can:

- **Wait on**, until the condition occurs; or
- **Notify** other waiting threads that the condition has occurred
  - *Very useful primitive for signaling between threads.*

Three operations on condition variables:

- wait() -- Block until another thread calls notify() or notifyAll() on the CV
- notify() -- Wake up *one* thread waiting on the CV
- notifyAll() -- Wake up *all* threads waiting on the CV

Java CVs and Locks

All condition variable operations **must** be within a synchronized block on the same object

```
/* Thread A */
synchronized (foo) {
  while (foo.counter < 10) {
    foo.wait();
  }
}
```

```
/* Thread B */
synchronized (foo) {
  foo.counter++;
  if (foo.counter >= 10) {
    foo.notify();
  }
}
```

Why is the “synchronized” necessary?

Java CVs and Locks

All condition variable operations **must** be within a synchronized block on the **same** object

```
/* Thread A */
synchronized (foo) {
  while (foo.counter < 10) {
    foo.wait();
  }
}
```

```
/* Thread B */
synchronized (foo) {
  foo.counter++;
  if (foo.counter >= 10) {
    foo.notify();
  }
}
```

Why is the "synchronized" necessary?

If no lock on Thread A...

- Thread might sleep just after another thread sets the counter value to 10!

If no lock on Thread B...

- No guarantee that increment and test of counter is atomic!
  - *Requiring CV operations to be done within a synchronized { } block prevents a lot of common programming mistakes.*

Java CVs and Locks

All condition variable operations **must** be within a synchronized block on the same object

```
/* Thread A */
synchronized (foo) {
  while (foo.counter < 10) {
    foo.wait();
  }
}
```

```
/* Thread B */
synchronized (foo) {
  foo.counter++;
  if (foo.counter >= 10) {
    foo.notify();
  }
}
```

What happens to the lock when you call wait() on the CV?

Java CVs and Locks

All condition variable operations **must** be within a synchronized block on the same object

```
/* Thread A */
synchronized (foo) {
  while (foo.counter < 10) {
    foo.wait();
  }
}
```

```
/* Thread B */
synchronized (foo) {
  foo.counter++;
  if (foo.counter >= 10) {
    foo.notify();
  }
}
```

What happens to the lock when you call wait() on the CV?

- Calling wait() **releases** the lock (atomically!) before blocking.
  - *Can't wait while holding the lock – Thread B could never run!*
- And, the lock is automatically **reclaimed** just before Thread A starts running again!

Bounded Buffer using CV's

```
int theArray[ARRAY_SIZE], size;
Object theLock;

void put(int val) {
    synchronized(theLock) {
        while (size == ARRAY_SIZE) {
          theLock.wait();
        }
        addItemToArray(val);
        size++;
        if (size == 1)
            theLock.notify();
    }
}
```

```
int get() {
    int item;
    synchronized(theLock) {
        while (size == 0) {
            theLock.wait();
        }
        item = getItemFromArray()
        size--;
        if (size == ARRAY_SIZE-1)
            theLock.notify();
    }
    return item;

}
```

Problems with this code??

Bounded Buffer using CV's

```
int theArray[ARRAY_SIZE], size;
Object theLock;

void put(int val) {
    synchronized(theLock) {
        while (size == ARRAY_SIZE) {
          theLock.wait();
        }
        addItemToArray(val);
        size++;
        if (size == 1)
            theLock.notify();
    }
}
```

```
int get() {
    int item;
    synchronized(theLock) {
        while (size == 0) {
            theLock.wait();
        }
        item = getItemFromArray()
        size--;
        if (size == ARRAY_SIZE-1)
            theLock.notify();
    }
    return item;
}
```

Assumes only a single thread calling put() or get() at a time!

If two threads call get(), then two threads call put(), only one will be woken up!!

How to fix this problem?

```
int theArray[ARRAY_SIZE], size;
Object theLock;

void put(int val) {
    synchronized(theLock) {
        while (size == ARRAY_SIZE) {
          theLock.wait();
        }
        addItemToArray(val);
        size++;
        if (size == 1)
            theLock.notifyAll();
    }
}
```

```
int get() {
    int item;
    synchronized(theLock) {
        while (size == 0) {
            theLock.wait();
        }
        item = getItemFromArray()
        size--;
        if (size == ARRAY_SIZE-1)
            theLock.notifyAll();
    }
    return item;

}
```

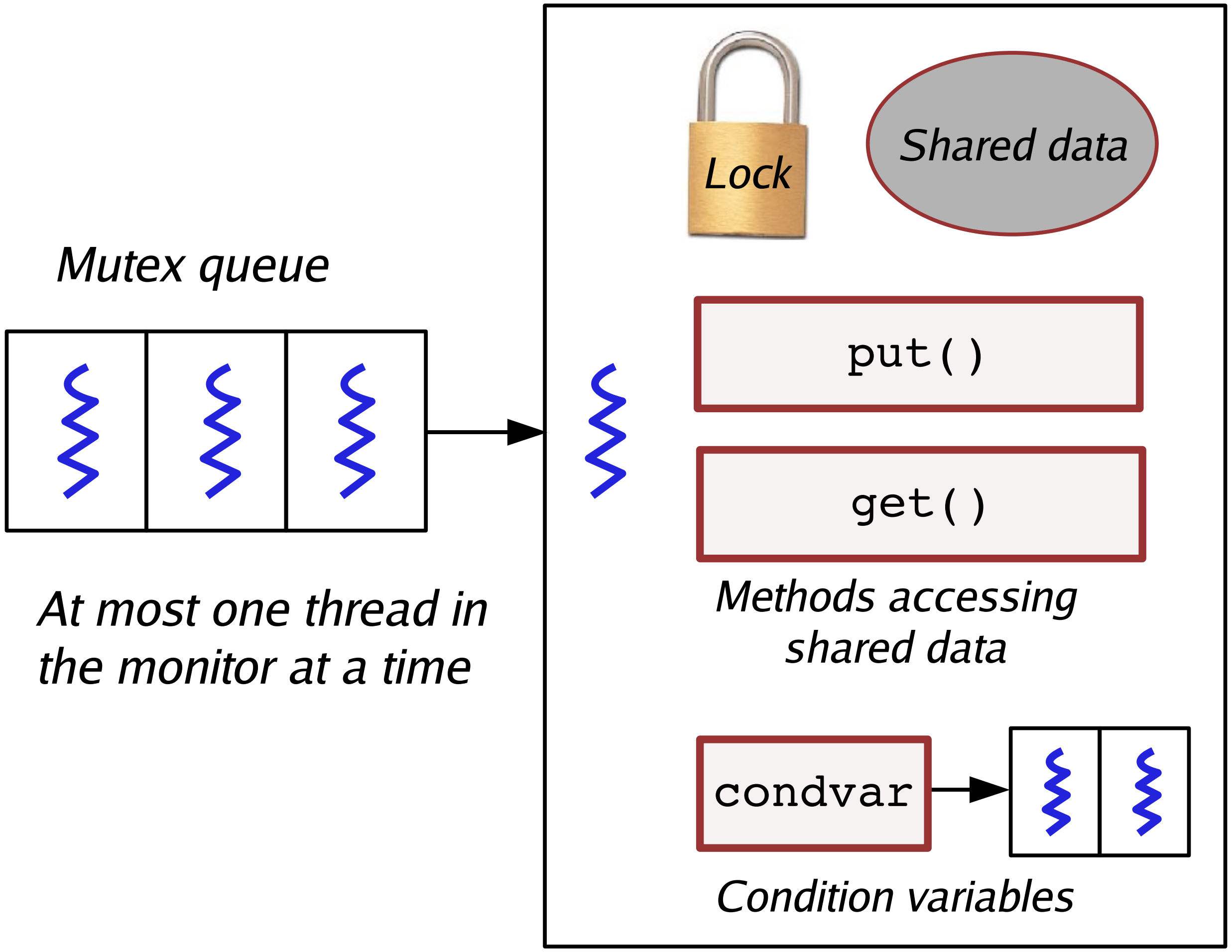
Using notifyAll() will cause **all** threads to wake up and re-check the condition variable.

- Of course, only one thread at a time can get the lock when it wakes up.
- Could be inefficient if a lot of threads are involved!

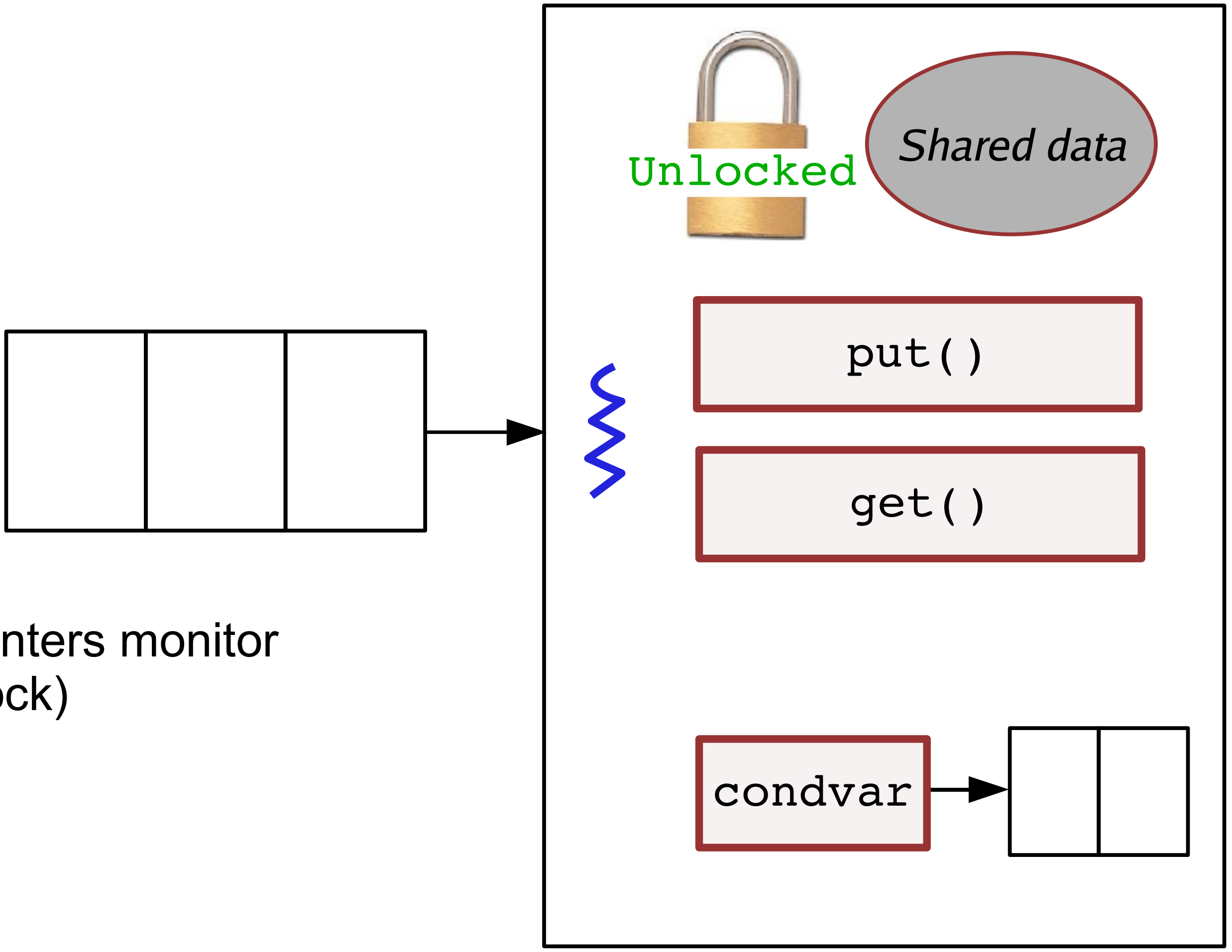
Monitors

This style of using locks and CV's to protect access to a shared object is often called a *monitor*

- Think of a monitor as a lock protecting an object, a series of methods, and associated condition variables.

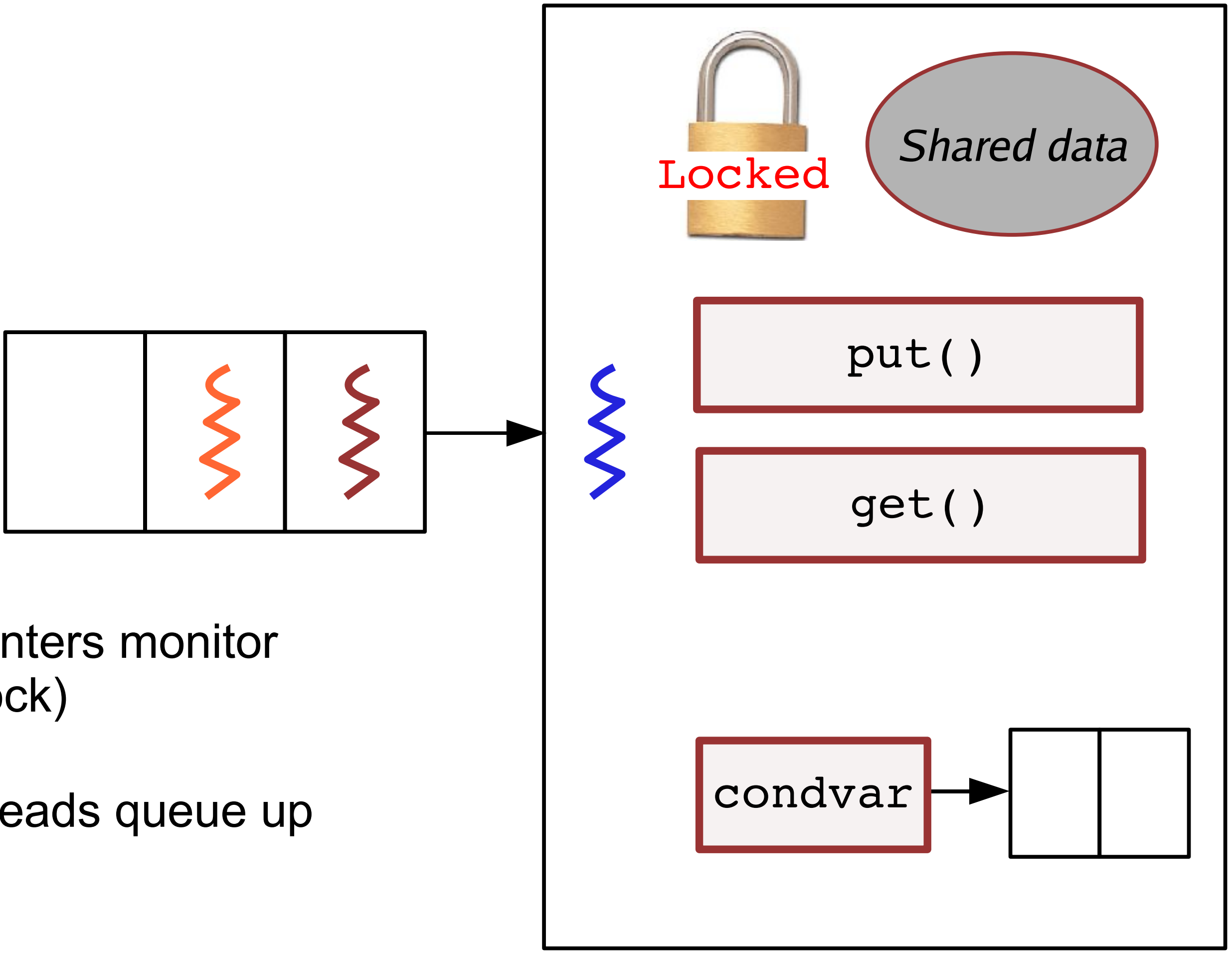

Lock
Shared data
Mutex queue
put()
get()
Methods accessing shared data
At most one thread in the monitor at a time
condvar
Condition variables

Monitors



1) Thread enters monitor
(grabs lock)

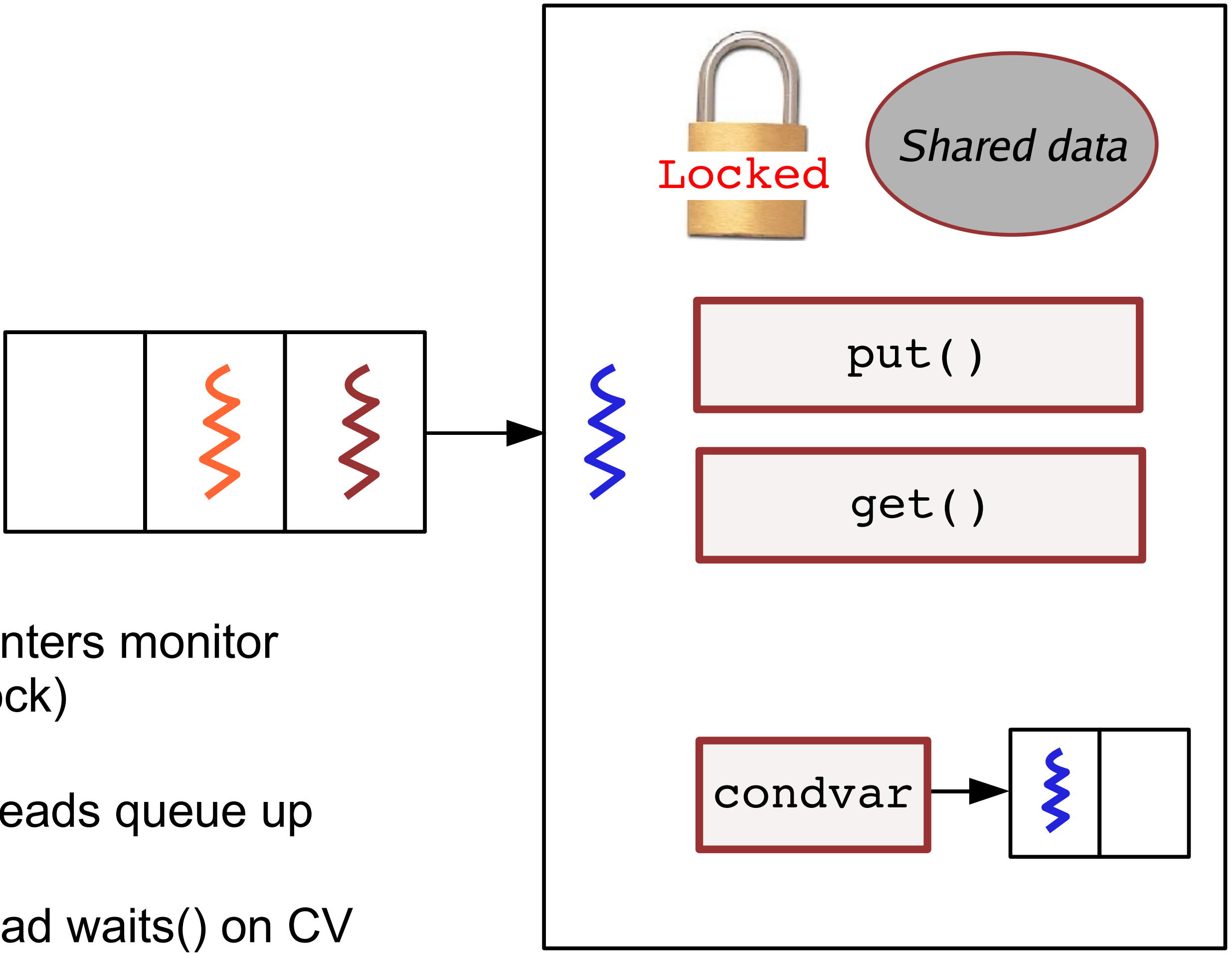
Monitors



1) Thread enters monitor
   (grabs lock)

2) Other threads queue up

Monitors

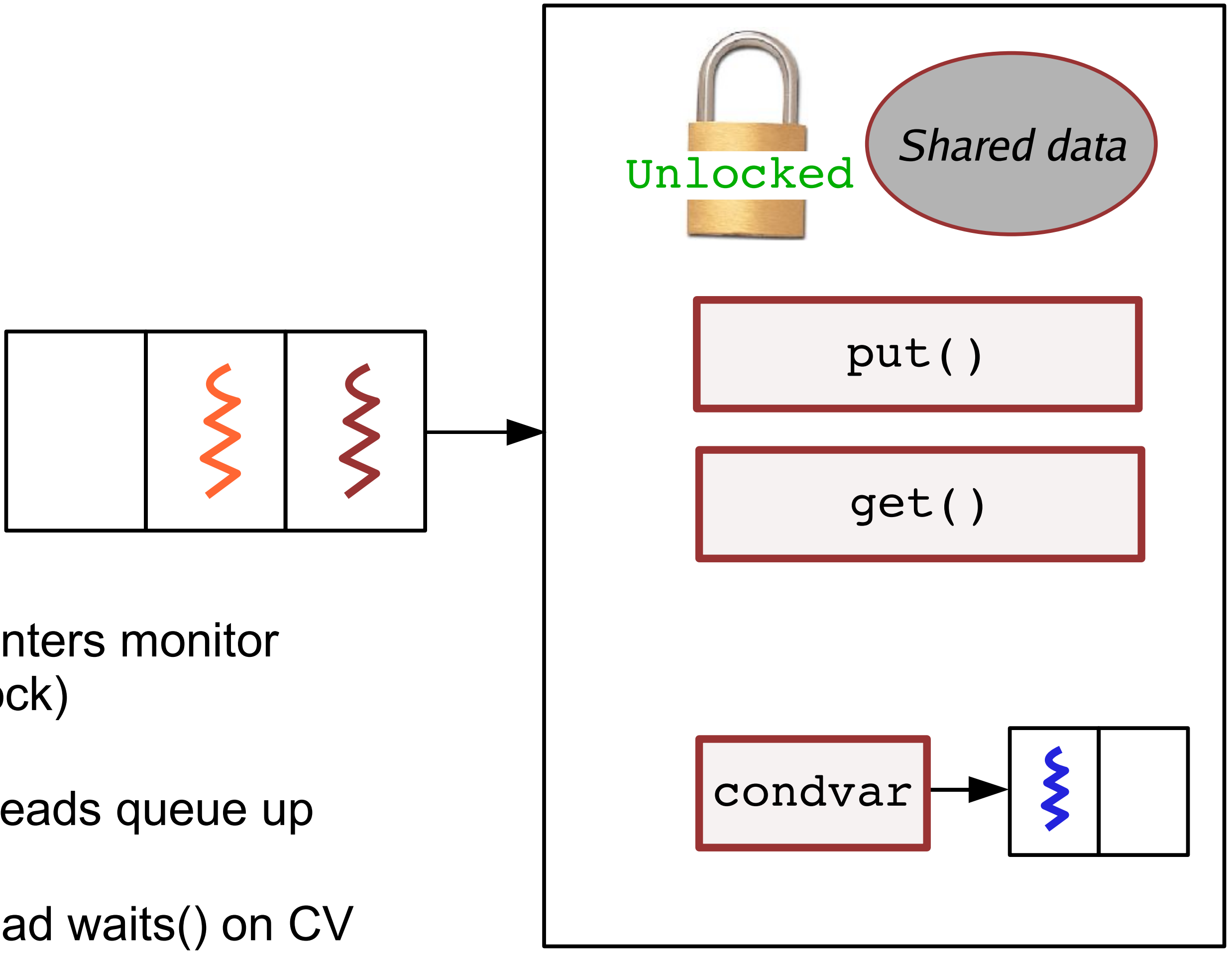


1) Thread enters monitor
   (grabs lock)

2) Other threads queue up

3) Blue thread waits() on CV

Monitors



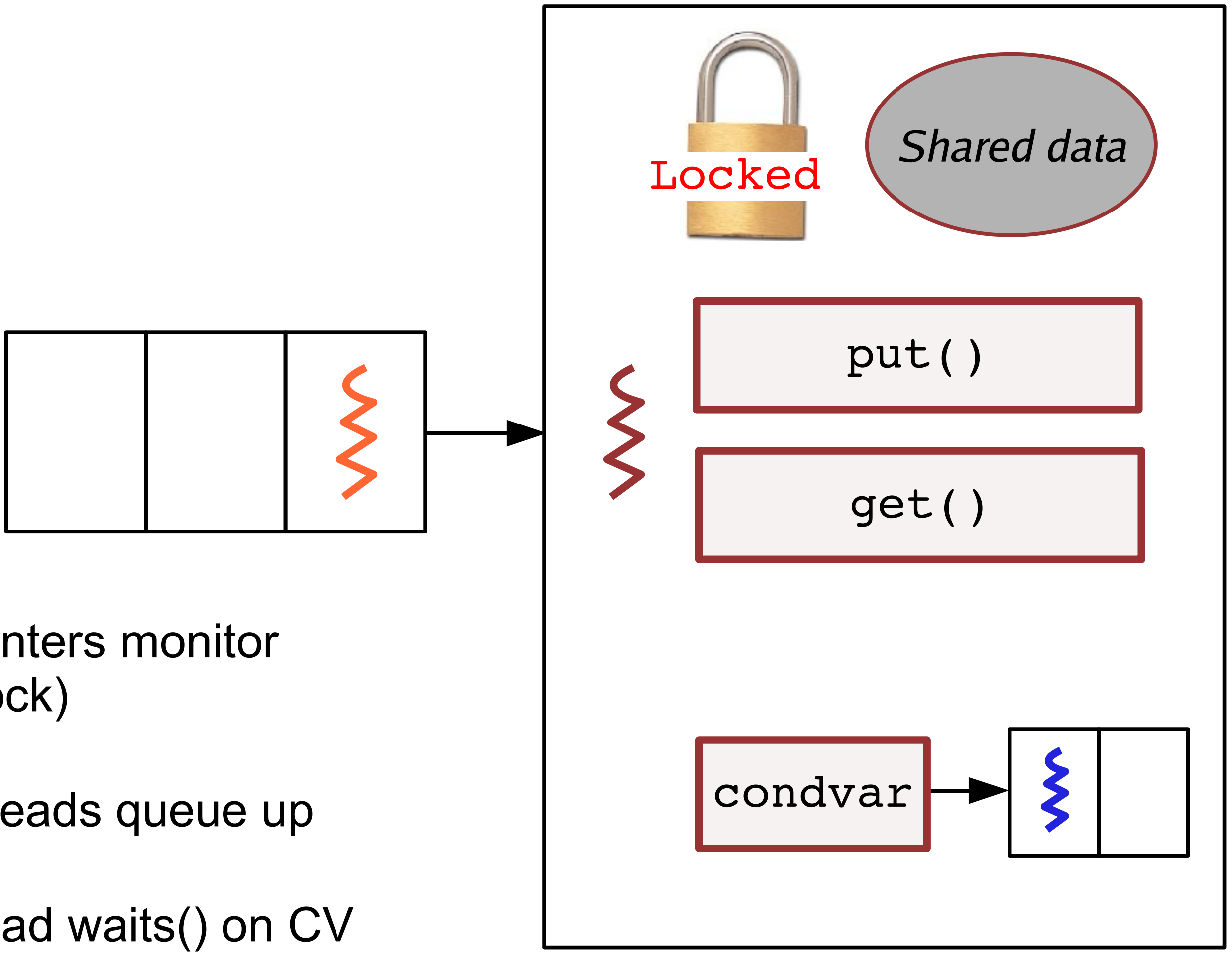
1) Thread enters monitor
   (grabs lock)

2) Other threads queue up

3) Blue thread waits() on CV

4) Next thread enters monitor

Monitors



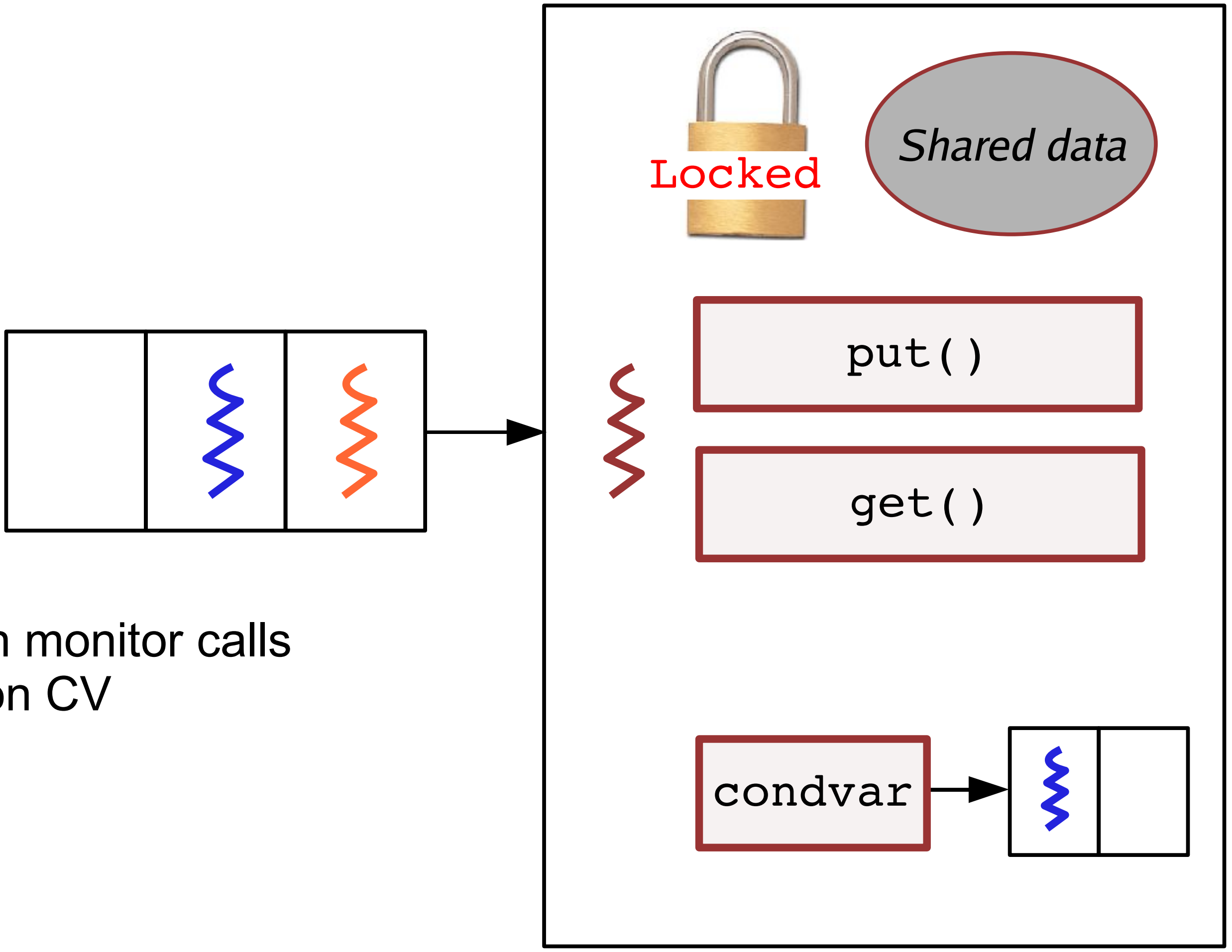
1) Thread enters monitor
   (grabs lock)

2) Other threads queue up

3) Blue thread waits() on CV

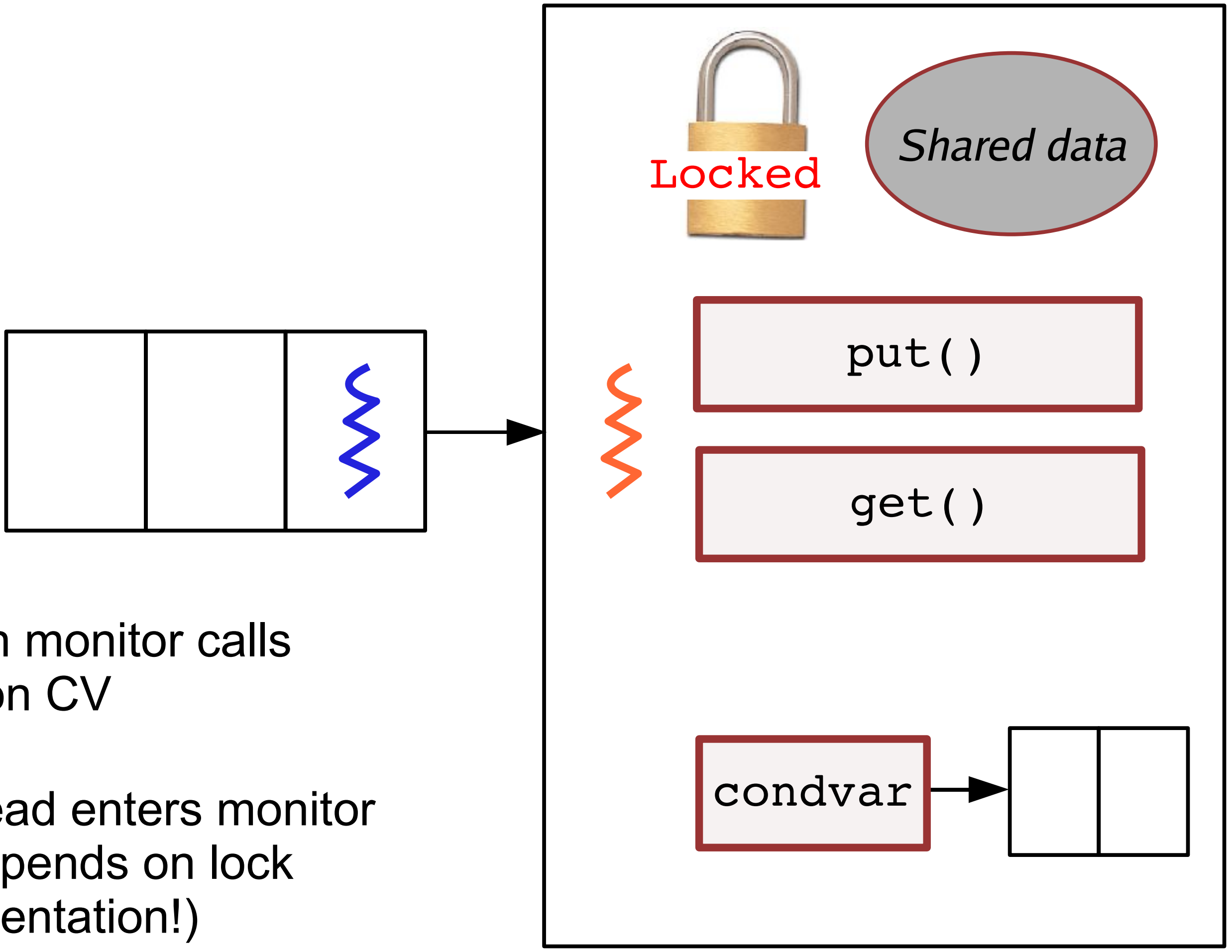
4) Next thread enters monitor

Monitors



5) Thread in monitor calls
notify() on CV

Monitors



5) Thread in monitor calls notify() on CV

6) Next thread enters monitor (order depends on lock implementation!)

Hoare vs. Mesa Monitor Semantics

The monitor notify() operation can have two different meanings:

Hoare monitors (1974)

- notify(CV) means to run the waiting thread *immediately*
- Causes notifying thread to block

Mesa monitors (Xerox PARC, 1980)

- notify(CV) puts waiting thread back onto the “ready queue” for the monitor
- But, notifying thread keeps running

Hoare vs. Mesa Monitor Semantics

The monitor notify() operation can have two different meanings:

Hoare monitors (1974)

- notify(CV) means to run the waiting thread *immediately*
- Causes notifying thread to block

Mesa monitors (Xerox PARC, 1980)

- notify(CV) puts waiting thread back onto the “ready queue” for the monitor
- But, notifying thread keeps running

What's the practical difference?

- In Hoare-style semantics, the “condition” that triggered the notify() will **always be true** when the awoken thread runs
  - *For example, that the buffer is now no longer empty*
- In Mesa-style semantics, awoken thread has to **recheck the condition**
  - *Since another thread might have beaten it to the punch*

We almost always use Mesa-style semantics.

- But the textbook discusses Hoare semantics.

The Big Picture

The point here is that getting synchronization right is *hard*

- Even some of your esteemed faculty members (ahem) have been known to get it wrong.

How to pick between locks, semaphores, condvars, monitors???

Locks are very simple for many cases.

- Issues: Maybe not the most efficient solution
- For example, can't allow multiple readers but one writer inside a standard lock.

Condition variables allow threads to sleep while holding a lock

- Just be sure you understand whether they use Mesa or Hoare semantics!

Semaphores provide pretty general functionality

- But also make it really easy to botch things up.

Java captures a lot of useful common operations with its use of *monitors* (compiler checking is nice too)

- But, not possible to implement everything directly with Java's primitives.

Next Lecture

Famous problems in synchronization

Race conditions, deadlock, and priority inversion

The THERAC-25 disaster

- A radiation machine used to treat cancer
- Had a software bug that actually killed several people.
- Came down to a race condition!

What happened to the Mars Pathfinder?

- Very subtle synchronization bug plagued its software