CS61: Systems Programming and Machine Organization
Harvard University, Fall 2009

Lecture 20:
Semaphores, Condition Variables, and Monitors

Prof. Matt Welsh

November 17, 2009

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

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

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;
  put_balance(account, balance);
  V(my_semaphore);
  return balance;
}
```

critical section

- A semaphore where the counter value is only 0 or 1 is called a *binary semaphore*.
  - Essentially the same as a lock.

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!

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?

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);
    }
```

Another thread might call P() while this thread is blocked, so we have to check the condition again when we wake back up!

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.

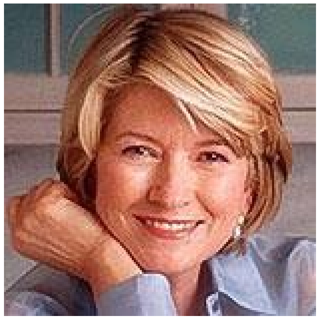
OK, but why are semaphores useful?

- 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.

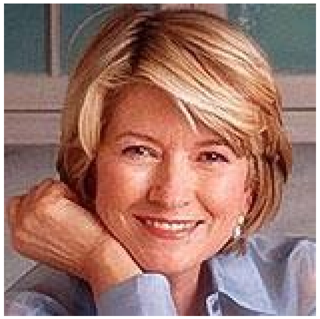
Producer needs to wait when buffer is full.

Consumer needs to wait when the buffer is empty.

The Producer/Consumer Problem

Also called the Bounded Buffer problem.

zzzzz….



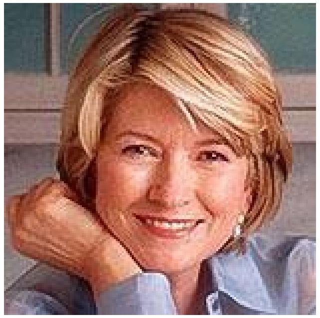
Producer pushes items into the buffer.

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...



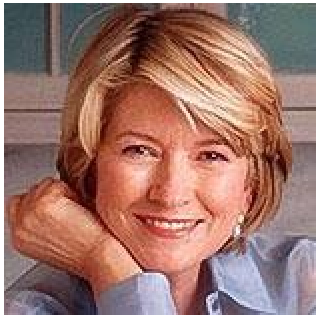
```
int count = 0;

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

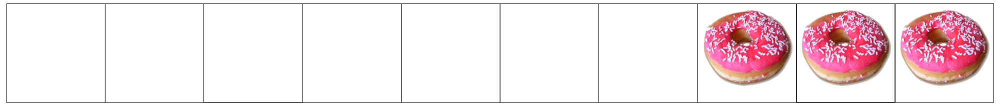
```
Consumer() {
    int item;
    while (TRUE) {
        if (count == 0) sleep();
        item = remove_item();
        count = count - 1;
        if (count == N-1)
            wakeup(producer);
        eat(item);
    }
}
```

- What's wrong with this code?

What if we context switch between the test and the sleep?

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);
}
```

```
Reader() {

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

    do_read();

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

Seems simple, but this code is **broken**. Let's see how...

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
    - *(And, not interfere with any readers...)*

What can happen if we context switch here?

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

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

```
Reader() {

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

    do_read();

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

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
    - *(And, not interfere with any readers...)*

Another Reader() could start and "readcount==1" never happens!

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

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

```
Reader() {

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

    do_read();

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

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
    - *(And, not interfere with any readers...)*

What can happen if we context switch here?

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

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

```
Reader() {

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

    do_read();

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

Reader/Writers

- Let's go back to the problem at the beginning of lecture.
  - Single shared object
  - Want to allow any number of threads t
  - But, only one thread should be able t
    - *(And, not interfere with any reader*

A Writer() could start, P the semaphore first, then subsequent Reader() threads would be able to get past the semaphore (since "readcount != 1")

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

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

```
Reader() {

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

    do_read();

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

Reader/Writers fixed

- Problem: Multiple Readers are accessing "readcount".
  - Solution: Make "increment, test, P" and "decrement, test, V" both atomic – using a mutex.

```
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);
```

Reader/Writers fixed

- Problem: Multiple Readers are accessing “readcount”.
  - Solution: Make “increment, test, P” and “decrement, test, V” both atomic – using a mutex.

```
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);
```

What if a Writer() is active, the first Reader() stalls on P(wrt), and additional Readers() try to enter?

Reader/Writers fixed

- Problem: Multiple Readers are accessing "readcount".
  - Solution: Make "increment, test, P" and "decrement, test, V" both atomic – using a mutex.

```
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);
```

The subsequent Reader() threads stall on P(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...

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 signal() or broadcast() on the CV
  - signal() -- Wake up *one* thread waiting on the CV
  - broadcast() -- Wake up *all* threads waiting on the CV

- In Pthreads, the CV type is a `pthread_cond_t`.
  - Use `pthread_cond_init()` to initialize
  - `pthread_cond_wait(&theCV, &someLock);`
  - `pthread_cond_signal(&theCV);`
  - `pthread_cond_broadcast(&theCV);`

Using Condition Variables

- All condition variable operations **must** be performed while a mutex is locked!!!

```
pthread_mutex_t myLock;
pthread_cond_t myCV;
int counter = 0;

/* Thread A */
pthread_mutex_lock(&myLock);

while (counter < 10) {
    pthread_cond_wait(&myCV,
                      &myLock);
  }

pthread_mutex_unlock(&myLock);
```

```
/* Thread B */
pthread_mutex_lock(&myLock);

counter++;
if (counter >= 10) {
    pthread_cond_signal(&myCV);
}

pthread_mutex_unlock(&myLock);
```

- Why is the lock necessary?

Using Condition Variables

- All condition variable operations **must** be performed while a mutex is locked!!!

```
pthread_mutex_t myLock;
pthread_cond_t myCV;
int counter = 0;

/* Thread A */
pthread_mutex_lock(&myLock);

while (counter < 10) {
    pthread_cond_wait(&myCV,
                      &myLock);
  }

pthread_mutex_unlock(&myLock);
```

```
/* Thread B */
pthread_mutex_lock(&myLock);

counter++;
if (counter >= 10) {
    pthread_cond_signal(&myCV);
}

pthread_mutex_unlock(&myLock);
```

- If no lock on Thread A...
  - Thread might wait 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 while holding a lock prevents a lot of common programming mistakes.*

Using Condition Variables

```
pthread_mutex_t myLock;
pthread_cond_t myCV;
int counter = 0;

/* Thread A */
pthread_mutex_lock(&myLock);

while (counter < 10) {
    pthread_cond_wait(&myCV,
                      &myLock);
  }

pthread_mutex_unlock(&myLock);
```

```
/* Thread B */
pthread_mutex_lock(&myLock);

counter++;
if (counter >= 10) {
    pthread_cond_signal(&myCV);
}

pthread_mutex_unlock(&myLock);
```

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

Using Condition Variables


```
pthread_mutex_t myLock;
pthread_cond_t myCV;
int counter = 0;

/* Thread A */
pthread_mutex_lock(&myLock);

while (counter < 10) {
    pthread_cond_wait(&myCV,
                      &myLock);
  }

pthread_mutex_unlock(&myLock);
```

```
/* Thread B */
pthread_mutex_lock(&myLock);

counter++;
if (counter >= 10) {
    pthread_cond_signal(&myCV);
}

pthread_mutex_unlock(&myLock);
```

Using Condition Variables

```
pthread_mutex_t myLock;
pthread_cond_t myCV;
int counter = 0;
```


```
/* Thread A */
pthread_mutex_lock(&myLock);

while (counter < 10) {
    pthread_cond_wait(&myCV,
                      &myLock);
  }

pthread_mutex_unlock(&myLock);
```

```
/* Thread B */
pthread_mutex_lock(&myLock);

counter++;
if (counter >= 10) {
    pthread_cond_signal(&myCV);
}

pthread_mutex_unlock(&myLock);
```

Using Condition Variables

wait() releases the lock while Thread A is sleeping!!!


```
pthread_mutex_t myLock;
pthread_cond_t myCV;
int counter = 0;

/* Thread A */
pthread_mutex_lock(&myLock);

while (counter < 10) {
    pthread_cond_wait(&myCV,
                      &myLock);
  }

pthread_mutex_unlock(&myLock);
```

```
/* Thread B */
pthread_mutex_lock(&myLock);

counter++;
if (counter >= 10) {
    pthread_cond_signal(&myCV);
}

pthread_mutex_unlock(&myLock);
```

Using Condition Variables


```
pthread_mutex_t myLock;
pthread_cond_t myCV;
int counter = 0;

/* Thread A */
pthread_mutex_lock(&myLock);

while (counter < 10) {
    pthread_cond_wait(&myCV,
                      &myLock);
  }

pthread_mutex_unlock(&myLock);
```

```
/* Thread B */
pthread_mutex_lock(&myLock);

counter++;
if (counter >= 10) {
    pthread_cond_signal(&myCV);
}

pthread_mutex_unlock(&myLock);
```

Using Condition Variables

```
pthread_mutex_t myLock;
pthread_cond_t myCV;
int counter = 0;
```


```
/* Thread A */
pthread_mutex_lock(&myLock);

while (counter < 10) {
    pthread_cond_wait(&myCV,
                      &myLock);
  }

pthread_mutex_unlock(&myLock);
```

```
/* Thread B */
pthread_mutex_lock(&myLock);

counter++;
if (counter >= 10) {
    pthread_cond_signal(&myCV);
}

pthread_mutex_unlock(&myLock);
```

Using Condition Variables

```
pthread_mutex_t myLock;
pthread_cond_t myCV;
int counter = 0;

/* Thread A */
pthread_mutex_lock(&myLock);

while (counter < 10) {
     pthread_cond_wait(&myCV,
                       &myLock);
  }

pthread_mutex_unlock(&myLock);
```


```
/* Thread B */
pthread_mutex_lock(&myLock);

counter++;
if (counter >= 10) {
     pthread_cond_signal(&myCV);
}

pthread_mutex_unlock(&myLock);
```

Using Condition Variables

```
pthread_mutex_t myLock;
pthread_cond_t myCV;
int counter = 0;

/* Thread A */
pthread_mutex_lock(&myLock);

while (counter < 10) {
    pthread_cond_wait(&myCV,
                      &myLock);
  }

pthread_mutex_unlock(&myLock);
```


```
/* Thread B */
pthread_mutex_lock(&myLock);

counter++;
if (counter >= 10) {
    pthread_cond_signal(&myCV);
}

pthread_mutex_unlock(&myLock);
```

Using Condition Variables

```
pthread_mutex_t myLock;
pthread_cond_t myCV;
int counter = 0;

/* Thread A */
pthread_mutex_lock(&myLock);

while (counter < 10) {
    pthread_cond_wait(&myCV,
                      &myLock);
}

pthrea                 ock);
```


```
/* Thread B */
pthread_mutex_lock(&myLock);

counter++;
if (counter >= 10) {
    pthread_cond_signal(&myCV);
}

pthread_mutex_unlock(&myLock);
```

Awake, but cannot proceed out of pthread_cond_wait() ... why?

The lock is still held by Thread B!

Using Condition Variables


```
pthread_mutex_t myLock;
pthread_cond_t myCV;
int counter = 0;

/* Thread A */
pthread_mutex_lock(&myLock);

while (counter < 10) {
   pthread_cond_wait(&myCV,
                     &myLock);
}

pthread_mutex_unlock(&myLock);
```

```
/* Thread B */
pthread_mutex_lock(&myLock);

counter++;
if (counter >= 10) {
    pthread_cond_signal(&myCV);
}

pthread_mutex_unlock(&myLock);
```

Using Condition Variables


```
pthread_mutex_t myLock;
pthread_cond_t myCV;
int counter = 0;

/* Thread A */
pthread_mutex_lock(&myLock);

while (counter < 10) {
    pthread_cond_wait(&myCV,
                      &myLock);
}

pthrea                   ock);
```

Can start running again. But one thing first...

```
/* Thread B */
pthread_mutex_lock(&myLock);

counter++;
if (counter >= 10) {
    pthread_cond_signal(&myCV);
}

pthread_mutex_unlock(&myLock);
```

Using Condition Variables


```
pthread_mutex_t myLock;
pthread_cond_t myCV;
int counter = 0;

/* Thread A */
pthread_mutex_lock(&myLock);

while (counter < 10) {
    pthread_cond_wait(&myCV,
                      &myLock);
}

pthread_mutex_unlock(&myLock);
```

*Thread A **re-acquires** the lock before it resumes execution!*

```
p                       (&myLock);

counter++;
if (counter >= 10) {
    pthread_cond_signal(&myCV);
}

pthread_mutex_unlock(&myLock);
```

Using Condition Variables

```
pthread_mutex_t myLock;
pthread_cond_t myCV;
int counter = 0;

/* Thread A */
pthread_mutex_lock(&myLock);

while (counter < 10) {
    pthread_cond_wait(&myCV,
                      &myLock);
}

pthread_mutex_unlock(&myLock);
```


```
/* Thread B */
pthread_mutex_lock(&myLock);

counter++;
if (counter >= 10) {
    pthread_cond_signal(&myCV);
}

pthread_mutex_unlock(&myLock);
```

Using Condition Variables


```
pthread_mutex_t myLock;
pthread_cond_t myCV;
int counter = 0;

/* Thread A */
pthread_mutex_lock(&myLock);

while (counter < 10) {
    pthread_cond_wait(&myCV,
                      &myLock);
  }

pthread_mutex_unlock(&myLock);
```

```
/* Thread B */
pthread_mutex_lock(&myLock);

counter++;
if (counter >= 10) {
    pthread_cond_signal(&myCV);
}

pthread_mutex_unlock(&myLock);
```

__Key ideas:__

- wait() on a CV releases the lock
- signal() on a CV wakes up a thread waiting on a lock
- The thread that wakes up has to re-lock before wait() returns

Bounded buffer using CVs

```
int theArray[ARRAY_SIZE], size;
pthread_mutex_t theLock;
pthread_cond_t theCV;

/* Initialize */
pthread_mutex_init(&theLock, NULL);
pthread_condvar_init(&theCV, NULL);

void put(int val) {
    pthread_mutex_lock(&theLock);
    while (size == ARRAY_SIZE) {
      pthread_cond_wait(&theCV,
                        &theLock);
    }
    addItemToArray(val);
    size++;
    if (size == 1) {
      pthread_cond_signal(&theCV);
    }
    pthread_mutex_unlock(&theLock);
}
```

```
int get() {
  int item;
  pthread_mutex_lock(&theLock);
  while (size == 0) {
    pthread_cond_wait(&theCV,
                      &theLock);
  }
  item = getItemFromArray();
  size--;
  if (size == ARRAY_SIZE-1) {
    pthread_cond_signal(&theCV);
  }
  pthread_mutex_unlock(&theLock);
  return item;
}
```

What's wrong with this code?

Bounded buffer using CVs

```
int theArray[ARRAY_SIZE], size;
pthread_mutex_t theLock;
pthread_cond_t theCV;

/* Initialize */
pthread_mutex_init(&theLock, NULL);
pthread_condvar_init(&theCV, NULL);

void put(int val) {
    pthread_mutex_lock(&theLock);
    while (size == ARRAY_SIZE) {
      pthread_cond_wait(&theCV,
                        &theLock);
    }
    addItemToArray(val);
    size++;
    if (size == 1) {
      pthread_cond_signal(&theCV);
    }
    pthread_mutex_unlock(&theLock);
}
```

```
int get() {
  int item;
  pthread_mutex_lock(&theLock);
  while (size == 0) {
    pthread_cond_wait(&theCV,
                      &theLock);
  }
  item = getItemFromArray();
  size--;
  if (size == ARRAY_SIZE-1) {
    pthread_cond_signal(&theCV);
  }
  pthread_mutex_unlock(&theLock);
  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;
pthread_mutex_t theLock;
pthread_cond_t theCV;

/* Initialize */
pthread_mutex_init(&theLock, NULL);
pthread_condvar_init(&theCV, NULL);

void put(int val) {
    pthread_mutex_lock(&theLock);
    while (size == ARRAY_SIZE) {
      pthread_cond_wait(&theCV,
                        &theLock);
    }
    addItemToArray(val);
    size++;

    pthread_cond_signal(&theCV);

    pthread_mutex_unlock(&theLock);
}
```

```
int get() {
  int item;
  pthread_mutex_lock(&theLock);
  while (size == 0) {
    pthread_cond_wait(&theCV,
                      &theLock);
  }
  item = getItemFromArray();
  size--;
  if (size == ARRAY_SIZE-1) {
    pthread_cond_broadcast(
                      &theCV);
  }
  pthread_mutex_unlock(&theLock);
  return item;
}
```

One fix: Always signal.

Less efficient: higher overhead.
Though it does not <u>hurt</u> to signal
a CV that no thread is waiting on.

How to fix this problem?

```
int theArray[ARRAY_SIZE], size;
pthread_mutex_t theLock;
pthread_cond_t theCV;

/* Initialize */
pthread_mutex_init(&theLock, NULL);
pthread_condvar_init(&theCV, NULL);

void put(int val) {
    pthread_mutex_lock(&theLock);
    while (size == ARRAY_SIZE) {
     pthread_cond_wait(&theCV,
                       &theLock);
    }
    addItemToArray(val);
    size++;
    if (size == 1) {
      pthread_cond_broadcast(&theCV);
    }
    pthread_mutex_unlock(&theLock);
}
```

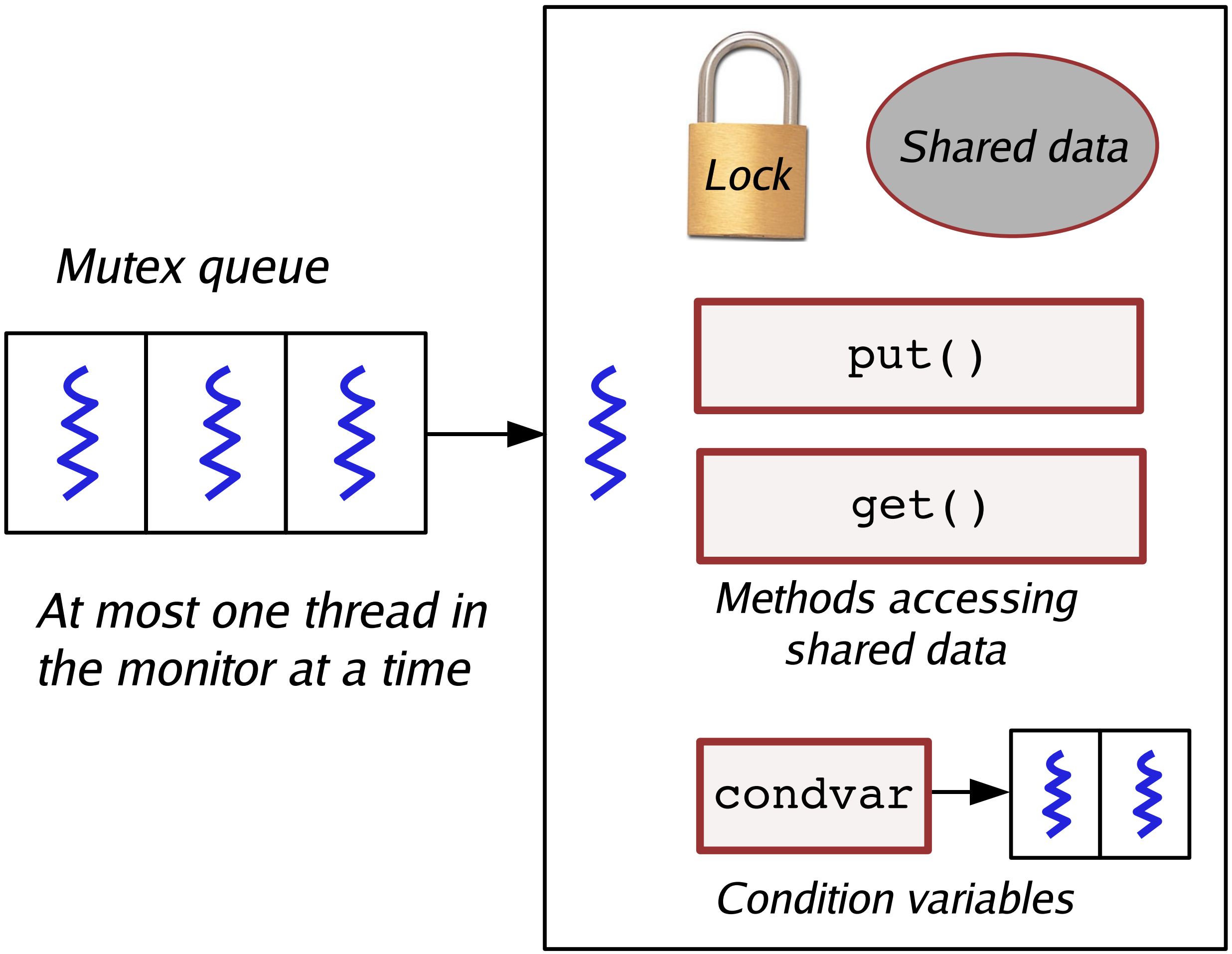
```
int get() {
  int item;
  pthread_mutex_lock(&theLock);
  while (size == 0) {
    pthread_cond_wait(&theCV,
                      &theLock);
  }
  item = getItemFromArray();
  size--;
  if (size == ARRAY_SIZE-1) {
    pthread_cond_broadcast(
                         &theCV);
  }
  pthread_mutex_unlock(&theLock);
  return item;
}
```

Another fix: use broadcast()

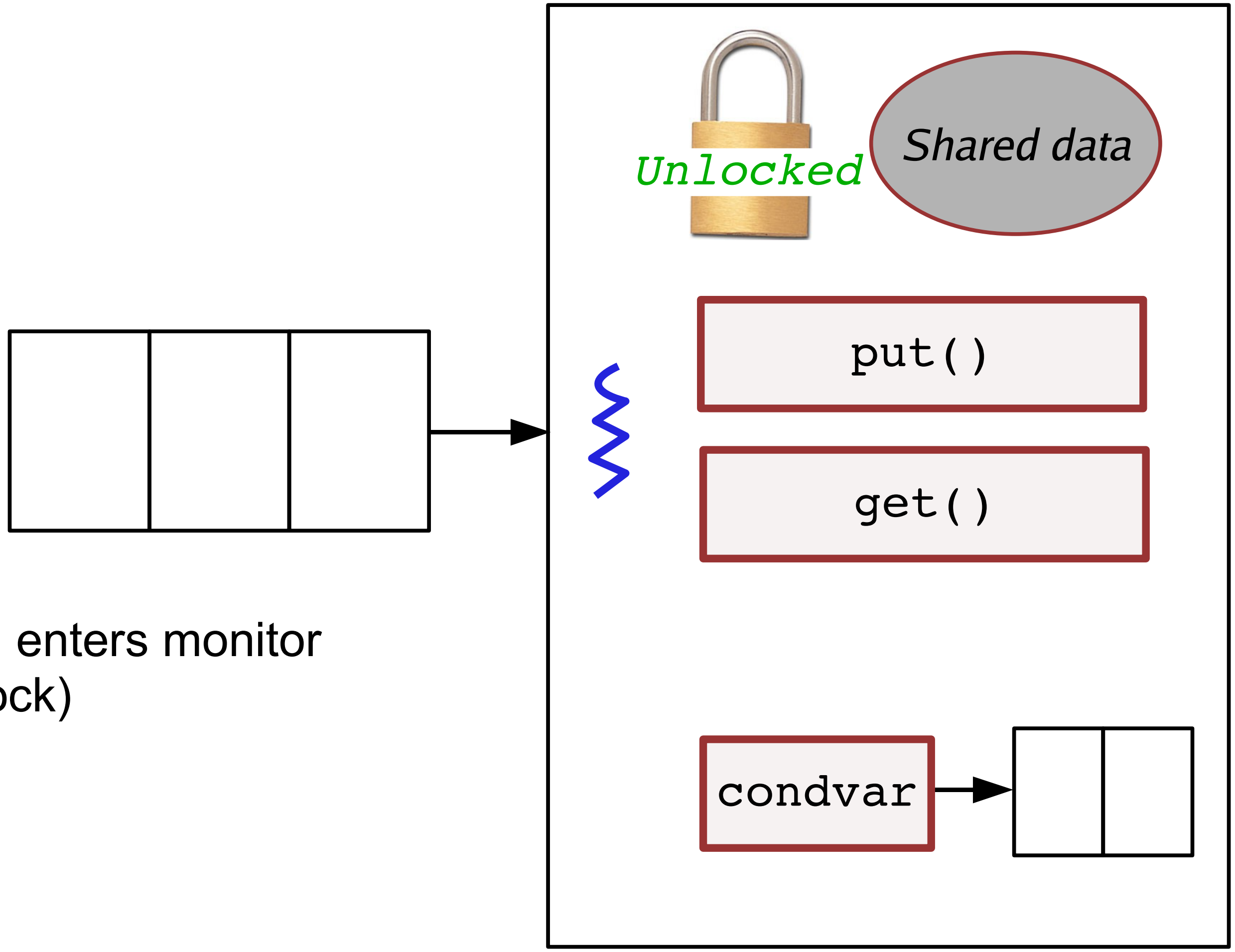
Wakes up all threads when the condition changes.
But note!!! Only **one** thread will grab the lock when it wakes up. The others wake up and immediately wait to acquire the lock again.

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.

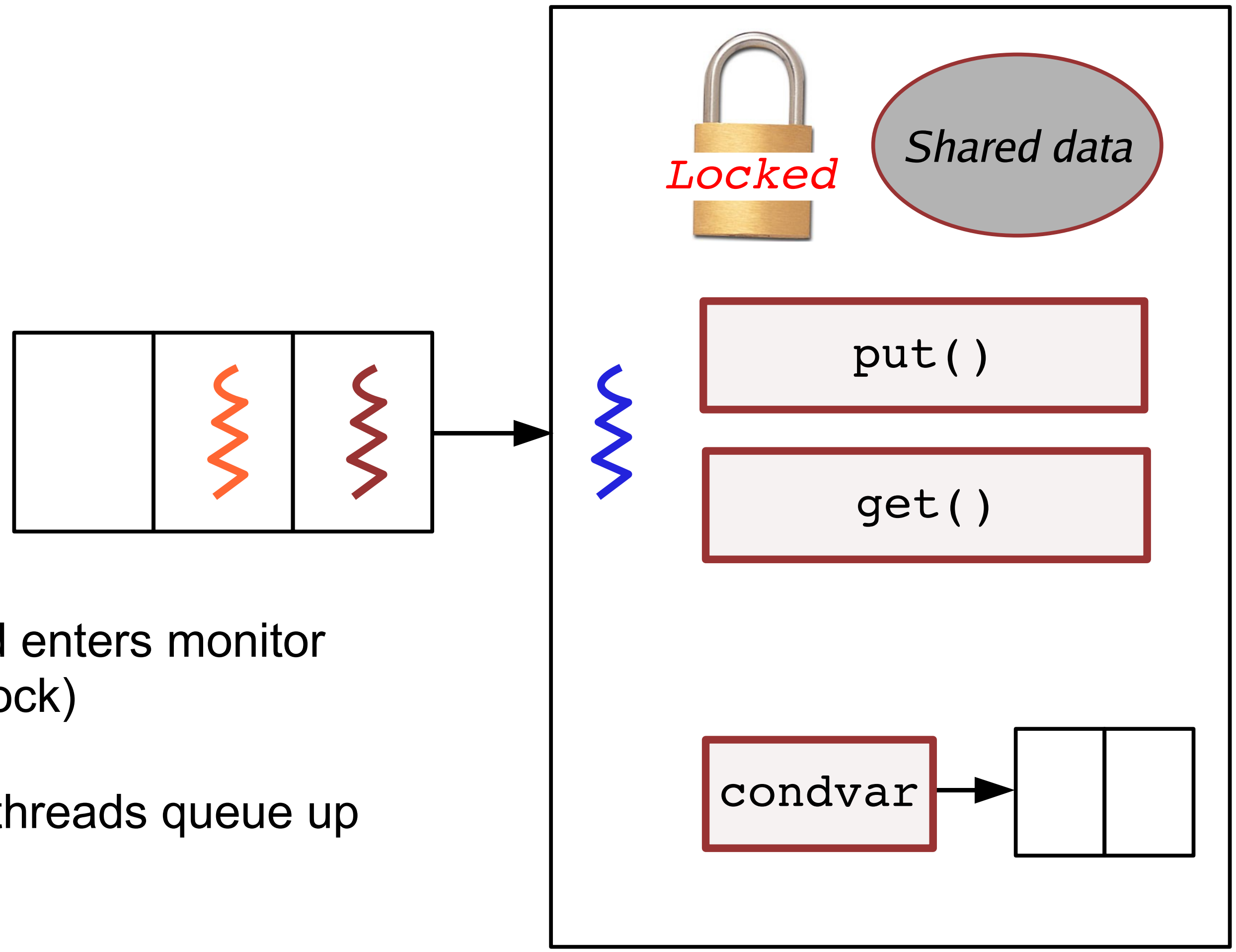


Monitors



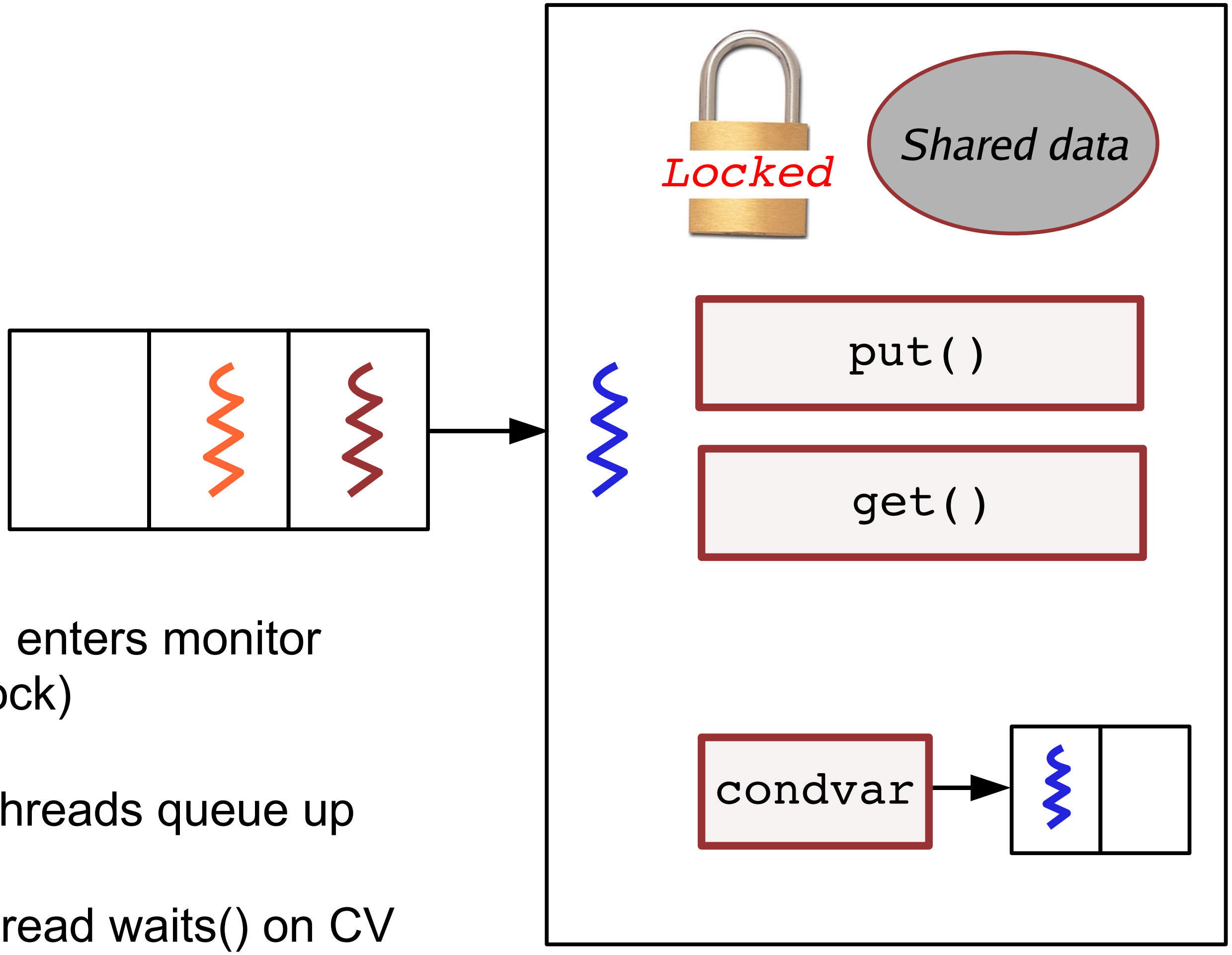
- 1) Thread enters monitor (grabs lock)

Monitors



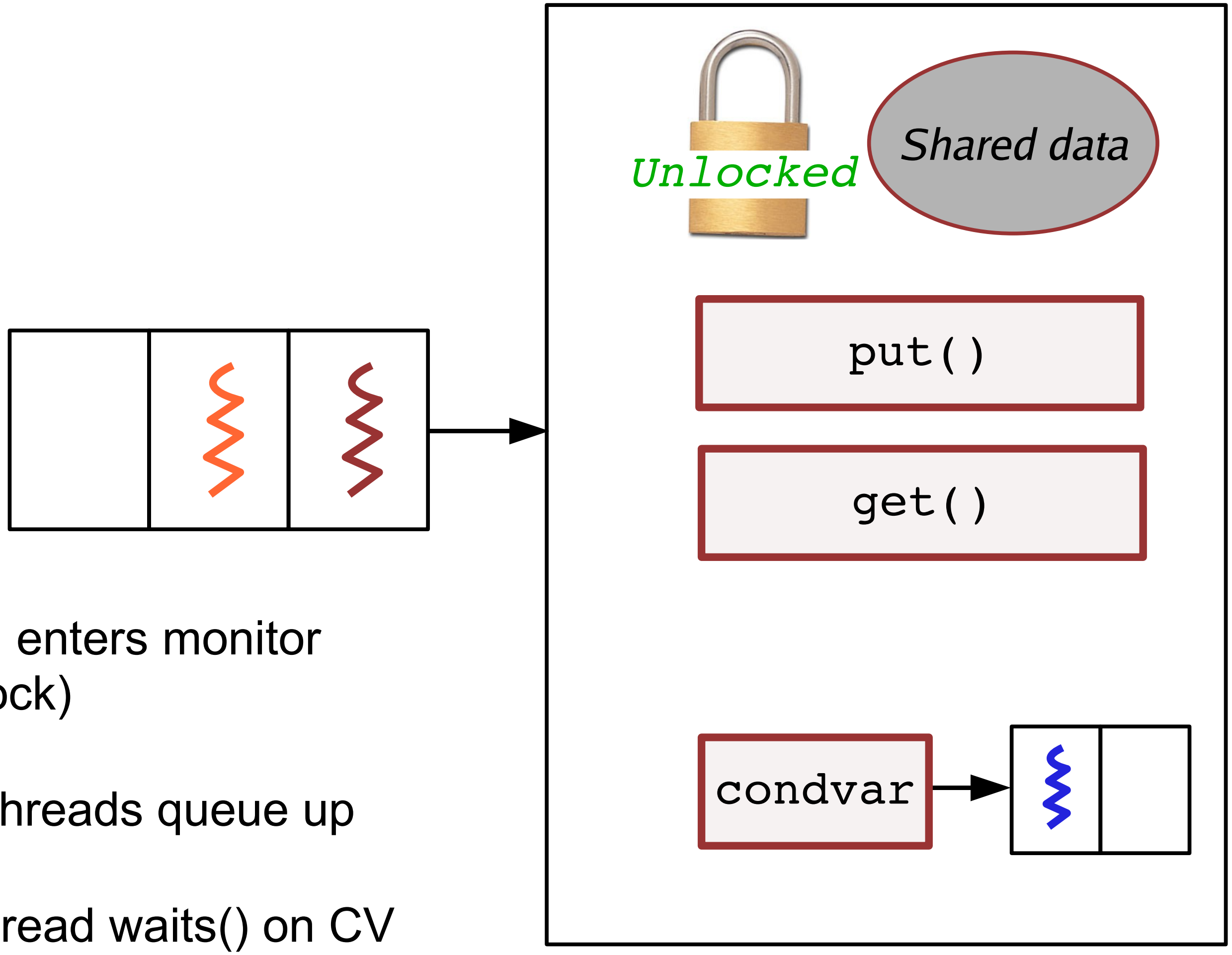
- 1) Thread enters monitor (grabs lock)
- 2) Other threads queue up

Monitors



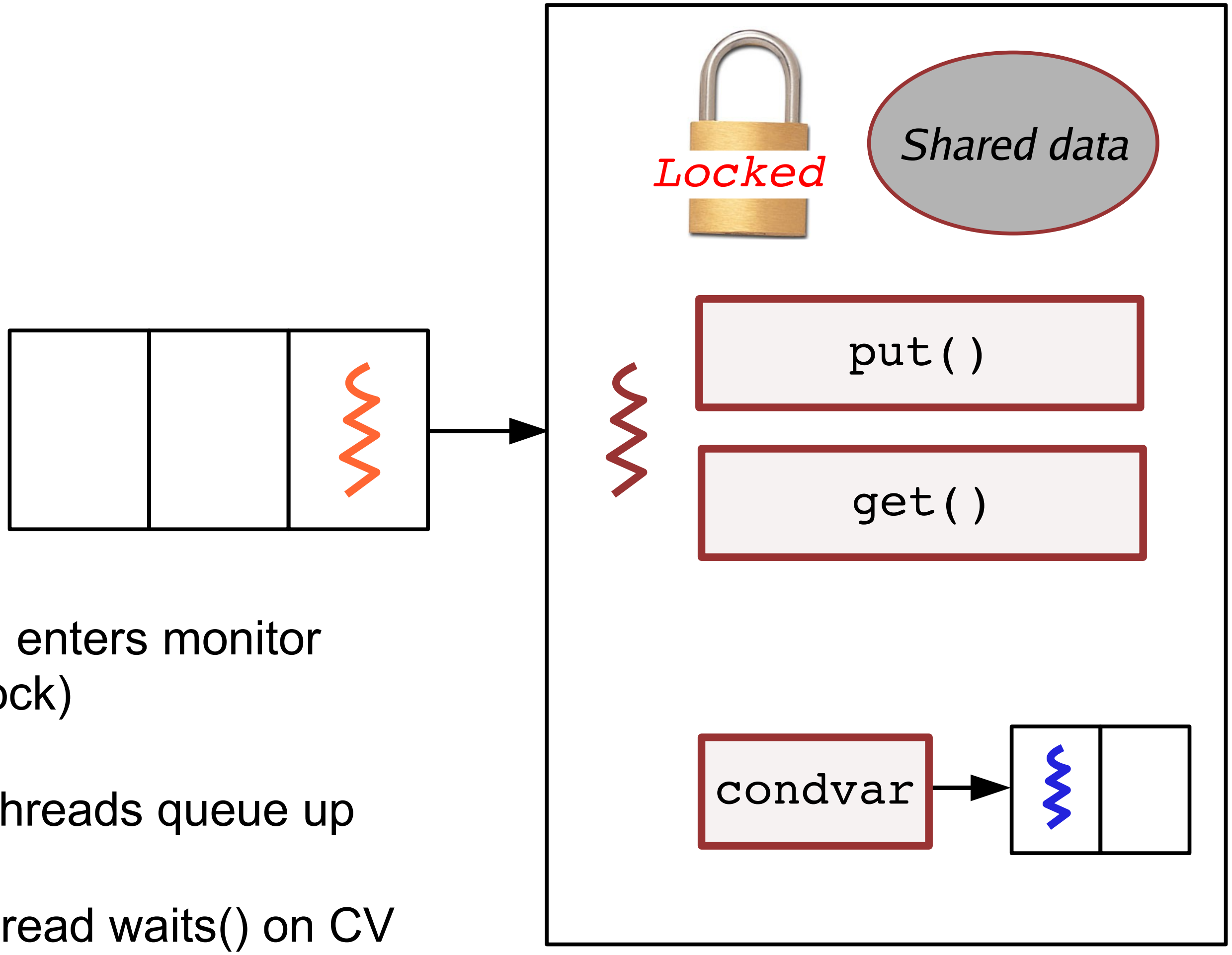
- 1) Thread enters monitor (grabs lock)
- 2) Other threads queue up
- 3) Blue thread waits() on CV

Monitors



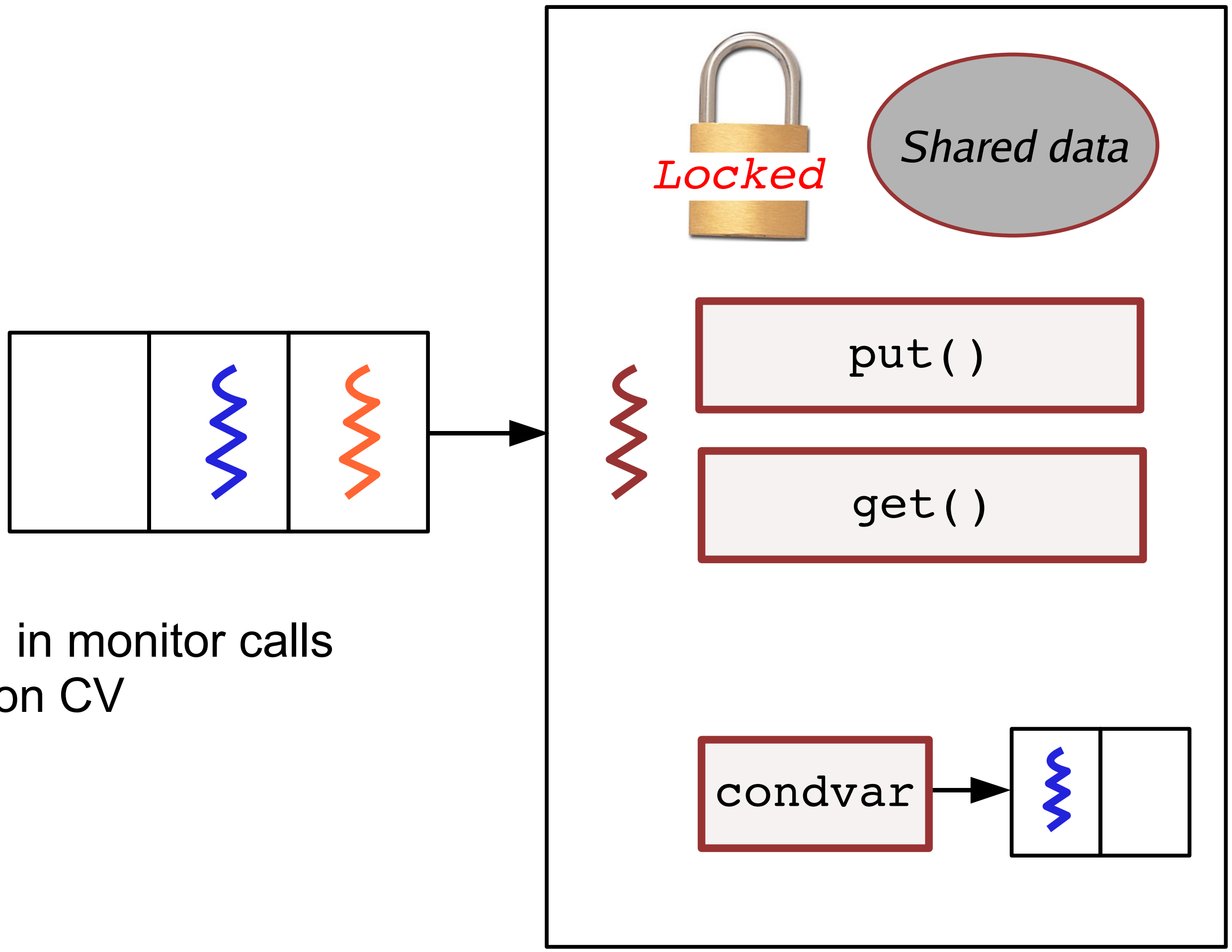
- 1) Thread enters monitor (grabs lock)
- 2) Other threads queue up
- 3) Blue thread waits() on CV
- 4) Next thread enters monitor

Monitors



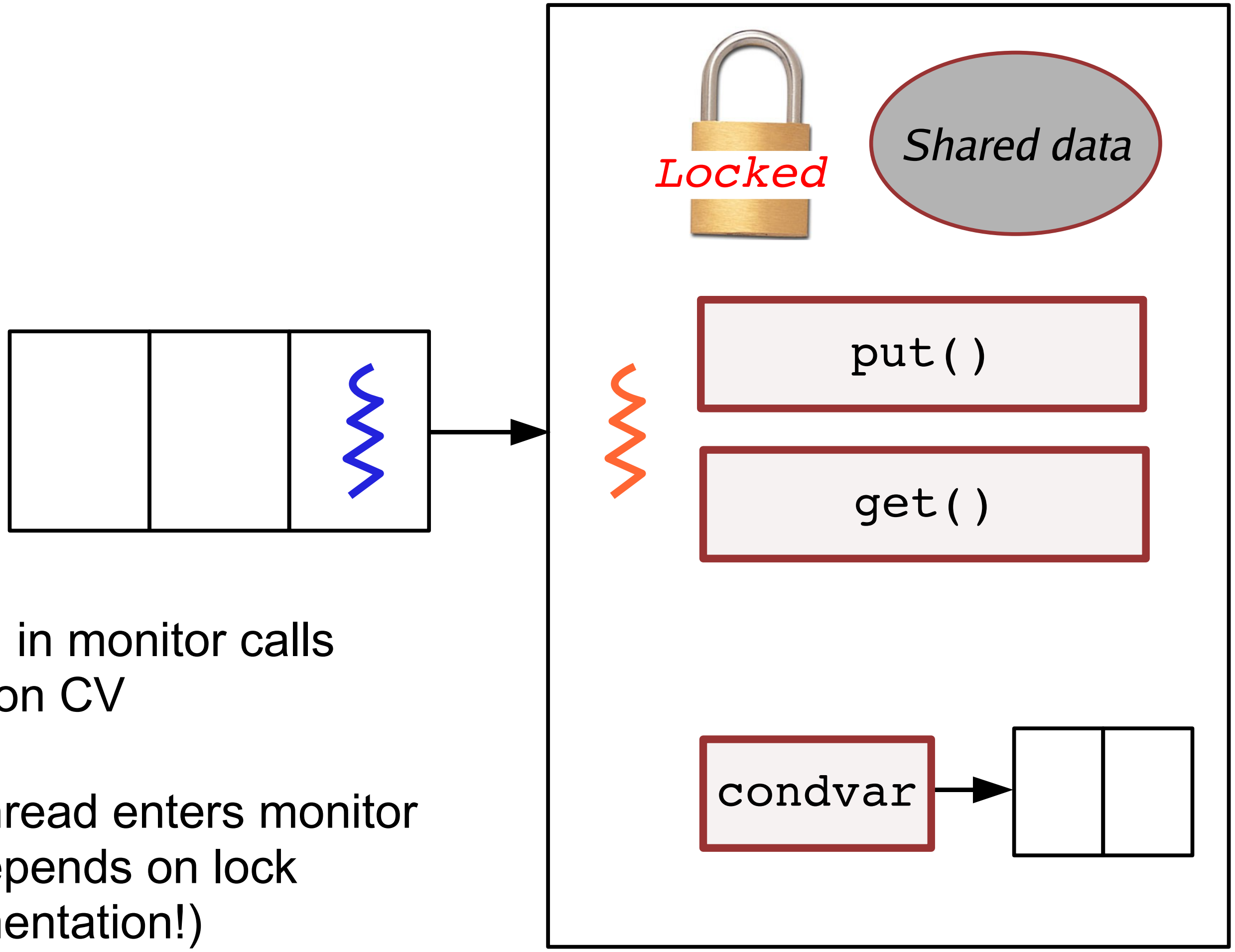
- 1) Thread enters monitor (grabs lock)
- 2) Other threads queue up
- 3) Blue thread waits() on CV
- 4) Next thread enters monitor

Monitors



- 5) Thread in monitor calls signal() on CV

Monitors



- 5) Thread in monitor calls signal() on CV
- 6) Next thread enters monitor (order depends on lock implementation!)

Hoare vs. Mesa Monitor Semantics

- The monitor signal() operation can have two different meanings:
- Hoare monitors (1974)
  - signal(CV) means to run the waiting thread *immediately*
  - Effectively "hands the lock" to the thread just signaled.
  - Causes the signaling thread to block
- Mesa monitors (Xerox PARC, 1980)
  - signal(CV) puts waiting thread back onto the "ready queue" for the monitor
  - But, signaling thread keeps running.
  - Signaled thread doesn't get to run until it can acquire the lock.
    - *This is what we almost always use – so do Pthreads, Java, C#, etc.*
- 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*

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.
- *Monitors* are a “pattern” for using locks and condition variables that is often very useful.

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