HARVARD
School of Engineering and Applied Sciences

Semaphores, Condition Variables, and Monitors

CS61, Lecture 19

Prof. Stephen Chong

November 8, 2011

Announcements

- Assignment 5 Bank lab
  - If you haven't yet told us who you are working with, please do it now!

Last time

- We looked at **locks**
  - Two operations: acquire and release
  - At most one thread can hold a lock at a time
  - Can use to enforce mutual exclusion and critical sections
  - Considered how to efficiently implement

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?

Today

- Semaphores
- Condition variables
- Monitors

Semaphores

- Higher-level synchronization construct
  - Designed by Edsger Dijkstra in the 1960's
- 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: **Increment** semaphore value by 1.

Semaphore Example

- Semaphores can be used to implement locks:

```
Semaphore my_semaphore = 1; // Initialize to nonzero
int withdraw(account, amount) {
  wait(my_semaphore);
  balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  signal(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
}
```

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

```
signal(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);
     }
```

wait() and signal() must be atomic actions!

Simple Semaphore Implementation

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

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

Why is this a while loop, and not an if?

wait could be called by another thread while this thread is waiting

```
signal(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);
     }
```

Semaphore Implementation

- How do we ensure that the semaphore implementation is atomic?
- One option: use a lock for wait() and signal()
  - Make sure that only one wait() or signal() can be executed by any process at a time
  - Need to be careful to release lock before sleeping, acquire lock on waking up
- Another option: hardware support

Why are semaphores useful?

- A binary semaphore (counter is always 0 or 1) is basically a lock.
  - Start with semaphore value = 1
  - acquire( ) = wait( )
  - release( ) = signal( )
- The real value of semaphores becomes apparent when the counter can be initialized to a value other than 0 or 1.

The Producer/Consumer Problem

- Also called the Bounded Buffer problem.

Mmmm... donuts

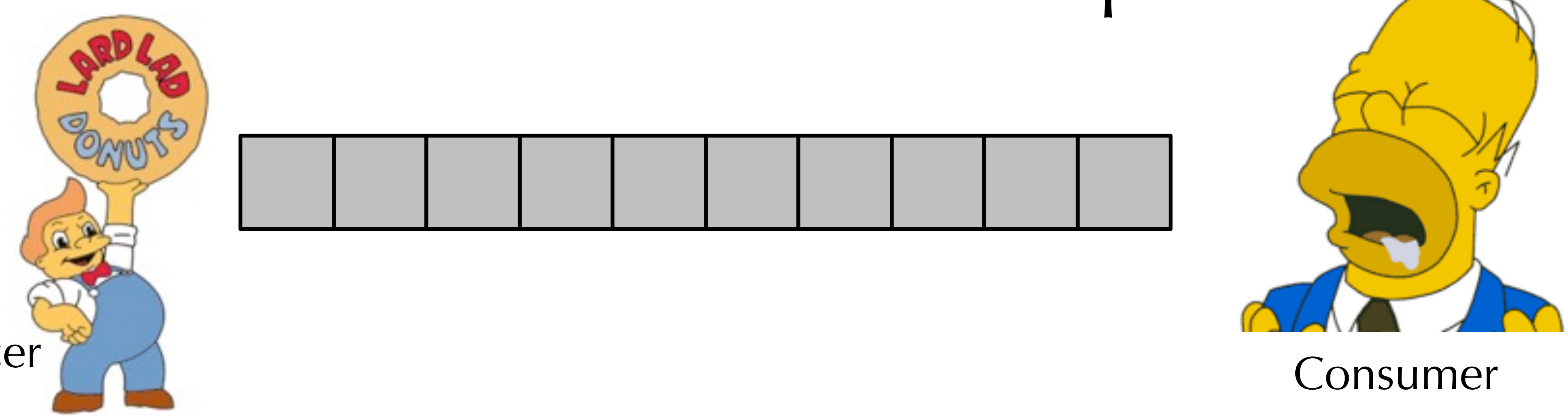
Producer

Consumer

- 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.

An implementation

Mmmm... donuts

Proc

```
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?

An implementation

Mmmm... donuts

Proc

```
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);
  }
}
```

Access to `count` not synchronized

What if we context switch between the test and sleep?

```
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?

An implementation with semaphores

Mmmm... donuts

Proc

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

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

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

Why is it important that `wait(empty)` is before `wait(mutex)`?

Otherwise a thread could acquire mutex and wait for empty; prevent another thread acquiring mutex. DEADLOCK! (more on this next week)

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() {
  wait(wrt);
  do_write();
  signal(wrt);
}
```

```
Reader() {

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

  do_read();

  readcount--;
  if (readcount == 0) {
    signal(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 time
    - (And, not interfere with any readers...)

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

Writer() {
  wait(wrt);
  do_write();
  signal(wrt);
}
```

```
Reader() {

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

  do_read();

  readcount--;
  if (readcount == 0) {
    signal(wrt);
  }

}
```

What if we context switch here?

Another thread might increment readcount, and readcount==1 never happens

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

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

Writer() {
  wait(wrt);
  do_write();
  signal(wrt);
}
```

```
Reader() {

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

  do_read();

  readcount--;
  if (readcount == 0) {
    signal(wrt);
  }

}
```

What if we context switch here?

A writer thread might get the `wrt` lock, and subsequent reader threads run without the lock!

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

Reader/Writers

- Problem: `readcount` is accessed by multiple threads concurrently without synchronization!
- Solution: Make "increment, test, wait" and "decrement, test, signal" atomic, by using a mutex.

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

Writer() {
  wait(wrt);
  do_write();
  signal(wrt);
}
```

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

Semaphore library

- There are POSIX semaphores, but they are not part of the pthreads library
- All semaphore functions are declared in `semaphore.h`
- The semaphore type is a `sem_t`.
- Intialize: `sem_init(&theSem, 0, initialVal);`
- Wait: `sem_wait(&theSem);`
- Signal: `sem_post(&theSem);`
- Get the current value of the semaphore:

```
sem_getvalue(&theSem, &result);
```

Issues with Semaphores

- Much of the power of semaphores derives from calls to wait() and signal() that are unmatched
  - See previous example!
  - Unlike locks, where acquire() and release() are always paired.
- This means it is a lot easier to get into trouble with semaphores.
  - Semaphores are a lot of rope to tie yourself in knots with…

Today

- Semaphores
- Condition variables
- Monitors

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.
- Condition variable indicates an event; cannot store or retrieve a value from a CV
- 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

```
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);
```

- In pthreads, all condition variable operations **must** be performed while a mutex is locked!!!
  - Why is the lock necessary?

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);
```

- If no lock on Thread A:
  - Might wait after another thread sets counter to 10
- If no lock on Thread B:
  - No guarantee that increment and test is atomic

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

```
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);
```

- `wait()` released the lock while Thread A is sleeping
  - That is why pthreads requires that the `myLock` is passed in

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);
```

- `signal()` wakes up Thread A, but Thread A cannot proceed. Why?
  - Thread A requires lock to continue. 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);
```

- `signal()` wakes up Thread A, but Thread A cannot proceed. Why?
  - Thread A requires lock to continue. 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);
```

- Once Thread B releases lock, Thread A can acquire it and continue running

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 the CV
  - The thread that wakes up has to re-acquire the lock before `wait()` returns

Bounded buffer using CVs

Mmmm... donuts

Pr

```
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);
}
```

What's wrong with this code?

```
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;
}
```

Bounded buffer using CVs

Pro

donuts

```
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);
}
```

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!!

```
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;
}
```

Bounded buffer using CVs

One fix: **always signal**

Less efficient but OK.

```
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--;

    pthread_cond_signal(&theCV);

    pthread_mutex_unlock(&theLock);
    return item;
}
```

Bounded buffer using CVs

Another fix: **use broadcast()**

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

Pro

```
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;
}
```

Today

- Semaphores
- Condition variables
- Monitors

Monitors

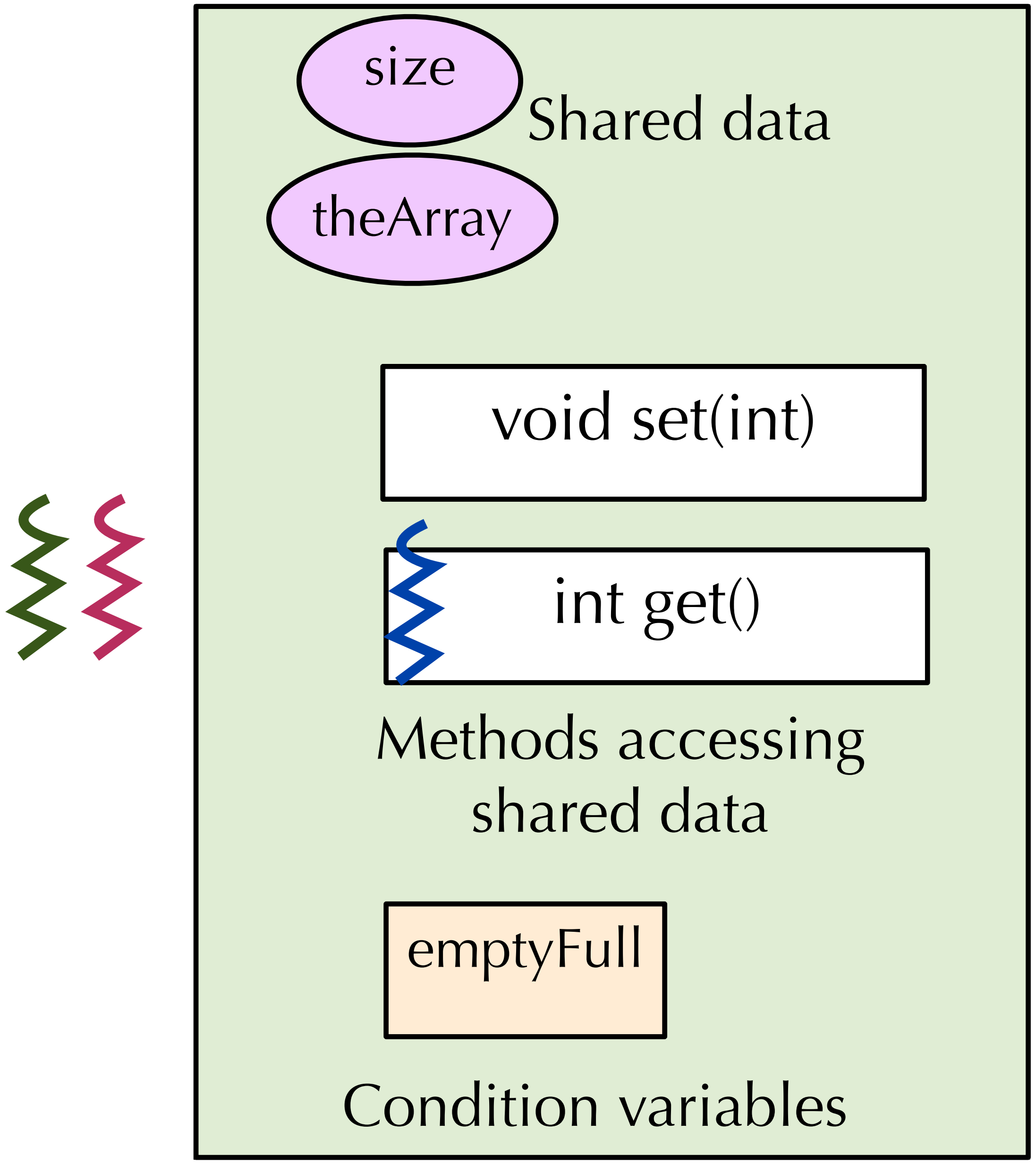
- A monitor uses this style of locks and condition variables to protect resources and coordinate threads
- A **monitor** is an object containing variables, condition variables, and methods
- At most one thread can be active in a monitor at a time

```
monitor M {
    int size, theArray[ARRAY_SIZE];
    ConditionVariable emptyFull;
    void put(int x) {
      if (size == ARRAY_SIZE) wait(emptyFull);
      theArray[size] = x;
      size++;
      if (size == 1) broadcast(emptyFull);
    }
    int get() {
      if (size == 0) wait(emptyFull);
      size--;
      if (size == ARRAY_SIZE-1) broadcast(emptyFull);
      return theArray[size];
    }
}
```

Monitors

1) Blue thread enters monitor

2) Other threads queue up

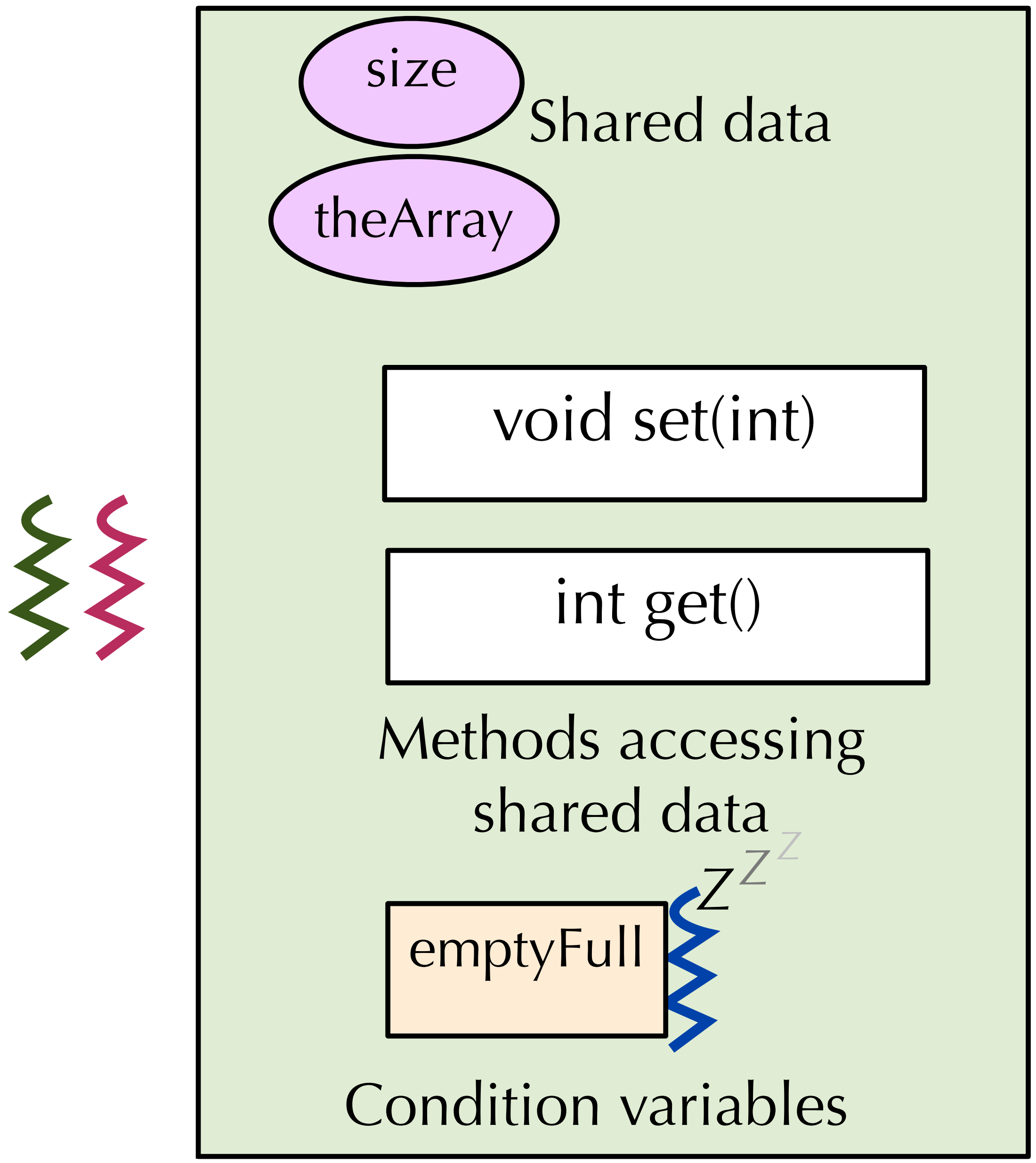


Monitors

1) Blue thread enters monitor

2) Other threads queue up

3) Blue thread waits on CV



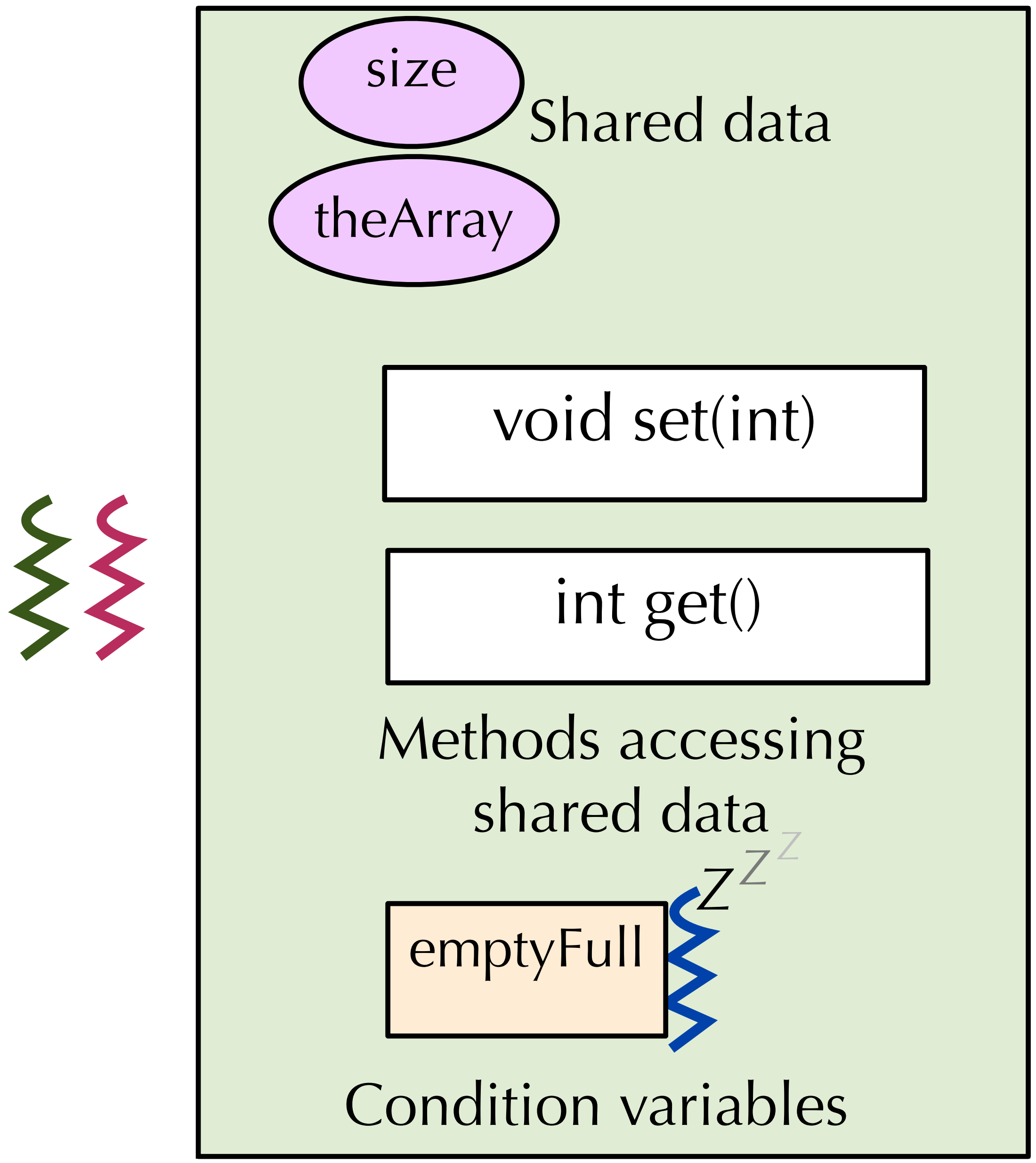
Monitors

1) Blue thread enters monitor

2) Other threads queue up

3) Blue thread waits on CV

4) Another thread (pink) can enter monitor



Monitors

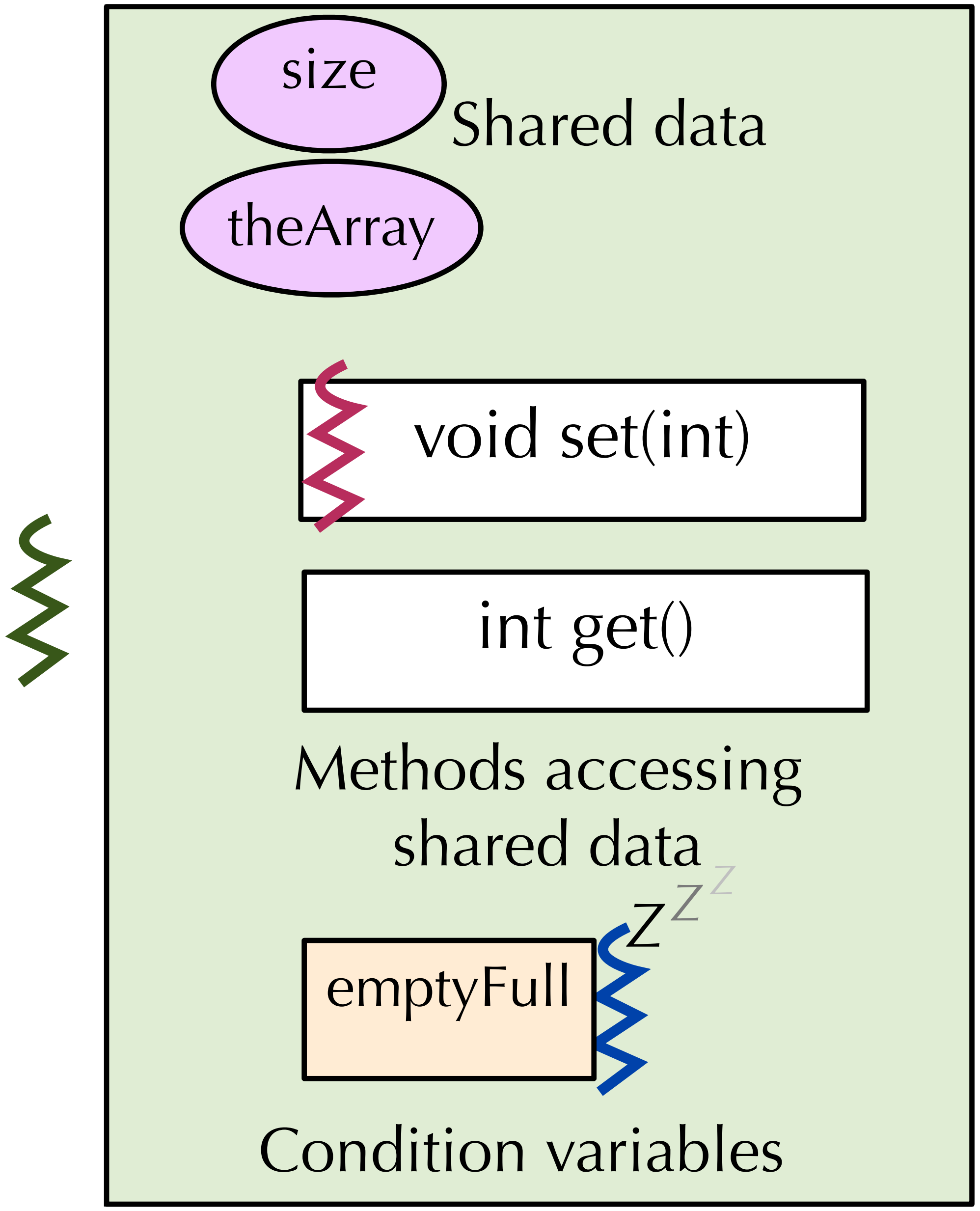
1) Blue thread enters monitor

2) Other threads queue up

3) Blue thread waits on CV

4) Another thread (pink) can enter monitor

5) Pink thread calls signal. What happens now?



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 snuck in and invalidated the condition

Monitors

1) Blue thread enters monitor

2) Other threads queue up

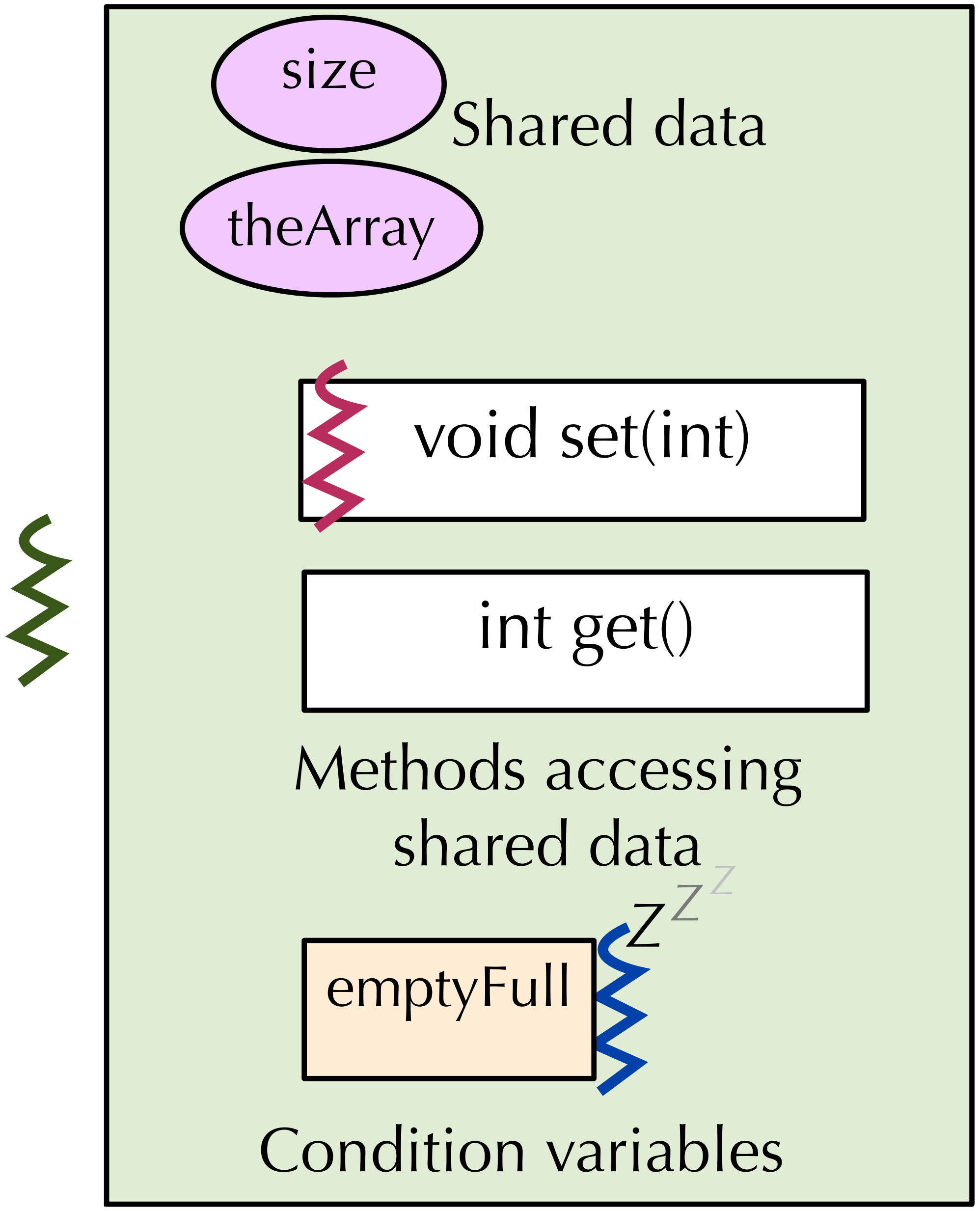
3) Blue thread waits on CV

4) Another thread (pink) can enter monitor

5) Pink thread calls signal. What happens now?

6) Pink thread leaves monitor

7) Another thread can enter monitor
(which depends on implementation)



Java thread synchronization

- Java uses a form of monitors
- Every object can be a lock and a condition variable
- A thread executing a method `m` of object `o` marked `synchronized` must acquire lock `o` before executing
- Given an object `o`, can call `o.wait()`, `o.notify()`, `o.notifyAll()`

Bounded buffer in Java

```
class BoundedBuffer {
    private int size;
    private int theArray[ARRAY_SIZE];

    public synchronized void put(int x) {
      while (size == ARRAY_SIZE) this.wait();
      theArray[size] = x;
      size++;
      if (size == 1) this.notifyAll();
    }

    public synchronized int get() {
      while (size == 0) this.wait();
      size--;
      if (size == ARRAY_SIZE-1) this.notifyAll();
      return theArray[size];
    }
}
```

- Almost, not quite. Some subtleties in using wait and notify.

The Big Picture

- Getting synchronization right is hard!
  - Even your TFs and faculty have been known to get it wrong.
  - Testing isn't enough.
  - Need to assume worst case: all interleavings are possible
- We need to synchronize for correctness
  - Unsynchronized code can cause incorrect behavior
  - But too much synchronization means threads spend a lot of time waiting, not performing productive work.

The Big Picture

- How to choose between locks, semaphores, condition variables, monitors?
- Locks are very simple and suitable 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