ECE 570/670

David Irwin

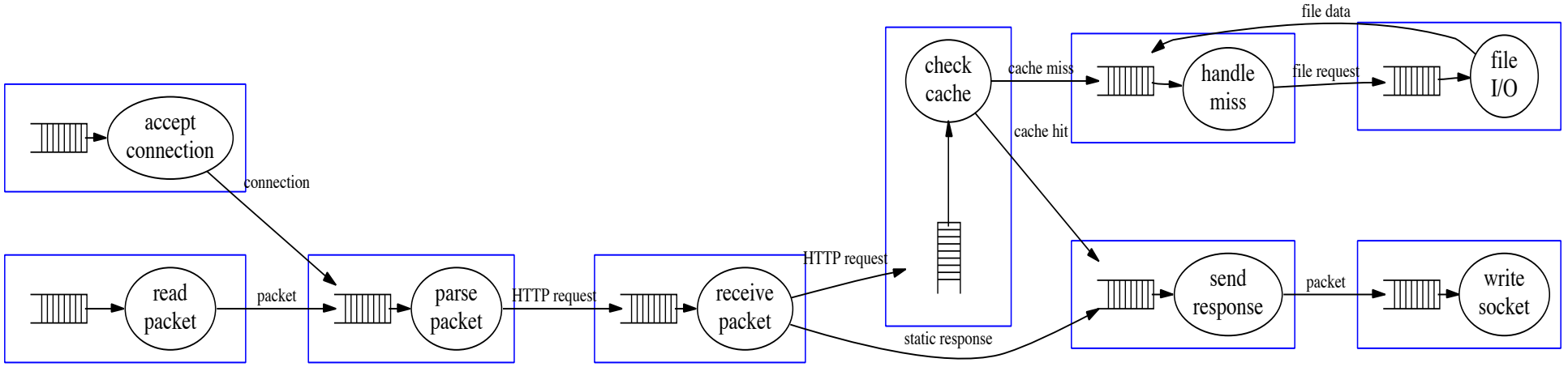
Lecture 6

Outline

- Virtualization paper discussion
  - Paper posted for Thursday
  - **End to End Arguments in Systems Design**
- ECE670 project signups
  - Only 1 group signed up so far
- Continue synchronization
- Maybe get to Assignment 1 discussion

Virtualization Considered Harmful: OS Design Directions for Well-Conditioned Services

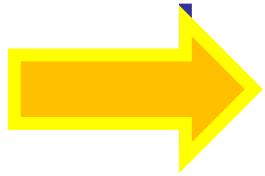
- New server architecture
  - Staged Event Driven Architecture (SEDA)
  - Set of event-driven stages separated by queues
- Benefits
  - Small number of threads (high concurrency)
  - Queues individually "load-conditioned"
  - Modularity of stages (aids debugging)



How about this?

I'm always holding a lock while accessing shared state.

ptr may not point to tail after unlock/lock.



```
enqueue () {
   lock (qLock)
   node *ptr;
   for (ptr=head;
        ptr->next!=NULL;
        ptr=ptr->next){}
   unlock (qLock);
   lock (qLock);
   ptr->next=new_element;
   new_element->next=NULL;
   unlock(qLock);
}
```

Lesson:

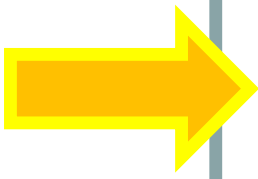
- Thinking about individual accesses is not enough
- Must reason about dependencies *between* accesses too

Next up

- Mutual exclusion is great and all, but…
  - It's not really expressive enough
- Ordering constraints
  - Often must wait for something to happen
  - Introduce monitors and condition variables

Intro to ordering constraints

- Say you want dequeue to wait while the queue is empty
- Can we just busy-wait?
  - No!
  - Still holding lock

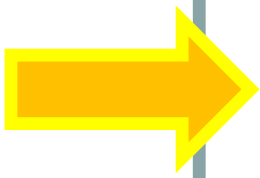


```
dequeue () {
  lock (qLock);
  element=NULL;
  while (head==NULL) {}
  // remove head
  element=head->next;
  head->next=NULL;
  unlock (qLock);
  return element;
}
```

Release lock before spinning?

What can go wrong?

- Another dequeuer could "steal" our element
- Head might be NULL when we try to remove entry

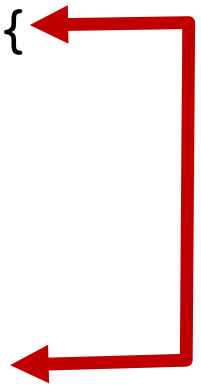


```
dequeue () {
   lock (qLock);
   element=NULL;
   unlock (qLock);
   while (head==NULL) {}
   lock (qLock);
   // remove head
   element=head->next;
   head->next=NULL;
   unlock (qLock);
   return element;
}
```

One more try

- Does it work?
  - Seems ok
- Why?
  - Shared state is protected
- What's wrong?
  - Busy-waiting
  - Wasteful

```
dequeue () {
  lock (qLock);
  element=NULL;
  while (head==NULL) {
    unlock (qLock);
    lock (qLock);
  }
  // remove head
  element=head->next;
  head->next=NULL;
  unlock (qLock);
  return element;
}
```



Ideal solution

- Would like dequeueing thread to "sleep"
  - Add self to "waiting list"
  - Enqueuer can wake up waiting threads when queue is non-empty
- **Problem:** what to do with the lock?
  - Can dequeueing thread sleep with lock?
    - No!
    - Enqueuer would never be able to add to queue

Release the lock before sleep?

```
enqueue () {
  acquire lock
  find tail of queue
  add new element
  if (dequeuer waiting){
    remove from wait list
    wake up dequeuer
  }
  release lock
}
```

```
dequeue () {
  acquire lock
  …
  if (queue empty) {
    release lock
    add self to wait list
    sleep
  }
  …
  release lock
}
```

Does this work?

Release the lock before sleep?

```
enqueue () {
  acquire lock
  find tail of queue
  add new element
  if (dequeuer waiting){
    remove from wait list
    wake up dequeuer
  }
  release lock
}
```

```
dequeue () {
  acquire lock
  …
  if (queue empty) {
    release lock
    add self to wait list
    sleep
  }
  …
  release lock
}
```

Thread can sleep forever!

Other problems?

Wait list is shared and unprotected. (bad idea)

Release the lock before sleep?

```
enqueue () {
  acquire lock
  find tail of queue
  add new element
  if (dequeuer waiting){
    remove from wait list
    wake up dequeuer
  }
  release lock
}
```

```
dequeue () {
  acquire lock
  …
  if (queue empty) {
    add self to wait list
    release lock
    sleep
  }
  …
  release lock
}
```

Release the lock before sleep?

```
enqueue () {
  acquire lock
  find tail of queue
  add new element
  if (dequeuer waiting){
    remove from wait list
    wake up dequeuer
  }
  release lock
}
```

2

```
dequeue () {
  acquire lock
  …
  if (queue empty) {
    add self to wait list
    release lock
    sleep
  }
  …
  release lock
}
```

1

3

Problem: missed wake-up

Note: this can be fixed, but it's messy

Two types of synchronization

- **As before we need to raise the level of abstraction**

1. Mutual exclusion
   - One thread doing something at a time
   - Use locks
2. Ordering constraints
   - Describes "before-after" relationships
   - One thread waits for another
   - Use monitors

Review

- First we discussed atomic actions
  - i.e., load, store at hardware layer
  - Guaranteed to happen without being interrupted
  - Not "strong" enough to provide mutual exclusion
- Then we discussed locks/mutexes
  - Provided mutual exclusion
  - Prevented two threads from accessing shared data simultaneously
  - Commonly used for "identical" threads running in parallel
  - Cannot provide ordering
- Now we are going to discuss monitors
  - Provides ordering (before-after relationship)

Monitor synchronization

1. Mutual exclusion
   - One thread doing something at a time
   - Use locks
2. Ordering constraints
   - Describes "before-after" relationships
   - One thread waits for another

- **Monitor:** a lock + its **condition variable**

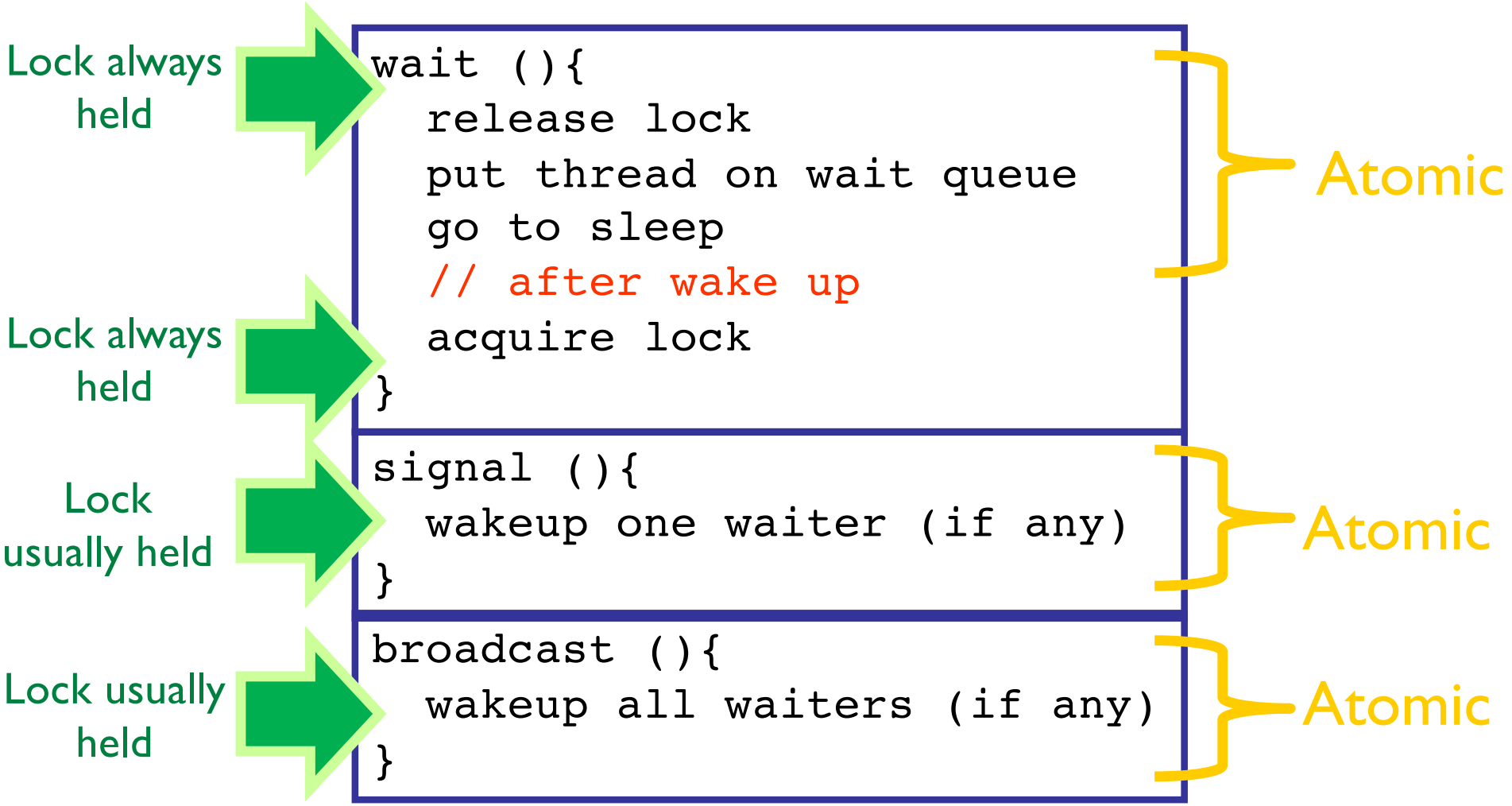
Locks and condition variables

- **Condition variable**
  - Lets threads sleep inside a critical section
- Internal atomic actions (for now, by definition)

```
// begin atomic
release lock
put thread on wait queue
go to sleep
// end atomic
```

- CV State = queue of waiting threads + one lock

Condition variable operations



CVs and invariants

- User programs
  - Ok to leave invariants violated before wait?
  - **No: wait releases the lock**
- Larger rule about returning from wait
  - Lock may have changed hands
  - State can change between wait entry and return
  - Don't make assumptions about shared state

Multi-threaded queue

```
enqueue () {
  acquire lock

  find tail of queue
  add new element

  signal (lock, CV)

  release lock
}
```

```
dequeue () {
  acquire lock

  if (queue empty) {
    wait (lock, CV)
  }

  remove item from queue
  release lock
  return removed item
}
```

Any problems with the "if" statement in dequeue?

Multi-threaded queue

```
enqueue () {
  acquire lock

  find tail of queue
  add new element

  signal (lock, CV)

  release lock
}
```

```
dequeue () {
  acquire lock

  if (queue empty) {
    // begin atomic wait
    release lock
    sleep and wait
    // end atomic wait
    re-acquire lock
  }

  remove item from queue
  release lock
  return removed item
}
```

Multi-threaded queue

```
enqueue () {
  acquire lock

  find tail of queue
  add new element

  signal (lock, CV)

  release lock
}
```

2

```
dequeue () {
  acquire lock
  …
  return removed item
}
```

3

```
dequeue () {
  acquire lock

  if (queue empty) {
    // begin atomic wait
    release lock
    sleep and wait
    // end atomic wait
    re-acquire lock
  }

  remove item from queue
  release lock
  return removed item
}
```

1

4

Multi-threaded queue

```
enqueue () {
  acquire lock

  find tail of queue
  add new element

  signal (lock, CV)

  release lock
}
```

```
dequeue () {
  acquire lock

  if (queue empty) {
    // begin atomic wait
    release lock
    sleep and wait
    // end atomic wait
    re-acquire lock
  }

  remove item from queue
  release lock
  return removed item
}
```

How to solve?

Multi-threaded queue

The "condition" in condition variable

```
enqueue () {
  acquire lock

  find tail of queue
  add new element

  signal (lock, CV)

  release lock
}
```

```
dequeue () {
  acquire lock

  while (queue empty) {
    wait (lock, CV)
  }

  remove item from queue
  release lock
  return removed item
}
```

Solve with a while loop ("loop before you leap")
Remember this for assignments …

Mesa vs. Hoare monitors

- So far, we've described Mesa monitors
  - After waking up, no special priority
  - Threads have to recheck condition
- Hoare semantics are "simpler"
  - But complicate implementation

Hoare semantics

- Condition guaranteed true after wakeup
  - i.e. no need to loop
- Waiter acquires lock before any threads run
  - (since lock protects condition)
  - Including the signaler!
- Signaler must give up lock and signal atomically
- What would this mean for invariants?
  - All invariants must be established before signal
- **We will use Mesa semantics**

Tips for using monitors

1. **List the shared data needed for problem**
2. **Figure out the locks**
   - 1 lock per group of shared data
3. **Bracket code that uses shared data with lock/unlock**
4. **Think about before-after conditions**
   - 1 condition variable per type of condition
   - CV's lock should protect data used to evaluate condition
   - Call wait when you need to wait on a condition
   - Loop to re-check condition when wait returns
   - Call signal when condition changes
   - Ensure invariants are established when lock is not held (unlock, wait)

Monitor Benefits

- Benefit of sleeping thread vs. blocked thread?
  - Sleeping thread is not wasting CPU cycles
- A note about condition variables
  - They represent some condition in our code
  - But they are not boolean expressions themselves
  - When we call wait or signal, we specify mutex and condition variable

Producer-consumer problem

- Producer makes something consumer wants
  - Goal: avoid lock-step (direct hand-off)

Delivery person
(producer)

Soda drinker
(consumer)

Problem: everyone's time is wasted.

Producer-consumer problem

- Instead, use a fixed-size, shared buffer (sometimes called bounded buffer)
  - Producer puts in (must wait when full)
  - Consumer takes out (must wait when empty)
  - Must synchronize access to buffer
- Examples
  - Unix pipes: cat file | grep -v the | grep from | wc -l
  - Interaction between hardware devices/buffers

Use a soda machine as a buffer

Solving producer-consumer

1. **What are the variables/shared state?**
   - Soda machine buffer
   - Number of sodas in machine (≤ MaxSodas)
2. **Locks?**
   - 1 to protect all shared state (sodaLock)
3. **Mutual exclusion?**
   - Only one thread can manipulate machine at a time
4. **Ordering constraints?**
   - Consumer must wait if machine is empty (CV hasSoda)
   - Producer must wait if machine is full (CV hasRoom)

Producer-consumer code

```
consumer () {
  lock (sodaLock)

  while (numSodas == 0) {
    wait (sodaLock,hasSoda)
  }

  take soda out of machine

  signal (hasRoom)

  unlock (sodaLock)
}
```

```
producer () {
  lock (sodaLock)

  while(numSodas==MaxSodas){
    wait (sodaLock, hasRoom)
  }

  add soda to machine

  signal (hasSoda)

  unlock (sodaLock)
}
```

Variations: looping producer

- Producer
  - Infinite loop ok?
  - Why/why not?
- Release lock in wait call

```
producer () {
   lock (sodaLock)
   while (1) {
      while(numSodas==MaxSodas){
         wait (sodaLock, hasRoom)
      }

      add soda to machine

      signal (hasSoda)
   }
   unlock (sodaLock)
}
```

Variations: resting producer

- Producer
  - Sleep ok?
  - Why/why not?
- Shouldn't hold locks during a slow operation

```
producer () {
   lock (sodaLock)
   while (1) {
      sleep (1 hour)
      while(numSodas==MaxSodas){
         wait (sodaLock, hasRoom)
      }

      add soda to machine

      signal (hasSoda)
   }
   unlock (sodaLock)
}
```

Variations: one CV?

```
consumer () {
  lock (sodaLock)

  while (numSodas == 0) {
    wait (sodaLock,hasRorS)
  }

  take soda out of machine

  signal (hasRorS)

  unlock (sodaLock)
}
```

```
producer () {
  lock (sodaLock)

  while(numSodas==MaxSodas){
    wait (sodaLock,hasRorS)
  }

  add soda to machine

  signal (hasRorS)

  unlock (sodaLock)
}
```

Two producers, two consumers: who consumes a signal?

Variations: one CV?

```
consumer () {
  lock (sodaLock)

  while (numSodas == 0) {
    wait (sodaLock,hasRorS)
  }

  take soda out of machine

  signal (hasRorS)

  unlock (sodaLock)
}
```

```
producer () {
  lock (sodaLock)

  while(numSodas==MaxSodas){
    wait (sodaLock,hasRorS)
  }

  add soda to machine

  signal (hasRorS)

  unlock (sodaLock)
}
```

Is it possible to have a producer and consumer both waiting?

max=1, cA and cB wait, pC adds/signals, pD waits, cA wakes

Variations: one CV?

```
consumer () {
  lock (sodaLock)

  while (numSodas == 0) {
    wait (sodaLock,hasRorS)
  }

  take soda out of machine

  signal (hasRorS)

  unlock (sodaLock)
}
```

```
producer () {
  lock (sodaLock)

  while(numSodas==MaxSodas){
    wait (sodaLock,hasRorS)
  }

  add soda to machine

  signal (hasRorS)

  unlock (sodaLock)
}
```

Is it possible to have a producer and consumer both waiting?

max=1, cA and cB wait, pC adds/signals, pD waits, cA wakes

cA removes/signals, cB wakes…but numSodas==0

Variations: one CV?

```
consumer () {
  lock (sodaLock)

  while (numSodas == 0) {
    wait (sodaLock,hasRorS)
  }

  take soda out of machine

  signal (hasRorS)

  unlock (sodaLock)
}
```

```
producer () {
  lock (sodaLock)

  while(numSodas==MaxSodas){
    wait (sodaLock,hasRorS)
  }

  add soda to machine

  signal (hasRorS)

  unlock (sodaLock)
}
```

How can we make the one CV solution work?

Variations: one CV?

```
consumer () {
  lock (sodaLock)

  while (numSodas == 0) {
    wait (sodaLock,hasRorS)
  }

  take soda out of machine

  broadcast (hasRorS)

  unlock (sodaLock)
}
```

```
producer () {
  lock (sodaLock)

  while(numSodas==MaxSodas){
    wait (sodaLock,hasRorS)
  }

  add soda to machine

  broadcast (hasRorS)

  unlock (sodaLock)
}
```

Use broadcast instead of signal

Broadcast vs signal

- Can I always use broadcast instead of signal?
  - Yes, assuming threads recheck condition
- Why might I use signal instead?
  - Efficiency (spurious wakeups)
  - May wakeup threads for no good reason

Java Monitors

```
class BoundedBuffer {
   final Lock lock = new ReentrantLock();
   final Condition notFull  =
    lock.newCondition();
   final Condition notEmpty =
    lock.newCondition();

   final Object[] items = new Object[100];
   int putptr, takeptr, count;

   public void put(Object x)
          throws InterruptedException {
     lock.lock();
     try {
       while (count == items.length)
         notFull.await();
       items[putptr] = x;
       if (++putptr == items.length)
         putptr = 0;
       count++;
       notEmpty.signal();
     } finally {
       lock.unlock();
     }
   }
```

```
public Object take()
       throws InterruptedException {
     lock.lock();
     try {
       while (count == 0)
         notEmpty.await();
       Object x = items[takeptr];
       if (++takeptr == items.length)
         takeptr = 0;
       count--;
       notFull.signal();
       return x;
     } finally {
       lock.unlock();
     }
   }
 }
```

C/C++ Monitors

```
#include <pthread.h>

pthread_mutex_t mutex =
          PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t notFull =
          PTHREAD_COND_INITIALIZER;
pthread_cond_t notEmpty =
          PTHREAD_COND_INITIALIZER;

int items[100];
int putptr, takeptr, count = 0;

void put(int x) {
  pthread_mutex_lock( &mutex );
  while( count == 100 )
     pthread_cond_wait( &notFull, &mutex );
  items[putptr] = x;
  if (++putptr == 100) putptr = 0;
  count++;
  pthread_cond_signal( &notEmpty );
  pthread_mutex_unlock( &mutex );
}
```

```
int take() {
   pthread_mutex_lock( &mutex );
   while (count == 0)
     pthread_cond_wait( &notEmpty, &mutex );
   int x = items[takeptr];
   if (++takeptr == 100) takeptr = 0;
   count--;
   pthread_cond_signal( &notFull );
   pthread_mutex_unlock( &mutex );
   return x;
}

int main(int argc, char* argv[]) {
   return 0;
}
```

g++ buffer.cc -pthread