# CMSC 330: Organization of Programming Languages

Threads, con't.

## What's Wrong with the Following?

```
static int count = 0;
static int x = 0;
```

```
Thread 1
while (x != 0);
x = 1;
count++;
x = 0;
```

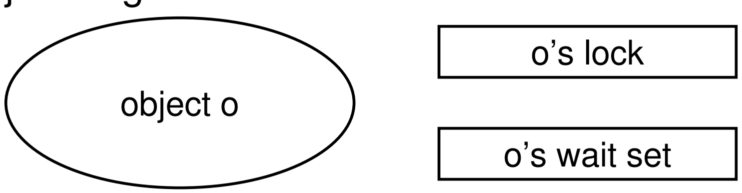
```
Thread 2
while (x != 0);
x = 1;
count++;
x = 0;
```

- Threads may be interrupted after the while but before the assignment x = 1
  - Both may think they "hold" the lock!
- This is *busy waiting*
  - Consumes lots of processor cycles



## Wait and NotifyAll (Java 1.4)

- Recall that in Java 1.4, use synchronize on object to get associated lock




- Objects also have an associated wait set



## Wait and NotifyAll (cont'd)

- o.wait()
  - Must hold lock associated with o
  - Release that lock
    - And no other locks
  - Adds this thread to wait set for lock
  - Blocks the thread
- o.notifyAll()
  - Must hold lock associated with o
  - Resumes all threads on lock's wait set
  - Those threads must reacquire lock before continuing
    - (This is part of the function; you don't need to do it explicitly)

# ReentrantLock Class (Java 1.5)

```
class ReentrantLock implements Lock { ... }
```

- Reentrant lock
  - Can be reacquired by same thread by invoking lock() up to 2147483648 times
  - To release lock, must invoke unlock() the same number of times lock() was invoked
- Reentrancy is useful because each method can acquire/release locks as necessary
  - No need to worry about whether callers already have locks
  - Discourages complicated coding practices to determine whether lock has already been acquired



# Reentrant Lock Example

```
static int count = 0;
static Lock l =
    new ReentrantLock();

void inc() {
  l.lock();
  count++;
  l.unlock();
}
```

```
int returnAndIncr() {
  int temp;

  l.lock();
  temp = count;
  inc();
  l.unlock();
  return temp;
}
```



# Deadlock

- *Deadlock* occurs when no thread can run because all threads are waiting for a lock
  - No thread running, so no thread can ever release a lock to enable another thread to run

```
Lock l = new ReentrantLock();
Lock m = new ReentrantLock();
```

| <u>Thread 1</u> | <u>Thread 2</u> |
|---|---|
| `l.lock();` | `m.lock();` |
| `m.lock();` | `l.lock();` |
| `...` | `...` |
| `m.unlock();` | `l.unlock();` |
| `l.unlock();` | `m.unlock();` |



# Deadlock (cont'd)

- Some schedules work fine
  - Thread 1 runs to completion, then thread 2
- But what if...
  - Thread 1 acquires lock l
  - The scheduler switches to thread 2
  - Thread 2 acquires lock m
- Deadlock!
  - Thread 1 is trying to acquire m
  - Thread 2 is trying to acquire l
  - And neither can, because the other thread has it

# Another Case of Deadlock

```
static Lock l = new ReentrantLock();

void f () throws Exception {
  l.lock();
  FileInputStream f =
     new FileInputStream("file.txt");
  // Do something with f
  f.close();
  l.unlock();
}
```

- l not released if exception thrown
  - Likely to cause deadlock some time later



# Solution: Use Finally

```
static Lock l = new ReentrantLock();

void f () throws Exception {
  l.lock();
  try {
    FileInputStream f =
       new FileInputStream("file.txt");
    // Do something with f
    f.close();
  }
  finally {
    // This code executed no matter how we
    // exit the try block
    l.unlock();
  }
}
```



# Producer/Consumer Design Pattern

- Suppose we are communicating with a shared variable
  - E.g., some kind of a buffer holding messages

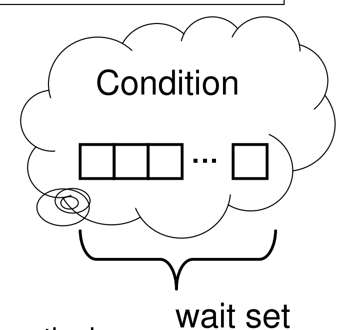
- One thread *produces* input to the buffer
- One thread *consumes* data from the buffer
- How do we implement this?
  - Use *condition variables*



# Conditions (Java 1.5)

```
interface Lock { Condition newCondition(); ... }
interface Condition {
  void await();
  void signalAll();
  ...
}
```

- Condition created from a Lock
- await called with lock held
  - Releases the lock
    - But not any other locks held by this thread
  - Adds this thread to wait set for lock
  - Blocks the thread
- signallAll called with lock held
  - Resumes all threads on lock's wait set
  - Those threads must reacquire lock before continuing
    - (This is part of the function; you don't need to do it explicitly)



## Producer/Consumer Example

```
Lock lock = new ReentrantLock();
Condition ready = lock.newCondition();
boolean valueReady = false;
Object value;
```

```
void produce(Object o) {
  lock.lock();
  while (valueReady)
    ready.await();
  value = o;
  valueReady = true;
  ready.signalAll();
  lock.unlock();
}
```

```
Object consume() {
  lock.lock();
  while (!valueReady)
    ready.await();
  Object o = value;
  valueReady = false;
  ready.signalAll();
  lock.unlock();
  return o;
}
```



## Use This Design

- This is the right solution to the problem
  - It's tempting to try to just use locks directly, but that's very hard to get right
  - Problems with other approaches are often very subtle
    - E.g., double-checked locking is broken



## Broken Producer/Consumer Example

```
Lock lock = new ReentrantLock();
boolean valueReady = false;
Object value;
```

```
void produce(object o) {
  lock.lock();
  while (valueReady);
  value = o;
  valueReady = true;
  lock.unlock();
}
```

```
Object consume() {
  lock.lock();
  while (!valueReady);
  Object o = value;
  valueReady = false;
  lock.unlock();
  return o;
}
```

A thread can wait with lock held – no way to make progress



## Broken Producer/Consumer Example

```
Lock lock = new ReentrantLock();
boolean valueReady = false;
Object value;
```

```
void produce(object o) {
  while (valueReady);
  lock.lock();
  value = o;
  valueReady = true;
  lock.unlock();
}
```

```
Object consume() {
  while (!valueReady);
  lock.lock();
  Object o = value;
  valueReady = false;
  lock.unlock();
  return o;
}
```

valueReady accessed without a lock held – race condition

## Broken Producer/Consumer Example

```
Lock lock = new ReentrantLock();
Condition ready = lock.newCondition();
boolean valueReady = false;
Object value;
```

```
void produce(object o) {
  lock.lock();
  if (valueReady)
    ready.await();
  value = o;
  valueReady = true;
  ready.signalAll();
  lock.unlock();
}
```

```
Object consume() {
  lock.lock();
  if (!valueReady)
    ready.await();
  Object o = value;
  valueReady = false;
  ready.signalAll();
  lock.unlock();
  return o;
}
```

What if there are multiple producers or consumers?



## Await and SignalAll Gotcha's

- await *must* be in a loop
  - Don't assume that when await returns conditions are met
- Avoid holding other locks when waiting
  - await only gives up locks on the object you wait on



## Producer/Consumer in Java 1.4

```
public class ProducerConsumer {

  private boolean valueReady = false;
  private Object value;

  synchronized void produce(Object o) {
    while (valueReady)
      wait();
    value = o;
    valueReady = true;
    notifyAll();
  }
  synchronized Object consume() {
    while (!valueReady)
      wait();
    valueReady = false;
    Object o = value;
    notifyAll();
    return o;
  }
}
```



## Key Ideas

- Multiple threads can run simultaneously
  - Either truly in parallel on a multiprocessor
  - Or can be scheduled on a single processor
    - A running thread can be pre-empted at any time
- Threads can share data
  - In Java, only fields can be shared
  - Need to prevent interference
    - Rule of thumb 1: You must hold a lock when accessing shared data
    - Rule of thumb 2: You must not release a lock until shared data is in a valid state
  - Overuse of synchronization can create deadlock
    - Rule of thumb: No deadlock if only one lock

# Guidelines for Programming w/Threads

- Synchronize access to shared data
- Don't hold multiple locks at a time
  - Could cause deadlock
- Hold a lock for as little time as possible
  - Reduces blocking waiting for locks
- While holding a lock, don't call a method you don't understand
  - E.g., a method provided by someone else, especially if you can't be sure what it locks
  - Corollary: document which locks a method acquires



# Ruby Threads – Thread Creation

- Create thread using Thread.new
  - New method takes code block argument

```
t = Thread.new { …body of thread… }
t = Thread.new (arg) { | arg | …body of thread… }
```

  - Join method waits for thread to complete

```
t.join()
```

- Example

```
myThread = Thread.new {
   sleep(1)              # sleep for 1 second
   puts( "New thread awake!")
   $stdout.flush         # flush makes sure output is seen
}
```



# Ruby Threads – Locks

- Monitor, Mutex
  - Object intended to be used by multiple threads
  - Methods are executed with mutual exclusion
    - As if all methods are synchronized
  - Monitor is reentrant, Mutex is not
- Create lock using Monitor.new
  - Synchronize method takes code block argument

```
require 'monitor.rb'
myLock = Monitor.new
myLock.synchronize {
        # myLock held during this code block
}
```



# Ruby Threads – Condition

- Condition derived from Monitor
  - Create condition from lock using new_cond
  - Sleep while waiting using wait_while, wait_until
  - Wake up waiting threads using broadcast
- Example

```
myLock = Monitor.new                    # new lock
myCondition = myLock.new_cond           # new condition
myLock.synchronize {
  myCondition.wait_while { y > 0 }      # wait as long as y > 0
  myCondition.wait_until { x != 0 }     # wait as long as x == 0
}
myLock.synchronize {
  myCondition.broadcast              # wake up all waiting threads
}
```

## Parking Lot Example

```
require "monitor.rb"
class ParkingLot
    def initialize     # initialize synchronization
        @numCars = 0
        @myLock = Monitor.new
        @myCondition = @myLock.new_cond
    end
    def addCar
      …
    end
    def removeCar
      …
    end
end
```



## Parking Lot Example

```
def addCar # do work not requiring synchronization
    @myLock.synchronize {
        @myCondition.wait_until { @numCars < MaxCars }
        @numCars = @numCars + 1
        @myCondition.broadcast
    }
end
def removeCar # do work not requiring synchronization
    @myLock.synchronize {
        @myCondition.wait_until { @numCars > 0 }
        @numCars = @numCars - 1
        @myCondition.broadcast
    }
end
```



## Parking Lot Example

```
garage = ParkingLot.new ()
valet1 = Thread.new {        # valet 1 drives cars into parking lot
   while …
     # do work not requiring synchronization
     garage.addCar()
   end
}
valet2 = Thread.new {        # valet 2 drives car out of parking lot
   while …
     # do work not requiring synchronization
     garage.removeCar()
   end
}
valet1.join()          # returns when valet1 exits
valet2.join()          # returns when valet2 exits
```



## Ruby Threads – Difference from Java

- Ruby thread can access all variables in scope when thread is created, including local variables
  - Java threads can only access object fields
- Exiting
  - All threads exit when main Ruby thread exits
  - Java continues until all non-daemon threads exit
- When thread throws exception
  - Ruby only aborts current thread (by default)
  - Ruby can also abort all threads (better for debugging)
    - Set Thread.abort_on_exception = true