Today

- Classic Synchronization Problem: Producer-Consumer
  - With Condition Variables
  - With Semaphores
  - In Java
- Classic Synchronization Problem: Dining Philosophers

Review

- What 3 synchronization mechanisms have we talked about?
  - How do they work?
- What problem did we solve?
- What does Java provide so that we can synchronize code?
- What problem were we in the process of solving?
  - What did we figure out so far?

Review: Synchronization Mechanisms

- Lock
- Condition Variables
- Semaphores

Review: Java Synchronization

- Monitors built in to every object, through inheritance from `Object` class
  - Mutual exclusion (locks)
  - Cooperation (condition variable)
  - Lock/critical sections with `synchronized` keyword
- `java.util.concurrent` classes
  - Lock
  - Condition
  - Semaphore

Review: Java Uses Mutexes and CVs

Every Java object has a mutex ("monitor") and condition variable ("CV") built in. You don't have to use it, but it's there.

```
public class Object {
    void notify(); /* signal */
    void notifyAll(); /* broadcast */
    void wait();
    void wait(long timeout);
}
```

`wait(timeout)` waits until the timeout elapses or another thread notifies.

A thread must own an object's monitor (**`synchronized`**) to call `wait`/`notify`, else the method raises an *`IllegalMonitorStateException`*.

```
public class PingPong {
   public synchronized void
     pingPong() {
      while(true) {
          notify();
          wait();
      }
   }
}
```


Review: Solving Producer-Consumer Problems

- What variables/shared state do we need?
  - Soda machine buffer
  - Number of sodas in machine (≤ maxSodas)
- Where do we need mutual exclusion?
  - Only one thread can manipulate machine at a time
  - 1 lock to protect all shared state (sodaLock)
- What are our ordering constraints?
  - Consumer must wait if machine is empty (CV hasSoda)
  - Producer must wait if machine is full (CV hasRoom)

Producer-Consumer Psuedocode

```
consumer () {

  take a soda from machine

}
```

```
producer () {

  add one soda to machine

}
```


Producer-Consumer Psuedocode: First Step

Wrap both of these calls in while loops

```
consumer () {

  lock.acquire();

  if( numSodas > 0 )
     numSodas--;

  lock.release();

}
```

```
producer () {

  lock.acquire();

  if( numSodas < maxSodas )
     numSodas++;

  lock.release();

}
```

Pretty good… But we're doing a bunch of busy-waiting

Solution: Condition Variables!

Producer-Consumer Psuedocode

```
consumer () {
  lock
  wait if empty

  take a soda from machine

  notify (not full)
  unlock
}
```

```
producer () {
  lock
  wait if full

  add one soda to machine

  notify (not empty)
  unlock
}
```


Producer-Consumer Code

```
consumer () {
  sodaLock.acquire()

  while (numSodas == 0) {
    hasSoda.wait(sodaLock)
  }    CV1              Mx

  numSodas--;

  hasRoom.signal()
    CV2
  sodaLock.release()
}
```

```
producer () {
  sodaLock.acquire()

  while(numSodas==MaxSodas){
    hasRoom.wait(sodaLock)
  }    CV2              Mx

  numSodas++;

  hasSoda.signal()
    CV1
  sodaLock.release()
}
```

SodaMachineWithConditionVariable.java

As Java Code

Shared State

```
public static final Lock sodaLock = new ReentrantLock();
public static final Condition hasRoom =
                                    sodaLock.newCondition();
public static final Condition hasSoda =
                                    sodaLock.newCondition();
```

ProducerThread

```
while( true ) {
    SodaMachine.sodaLock.lock();

    while(SodaMachine.numSodas==SodaMachine.maxSodas){
        try {
            SodaMachine.hasRoom.await();
        } catch( Exception e ) {
            e.printStackTrace();
        }
    }

    SodaMachine.numSodas++;
    SodaMachine.hasSoda.signal();
    SodaMachine.sodaLock.unlock();
}
```

>1 Resource, >1 Consumers

The signal should be a *broadcast*
if the producer can produce more than one resource,
and there are multiple consumers.

```
consumer () {
  sodaLock.acquire()

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

  take a soda from machine

  hasRoom.signal()

  sodaLock.release()
}
```

```
producer () {
  sodaLock.acquire()

  while(numSodas==maxSodas){
    hasRoom.wait(sodaLock)
  }

  fill machine with soda

  hasSoda.broadcast()

  sodaLock.release()
}
```

Broadcast vs signal

- Can I always use broadcast instead of signal?
  - Yes, assuming threads recheck condition
  - And they should: "loop before you leap"!
  - Another thread could get to the lock before wait returns
- Why might I use signal instead?
  - Efficiency -- May wakeup threads for no good reason
    - Those threads will then be put back to sleep

Condition Variable Design Pattern

```
methodThatWaits() {
   lock.acquire();

   // Read/write shared state

   while (
     !testSharedState() ) {
      cv.wait(lock);
   }

   // Read/write shared state

   lock.release();
}
```

```
methodThatSignals() {
   lock.acquire();

   // Read/write shared state

   // If testSharedState is
   // now true
   cv.signal(lock);

   // Read/write shared state

   lock.release();
}
```

Summary: Condition Variables

- Condition variable is *memoryless*
  - If signal when no one is waiting, no op
- wait ***atomically*** releases lock
  - What if not atomic: wait queue, then release?
  - What if not atomic: release, then wait queue?

```
wait (lock){
  release lock                Atomic
  put thread on wait queue
  go to sleep
  // after wake up
  acquire lock
}
```


Summary: Condition Variables

- When a thread is woken up from wait, it may not run immediately
  - Signal/broadcast puts thread on *ready* (not running) list
  - When lock is released, anyone might acquire it
- Benefit: simplifies implementation
  - Of condition variables and locks
  - Of code that uses condition variables and locks

Using Condition Variables

- Document the condition(s) associated with each CV.
  - What are the waiters waiting for?
  - When can a waiter expect a signal?

- ALWAYS hold lock when calling wait, signal, broadcast
  - Condition variable is sync for shared state
  - ALWAYS hold lock when accessing shared state

Using Condition Variables

- Wait MUST be in a loop if you're waiting for shared state to be in a certain state
  - "Loop before you leap!"

```
while( needToWait() ) {
    condition.wait(lock);
}
```

  - Another thread may beat you to the mutex.
  - The signaler may be careless.
    - Some thread packages have "spurious wakeups": 2 threads woken up, though a single signal has taken place
  - A single CV may have multiple conditions
  - Signals on CVs do not stack!
    - A signal will be lost if nobody is waiting: always check the wait condition before calling wait.

PRODUCER CONSUMER W/ SEMAPHORES

Producer/Consumer with Semaphores?

- To start:
  - Just one resource produced/consumed

Basic Producer/Consumer

```
empty = Semaphore(1);
full = Semaphore(0);
int buf;
```

```
void Produce(int m) {
    empty.P();
    buf = m;
    full.V();
}
```

```
int Consume() {
    int m;
    full.P();
    m = buf;
    empty.V();
    return m;
}
```

- This use of a semaphore pair is called a **split binary semaphore**
  - sum of the values is always 1
- **It is the same as ping-pong**: producer and consumer access the buffer in strict alternation

What non-semaphore statement happens first?

P-C Analysis: Multiple Resources

- This time: more than one resource can be stored
- Same before-after constraints
  - If buffer empty, consumer waits for producer
  - If buffer full, producer waits for consumer
- What data do we need?
- How can we use semaphores to synchronize?

P-C Analysis: Multiple Resources

- This time: more than one resource can be stored
- Same before-after constraints
  - If buffer empty, consumer waits for producer
  - If buffer full, producer waits for consumer
- Data
  - maxSodas, numSodas, sodaBuffer
- Semaphores
  - mutex (binary semaphore)
  - fullSlots (counts number of full slots)
  - emptySlots (counts number of empty slots)

P-C Analysis: Multiple Resources

- What should the semaphores' initial values be?
  - Mutual exclusion
    - Semaphore mutex (?)
  - Machine is initially empty
    - Semaphore fullSlots (?)
    - Semaphore emptySlots (?)

P-C Analysis: Multiple Resources

- Initial semaphore values
  - Mutual exclusion
    - Semaphore mutex (1)
  - Machine is initially empty
    - Semaphore fullSlots (0)
    - Semaphore emptySlots (MaxSodas)

Called *counting* semaphores

Producer-Consumer with Semaphores

```
Semaphore fullSlots(0), emptySlots(MaxSodas)
```

```
producer () {
   // wait for empty slot
   emptySlots.P()

   put one soda in

   // signal item arrival
   fullSlots.V()
}
```

```
consumer () {
   // wait for item arrival
   fullSlots.P()

   take one soda out

   //signal empty slot
   emptySlots.V()
}
```

Semaphores give us elegant full/empty synchronization.

Is that enough?

No – still need to synch access to shared state

Producer-Consumer with Semaphores and Mutex

```
Semaphore mutex(1), fullSlots(0), emptySlots(MaxSodas)
```

```
producer () {
  // wait for empty slot
  emptySlots.P()
  // lock shared state
  mutex.P()
  put one soda in
  mutex.V()

  // signal item arrival
  fullSlots.V()
}
```

```
consumer () {
  // wait for item arrival
  fullSlots.P()
  // lock shared state
  mutex.P()
  take one soda out
  mutex.V()

  //signal empty slot
  emptySlots.V()
}
```

Consider this solution… Does this work?

Yes! What if we switched the mutex calls to begin and end the method?

Producer-Consumer with Semaphores and Mutex: Swapped Calls

```
Semaphore mutex(1), fullSlots(0), emptySlots(MaxSodas)
```

```
producer () {
   // lock shared state
   mutex.P()

   // wait for empty slot
   emptySlots.P()
   put one soda in
   // signal item arrival
   fullSlots.V()

   mutex.V()
}
```

```
consumer () {
  //lock shared state
  mutex.P()

  // wait for item arrival
  fullSlots.P()
  take one soda out
  //signal empty slot
  emptySlots.V()

  mutex.V()
}
```

Consider this solution… Does this work?

Analysis: Producer-Consumer with Semaphores and Mutex: Swapped Calls

```
Semaphore mutex(1), fullSlots(0), emptySlots(MaxSodas)
```

```
producer () {
   // lock shared state
   mutex.P()

   // wait for em
   emptySlots.P()
   put one soda i
   // signal item arrival
   fullSlots.V()

   mutex.V()
}
```

```
consumer () {
  //lock shared state
  mutex.P()

  // wait for item arrival
  fullSlots.P()
  take one soda out
  //signal empty slot
  emptySlots.V()

  mutex.V()
}
```

Waits but still holds the lock!

Does not work.
Can cause ***deadlock.*** Order of the down calls matters.

Analysis: Producer-Consumer with Semaphores and Mutex

```
Semaphore mutex(1), fullSlots(0), emptySlots(MaxSodas)
```

```
producer () {
  // wait for empty slot
  emptySlots.P()
  // lock shared state
  mutex.P()
  put one soda in
  mutex.V()

  // signal item arrival
  fullSlots.V()
}
```

```
consumer () {
  // wait for item arrival
  fullSlots.P()
  // lock shared state
  mutex.P()
  take one soda out
  mutex.V()

  //signal empty slot
  emptySlots.V()
}
```

Does the order of the up calls matter?

Not for correctness. (possible efficiency issues)

Analysis: Producer-Consumer with Semaphores and Mutex

```
Semaphore mutex(1), fullSlots(0), emptySlots(MaxSodas)
```

```
producer () {
  // wait for empty slot
  emptySlots.P()
  // lock shared state
  mutex.P()
  put one soda in
  mutex.V()

  // signal item arrival
  fullSlots.V()
}
```

```
consumer () {
  // wait for item arrival
  fullSlots.P()
  // lock shared state
  mutex.P()
  take one soda out
  mutex.V()

  //signal empty slot
  emptySlots.V()
}
```

Does this work with multiple consumers and/or producers?

Yes!...

Analysis: Producer-Consumer with Semaphores and Mutex

```
Semaphore mutex(1), fullSlots(1), emptySlots(MaxSodas-1)
```

```
producer () {
  // wait for empty slot
  emptySlots.P()
  // lock shared state
  mutex.P()
  put one soda in
  mutex.V()

  // signal item arrival
  fullSlots.V()
}
```

```
consumer () {
  // wait for item arrival
  fullSlots.P()
  // lock shared state
  mutex.P()
  take one soda out
  mutex.V()

  //signal empty slot
  emptySlots.V()
}
```

What if 1 full slot and multiple consumers call down?
Only one will see semaphore at 1, rest see at 0.

SodaMachineWithSemaphores.java

In Java Code

Shared State

```
public static final Semaphore sodaLock = new Semaphore(1);
public static final Semaphore emptySlots =
                                    new Semaphore(maxSodas);
public static final Semaphore fullSlots = new Semaphore(0);
```

Producer Thread

```
while( true ) {
    try {
        SodaMachineWithSemaphore.emptySlots.acquire();
        SodaMachineWithSemaphore.sodaLock.acquire();
    } catch( InterruptedException e ) {
        e.printStackTrace();
    }

    SodaMachineWithSemaphore.numSodas++;

    SodaMachineWithSemaphore.sodaLock.release();

    SodaMachineWithSemaphore.fullSlots.release();
}
```


Looking Ahead

- Synchronization Assignment – Due next Monday
  - Part 1: Discussion/pseudocode
  - Part 2: implementation in Java