Today

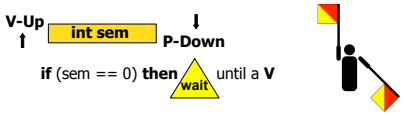
- Synchronization Mechanisms
  - Semaphores
- Implementation in Java

Review

- What is a semaphore?

Semaphore

- A ***semaphore*** is a hidden atomic integer counter with only increment/up (V) and decrement/down (P) operations.
  - Book calls V `signal` and P `wait`
- Decrement blocks iff the count is zero.
- Semaphores handle all of your synchronization needs with one elegant but confusing abstraction.



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**: the sum of the values is always 1

Basic producer/consumer is called **rendezvous**:
one producer, one consumer, and one item at a time.
It is the same as ping-pong:
producer and consumer access the buffer in strict alternation.

Prod.-cons. with semaphores

- 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 structures/synchronization mechanisms should we use?

Prod.-cons. with semaphores

```
Semaphore fullBuffers(0), emptyBuffers(MaxSodas)

consumer () {
  // wait for item arrival
  fullBuffers.P()

  take one soda out

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

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

  put one soda in

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

Semaphores give us elegant full/empty synchronization.
Is that enough?

Prod.-cons. with semaphores

```
Semaphore fullBuffers(0), emptyBuffers(MaxSodas)

consumer () {
  // wait for item arrival
  fullBuffers.P()

  take one soda out

  //signal empty slot
  emptyBuffers.V()
}

producer () {
  // wait for empty slot
  emptyBuffers.P()

  put one soda in

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

We could have two threads accessing the "soda buffer".

Why wasn't this an issue in the one-resource version of the problem?

Prod.-cons. with semaphores

```
Semaphore mutex(1),fullBuffers(0),emptyBuffers(MaxSodas)

consumer () {
  fullBuffers.P()

  mutex.P()

  take soda out

  mutex.V()

  emptyBuffers.V()
}

producer () {
  emptyBuffers.P()

  mutex.P()

  put soda in

  mutex.V()

  fullBuffers.V()
}
```


Prod.-cons. with semaphores

```
Semaphore mutex(1),fullBuffers(0),emptyBuffers(MaxSodas)

consumer () {
  mutex.P()          (1)

  fullBuffers.P()

  take soda out

  emptyBuffers,V()

  mutex.V()
}

producer () {
  mutex.P()          (2)

  emptyBuffers.P()

  put soda in

  fullBuffers.V()

  mutex.V()
}
```

Does the order of the down calls matter?
Yes. Can cause "deadlock."

Prod.-cons. with semaphores

```
Semaphore mutex(1),fullBuffers(0),emptyBuffers(MaxSodas)

consumer () {
  fullBuffers.P()

  mutex.P()

  take soda out

  emptyBuffers.V()

  mutex.V()
}

producer () {
  emptyBuffers.P()

  mutex.P()

  put soda in

  fullBuffers.V()

  mutex.V()
}
```

Does the order of the up calls matter?
Not for correctness, possible efficiency issues.

Prod.-cons. with semaphores

```
Semaphore mutex(1),fullBuffers(0),emptyBuffers(MaxSodas)

consumer () {
  fullBuffers.P()

  mutex.P()

  take soda out

  mutex.V()

  emptyBuffers.V()
}

producer () {
  emptyBuffers.P()

  mutex.P()

  put soda in

  mutex.V()

  fullBuffers.V()
}
```

What about multiple consumers and/or producers?
Doesn't matter; solution stands.

Prod.-cons. with semaphores

```
Semaphore mtx(1),fullBuffers(1),emptyBuffers(MaxSodas-1)

consumer () {
  down (fullBuffers)

  down (mutex)

  take soda out

  up (mutex)

  up (emptyBuffers)
}

producer () {
  down (emptyBuffers)

  down (mutex)

  put soda in

  up (mutex)

  up (fullBuffers)
}
```

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

SYNCHRONIZING JAVA CODE

Java Synchronization

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

Lock

Returns	Method	Description
void	lock()	Acquires the lock.
Condition	newCondition()	Returns a new Condition instance that is bound to this Lock instance.
void	unlock()	Releases the lock.

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/Lock.html

Condition API

Returns	Method	Description
void	await()	Causes the current thread to wait until it is signalled or interrupted.
void	signal()	Wakes up one waiting thread.
void	signalAll()	Wakes up all waiting threads.

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/Condition.html

Example Use

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

BoundedBuffer.java

Semaphore API

`Semaphore(int permits)` –
Creates a Semaphore with the given number of permits and nonfair fairness setting.

Returns	Method	Description
void	acquire()	Acquires a permit from this semaphore, blocking until one is available, or the thread is interrupted.
void	release()	Releases a permit, returning it to the semaphore.

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Semaphore.html

Synchronization Problem

- Consider two threads
  - threadA generates a value of X
  - threadB uses the value of X to calculate the value of Y
- Assume: X=1, Y=0 are stored in the address space shared by the threads.
- A serialization constraint is necessary in order to ensure proper execution

Looking Ahead

- Synchronization assignment
  - Due Wednesday
- Project 4 – out soon!