# Today

- Synchronization in Java
- Classic Synchronization Problems
  - Producer-Consumer

# Review

- What two synchronization mechanisms did we discuss?
  - What are their APIs?
- What problem did we solve with these mechanisms that we could not solve with locks?

# Review: Condition Variables

- ***Condition variable (CV):*** Data structure that allows thread to check if some condition is true before continuing execution
  - Allows waiting ***inside*** a critical section
- Condition Variable API
  - **`wait`**: block until condition becomes true
  - **`signal`**: signal that the condition is true
    - also called **`notify`**
    - Wake up one waiting thread
  - May also define a **`broadcast`** (**`notifyAll`**)
    - Signal *all* waiting threads

# Condition Variable Operations

Lock always held

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

Atomic

Lock always held

Lock usually held

```
signal () {
  wakeup one waiter (if any)
}
```

Atomic

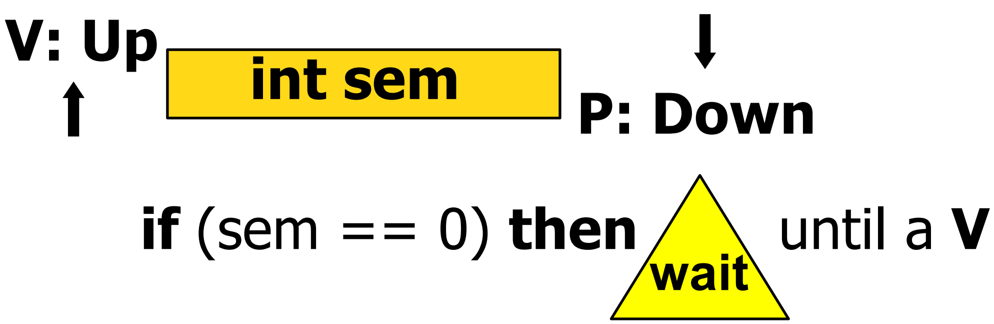
Lock usually held

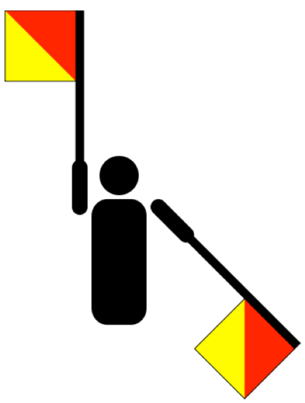
```
broadcast () {
  wakeup all waiters (if any)
}
```

Atomic

# Review: Semaphore

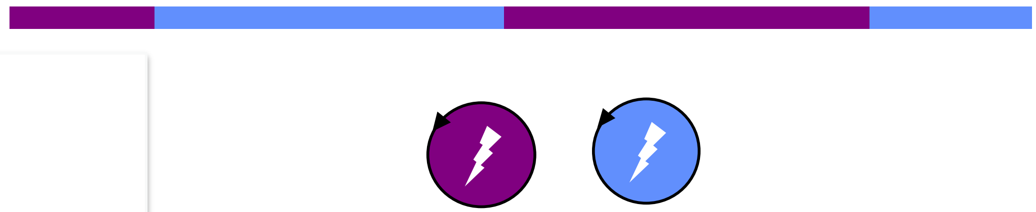
- A ***semaphore*** is a hidden atomic integer counter with only increment/up (V) and decrement/down (P) operations.
  - Book calls V `signal` and P `wait`
- Decrement blocks *iff* the count is zero.
- Semaphores handle all of your synchronization needs with one *elegant* but *confusing* abstraction.



# Review: Ping Pong using a Condition Variable

```
turn = purple;
```



```
void
PingPong() {
   mx.acquire();
   while(not done) {
      while(!myTurn)
          cv.wait(mx);
      do stuff;
      turn = blue;
      cv.signal();
    }
    mx.release();
}
```

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

```
signal (){
   wakeup one waiter (if any)
}
```

Blue's code is similar, with change to turn.

# Review: Ping Pong with Semaphores

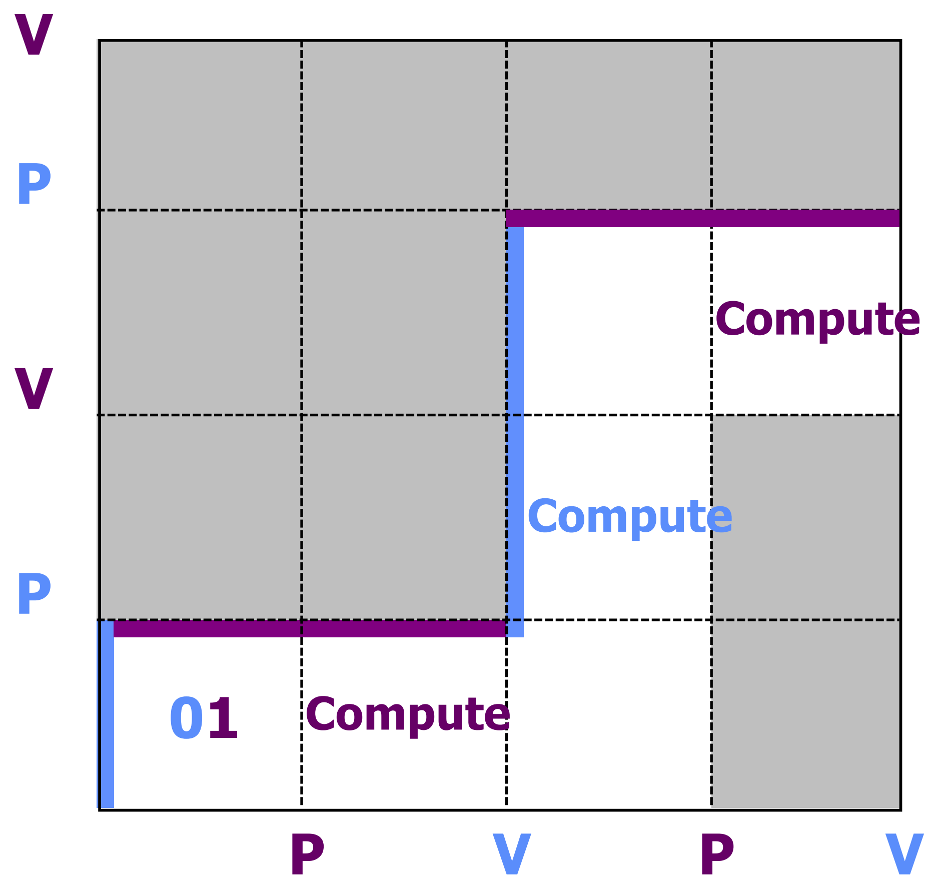
```
blue = Sempahore(0);
purple = Semaphore(1);
```

```
void
PingPong() {
    while(not done) {
        blue.P();
        Compute();
        purple.V();
    }
}
```

```
void
PingPong() {
    while(not done) {
        purple.P();
        Compute();
        blue.V();
    }
}
```

# Review: Ping Pong with Semaphores

The threads compute in strict alternation.



# SYNCHRONIZING JAVA CODE

# 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

# 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();
       }
    }
}
```

# Ping Pong Using a Condition Variable in Java

```
public synchronized void pingPong() {
    while(true) {
        // do something
        notify();
        wait();
    }
}
```

Implicit acquire() of this object's lock to start method

Implicit release() of this object's lock to end method

**Interchangeable lingo:**

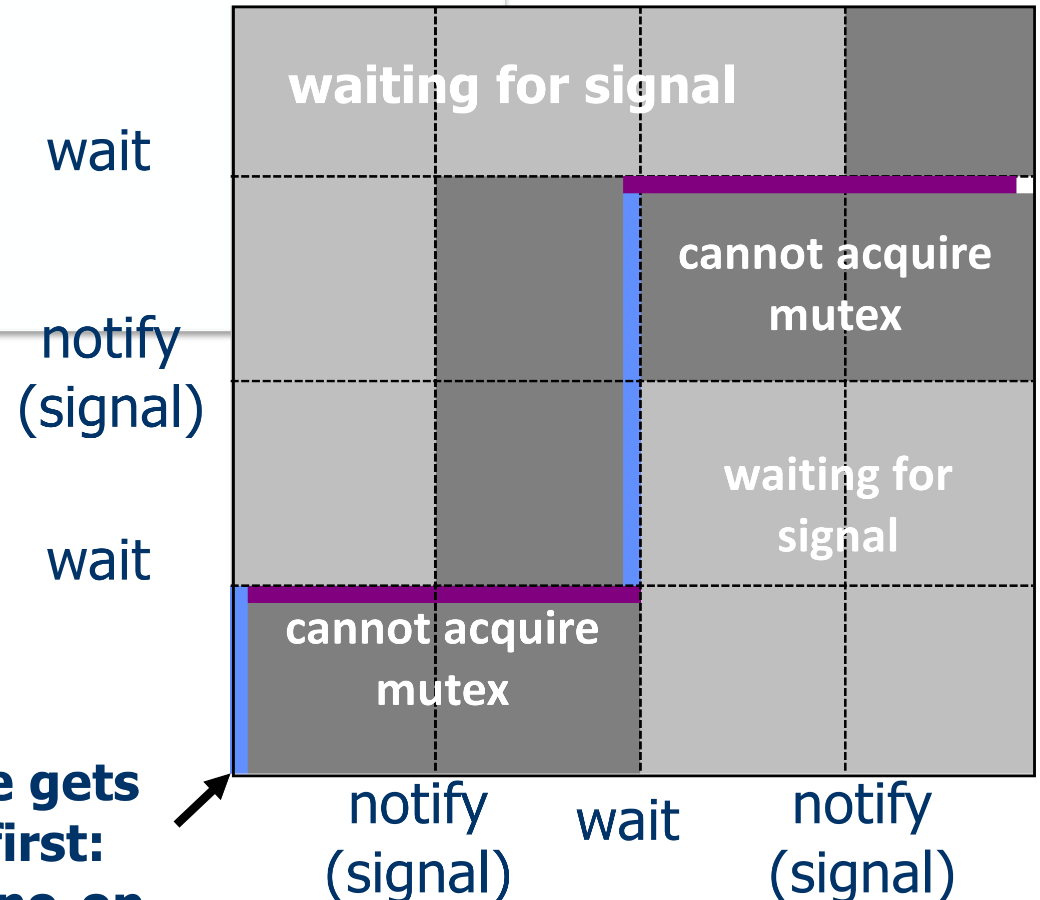
synchronized == mutex == lock

monitor == mutex+CV

notify == signal

# Ping Pong Using a Condition Variable in Java

```
public synchronized void pingPong() {
   while(true) {
      // do something
      notify();
      wait();
   }
}
```




**Suppose blue gets the mutex first: its notify is a no-op.**

# Ping Pong Using a Condition Variable in Java

```
public synchronized void pingPong() {
   while(true) {
      // do something
      notify();
      wait();
   }
}
```

Requires that two threads can execute this method on the **same** object

PingPongLock.java

# Java Synchronization

```
public void pingPong() {
   synchronized( someObject ) {
       while(true) {
          // do something
          notify();
          wait();
       }
   }
}
```

Implicit acquire() of someObject's lock to start block of code

Implicit release() of this someObject's lock to end block

someObject must be a shared variable

PingPong.java

# Monitors and mutexes are “equivalent”

- Entry to a monitor (e.g., a Java synchronized block) is equivalent to Acquire of an associated mutex.
  - Lock on entry
- Exit of a monitor is equivalent to Release.
  - Unlock on exit (or at least “return the key”…)
- Note: exit/release is implicit and automatic if the thread exits synchronized code by a Java exception.
  - Much less error-prone then explicit release
  - Can’t “forget” to unlock / “return the key”.
  - Language-integrated support is a plus for Java.

# Monitors and mutexes are "equivalent"

- Mutexes are more flexible because we can choose which mutex controls a given piece of state.
  - E.g., in Java we can use one object's monitor to control access to state in some other object.
  - Perfectly legal! So "monitors" in Java are more properly thought of as mutexes.
- Caution: this flexibility is also more dangerous!
  - It violates modularity: can code "know" what locks are held by the thread that is executing it?
  - Nested locks may cause deadlock (later).
- Keep your locking scheme simple and local!
  - Java ensures that each Acquire/Release pair (synchronized block) is contained within a method, which is good practice.

# 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

# 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

# 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

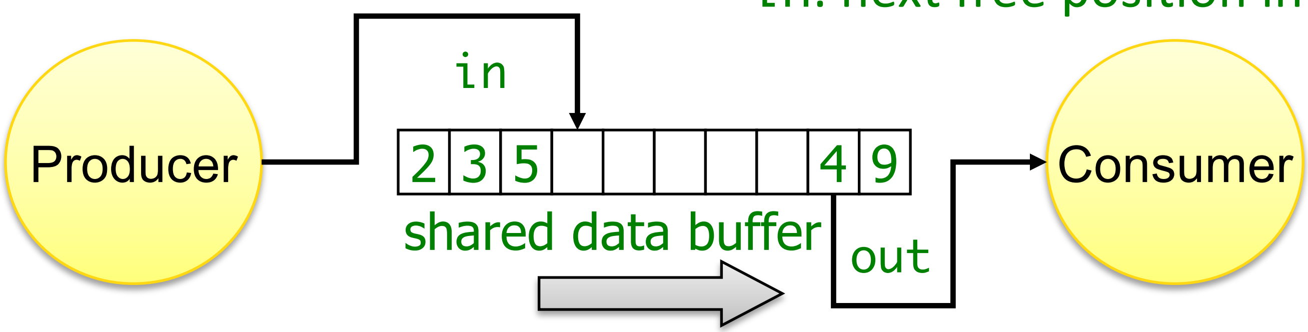
Producer-Consumer

# CLASSIC PROBLEMS

# Producer-Consumer Problem

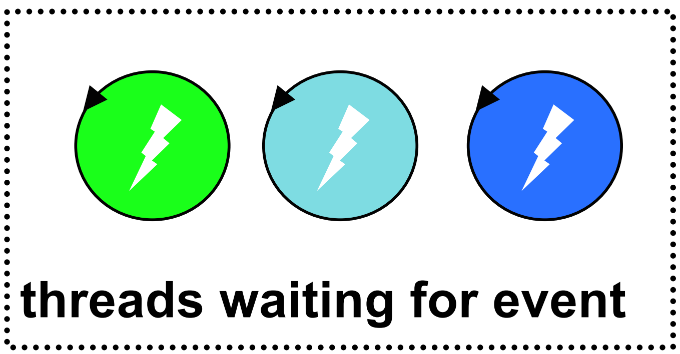
- Have a producer thread creating the items
- Have a consumer thread consuming the items
- Common synchronization problem

One implementation: Circular buffer
`out`: first full position in the buffer
`in`: next free position in the buffer



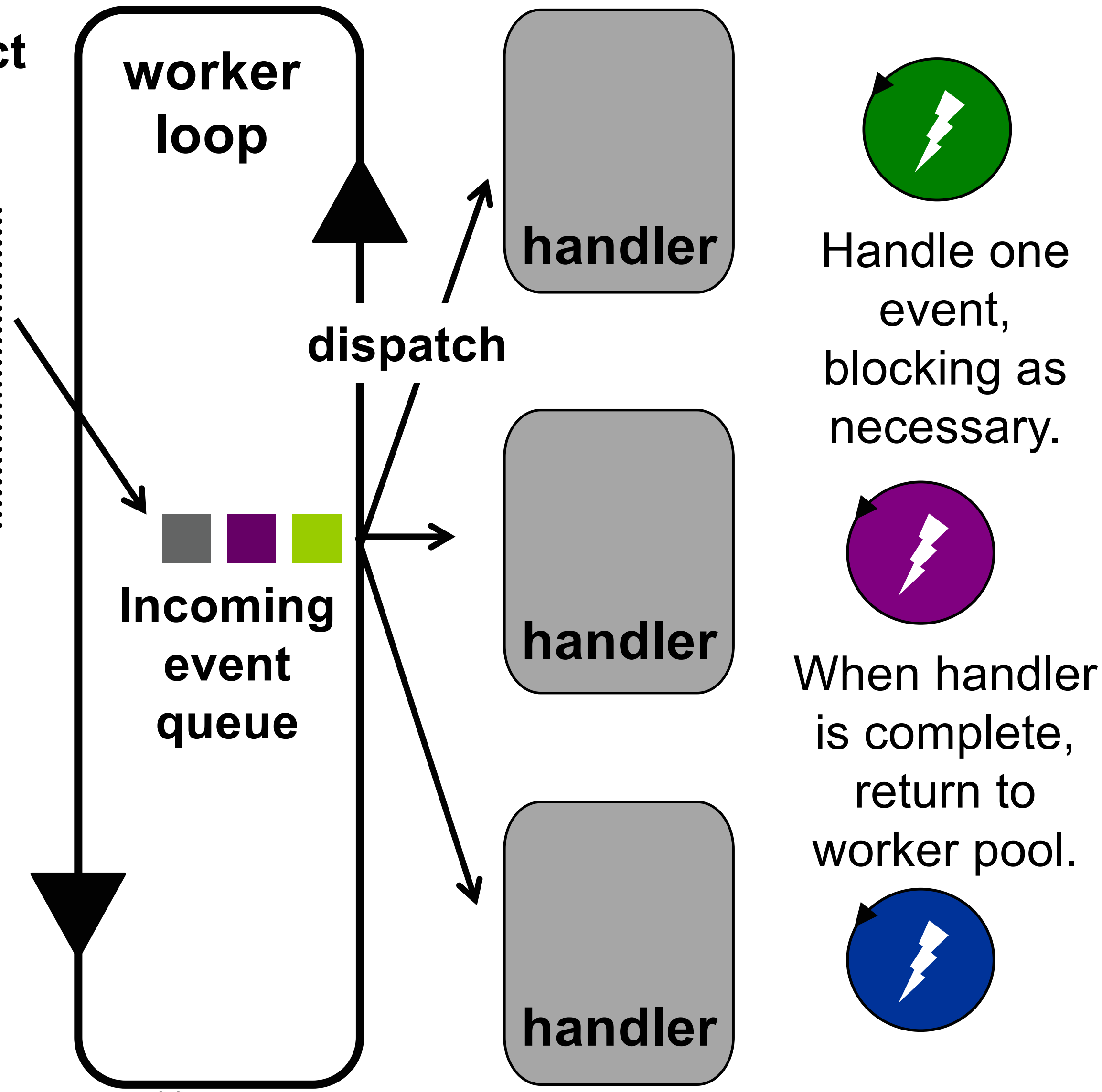
# Producer-Consumer Example: Event/Request Queue

**We can use a mutex to protect a shared event queue.**
"Lock it down."




- But how will worker threads wait on an empty queue?
- How to wait for arrival of the next event?

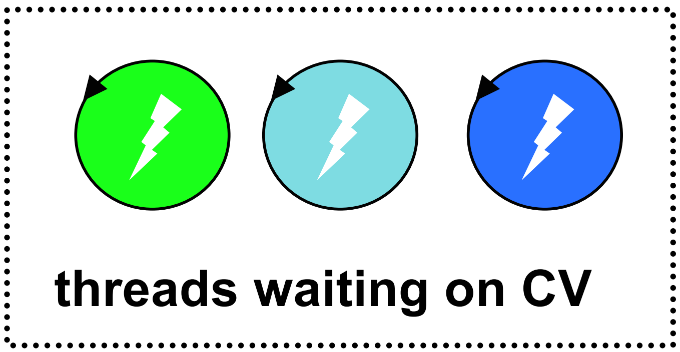
We need suitable primitives to **sleep** (block) for a *condition* and **wakeup** when the *condition* is satisfied.

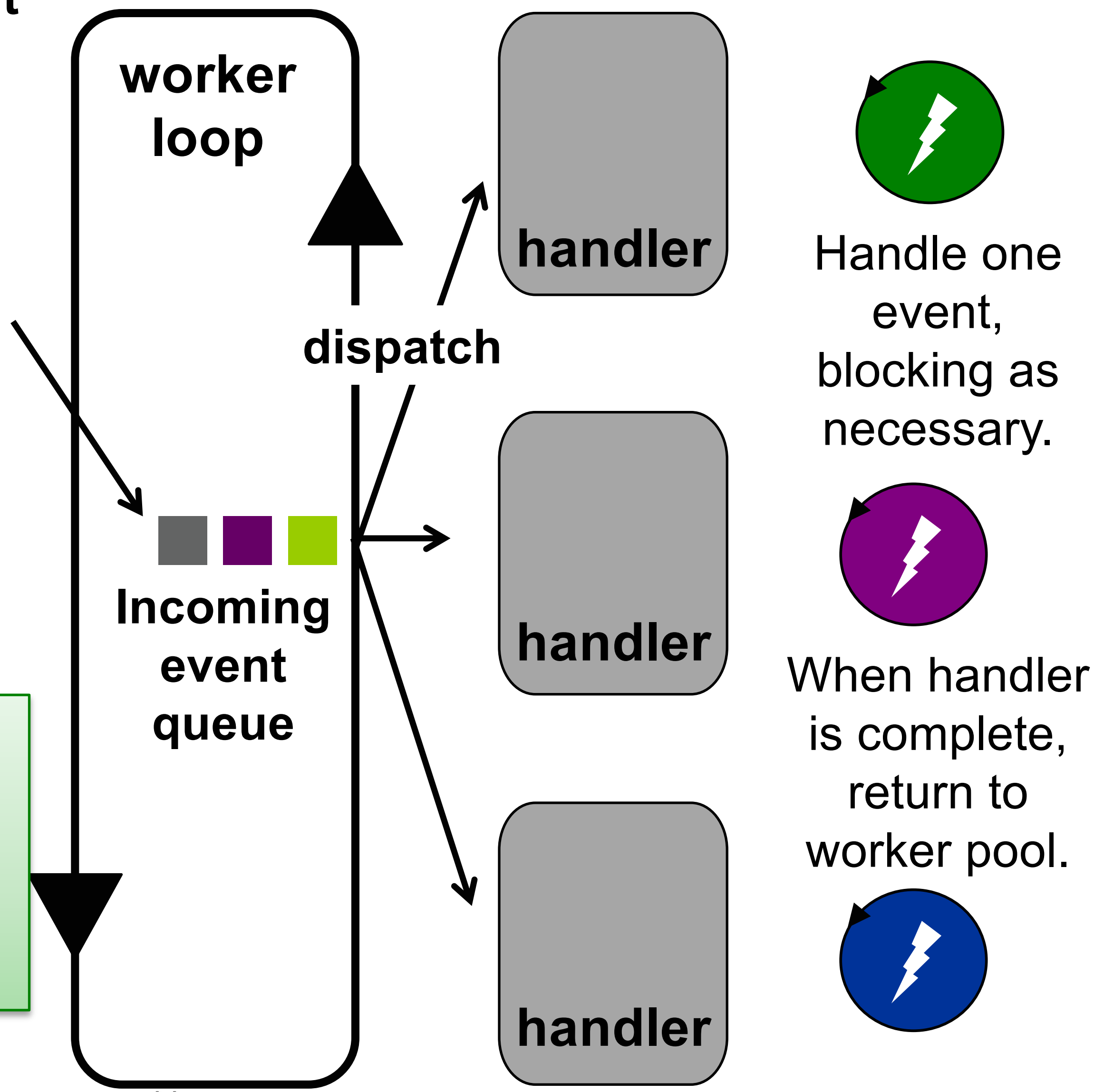


# Producer-Consumer Example: Event/Request Queue

**We can synchronize an event queue with a mutex/CV pair.**

Protect the event queue data structure itself with the mutex.




Workers **wait** on the CV for next event if the event queue is empty.
**Signal** the CV when a new event arrives.

# Producer-Consumer Problem

- Example: Soda machine
  - Producer adds a soda
  - Consumer removes a soda

```
consumer () {

  take a soda from machine

}
```

```
producer () {

  add one soda to machine

}
```

# Solving Producer-Consumer Problems

- What variables/shared state do we need?
- Where do we need mutual exclusion?
  - What is our critical section?
  - How many locks do we need?
- What are our ordering constraints?

# 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

```
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

  take a soda from machine

  hasRoom.signal()
    CV2

  sodaLock.release()
}
```

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

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

  add one soda to machine

  hasSoda.signal()
    CV1

  sodaLock.release()
}
```

# >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

  signal(hasRoom)

  sodaLock.release()
}
```

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

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

  fill machine with soda

  broadcast(hasSoda)

  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 wait, then release?
  - What if release, then wait?

```
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 – "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.

# Looking Ahead

- Project 3 due today
- Synchronization Assignment
  - Part 1: Discussion/pseudocode
  - Part 2: implementation in Java