Today

- Synchronization Mechanisms
  - Mutex
  - Condition Variables
  - Semaphores
  - Monitors

Review

- What are the synchronization mechanisms we covered?
  - When would you use them?
- How do we synchronize Java code?

Synchronization Mechanisms

- Mutex/lock
  - Mutual exclusion: only one thread can access a resource at a time
- Signaling mechanisms:
  - Condition Variable
  - Semaphore
- Monitor: lock/CV combo

Java uses mutexes and CVs

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

Interchangeable lingo
monitor == mutex+CV

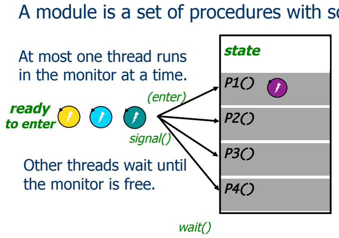
```
public class Object {
    void notify(); /* signal */
    void notifyAll(); /* broadcast */
    void wait();
    void wait(long timeout);
}
```

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

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

Roots: monitors

A ***monitor*** is a module in which execution is serialized.
A module is a set of procedures with some private state.



[Brinch Hansen 1973]
[C.A.R. Hoare 1974]

Java **synchronized** allows finer control over the entry/exit points.
Each Java object is its own "module": objects of a Java class share methods of the class but have private state and a private monitor.

Monitors and mutexes are "equivalent"

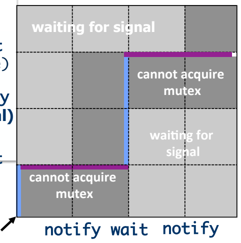
- Entry to a monitor (e.g., a Java synchronized block) is equivalent to `Acquire` of an associated mutex.
  - Lock on entry
- Exit of a monitor is equivalent to `Release`.
  - Unlock on exit (or at least "return the key"…)
- exit/release is implicit and automatic if the thread exits synchronized code via an 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
- 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.

Ping-Pong using a condition variable in Java

```
public void pingPong() {
  synchronized (monitor) {
    monitor.notify();
    try {
      monitor.wait();
    } catch (InterruptedException e)
    {
      e.printStackTrace();
    }
  }
}
```



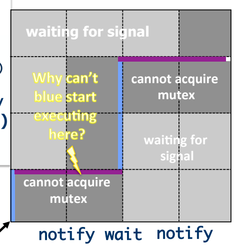
Interchangeable lingo:
synchronized == mutex

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

PingPong.java

Ping-Pong using a condition variable in Java

```
public void pingPong() {
  synchronized (lock) {
    lock.notify();
    try {
      lock.wait();
    } catch (InterruptedException e)
    {
      e.printStackTrace();
    }
  }
}
```



Interchangeable lingo:
synchronized == mutex

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

Implementing Semaphore

```
void P() {
    s = s - 1;
}

void V() {
    s = s + 1;
}
```

Step 0.
Increment and decrement operations on a counter.

But how to ensure that these operations are **atomic**, with **mutual exclusion** and no **races**?

How to implement the blocking (**sleep/wakeup**) behavior of semaphores?

Implementing Semaphore

```
void P() {
   synchronized(this) {
       ….
       s = s - 1;
   }
}

void V() {
   synchronized(this) {
      s = s + 1;
      ….
   }
}
```

Step 1.
Use a **mutex** so that increment (V) and decrement (P) operations on the counter are **atomic**.

Implementing Semaphore

```
synchronized void P() {

     s = s - 1;

}

synchronized void V() {

     s = s + 1;

}
```

Step 1 Alternative
Use a **mutex** so that increment (V) and decrement (P) operations on the counter are **atomic**.

Implementing Semaphore

```
synchronized void P() {
    while (s == 0)
        wait();
    s = s - 1;
}

synchronized void V() {
    s = s + 1;
    if (s == 1)
        notify();
}
```

Step 2.
Use a condition variable to add sleep/wakeup synchronization around a zero count.

Implementing Semaphore

```
synchronized void P() {
    while (s == 0)
        wait();
    s = s - 1;
    ASSERT(s >= 0);
}

synchronized void V() {
    s = s + 1;
    signal();
}
```

Loop before you leap!
Understand why the **while** is needed, and why an **if** is not good enough.

Wait releases the monitor/mutex and blocks until a **signal**.

Signal wakes up one waiter blocked in **P**, if there is one, else the **signal** has no effect: it is forgotten.

This code constitutes a proof that monitors (mutexes and condition variables) are at least as powerful as semaphores.

Implementing Semaphore

```
synchronized void P() {
    while (s == 0)
        wait();
    s = s - 1;
    ASSERT(s >= 0);
}

synchronized void V() {
    s = s + 1;
    signal();
}
```

Loop before you leap!
Understand why the **while** is needed, and why an **if** is not good enough.

Wait releases the monitor/mutex and blocks until a **signal**.

Signal wakes up one waiter blocked in **P**, if there is one, else the **signal** has no effect: it is forgotten.

This code constitutes a proof that monitors (mutexes and condition variables) are at least as powerful as semaphores.

Book shows how monitors can be implemented using semaphores, so …

Binary Semaphores vs. Mutex

- A binary semaphore is similar to a mutex, but …

Binary Semaphores vs. Mutex

- A binary semaphore is similar to a mutex, but …
- Mutex has an *owner*
  - Only the owner can acquire/release the lock
- Semaphores: anyone *could* release the lock

Semaphores vs. Condition Variables

- Semaphores are “prefab CVs” with an atomic integer.
- V(Up) differs from signal (notify) in that …?
- P(Down) differs from wait in that …?

Semaphores vs. Condition Variables

- Semaphores are "prefab CVs" with an atomic integer.
- V(Up) differs from signal (notify) in that:
  - **Signal** has no effect if no thread is waiting on the condition.
    - Condition variables are ***not*** variables! They have no value!
  - **Up** has the same effect whether or not a thread is waiting.
    - Semaphores retain a *memory* of calls to Up.
- P(Down) differs from wait in that:
  - **Down** checks the condition and blocks only if necessary.
    - No need to recheck the condition after returning from Down.
    - The wait condition is defined internally, but is limited to a counter.
  - **Wait** is explicit: it does not check the condition itself, ever.
    - Condition is defined externally and protected by integrated mutex.

Monitors vs. semaphores

- Monitors
  - Separate mutual exclusion and wait/signal
- Semaphores
  - Provide both with same mechanism
- Semaphores are more "elegant"
  - Can be harder to program

Monitors vs. semaphores

```
// Monitors
mutex.lock()

while (condition) {
   cv.wait(mutex)
}

mutex.unlock()
```

```
// Semaphores
semaphore.down()
```

- Where are the conditions in both?
- Which is more flexible?
- Why do monitors need a lock, but not semaphores?

Monitors vs. semaphores

```
// Monitors
mutex.lock()

while (condition) {
   cv.wait(mutex)
}

mutex.unlock()
```

```
// Semaphores
semaphore.down()
```

- When are semaphores appropriate?
  - When shared integer maps naturally to problem at hand, when condition involves a count of one thing

Java Manual

"When waiting upon a Condition, a 'spurious wakeup' is permitted to occur, in general, as a concession to the underlying platform semantics.

This has little practical impact on most application programs as a Condition should always be waited upon in a loop, testing the state predicate that is being waited for."

What does this code do?

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

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

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

Barrier

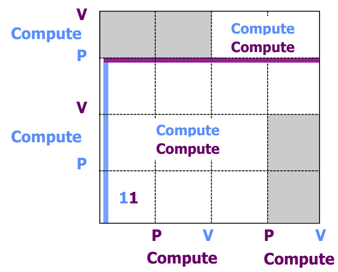
```
blue = Semaphore(1);
purple = Semaphore(1);
```

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

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

Neither thread can advance to the next iteration until its peer completes the current iteration.

Barrier with semaphores



Neither thread can advance to the next iteration until its peer completes the current iteration.

Synchronization: layering

Concurrent Applications

Semaphores Locks Condition Variables

Interrupt Disable Atomic Read/Modify/Write Instructions

Multiple Processors Hardware Interrupts

Looking Ahead

- Wed: Synchronization Assignment
- Project 4 out on Wednesday