CS 475
Operating Systems

UNIVERSITY of PUGET SOUND

Est. 1888

Department of Mathematics
and Computer Science

Lecture 6
Synchronization (Part III)
Monitors

Problem: Synchronization Is Hard

- **Original Problem:** Want to support thread coordination and mutual exclusion
  - Semaphores gave us a way!
  - But they're error-prone, hard to use.
    - Easy to deadlock
    - (Was it obvious at the time of coding?)

- **Want:** Even higher level language support
  - It'd be nice to avoid setting up locks and semaphores at all!

Goals for This Lecture...

- Basic Problem Definition
- Mechanisms to Control Access to Shared Resources
  - Low Level Mechanisms:
    - Busy-Waiting (Spin) Locks
    - Self-Blocking Locks
  - High Level Mechanisms:
    - Semaphores
    - Condition Variables and Monitors

Bank Example

- Joint bank account
  - Bank must ensure synchronization when multiple people (threads) access ATMs simultaneously.

- 3 Threads (People) at different ATMs:
  - T1: withdraw(600)
  - T2: withdraw(300)
  - T3: deposit(500)

```
// shared vars
int balance = 0;

void withdraw(int amt) {
    balance -= amt;
}

void deposit(int amt) {
    balance += amt;
}
```

- *What could go wrong if T1, T2, T3 run concurrently? (It helps to remember that* **-=** *and* **+=** *aren't atomic)*

High-Level Synchronization Mechanisms

- *Monitors (early 1970s)*
  - Developed by C.A.R. (Tony) Hoare.
  - Combines a *lock* and *condition variables* for coordinating threads' access to shared data
  - Has a set of functions that are provided with ***mutual exclusion*** within the monitor.

```
monitor name {
   << shared variables >>

   <<mutually excl. functions>>
   func1(...) {
      ...
   }

   func2(...) {
      ...
   }
}
```

High-Level Synchronization Mechanisms (2)

- *Monitors* (cont.)
  - A monitor lock ensures that **one** thread is actively running within the monitor!
  - All other threads wait in a **queue** to gain entry to the monitor.
  - When thread finishes an operation, it releases the monitor lock

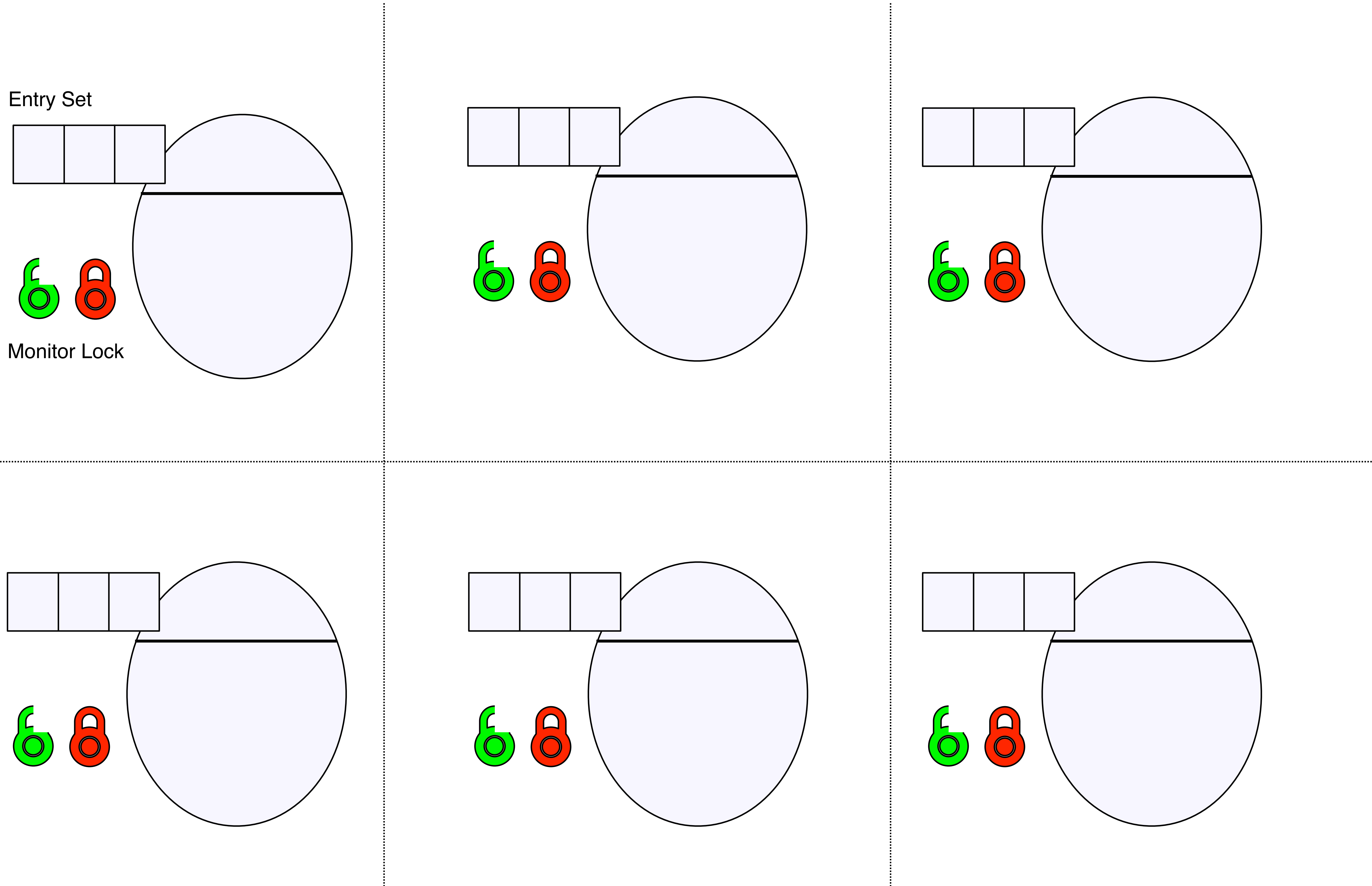
```
monitor name {

  << shared variables >>

  <<mutually excl. functions>>
  func1(...) {
      ...
  }

  func2(...) {
      ...
  }

}
```

Bank Code (This Works!)

- Set up:
  - Shared Variable: balance
  - Race conditions can't occur within withdraw() and deposit()

```
monitor Bank {

   // shared vars
   int balance = 0;

   void withdraw(int amt) {
        balance -= amt;
   }

   void deposit(int amt) {
        balance += amt;
   }
}
```

Bank Example (Mutual Exclusion)



Bank Example

- Joint bank account
  - Bank must ensure synchronization when multiple people access ATMs

Now consider this additional "Correctness Property":

Don’t let customers `withdraw()` if there's not enough $ in account.

(i.e., make customers **wait** to `withdraw()` until there is enough $.)

Integration of Condition Variables

UNIVERSITY *of* PUGET SOUND

- *Monitors*
  - But that "correctness property" is not at all about mutual exclusion.
  - Rather, it's about the coordination of threads (like what semaphores are for!)
  - To coordinate among threads, we use *condition variables* inside monitors.

```
monitor name {

  << shared variables >>

  <<mutually excl. functions>>
  func1(...) {
     ...
  }

  func2(...) {
     ...
  }

  << condition variables >>

}
```

Condition Variables

- *Condition Variables (CV)*
  - They don't store any values
  - Each CV associates a queue of waiting threads inside the monitor

- A condition variables supports three operations:
  - `wait()`: atomically unlock monitor, blocks the calling thread, and places it on the CV's queue.
  - `notify()`: wake up ***one*** thread waiting on CV.
    - Awoken thread competes for re-entry, but starts execution from where it left off.
  - `notifyAll()`: wake up ***all*** threads waiting on CV.
    - Let all threads compete for re-entry.

Monitor Solution

- Try this schedule:
  - T1: withdraw(600)
  - T2: withdraw(300)
  - T3: deposit(500)
  - T4: deposit(500)

```
monitor Bank {

    // shared vars
    int balance = 0;

    void withdraw(int amt) {
        while (balance < amt) {
           notEnough.wait();
        }
        balance -= amt;
    }

    void deposit(int amt) {
       balance += amt;
       if (balance > 0) {
           notEnough.notifyAll();
       }
    }

    // condition vars
    Condition notEnough;

}
```

Monitor Solution

T4 | T3 | T2 | T1

- Try this schedule:
  - T1: withdraw(600) - ready
  - T2: withdraw(300) - ready
  - T3: deposit(500) - ready
  - T4: deposit(500) - ready

```
monitor Bank {

    // shared vars
    int balance = 0;

    void withdraw(int amt) {
        while (balance < amt) {
            notEnough.wait();
        }
        balance -= amt;
    }

    void deposit(int amt) {
       balance += amt;
       if (balance > 0) {
           notEnough.notifyAll();
       }
    }

    // condition vars
    Condition notEnough;
}
```

| | | T1 |

***notEnough's** wait queue*

Current state:

Monitor is initially unlocked

Monitor Solution

T4 | T3 | T2

- Try this schedule:
  - **T1: withdraw(600) - running**
  - T2: withdraw(300) - ready
  - T3: deposit(500) - ready
  - T4: deposit(500) - ready

```
monitor Bank {

    // shared vars
    int balance = 0;

    void withdraw(int amt) {
        while (balance < amt) {
            notEnough.wait();
        }
        balance -= amt;
    }

    void deposit(int amt) {
       balance += amt;
       if (balance > 0) {
           notEnough.notifyAll();
       }
    }

    // condition vars
    Condition notEnough;

}
```

T1

Current state:

T1 acquires the monitor lock and runs withdraw(600)

Monitor Solution

	T4	T3	T2

- Try this schedule:
  - T1: withdraw(600) - *wait* -
  - T2: withdraw(300) - ready
  - T3: deposit(500) - ready
  - T4: deposit(500) - ready

Current state:

T1 must wait on notEnough.

Places itself on the notEnough queue and unlocks the monitor atomically.

```
monitor Bank {

    // shared vars
    int balance = 0;

    void withdraw(int amt) {
        while (balance < amt) {
            notEnough.wait();
        }
        balance -= amt;
    }

    void deposit(int amt) {
       balance += amt;
       if (balance > 0) {
           notEnough.notifyAll();
       }
    }

    // condition vars
    Condition notEnough;
}
```

		T1

Monitor Solution

T4 T3

- Try this schedule:
  - T1: withdraw(600) - wait -
  - **T2: withdraw(300) - running**
  - T3: deposit(500) - ready
  - T4: deposit(500) - ready

Current state:

T2 acquires the monitor lock and runs withdraw(300)

```
monitor Bank {

    // shared vars
    int balance = 0;

    void withdraw(int amt) {
        while (balance < amt) {
            notEnough.wait();
        }
        balance -= amt;
    }

    void deposit(int amt) {
       balance += amt;
       if (balance > 0) {
           notEnough.notifyAll();
       }
    }

    // condition vars
    Condition notEnough;

}
```

T2

T1

Monitor Solution

		T4	T3

- Try this schedule:
  - T1: withdraw(600) - *wait* -
  - T2: withdraw(300) - *wait* -
  - T3: deposit(500) - ready
  - T4: deposit(500) - ready

Current state:

T2 must also wait on notEnough.

Places itself on the notEnough queue and unlocks the monitor atomically.

```
monitor Bank {

    // shared vars
    int balance = 0;

    void withdraw(int amt) {
        while (balance < amt) {
            notEnough.wait();
        }
        balance -= amt;
    }

    void deposit(int amt) {
       balance += amt;
       if (balance > 0) {
           notEnough.notifyAll();
       }
    }

    // condition vars
    Condition notEnough;
}
```

	T2	T1

Monitor Solution

T4

- Try this schedule:
  - T1: withdraw(600) - wait -
  - T2: withdraw(300) - wait -
  - **T3: deposit(500) - running**
  - T4: deposit(500) - ready

Current state:

T3 acquires the monitor lock and runs deposit(500)

```
monitor Bank {

    // shared vars
    int balance = 500;

    void withdraw(int amt) {
        while (balance < amt) {
            notEnough.wait();
        }
        balance -= amt;
    }

    void deposit(int amt) {
        balance += amt;
        if (balance > 0) {
            notEnough.notifyAll();
        }
    }

    // condition vars
    Condition notEnough;
}
```

T3

T2 T1

Monitor Solution

- Try this schedule:
  - T1: withdraw(600) - *wakeup!*
  - T2: withdraw(300) - *wakeup!*
  - **T3: deposit(500) - running**
  - T4: deposit(500) - ready

Current state:

*T1 notifies all threads waiting on notEnough. (What if just **notify**() was used?)*

```
monitor Bank {

    // shared vars
    int balance = 500;

    void withdraw(int amt) {
        while (balance < amt) {
            notEnough.wait();
        }
        balance -= amt;
    }

    void deposit(int amt) {
       balance += amt;
       if (balance > 0) {
           notEnough.notifyAll();
       }
    }

    // condition vars
    Condition notEnough;

}
```

Monitor Solution

T4 T2 T1

- Try this schedule:
  - T1: withdraw(600) - ready
  - T2: withdraw(300) - ready
  - ~~T3: deposit(500)~~ - done -
  - T4: deposit(500) - ready

Current state:

T1, T2 re-enter entry set with priority.

T3 exits and unlocks the monitor.

```
monitor Bank {

    // shared vars
    int balance = 500;

    void withdraw(int amt) {
        while (balance < amt) {
            notEnough.wait();
        }
        balance -= amt;
    }

    void deposit(int amt) {
        balance += amt;
        if (balance > 0) {
            notEnough.notifyAll();
        }
    }

    // condition vars
    Condition notEnough;

}
```

T3

Monitor Solution

- Try this schedule:
  - **T1: withdraw(600) - running**
  - T2: withdraw(300) - ready
  - ~~T3: deposit(500)~~ - done -
  - T4: deposit(500) - ready

Current state:

T1 acquires monitor lock.

T1 picks up from where it left off in the loop. Goes right back to check the looping condition. Must wait again.

```
monitor Bank {

    // shared vars
    int balance = 500;

    void withdraw(int amt) {
        while (balance < amt) {
            notEnough.wait();
        }
        balance -= amt;
    }

    void deposit(int amt) {
       balance += amt;
       if (balance > 0) {
           notEnough.notifyAll();
       }
    }

    // condition vars
    Condition notEnough;

}
```

T1

Monitor Solution

T4 | T2

- Try this schedule:
  - T1: withdraw(600) - *wait* -
  - T2: withdraw(300) - ready
  - ~~T3: deposit(500)~~ - done -
  - T4: deposit(500) - ready

```
monitor Bank {

    // shared vars
    int balance = 500;

    void withdraw(int amt) {
        while (balance < amt) {
            notEnough.wait();
        }
        balance -= amt;
    }

    void deposit(int amt) {
        balance += amt;
        if (balance > 0) {
            notEnough.notifyAll();
        }
    }

    // condition vars
    Condition notEnough;    T1

}
```

Current state:

T1 back in the notEnough queue.
Unlocks monitor atomically.

Monitor Solution

T4

- Try this schedule:
  - T1: withdraw(600) - wait -
  - **T2: withdraw(300) - running**
  - ~~T3: deposit(500)~~ - done -
  - T4: deposit(500) - ready

Current state:

T2 acquires monitor lock.

T2 picks up from where it left off in the loop. Breaks out of loop!

Updates balance.

```
monitor Bank {

    // shared vars
    int balance = 200;

    void withdraw(int amt) {
        while (balance < amt) {
            notEnough.wait();
        }
        balance -= amt;
    }

    void deposit(int amt) {
        balance += amt;
        if (balance > 0) {
            notEnough.notifyAll();
        }
    }

    // condition vars
    Condition notEnough;

}
```

T2

T1

Monitor Solution

T4

- Try this schedule:
  - T1: withdraw(600) - wait -
  - ~~T2: withdraw(300)~~ - done -
  - ~~T3: deposit(500)~~ - done -
  - T4: deposit(500) - ready

Current state:

T2 exits, and unlocks the monitor atomically.

```
monitor Bank {

    // shared vars
    int balance = 200;

    void withdraw(int amt) {
        while (balance < amt) {
            notEnough.wait();
        }
        balance -= amt;
    }

    void deposit(int amt) {
        balance += amt;
        if (balance > 0) {
            notEnough.notifyAll();
        }
    }

    // condition vars
    Condition notEnough;

}
```

T1

T2

Monitor Solution

- Try this schedule:
  - T1: withdraw(600) - *wakeup!*
  - ~~T2: withdraw(300)~~ - done -
  - ~~T3: deposit(500)~~ - done -
  - **T4: deposit(500) - running**

```
monitor Bank {

    // shared vars
    int balance = 700;

    void withdraw(int amt) {
        while (balance < amt) {
            notEnough.wait();
        }
        balance -= amt;
    }

    void deposit(int amt) {
        balance += amt;
        if (balance > 0) {
            notEnough.notifyAll();
        }
    }

    // condition vars
    Condition notEnough;

}
```

T4

T1

Current state:

T4 acquires monitor lock.

T4 runs deposit(500). Updates the balance and notifies all.

Monitor Solution

T1

- Try this schedule:
  - T1: withdraw(600) - ready
  - ~~T2: withdraw(300)~~ - done -
  - ~~T3: deposit(500)~~ - done -
  - ~~T4: deposit(500)~~ - done -

Current state:

T1 re-enters entry set.

T4 exits and unlocks monitor.

```
monitor Bank {

    // shared vars
    int balance = 700;

    void withdraw(int amt) {
        while (balance < amt) {
            notEnough.wait();
        }
        balance -= amt;
    }

    void deposit(int amt) {
        balance += amt;
        if (balance > 0) {
            notEnough.notifyAll();
        }
    }

    // condition vars
    Condition notEnough;

}
```

T4

Monitor Solution

- Try this schedule:
  - **T1: withdraw(600) - running**
  - ~~T2: withdraw(300)~~ - done -
  - ~~T3: deposit(500)~~ - done -
  - ~~T4: deposit(500)~~ - done -

```
monitor Bank {

    // shared vars
    int balance = 100;

    void withdraw(int amt) {
        while (balance < amt) {
            notEnough.wait();
        }
        balance -= amt;
    }

    void deposit(int amt) {
        balance += amt;
        if (balance > 0) {
            notEnough.notifyAll();
        }
    }

    // condition vars
    Condition notEnough;

}
```

T1

Current state:

T1 acquires monitor lock.

T1 updates balance (finally!)

Monitor Solution

- Try this schedule:
  - ~~T1: withdraw(600)~~ - done
  - ~~T2: withdraw(300)~~ - done -
  - ~~T3: deposit(500)~~ - done -
  - ~~T4: deposit(500)~~ - done -

Current state:

T1 exits, unlocks the monitor.

Balance correctly reflects $100.

```
monitor Bank {

    // shared vars
    int balance = 100;

    void withdraw(int amt) {
        while (balance < amt) {
            notEnough.wait();
        }
        balance -= amt;
    }

    void deposit(int amt) {
       balance += amt;
       if (balance > 0) {
           notEnough.notifyAll();
       }
    }

    // condition vars
    Condition notEnough;

}
```

T1

Goals for This Lecture...

- Basic Problem Definition
- Mechanisms to Control Access to Shared Resources
  - Low Level Mechanisms:
    - Busy-Waiting (Spin) Locks
    - Self-Blocking Locks
  - High Level Mechanisms:
    - Semaphores
    - Condition Variables and Monitors
      - Example: Java

Java's Synchronization Support

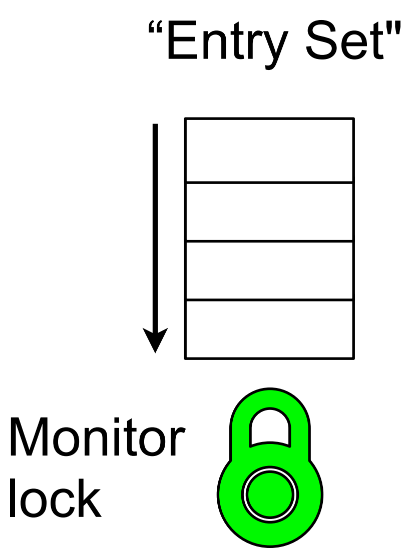
- The **`synchronized`** keyword
  - Java allows methods to be declared to be **`synchronized`**
  - Also allows definition of **`synchronized blocks`**

- Every Java `Object` is associated with a *monitor (or "intrinsic") lock*
  - When a method is declared to be **`synchronized`**, a thread calling the method must first acquire the intrinsic lock.
    - Or it waits in the Entry Set (This is just like Monitors!)
  - Synchronized Method Syntax:

```
public synchronized returnType method(...) {
   // everything you do here is mutually exclusive
}
```

Java's Synchronization Support (Cont.)

- *Java implements the monitor structure!*
  - Guarantees mutex when a thread calls it (acquires monitor lock)
  - Monitor lock is released when thread exits the method




```java
public class Counter {
   private int val = 0;

   public synchronized void inc() {
      // do stuff with Counter object locked
      val++;
      // unlock right before leaving method
   }

   public int getval() {
      // this method doesn't require synchronization
      // threads don't compete for the monitor lock
      return val;
   }
}
```

Java’s Synchronization Support (Cont.)

- Sometimes you don't *need* mutual exclusion on the *whole* method body.
  - You can use a *synchronized block* if only a critical section of code within the method needs to be locked.
  - **Synchronized Block Syntax:**

```
public void someMethod() {

   synchronized(someObject) {
      // acquires intrinsic lock on some object object
      // releases lock after you leave the block
   }
}
```

Synchronized Blocks

- These two code snippets are equivalent:
  - If you just want to run a block of code mutually exclusively inside the current object, just use "**this**"

```
public class Counter {
   private int val = 0;

   public void inc() {
      // lock up the current object
      synchronized(this) {
          val++;
      }
   }
}
```

```
public class Counter {
   private int val = 0;

   public synchronized void inc() {
      val++;
   }
}
```

Synchronized Blocks (2)

- But synchronized blocks buy us more flexibility.
  - There may be lots of code before and after the critical section that can be run without synchronization.

```
public class Counter {
   private int val = 0;

   public void inc() {
      // stuff that doesn't need synced

      synchronized(this) {
          // 'this' object is now locked!
         val++;
      }
      // 'this' Counter object is released!

      // more stuff that doesn't need synced
}
```

What about Condition Variables?

- *Every Object* can *also* be used as a Condition Variable (CV)!
  - In addition to a monitor lock (and queue), every object additionally has a CV wait queue. (Yes, an object can have two queues!)

- Three operations (API for `Object` class):
  - **`wait()`** - Waits until another thread calls `notify()` or `notifyAll()`
  - **`notify()`** - Wake up *one* thread that's waiting
  - **`notifyAll()`** - Wake up *all* threads that's waiting

Java's Condition Variables (CVs)

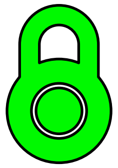
- Rule: All CV operations **must be done** in a `synchronized` block that locks the CV up.

- **Syntax:** Assume below that **notEnough** is an Object variable.

```
public void someMethod() {
   synchronized(notEnough) {
      notEnough.wait();
   }
}

public void someOtherMethod() {
   synchronized(notEnough) {
      notEnough.notifyAll();
   }
}
```

Java's Condition Variables (CVs)

- Rule: All CV operations <u>**must be done**</u> in a **`synchronized`** block that locks the CV up.



```java
// This code is correct

public class Counter {
   private int val = 0;   // shared data
   private Object foo = new Object(); //  CV

   public void dec() {
       synchronized (foo) {
           while (val < 10) {
               foo.wait();
           }
           val--;
       }
   }

   public void inc() {
       synchronized (foo) {
           val++;
           if (val >= 10) {
               foo.notify();
           }
       }
   }
}
```

Java's Condition Variables (CVs)

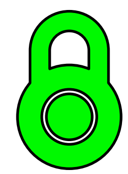
- Restoring the synchronized blocks on CV foo...

Assume the same scenario before:
Thread T1 is running `dec()` and needs to wait. But OS context switches to T2 right before `foo.wait()` is called.

However, T2 cannot make progress in `inc()` because T1 still holds the lock on **foo**!

Eventually, OS must switch back to T1. Now it waits, releasing **foo**'s lock.

T2 gets to run (finally), and notifies T1 to wake up.

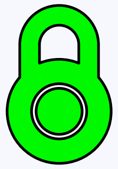


```java
public class Counter {
   private int val = 0;  // shared data
   private Object foo = new Object(); //  CV

   public void dec() {
       synchronized (foo) {
           while (val < 10) {
               foo.wait();
           }
           val--;
       }
   }

   public void inc() {
       synchronized (foo) {
           val++;
           if (val >= 10) {
               foo.notify();
           }
       }
   }
}
```

Another Common Problem

- Problem: Using synchronized methods/block when you don't need to.
  - Say I added the **synchronized** keyword to the method declarations.

```
public class Counter {
   private int val = 0;  // shared data
   private Object foo = new Object();

   public synchronized void doStuff() {
     synchronized (foo) {
         while (foo.count < 10)
           foo.wait();
         val--;
      }
   }

   public synchronized void doSomething() {
     synchronized(foo) {
         val++;
         if (foo.count >= 10)
            foo.notify();
      }
    }
}
```



foo's lock

Another Common Problem

- Recall that a **synchronized** method is just like having a giant **synchronized** block on **this**.
  - That means there's now monitor lock too

Thread 1 Runs doStuff()

Thread 2 Runs doSomething()

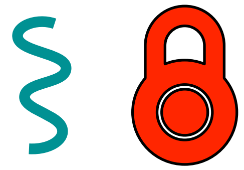
Counter's lock (monitor lock)

```
public class Counter {
  private int val = 0;  // shared data
  private Object foo = new Object();

  public void doStuff() {
    synchronized(this) {
      synchronized (foo) {
         while (foo.count < 10)
           foo.wait();
         val--;
      }
    }
  }

  public void doSomething() {
    synchronized(this) {
      synchronized(foo) {
         val++;
         if (foo.count >= 10)
            foo.notify();
      }
    }
  }
```

foo's lock

Another Common Problem

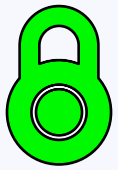
T1



Counter's
lock
(monitor lock)

Thread 1 Runs doStuff():
- Acquires monitor lock.

```java
public class Counter {
  private int val = 0;  // shared data
  private Object foo = new Object();

  public void doStuff() {
    synchronized(this) {
      synchronized (foo) {
         while (foo.count < 10)
           foo.wait();
         val--;
      }
    }
  }

  public void doSomething() {
    synchronized(this) {
      synchronized(foo) {
         val++;
         if (foo.count >= 10)
            foo.notify();
      }
    }
  }
```



foo's
lock

Another Common Problem

Thread 1 Runs doStuff():
- Acquires monitor lock.
- Acquires foo's lock

Counter's lock (monitor lock)

```
public class Counter {
  private int val = 0;  // shared data
  private Object foo = new Object();

  public void doStuff() {
    synchronized(this) {
      synchronized (foo) {
         while (foo.count < 10)
           foo.wait();
         val--;
      }
    }
  }

  public void doSomething() {
    synchronized(this) {
      synchronized(foo) {
         val++;
         if (foo.count >= 10)
            foo.notify();
      }
    }
  }
```

T1

foo's lock

Another Common Problem

Thread 1 Runs doStuff():

- Acquires monitor lock.
- Acquires foo's lock
- Waits on **foo**. Unlocks **foo**.
  (Monitor lock is still held!)

Counter's lock (monitor lock)

```java
public class Counter {
  private int val = 0;   // shared data
  private Object foo = new Object();

  public void doStuff() {
    synchronized(this) {
      synchronized (foo) {
         while (foo.count < 10)
           foo.wait();
         val--;
      }
    }
  }

  public void doSomething() {
    synchronized(this) {
      synchronized(foo) {
         val++;
         if (foo.count >= 10)
            foo.notify();
      }
    }
  }
```

foo's lock

T1 (queued

Another Common Problem

T2

Counter's lock (monitor lock)

```java
public class Counter {
  private int val = 0;  // shared data
  private Object foo = new Object();

  public void doStuff() {
    synchronized(this) {
      synchronized (foo) {
        while (foo.count < 10)
          foo.wait();
        val--;
      }
    }
  }

  public void doSomething() {
    synchronized(this) {
      synchronized(foo) {
        val++;
        if (foo.count >= 10)
          foo.notify();
      }
    }
  }
```

foo's lock

T1 (queued

Thread 1 Runs doStuff():

- Acquires monitor lock.
- Acquires foo's lock
- Waits on **foo**. Unlocks **foo**. (Monitor lock is still held!)

Thread 2 Runs doSomething():

- Attempts to acquire monitor lock.

Another Common Problem

T2
(queued here)

Counter's
lock
(monitor lock)

```java
public class Counter {
  private int val = 0;  // shared data
  private Object foo = new Object();

  public void doStuff() {
    synchronized(this) {
      synchronized (foo) {
         while (foo.count < 10)
           foo.wait();
         val--;
      }
    }
  }

  public void doSomething() {
    synchronized(this) {
      synchronized(foo) {
         val++;
         if (foo.count >= 10)
            foo.notify();
      }
    }
  }
```

foo's
lock

T1
(queued

Thread 1 Runs doStuff():
- Acquires monitor lock.
- Acquires foo's lock
- Waits on **foo**. Unlocks **foo**. (Monitor lock is still held!)

Thread 2 Runs doSomething():
- Attempts to acquire monitor lock.
- Must wait at monitor entry. Queues up in entry set.
- **Deadlock.**

Goals for This Lecture...

- Basic Problem Definition
- Mechanisms to Control Access to Shared Resources
  - Low Level Mechanisms:
    - Busy-Waiting (Spin) Locks
    - Self-Blocking Locks
  - High Level Mechanisms:
    - Semaphores
    - Condition Variables and Monitors
      - Example: The Bank Example revisited in Java.

(Java Solution to the Bank Problem)

```
public class TheBank {
    private int balance = 0;
    private Object notEnough = new Object();  //condition variable

    public void withdraw(int amt) {
       synchronized(notEnough) {   // lock on notEnough CV; No lock on TheBank
          while (balance < amt) {
            try {
               notEnough.wait();  //unlock notEnough and wait (atomically!)
            } catch (InterruptedException e) {
              e.printStackTrace();
            }
          }
          balance -= amt;
          System.out.println("Withdrew $" + amt + ". Now $" + balance);
        }
    }

    public void deposit(int amt) {
        synchronized(notEnough) {   // lock on notEnough CV; No lock on TheBank
          balance += amt;
          if (balance > 0)
            notEnough.notifyAll();
          System.out.println("Deposited $" + amt + ". Now $" + balance);
        }
    }
}
```

Customer Threads

- Thread Objects need to implement **Runnable**

```java
public class Customer implements Runnable {
   private String name;
   private TheBank myBank;

   public Customer(String name, TheBank bank) {
      this.name = name;
      myBank = bank;
   }

   /** Runs automatically on thread start */
   @Override
   public void run() {
      while (true) {
         if (name.equals("Adam")) {
           // Adam always (synchronously) deposits $1
           myBank.deposit(1);
         }
         else {
            // Brad and America always (synchronously) withdraw $3
           myBank.withdraw(3);
         }
      }
   }
}
```

Run the Customer Threads! (**Thread** Object)

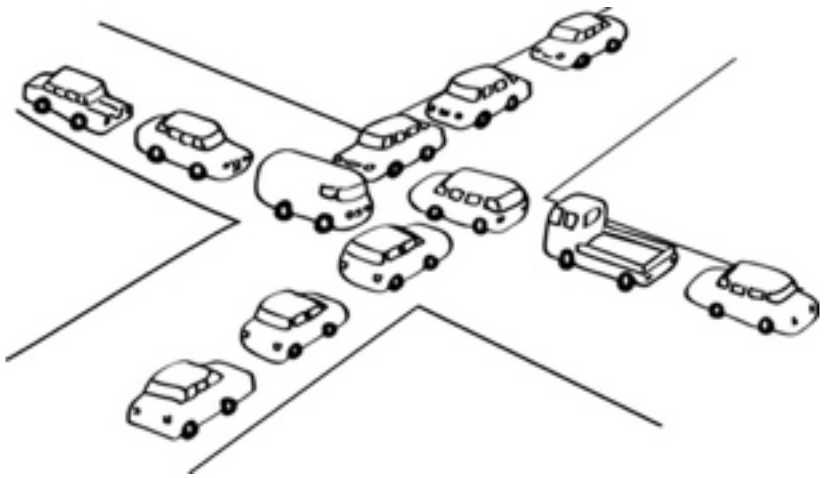
```
public class Tester {

  private static final String[] cust_names = {"Adam", "Brad", "America"};

  public static void main(String[] args) {
    TheBank syncedBank = new TheBank();

    // start the threads
    Thread[] customers = new Thread[cust_names.length];
    for (int i = 0; i < cust_names.length; i++) {
      // Creating Thread objects -- need to encapsulate a Runnable object
      customers[i] = new Thread(new Customer(cust_names[i], syncedBank));

      // Not a typo -- call start(), not run()
      customers[i].start();
    }

    //join the customer threads
    for (int i = 0; i < cust_names.length; i++) {
      try {
       customers[i].join();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    System.out.println("Done!");
  }
}
```

Java Code Examples

- You can check out the Java code examples.
  - https://github.com/davidtchiu/cs475-lec-producerConsumerJava
  - https://github.com/davidtchiu/cs475-lec-theBankJava

In Conclusion...

- Synchronization is hard
  - But increasingly necessary for today's programmers
    - Tough to get it right because an incorrect execution may only happen very, very rarely
  - Java makes things a little easier
- Because <u>waiting</u> is an essential mechanism, threads could wait on each other forever
  - How might we deal with *deadlocks*?



Administrivia 3/27

- Announcements:
  - Hwk 6 extended to tomorrow!
- Last time...
  - Binary vs counting semaphores
  - Solved bounded-buffer problem using semaphores
- Today:
  - Monitors and condition variables
  - Java

Administrivia 3/29

- Announcements:
  - Review Study Guide #2 is up on canvas
- Last time...
  - Semaphores are hard to set up and use. Error-prone.
    - What can we build using semaphores that makes sync easier?
    - Monitors & Condition variables
- Today
  - Synchronization in Java
  - Synchronized methods, synchronized blocks
  - Multithreading syntax in Java
  - *(On exams: You do not need to know locks and semaphores in Java)*

Administrivia 3/29 (Cont)

- Faculty candidate talk
  - Voronoi Cells of Varieties in Various Distances
  - Dr. Maddie Weinstein, Stanford University
  - Faculty candidate in Math

- Where: TH 391
- When: Thursday 3/30 (tomorrow!), 4-5pm