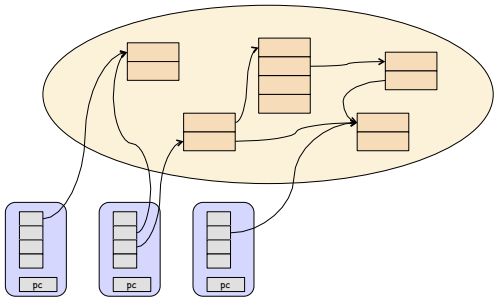
Concurrency, part 2

CSCI 334
Stephen Freund

Shared-Memory Concurrency



Shared-Memory Concurrency

- Critical Section
  - coded in which process may access shared resource
- Race Condition
  - inconsistent behavior if two actions are interleaved
- Mutual Exclusion
  - allow only one process in critical section
  - process may need to wait for another to exit crit. section
- Deadlock
  - occurs when no process can proceed

Deadlock

Thread 1: `a.transfer(b,n)`
Thread 2: `b.transfer(a,n)`

```
class Account {
  int balance;

  synchronized void add(int n) {
    balance += n;
  }

  synchronized void transfer(Account other,
                             int n) {
    balance -= n;
    other.add(n);
  }
```

```
acquire(a)
a.bal -= n
wait for b

acquire(b)
b.bal -= n
wait for a
```

java.lang.StringBuffer...

```
public final class StringBuffer {
  int count;
  char chars[];
  synchronized int length() { return count; }
  synchronized int getChars(...) { ... }

  synchronized StringBuffer append(StringBuffer sb) {
    int len = sb.length();
    ...
    sb.getChars(0, len, value, count);
    ...
 }
}
```

Interrupting Threads

```
class Example {
  public static void main(String[] args) {
    Buffer<String> buffer = new Buffer<String>(5);
    Producer prod = new Producer(buffer);
    Consumer cons = new Consumer(buffer);
    prod.start();
    cons.start();
    try {
        prod.join();
        cons.interrupt();
    } catch (InterruptedException e) {
        System.out.println("...");
    }
  }
}
```

Consumers

```
class Consumer extends Thread {
    private final Buffer<Character> buffer;

    public Consumer(Buffer<Character> b) {
        buffer = b;
    }

    public void run() {
        while (true) {
            char c = buffer.delete();
            System.out.print(c);
        }
    }
}
```

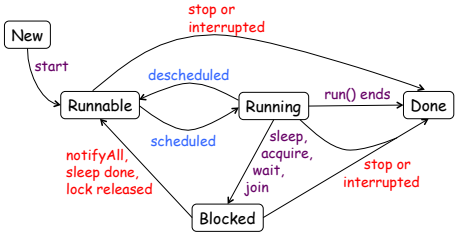
Consumers With Handler

```
class Consumer extends Thread {
    private final Buffer<Character> buffer;

    public Consumer(Buffer<Character> b) {
        buffer = b;
    }

    public void run() {
        try {
            while (true) {
                char c = buffer.delete();
                System.out.print(c);
            }
        } catch (InterruptedException e) {
            // thread interrupted, so stop loop
        }
    }
}
```

Thread States



Self control
JVM Scheduler
External control

Safe Buffer Ops

```
class Buffer<T> {

  private T[] elementData;
  private int elementCount;
  private int start;
  private int end;

  synchronized void insert(T t) throws InterruptedException {
    while (elementCount == elementData.length) wait();
    end = (end + 1) % elementData.length;
    elementData[end] = t;
    elementCount++;
    notifyAll();
  }

  synchronized T delete() throws InterruptedException {
    while (elementCount == 0) wait();
    T elem = elementData[start];
    start = (start + 1) % elementData.length;
    elementCount--;
    notifyAll();
    return elem;
  }
}
```

Atomicity Demo

java.util.StringBuffer...

```
public final class StringBuffer {
  int count;
  char chars[];
  synchronized int length() { return count; }
  synchronized int getChars(...) { ... }

  synchronized StringBuffer append(StringBuffer sb) {
    int len = sb.length();
    ...
    sb.getChars(0, len, value, count);
    ...
  }
}
```

http://www.docjar.com/html/api/java/lang/StringBuffer.java.html

Atomic As a Language Feature

```
class Account {
  int balance;

  atomic void add(int n) {
    balance += n;
  }

  atomic void transfer(Account other, int n) {
    balance -= n;
    other.add(n);
  }
```

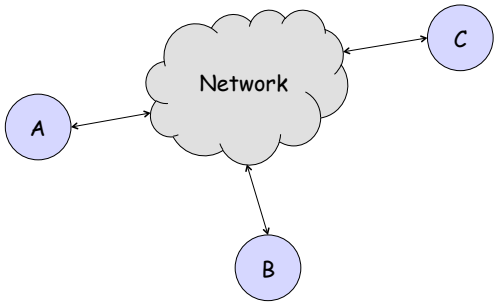
Pessimistic Atomicity Implementation

```
class Account {
  int balance;

  void add(int n) {
    synchronized(global_lock) {
      balance += n;
    }
  }

  void transfer(Account other, int n) {
    synchronized(global_lock) {
      balance -= n;
      other.add(n);
    }
  }
```

What's good? What's bad? Alternatives?



Simple Actor

```
class SimpleActor(val verb: String) extends Actor {
  def act() = {
    for (i <- 1 to 5) {
      println("I'm " + verb + "ing");
      Thread.sleep(1000);
    }
  }
  start();
}
```

Parroting Actor

```
class Parrot extends Actor {
  def act() = {
    loop {
      react {
        case msg => println("Recieved: " + msg);
      }
    }
  }
  start();
}
...
val p = new Parrot();
p ! "moo";
```

Matching Messages

```
abstract class Message { }
case class Hello() extends Message { }
case class Number(val n : Int) extends Message { }

class FussyParrot extends Actor {
  def act() = {
    loop {
      react {
        case Hello     => println("Hello to you too");
        case Number(n) => println("Number " + n);
      }
    }
  }
  start();
}
```

Bank Account

```
abstract class Message { }
case class DepositAmt(n : Int) extends Message;
case class GetBalance() extends Message;

class Account(var balance : Int) extends Actor {
  def act = {
    loop {
      react {
        case DepositAmt(i) => balance = balance + i;
        case GetBalance()  => sender ! balance;
      }
    }
  }
  start();
}
```

Receive, sender, rendezvous...

```
class PickANumber extends Actor {
  def act() = {
    var done = false;

    println("Send me an upper bound...");
    val num = receive { case n : Int => Random.nextInt(n); }

    while (!done) {
      receive {
        case i : Int if (i == num) => sender ! "You Win."; done = true;
        case i : Int if (i < num)  => sender ! "Too Low.";
        case i : Int if (i > num)  => sender ! "Too High.";
      }
    }
    println("Done...");
  }
  start()
}
```