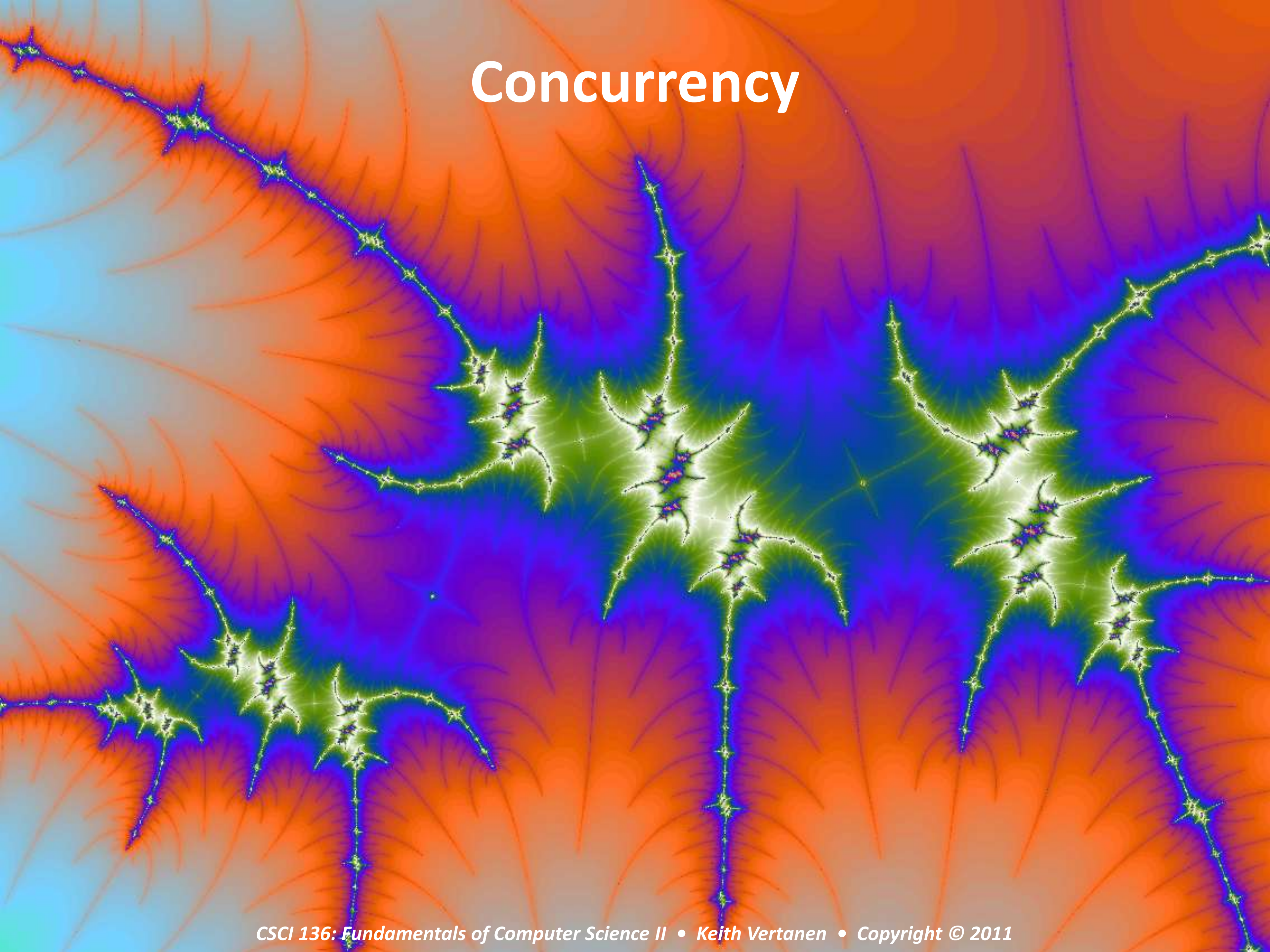

Concurrency
CSCI 136: Fundamentals of Computer Science II • Keith Vertanen • Copyright © 2011

Overview

- Multi-threaded programs
  - Multiple simultaneous paths of execution
    - Seemingly at once (single core)
    - Actually at the same time (multiple cores)
- Concurrency issues
  - The dark side of threading
  - Unpredictability of thread scheduler can get you
  - Protecting shared data using `synchronized` methods
- Deadlock
  - The really dark side of threading

Programming activity

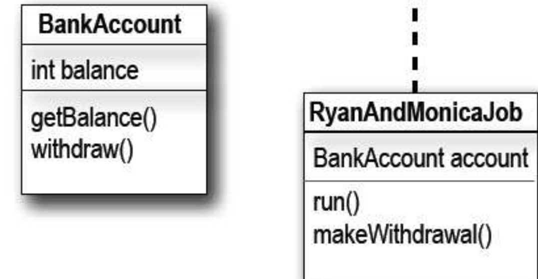
- Build a class that implements a worker thread
  - Draw something in unit box
  - Sleep
  - Change something about the drawing
  - Repeat forever
  - Don't worry about erasing (don't call `StdDraw.clear()`)
- Email me your completed class
  - I'll integrate into my ThreadZoo program

Trouble in concurrency city

- Ryan and Monica problem
  - Two people (threads)
  - Sharing data that exists in single bank account object
  - Always check balance before withdrawing
  - Only withdraw if enough funds

```
class BankAccount {
   private int balance = 100;

   public int getBalance() {
      return balance;
   }
   public void withdraw(int amount) {
      balance = balance - amount;
   }
}
```

The account starts with a balance of $100.



Making the withdrawal

```
private void makeWithdrawal(int amount) {
    if (account.getBalance() >= amount) {
    System.out.println(Thread.currentThread().getName() + " is about to withdraw");
     try {
       System.out.println(Thread.currentThread().getName() + " is going to sleep");
       Thread.sleep(500);
     } catch(InterruptedException ex) {ex.printStackTrace(); }
     System.out.println(Thread.currentThread().getName() + " woke up.");
     account.withdraw(amount);
     System.out.println(Thread.currentThread().getName() + " completes the withdrawal");
     }
   else {
     System.out.println("Sorry, not enough for " + Thread.currentThread().getName());
   }
  }
}
```

Check the account balance, and if there's not enough money, we just print a message. If there IS enough, we go to sleep, then wake up and complete the withdrawal, just like Ryan did.

We put in a bunch of print statements so we can see what's happening as it runs.

Firing up Ryan and Monica

```
public class RyanAndMonicaJob implements Runnable {

  private BankAccount account = new BankAccount();

  public static void main (String [] args) {
     RyanAndMonicaJob  theJob = new RyanAndMonicaJob();
     Thread one = new Thread(theJob);
     Thread two = new Thread(theJob);
     one.setName("Ryan");
     two.setName("Monica");
     one.start();
     two.start();
   }

 public void run() {
  for (int x = 0; x < 10; x++) {
     makeWithdrawal(10);
     if (account.getBalance() < 0) {
       System.out.println("Overdrawn!");
     }
   }
}
```

There will be only ONE instance of the RyanAndMonicaJob. That means only ONE instance of the bank account. Both threads will access this one account.

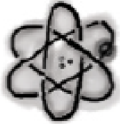
Instantiate the Runnable (job)

Make two threads, giving each thread the same Runnable job. That means both threads will be accessing the one account instance variable in the Runnable class.

In the run() method, a thread loops through and tries to make a withdrawal with each iteration. After the withdrawal, it checks the balance once again to see if the account is overdrawn.

Locks

- Only one person in makeWithdrawal at a time!
  - Tell Java this using `synchronized` keyword



```
private synchronized void makeWithdrawal(int amount) {

    if (account.getBalance() >= amount) {
       System.out.println(Thread.currentThread().getName() + " is about to withdraw");
       try {
          System.out.println(Thread.currentThread().getName() + " is going to sleep");
          Thread.sleep(500);
       } catch(InterruptedException ex) {ex.printStackTrace(); }
       System.out.println(Thread.currentThread().getName() + " woke up.");
       account.withdraw(amount);
       System.out.println(Thread.currentThread().getName() + " completes the withdrawl");
    } else {
      System.out.println("Sorry, not enough for " + Thread.currentThread().getName());
    }
}
```

The `synchronized` keyword means that a thread needs a key in order to access the synchronized code.

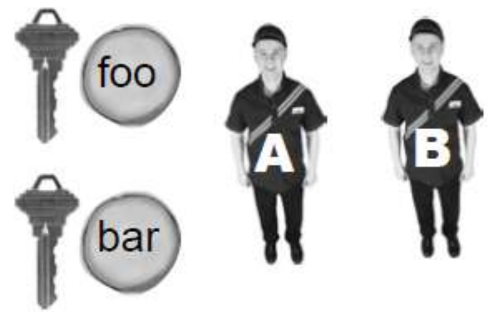
Lost update problem

```
class TestSync implements Runnable {

    private int balance;

    public void run() {
       for(int i = 0; i < 50; i++) {
        increment();
        System.out.println("balance is " + balance);
      }
    }

    public void increment() {
       int i = balance;
       balance = i + 1;
    }
 }


public class TestSyncTest {
   public static void main (String[] args) {
     TestSync job = new TestSync();
     Thread a = new Thread(job);
     Thread b = new Thread(job);
     a.start();
     b.start();
   }
}
```

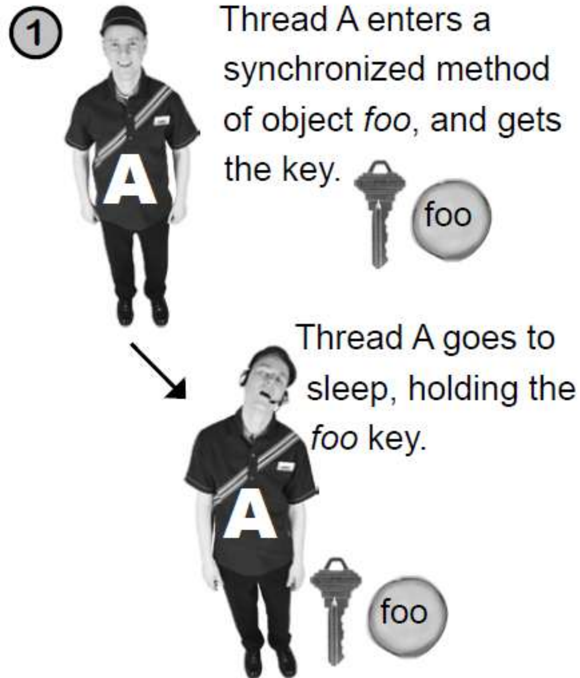
each thread runs 50 times, incrementing the balance on each iteration

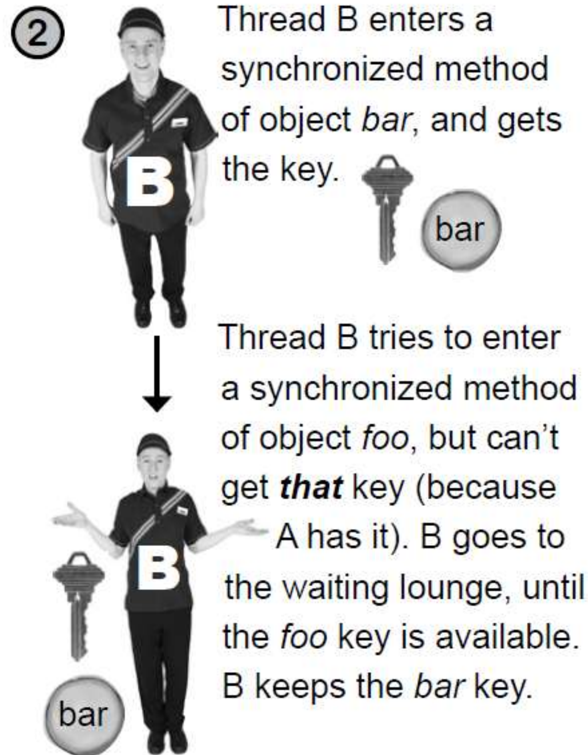
Here's the crucial part! We increment the balance by adding 1 to whatever the value of balance was AT THE TIME WE READ IT (rather than adding 1 to whatever the CURRENT value is)

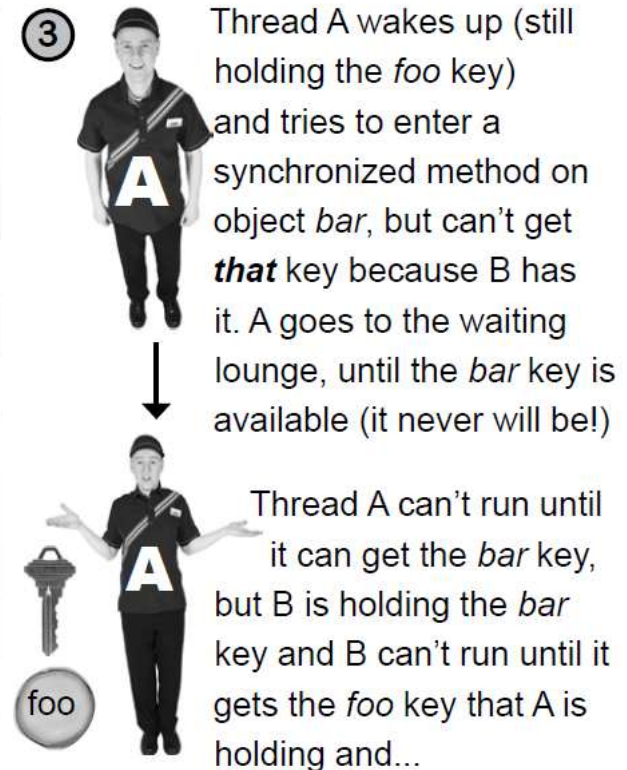
Deadlock



- Deadlock
  - Causes program to stop doing anything useful
  - All you need is two objects and two threads







Programming activity

- Goal: Program that increments/decrements all the integers in an array
  - Create class `NumHolder` that holds array of 100 integers
    - Create `increment()` and `decrement()` methods
      - Methods go through all 100 integers and ++ or – them
    - Create `run()` method
      - Randomly call `increment()` and `decrement()` 1000 times
  - Create main program in `NumHolderLaunch`
    - Create a single `NumHolder` object
    - Create two threads
    - Print out `NumHolder` object
    - Start threads, wait for them to finish
    - Print out `NumHolder` again