Concurrency with Threads and Actors

Adapted from slides by Steve Freund

Concurrency and Race Conditions

```java
int bal = 0;

Thread 1:   Thread 2:
\[
\begin{align*}
   t1 &= bal \\
   bal &= t1 + 10 \\
   t2 &= bal \\
   bal &= t2 - 10 \\
\end{align*}
\]

bal == 0
```

```java
int bal = 0;

Thread 1:   Thread 2:
\[
\begin{align*}
   t1 &= bal \\
   bal &= t1 + 10 \\
   t2 &= bal \\
   bal &= t2 - 10 \\
\end{align*}
\]

bal == -10
```
Concurrency and Race Conditions

Lock m = new Lock();
int bal = 0;

Thread 1
synchronized(m) {
  t1 = bal
  bal = t1 + 10
}

Thread 2
synchronized(m) {
  t2 = bal
  bal = t2 - 10
}

Account Monitor

Concurrent and Race Conditions

Thread 1
Thread 2

Account Monitor

acquire lock of
receiver object

class Account {
  private int balance;

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

  public synchronized String toString() {
    return "balance = " + balance;
  }
}

Producer-Consumer Buffers

• Buffer with finite size
  • Producers add values to it
  • Consumers remove values from it
• Used "everywhere"
  • buffer messages on network, OS events, events in simulation,
    messages between threads...

Java Buffer

public class Buffer<T> {

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

  static
  private static int size = 10;

  public Buffer() {
    elementData = (T[]) new Object[size];
    elementCount = 0;
  }

  public synchronized void add(T t) {
    elementData[putIndex(0)] = t;
    elementCount++;
  }

  public synchronized T remove() {
    return (T) elementData[getIndex(0)];
  }

  private int getIndex(int i) {
    return (start + i) & (size - 1);
  }

  private int putIndex(int i) {
    return (start + i + 1) & (size - 1);
  }
}

Network

• Buffer with finite size
  • Producers add values to it
  • Consumers remove values from it
• Used "everywhere"
  • buffer messages on network, OS events, events in simulation,
    messages between threads...

Web Server

database

Thread 1
Thread 2

Network
Java Buffer

```java
public class Buffer<T> {
    private T[] elementData;
    private int elementCount;
    private int start;
    private int end;

    public void insert(String str) {
        // Code to insert string into Buffer
    }

    public String delete() {
        // Code to delete element from Buffer
        return str;
    }
}
```
Consumers

```java
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);
        }
    }
}
```

Producers

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

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

    public void run() {
        while (moreData()) {
            char c = next();
            buffer.insert(c);
        }
    }
}
```

Using Buffers

```java
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();
    }
}
```

Unsafe Buffer Ops

```java
public class Buffer<T> {
    private T[] elementData;
    private int elementCount;
    private int start;
    private int end;

    public void insert(T t) {
        end = (end + 1) % elementData.length;
        elementData[end] = t;
        elementCount++;
    }

    public T delete() {
        T elem = elementData[start];
        start = (start + 1) % elementData.length;
        elementCount--;
        return elem;
    }
    ...
```
Safe Buffer Ops

```java
public class Buffer<T> {
    private T[] elementData;
    private int elementCount;
    private int start;
    private int end;

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

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

Consumers With Handler

```java
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
        }
    }
}
```

Accounts again

```java
class Account {
    int balance;

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

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

Deadlock

```java
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);
    }
```
Message-passing, actors, "shared nothing"

Communication via explicitly sending/receiving messages.

Threads/processes/actors have their own private state.

Simple Actor

```scala
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

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

Matching Messages

```scala
abstract class Message {
}
case class Hello() extends Message {
  def act() = {
    loop {
      react {
        case Hello => println("Hello to you too");
      }
    }
    start();
  }
}
case class Num(n: Int) extends Message {
  def act() = {
    loop {
      react {
        case Hello => println("Hello to you too");
        case Num(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();

  }
}

Back and forth

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();

  }
}