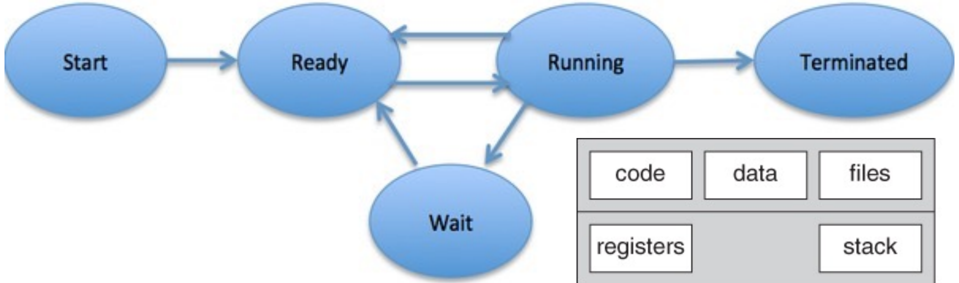
CSE201: Advanced Programming

Lecture 18: Thread Creation

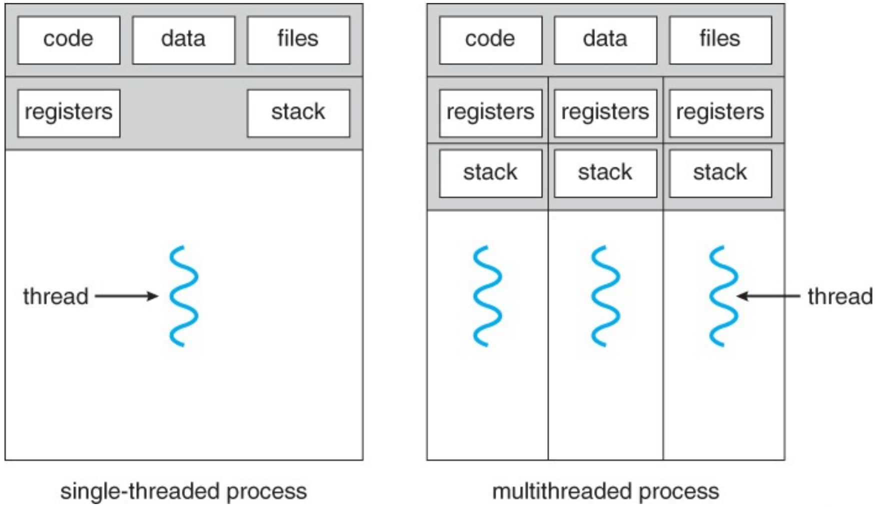
Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

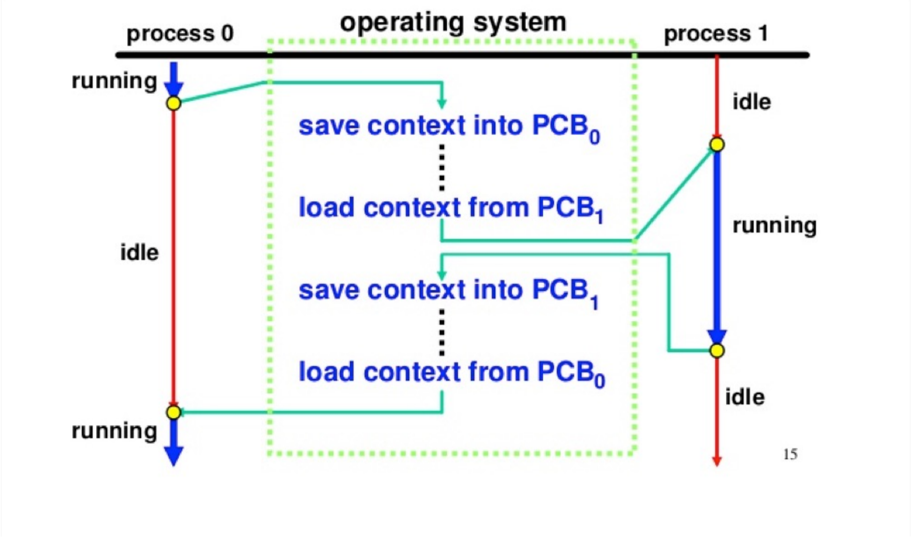
© Vivek Kumar

Last Lecture



- Processes
  - Program in execution
  - Heavy weight
- Threads
  - A lightweight process
  - Share resources inside the parent process
    - Code
    - Global variables
    - File
- Advantages of multithreading
  - Responsiveness
    - Even if part of program is blocked or performing lengthy operation, multithreading allows the program to continue
  - Economical resource sharing
    - Threads share memory and resources of their parent process which allows multiple tasks to be performed simultaneously inside the process
  - Utilization of multicores
    - Easily scale on modern multicore processors





Today’s Lecture

- How to create your own thread in Java

Creating Threads in Java

```
public class MyThread implements java.lang.Runnable {
    ...........
    @Override
    public void run() { ..... }
}
```

```
public class MyThread extends java.lang.Thread {
    ...........
    @Override
    public void run() { ..... }
}
```

- There are two ways to create your own **Thread** object
  - Implementing the **Runnable** interface
  - Subclassing the **Thread** class and instantiating a new object of that class
- In both cases the **run()** method should be implemented

© Vivek Kumar

Sequential Array Sum Implementation

```
public class ArraySum {
    int[] array;
    int sum, low, high;
    public ArraySum(int[] arr, int l, int h) {
        array=arr; sum=0; low=l; high=h;
    }
    //assume array.length%2=0
    public void calculate() {
        for(int i=low; i<high; i++)
            sum += array[i];
    }
    public int getResult() { return sum; }
    public static void main(String[] args)
    {
      int size; int[] array; //allocated (size) & initialized
      ArraySum asum = new ArraySum(array, 0, size);
      asum.calculate();
      int result = asum.getResult();
    }
}
```

- This is a sequential code to find the sum of elements in an array
- Can we use multithreading here?
  - Which part of the code we can parallelize?
  - As the length of array grows huge, the execution time will start increasing

Parallel Array Sum Implementation (1/6)

```java
public class ArraySum implements Runnable {
    int[] array;
    int sum, low, high;
    public ArraySum(int[] arr, int l, int h) {
        array=arr; sum=0; low=l; high=h;
    }
    //assume array.length%2=0
    public void calculate() {
        for(int i=low; i<high; i++)
            sum += array[i];
    }
    public int getResult() { return sum; }
    public static void main(String[] args)
    {
      int size; int[] array; //allocated (size) & initialized
      ArraySum asum = new ArraySum(array, 0, size);
      asum.calculate();
      int result = asum.getResult();
    }
}
```

- Lets parallelize the execution of “calculate” method by implementing `Runnable` interface
  - This method is the performance bottleneck as array length grows huge
- Step-1
  - Implement `java.lang.Runnable` interface

© Vivek Kumar

Parallel Array Sum Implementation (2/6)

```
public class ArraySum implements Runnable {
    int[] array;
    int sum, low, high;
    public ArraySum(int[] arr, int l, int h) {
        array=arr; sum=0; low=l; high=h;
    }
    //assume array.length%2=0
    public void run() {
        for(int i=low; i<high; i++)
            sum += array[i];
    }
    public int getResult() { return sum; }
    public static void main(String[] args)
    {
      int size; int[] array; //allocated (size) & initialized
      ArraySum asum = new ArraySum(array, 0, size);
      asum.calculate();
      int result = asum.getResult();
    }
}
```

- Step-2
  - Implement the method “public void run()”
  - This abstract method is in Runnable interface (no other methods there)
  - For simplicity, we will rename “calculate” method in this example to “run”
    - Note that run() method is of void type
    - In next lecture we will see how to return results (or objects) from Threads

Parallel Array Sum Implementation (3/6)

```
public class ArraySum implements Runnable {
    int[] array;
    int sum, low, high;
    public ArraySum(int[] arr, int l, int h) {
        array=arr; sum=0; low=l; high=h;
    }
    //assume array.length%2=0
    public void run() {
        for(int i=low; i<high; i++)
            sum += array[i];
    }
    public int getResult() { return sum; }
    public static void main(String[] args)
    {
      int size; int[] array; //allocated (size) & initialized
      ArraySum left = new ArraySum(array, 0, size/2);
      ArraySum right = new ArraySum(array, size/2, size);
      Thread t1 = new Thread(left);
      Thread t2 = new Thread(right);


    }
}
```

- Step-3
  - Create two threads (t1 & t2)
  - `java.lang.Thread` class
  - t1 will calculate the sum of left half of the array and t2 will calculate the sum of right half of array
    - Before creating t1 and t2 we must create objects of Runnable type that should be passed to the Thread constructor

Parallel Array Sum Implementation (4/6)

```
public class ArraySum implements Runnable {
    int[] array;
    int sum, low, high;
    public ArraySum(int[] arr, int l, int h) {
        array=arr; sum=0; low=l; high=h;
    }
    //assume array.length%2=0
    public void run() {
        for(int i=low; i<high; i++)
            sum += array[i];
    }
    public int getResult() { return sum; }
    public static void main(String[] args)
    {
      int size; int[] array; //allocated (size) & initialized
      ArraySum left = new ArraySum(array, 0, size/2);
      ArraySum right = new ArraySum(array, size/2, size);
      Thread t1 = new Thread(left);
      Thread t2 = new Thread(right);
      t1.start(); t2.start();

    }
}
```

- Step-4
  - Start both the threads by calling the start() method in Thread class
  - JVM now allows this thread to being its execution
  - JVM calls the run() method of this thread
    - Thread class also implements Runnable interface but has empty bodied run()
    - When a Thread is created using a Runnable object (as in this example), then run() implementation of that Runnable object is called

Parallel Array Sum Implementation (5/6)

```
public class ArraySum implements Runnable {
    int[] array;
    int sum, low, high;
    public ArraySum(int[] arr, int l, int h) {
        array=arr; sum=0; low=l; high=h;
    }
    //assume array.length%2=0
    public void run() {
        for(int i=low; i<high; i++)
            sum += array[i];
    }
    public int getResult() { return sum; }
    public static void main(String[] args)
                          throws InterruptedException {
      int size; int[] array; //allocated (size) & initialized
      ArraySum left = new ArraySum(array, 0, size/2);
      ArraySum right = new ArraySum(array, size/2, size);
      Thread t1 = new Thread(left);
      Thread t2 = new Thread(right);
      t1.start(); t2.start();
      t1.join(); t2.join();

    }
}
```

- Step-5
  - Wait for both the threads to complete their execution (i.e. wait for them to finish execution of run method)
    - join() method from Thread class is used for this purpose
    - join() method throws checked exception and hence main() must declare that

Parallel Array Sum Implementation (6/6)

```
public class ArraySum implements Runnable {
    int[] array;
    int sum, low, high;
    public ArraySum(int[] arr, int l, int h) {
        array=arr; sum=0; low=l; high=h;
    }
    //assume array.length%2=0
    public void run() {
        for(int i=low; i<high; i++)
            sum += array[i];
    }
    public int getResult() { return sum; }
    public static void main(String[] args)
                          throws InterruptedException {
      int size; int[] array; //allocated (size) & initialized
      ArraySum left = new ArraySum(array, 0, size/2);
      ArraySum right = new ArraySum(array, size/2, size);
      Thread t1 = new Thread(left);
      Thread t2 = new Thread(right);
      t1.start(); t2.start();
      t1.join(); t2.join();
      int result = left.getResult() + right.getResult();
    }
}
```

- Step-6
  - Sum the partial results from each threads to get the final results
- What would happen if you call t1.start() followed by t1.join() and then similarly for thread t2?
  - Although there are two threads, still the program is sequential!
- Can you write this same program with more than two threads?

© Vivek Kumar

Parallel Array Sum By Subclassing Thread

```
public class ArraySum extends Thread {
    int[] array;
    int sum, low, high;
    public ArraySum(int[] arr, int l, int h) {
        array=arr; sum=0; low=l; high=h;
    }
    //assume array.length%2=0
    @Override
    public void run() {
        for(int i=low; i<high; i++)
            sum += array[i];
    }
    public int getResult() { return sum; }
    public static void main(String[] args)
                          throws InterruptedException {
      int size; int[] array; //allocated (size) & initialized
      ArraySum t1 = new ArraySum(array, 0, size/2);
      ArraySum t2 = new ArraySum(array, size/2, size);
      t1.start(); t2.start();
      t1.join(); t2.join();
      int result = t1.getResult() + t2.getResult();
    }
}
```

- Only three changes are required
  1. Instead of implementing Runnable, now the ArraySum class will extend Thread class
  2. Override the run() method as Thread class also has empty-body implementation of run()
  3. ArraySum objects are themselves Thread objects and hence now no need to explicitly call constructor of Thread class

© Vivek Kumar

Runnable v/s Subclassing Thread

- **Multiple inheritance is not allowed in Java** hence if our ArraySum class extends Thread then it cannot extend any other class. By implementing Runnable our ArraySum can easily extend any other class
- **Subclassing is used in OOP to add additional feature**, modifying or improving behavior. If no modifications are being made to Thread class then use Runnable interface
- **Thread can only be started once**. Runnable is better as same object could be passed to different threads
- If just run() method has to be provided then **extending Thread class is an overhead for JVM**

© Vivek Kumar

Question: Any Issues Below?

```
.......

class MyClass1 implements Runnable {
    .......
}
class MyClass2 extends Thread {
 .......
}

.......
MyClass1 MyClass1Object = new MyClass1();
Thread t1 = new Thread(MyClass1Object);
t1.run()

MyClass2 t2 = new MyClass2();
t2.run();
```

- What would happen if we directly call run() method from Runnable or Thread object instead of start() and join()?
  - Neither a compilation or runtime error
  - No thread is created by JVM!
  - Sequential execution
  - Calling start() method is mandatory !

Question: Any Issues Below?

```
.......

class MyClass1 implements Runnable {
    .......
}
class MyClass2 extends Thread {
 .......
}

.......
MyClass1 MyClass1Object = new MyClass1();
Thread t1 = new Thread(MyClass1Object);
t1.start();
t1.start();

MyClass2 t2 = new MyClass2();
t2.start();
t2.start();
```

- start() method cannot be invoked more than once
  - A thread can’t be restarted
  - Exception generated at runtime
    - IllegalThreadStateException
- Although we can create several threads with the same runnable type object
  - Advantage of implementing Runnable over extending Thread

© Vivek Kumar

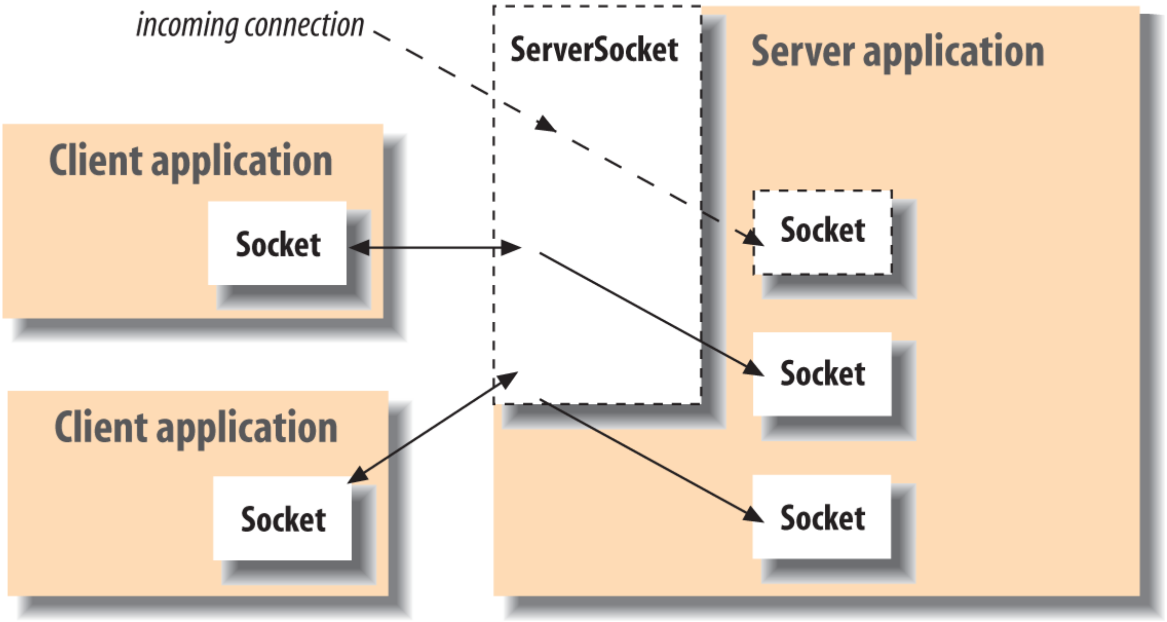
Fibonacci Number Calculation

```
// Sequential Implementation of Fibonacci
public class Fibonacci {
    int result, n;
    public Fibonacci(int n) { this.n = n; }
    public static int fib(int n) {
        if(n<2) return n;
        else return fib(n-1) + fib(n-2);
    }
    public void calculate() {
        result = fib(n);
    }
    public int getResult() { return result; }
    public static void main(String[] args) {
      int n = 40;
      Fibonacci fib = new Fibonacci(n);
      int result = fib.getResult();
    }
}
```

Is this an efficient implementation of parallel Fibonacci ??

```
// Parallel Implementation of Fibonacci
public class Fibonacci implements Runnable {
    int result, n;
    public Fibonacci(int n) { this.n = n; }
    public static int fib(int n) {
        if(n<2) return n;
        else return fib(n-1) + fib(n-2);
    }
    public void run() {
        result = fib(n);
    }
    public int getResult() { return result; }
    public static void main(String[] args)
                              throws InterruptedException {
      int n = 40;
      Fibonacci left = new Fibonacci(n-1);
      Fibonacci right = new Fibonacci(n-2);
      Thread t1 = new Thread(left);
      Thread t2 = new Thread(right);
      t1.start(); t2.start();
      t1.join(); t2.join();
      int result = left.getResult() + right.getResult();
    }
}
```

Multithreading in Socket Programming



- Sockets provide the communication mechanism between two computers that are connected using a network
  - A two-way communication protocol
  - Communication between two processes
- A client program creates a socket on its end of the communication and attempts to connect that socket to a server
- When the connection is made, the server creates a socket object on its end of the communication
- The client and the server can now communicate by writing to and reading from the socket

© Vivek Kumar

Multithreaded Server Application

```
import java.io.*;
import java.net.*;
public class Server  {

  public static void main(String args[ ])
      throws IOException {
    /* create a server socket
       bound to the specified port 1234 */
    ServerSocket me = new ServerSocket(1234);
    /* Server is now listening
       for incoming client’s request */
    while (true) {
      /* Connection is established */
       Socket connection = me.accept();
       System.out.println("Connected");
       /* Spawn a thread for every
          connecting client */
     Thread t=new Thread(new Handler(connection));
       t.start();
    }
  }
}
```

```
class Handler implements Runnable {
  Socket connection;
  Handler(Socket connection) {
    this.connection = connection;
  }
  public void run() throws IOException {
    DataOutputStream out = null;
    try {
     out=new DataOutputStream(connection.getOutputStream());
     out.writeUTF("Hello Client!!");
    } finally {
      out.close();
      connection.close();
    }
  }
}
```

© Vivek Kumar

Client Application

```
import java.io.*;
import java.net.*;

public class Client {

    public static void main(String args[ ])
                              throws IOException {
        String serverName = "localhost"; //or remote IP Address
        int port = 1234; // should be same as used in server
        /* Connect to server that is already listening */
        Socket server = new Socket(serverName, port);
        System.out.println("Just connected to " +
                      server.getRemoteSocketAddress());
        DataInputStream in = new
                      DataInputStream(server.getInputStream());
        System.out.println("Server says " + in.readUTF());
        in.close();
        /* close connection with server */
        server.close();
    }
}
```

- Why our server application was missing the join() for the threads it spawned for every new client connection ?
    - Will the server be able to serve multiple clients in parallel?

© Vivek Kumar

Some Other Methods in Thread

- `static Thread currentThread()`
    - Returns a reference to the currently executing thread object
- `long getId()`
    - Returns the identifier of this thread
- `static void sleep(long millisec)`
    - Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds

© Vivek Kumar

Scheduling Task Launch

- The classes Timer and TimerTask are part of the java.util package
- Useful for
  - performing a task after a specified delay
  - performing a sequence of tasks at constant time intervals

Scheduling Task Launch

- java.util.Timer
  - Delay the execution of a task until the specified time
- java.util.TimerTask
  - Abstract class that implements Runnable
  - Subclass TimerTask (similar to subclassing Thread) and provide a concrete implementation of run() method
- Use Timer instance to schedule this TimerTask

Scheduling Task Launch

```
import java.util.*;
public class Reminder {
    Timer timer;
    public Reminder(int seconds) {
        timer = new Timer();
        timer.schedule(new RemindTask(), seconds*1000);
    }

    class RemindTask extends TimerTask {
        public void run() {
            System.out.println("Time's up!");
            // Terminate the timer thread
            // or set the timer as daemon
            timer.cancel();
        }
    }

    public static void main(String args[]) {
        new Reminder(5);
        System.out.println("Task scheduled.");
    }
}
```

- The schedule method of a timer can get as parameters:
  - Task, time
  - Task, time, period
  - Task, delay
  - Task, delay, period
- A Timer thread can be stopped in the following ways:
  - Apply cancel() on the timer
  - Make the thread a daemon

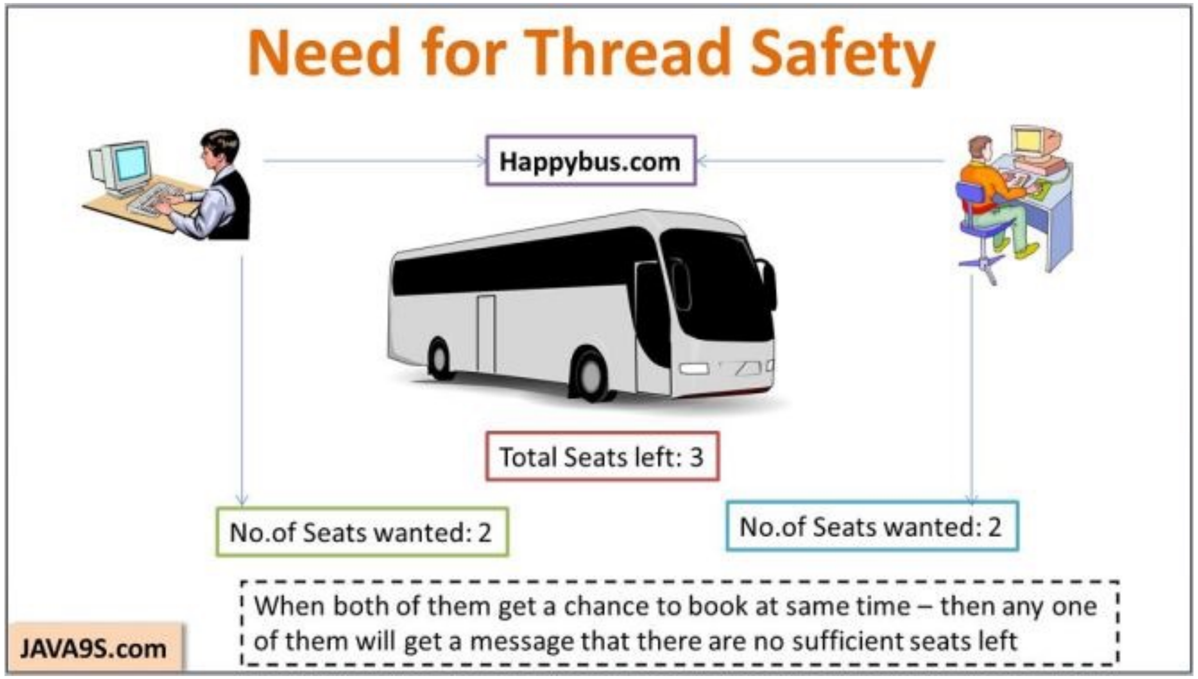
© Vivek Kumar

How Timer is Different Than Sleep

- TimerTask can be canceled anytime
- Easy to create recurring (repeating) task
- Better code readability
- Cannot generate InterruptedException unlike Thread.sleep
- More precise than Thread.sleep

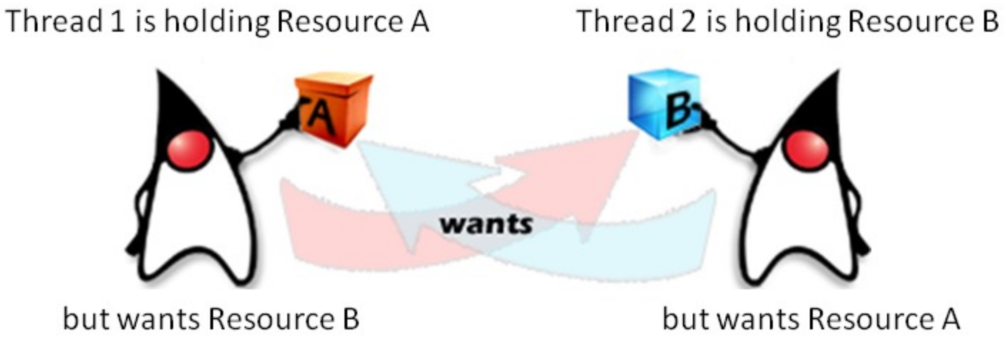
© Vivek Kumar

Disadvantages of Multithreading



Need for Thread Safety

When both of them get a chance to book at same time – then any one of them will get a message that there are no sufficient seats left



Thread 1 is holding Resource A
Thread 2 is holding Resource B
A
B
wants
but wants Resource B
but wants Resource A

© Vivek Kumar

- It is hard to debug and test a multithreaded program
- Sometimes unpredictable results
  - Race conditions
    - Lecture 20
- Chances of deadlock
  - Lecture 20

Next Lecture

- Thread pool in Java
  - java.util.* classes specific to ThreadPool implementation