CSE201: Monsoon 2024
Advanced Programming

Lecture 17: Intro to Multithreading

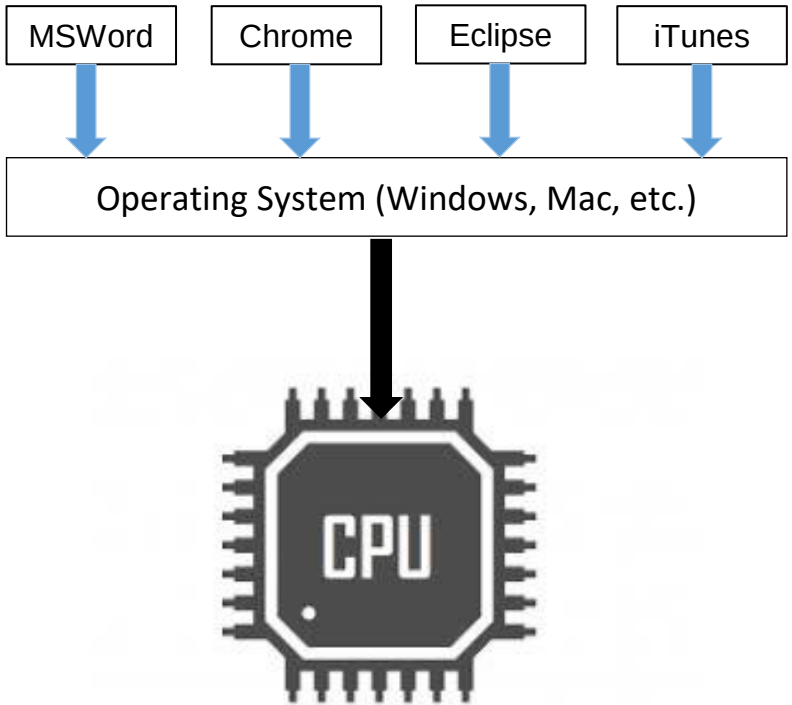
Dr. Arun Balaji Buduru

Founding Head, Usable Security Group (USG)

Associate Professor, Dept. of CSE | HCD

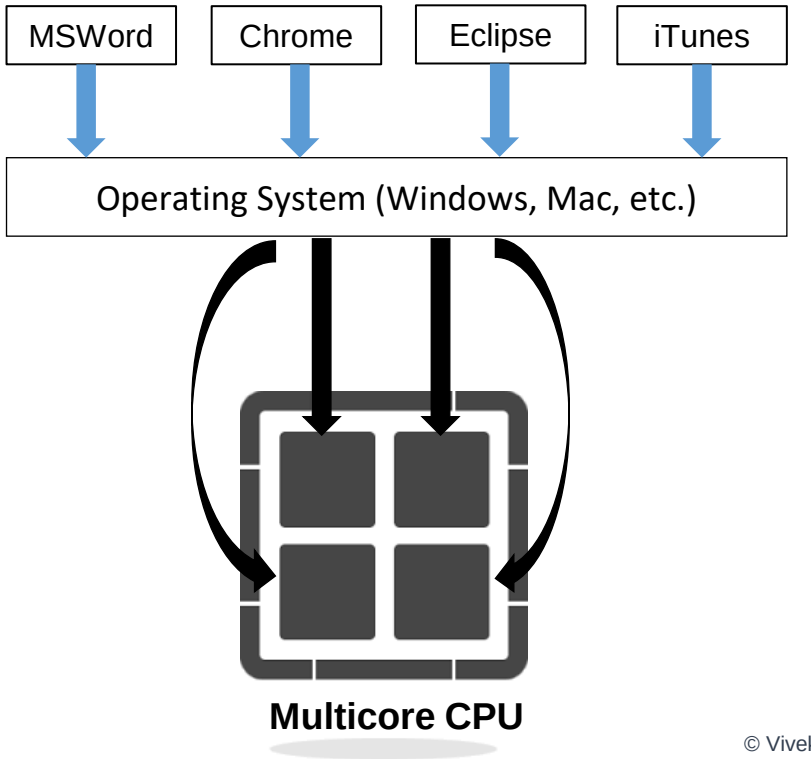
IIIT-Delhi, India

Your Laptop is Multitasking (1/2)



- Multitasking allows many more tasks to be run than there are CPUs
- In the case of a computer with a single CPU, only one task is said to be running at any point in time, meaning that the CPU is actively executing instructions for that task
- Multitasking solves the problem by scheduling which task may be the one running at any given time, and when another waiting task gets a turn

Your Laptop is Multitasking (1/2)

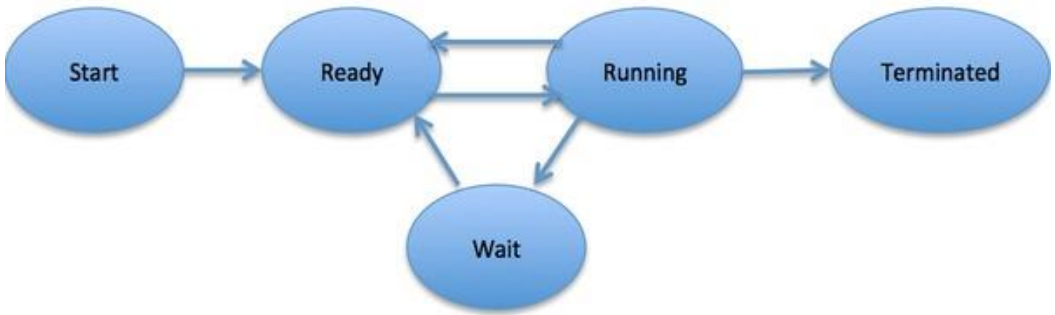


- In case of multicore processor each core will be running one task
- Here too OS will be multitasking by deciding which task runs on which core

Process

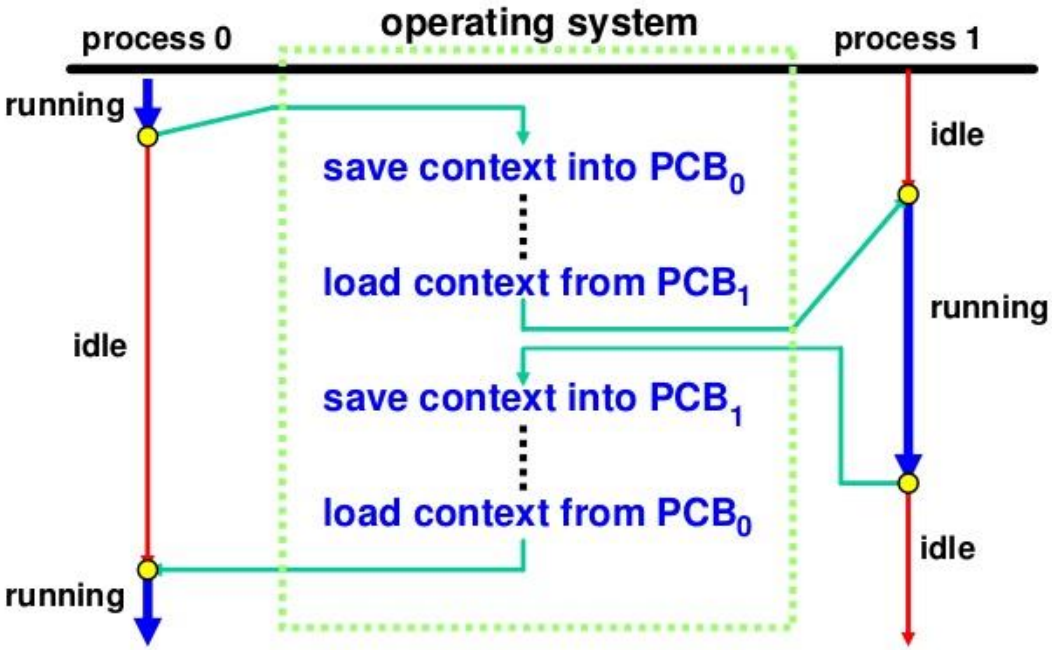
- Process is a program in execution
  - i.e., an icon of M.S. Word on your desktop isn’t a process. It would become a process once you have launched it
- Each process has its own address/memory space
- Separate processes don’t have access to each other’s memory space
  - O.S. can help in creating a communication channel between processes if there be any need

Process Lifecycle



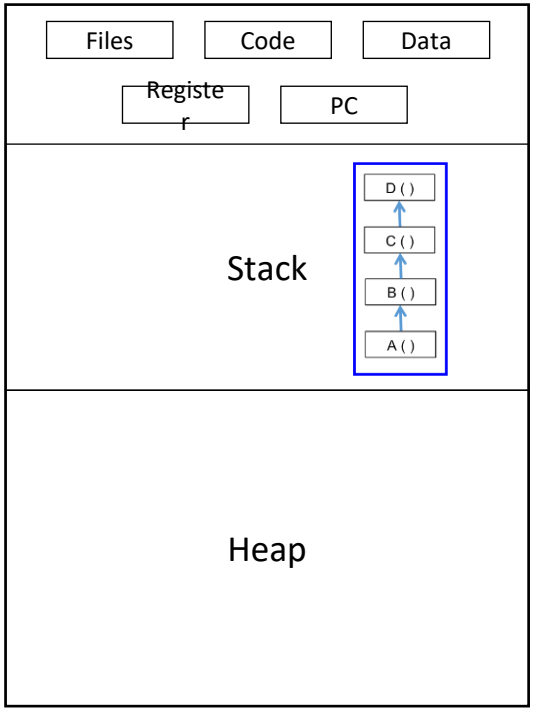
- Diagram on left shows process life-cycle
  - New – process being created
  - Ready – waiting for a free processor
  - Running – instructions are executing
  - Waiting – waiting for some event (I/O, etc.)
  - Terminated – execution is completed

Context Switch



- Switching the CPU to another process requires saving the state of the old process and loading the saved state for the new process
- A Process Control Block (PCB) is a data structure maintained by the Operating System to keep track of processes (e.g., state, id, CPU registers etc.)

Process Structure



- Process contains:
  - Code
    - Program instructions
  - Data
    - Global variables in program
  - Program counter (PC)
    - Address of currently executing program instruction
  - Registers
    - CPU registers
  - Stack
    - Local variables
    - caller-callee relationship between function

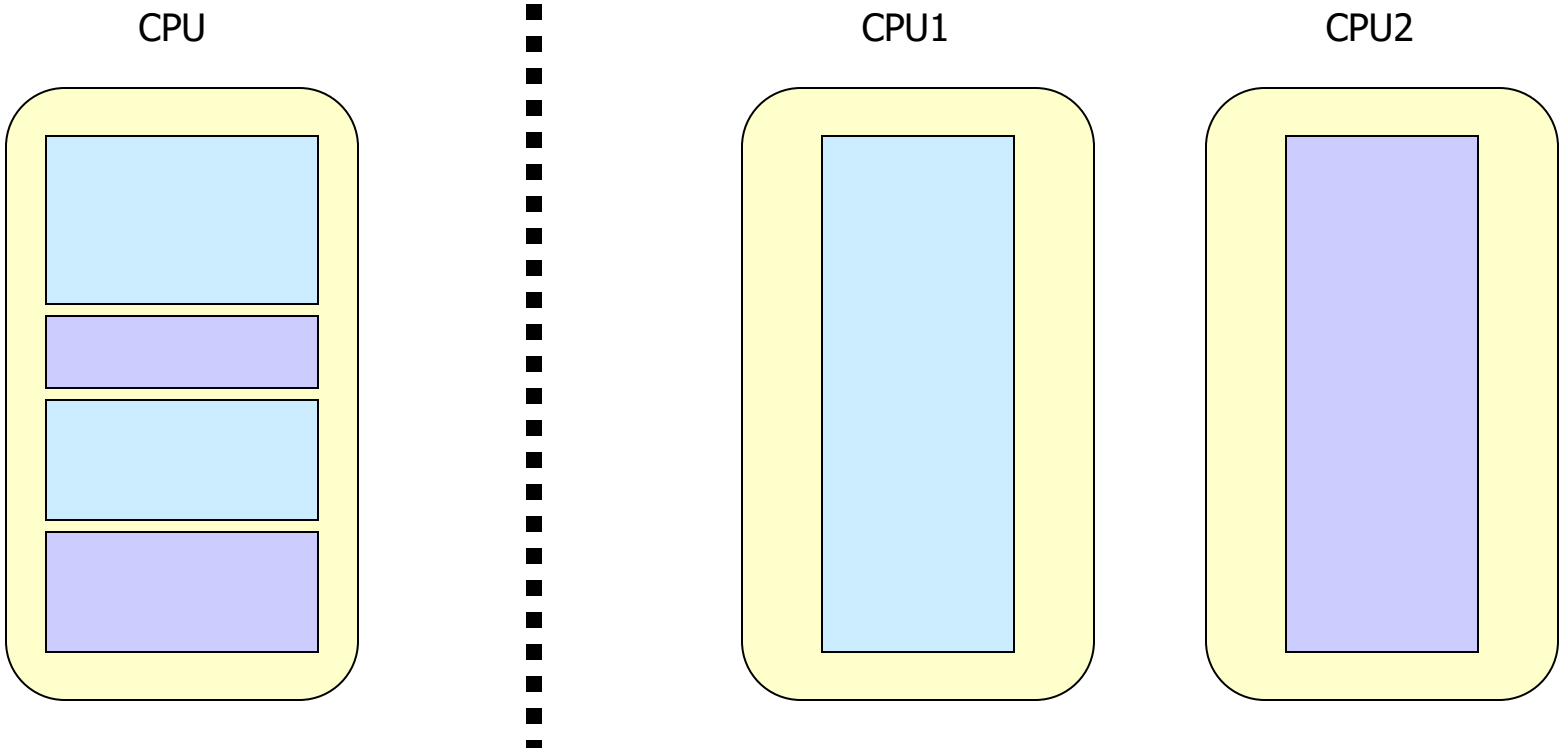
Thread – A Lightweight Process

- Processes are heavyweight
  - Personal address space (allocated memory)
  - Communication across process always requires help from Operating System
- Threads are lightweight
  - Share resources inside the parent process (code, data and files)
    - Easy to communicate across sibling threads!
  - They have their own personal stack (local variables, caller-callee relationship between function)
    - Each thread is assigned a different job in the program
- A process can have one or more threads

Uses of Threads in Java

- Video game application (one process)
  - one thread for graphics
  - one thread for user interaction
  - one thread for networking with peers on internet
- Parallel programming
  - Speeding up the program execution by using multiple threads
  - A program to find the array sum can be made faster by subdividing the array among **`N`** threads (total indices at each thread = **`array.length/N`**). Each thread will find its local sum and later these local sums will be combined to find the total sum
- Producer-consumer type applications
  - News reporter (producer) sends report to new agency and one or more office clerks (consumers) will process each items and upload to Twitter, Facebook, Website, or telecast on TV
- Client-server type applications
  - Server could create a new thread for listening to a new client

Concurrency vs. Parallelism



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

Class Thread in Java

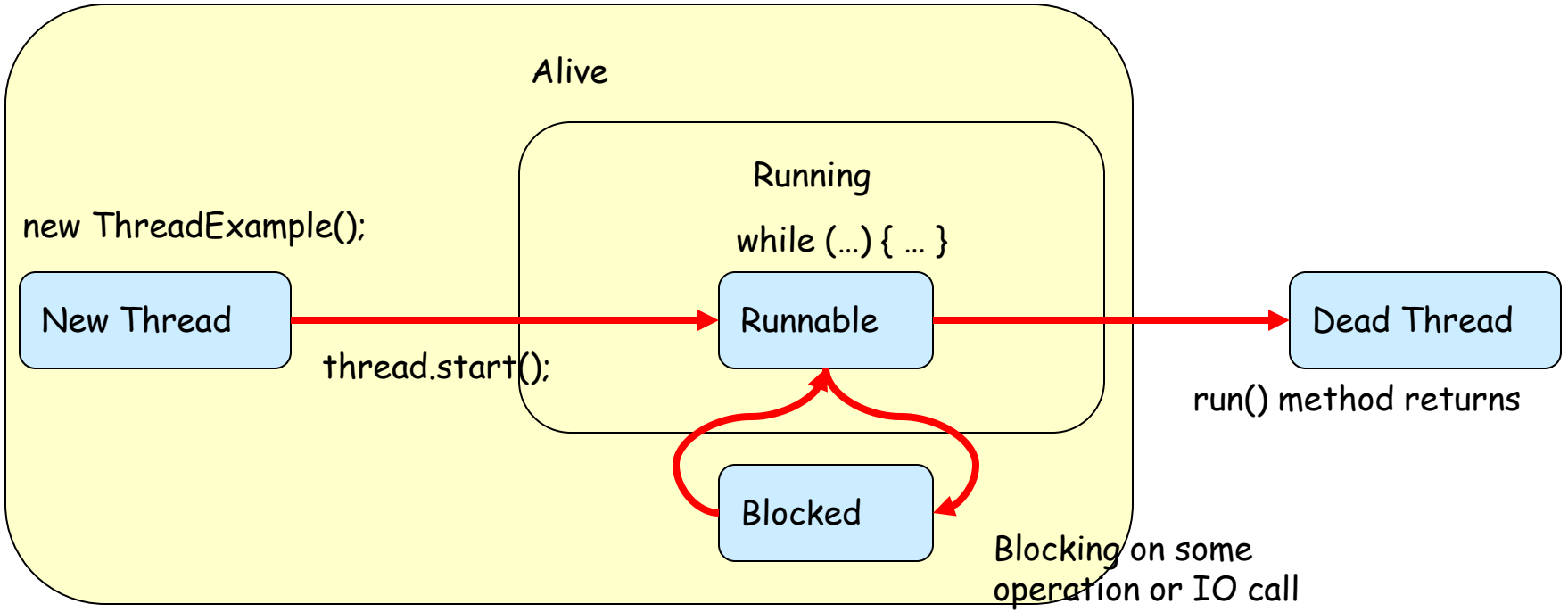
Constructors of Thread class

1. **Thread** ( )
2. **Thread** (*String str*)
3. **Thread** (*Runnable r*)
4. **Thread** (*Runnable r*, *String str*)

You can create new thread, either by extending Thread class or by implementing Runnable interface. Thread class also defines many methods for managing threads. Some of them are,

Method	Description
setName()	to give thread a name
getName()	return thread's name
getPriority()	return thread's priority
isAlive()	checks if thread is still running or not
join()	Wait for a thread to end
run()	Entry point for a thread
sleep()	suspend thread for a specified time
start()	start a thread by calling run() method

Thread State Diagram (Lifecycle)



Thread Priority

- Every thread has a priority (or seniority)
- The highest priority runnable thread is always selected by JVM for execution above lower priority threads
- The priority values range from 1 to 10, in increasing priority
- When a thread is created, it inherits the priority of the thread that created it
- The priority can be adjusted subsequently using the **setPriority()** method
- The priority of a thread may be obtained using **getPriority()**
- Pre-defined priority constants in Thread class:
  - MIN_PRIORITY=1
  - MAX_PRIORITY=10
  - NORM_PRIORITY=5

Java Application Thread

- When we execute an application:
  - The JVM creates a Thread object whose task is defined by the **main()** method
  - It starts the thread with NORM_PRIORITY
  - The thread executes the statements of the program one by one until the method returns and the thread dies

Daemon Threads

- Daemon threads are “background” threads, that provide services to other threads, e.g., the garbage collection thread
- JVM will not exit if non-Daemon threads are executing
- JVM will exit if only Daemon threads are executing
- Daemon threads die when the JVM exits

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

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)

```
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

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?

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

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**

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

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();
    }
}
```

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

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

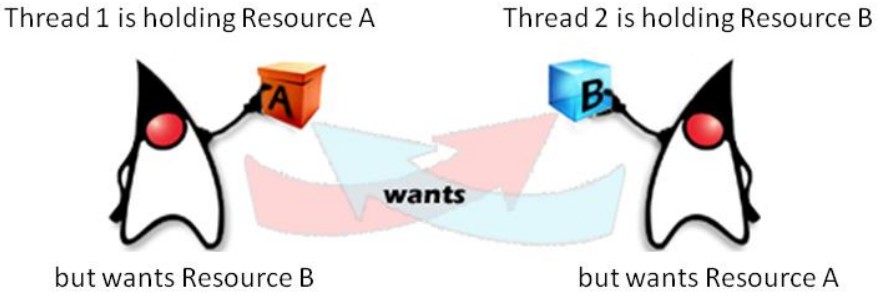
```
import java.util.*;
public class Reminder {
    Timer timer;
    public Reminder(int seconds) {
        timer = new Timer();
        timer.schedule(new RemindTask(), seconds*1000);
    }

    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

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

Disadvantages of Multithreading



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