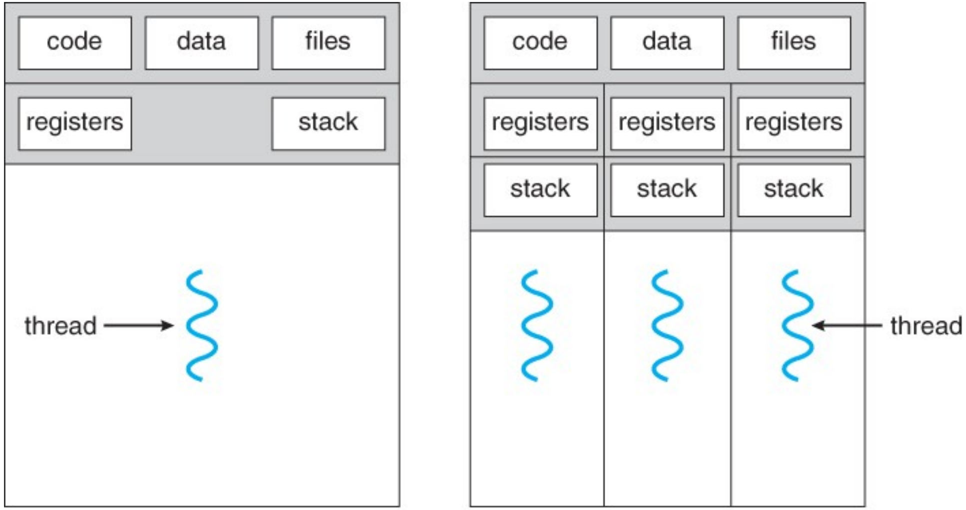
# CSE201: Monsoon 2020
# Advanced Programming

# Lecture 26: Endterm Review-2

Raghava Mutharaju (Section-B)
Vivek Kumar (Section-A)
CSE, IIIT-Delhi
raghava.mutharaju@iiitd.ac.in

# Topic-5: Multithreading

# Processes and Threads



single-threaded process  multithreaded 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

# Creating Threads in Java

- 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

# Parallel Array Sum By Implementing Runnable Interface

```
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();
   }
}
```

- Implement `java.lang.Runnable` interface
- Implement the method “public void run()”
- Create two threads (t1 & t2)
  - 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
- Start both the threads by calling the start() method in Thread class
- Wait for both the threads to complete their execution by calling join() method
- Sum the partial results from each threads to get the final results

# Parallel Array Sum By Subclassing Thread

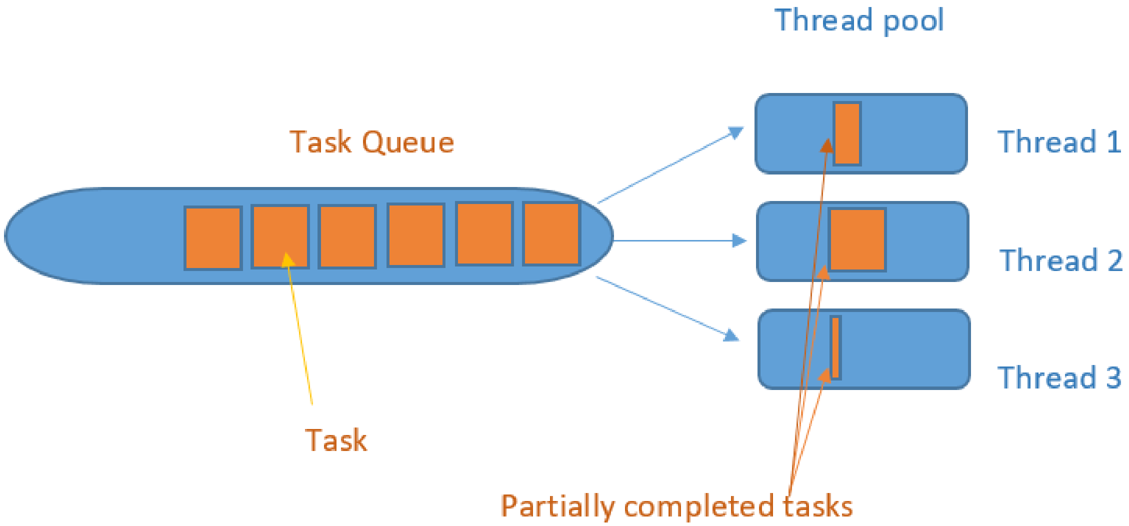
```
public class ArraySum extends Thread {
    int[] array;
    int sum, low, high;
    public ArraySum(int[] arr, int l, int h) {
        array=arr; sum=0; low=l; high=h;
    }
    //assume array.length%2=0
    @Override
    public void run() {
        for(int i=low; i<high; i++)
            sum += 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**

# Introduction to Thread-Pool




- Thread-pool consists of a fixed number of threads
  - Provided by the Java runtime
- User application creates “task” rather than threads
- These tasks are added to a task-pool
- Free threads from thread-pool takes out a task from task-pool and execute it



Fig. Source: http://www.geek-programmer.com/what-are-thread-pools-in-java/

# Parallel Array Sum Using Java ExecutorServices

```
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
   ExecutorService exec = Executors.newFixedThreadPool(2);
     ArraySum left = new ArraySum(array, 0, size/2);
     ArraySum right = new ArraySum(array, size/2, size);
     exec.execute(left); exec.execute(right);
     if(!exec.isTerminated()) {
         exec.shutdown();
         exec.awaitTermination(5L, TimeUnit.SECONDS);
     }
     int result = left.getResult() + right.getResult();
    }
}
```

- An `ExecutorService` is a group of thread objects (thread pool), each running some variant of the following
  - `while (....) { get work and run it; }`
- `ExecutorService` methods:
  - `isTerminated`
    - Returns true if all tasks are terminated following the shutdown
  - `awaitTermination`
    - Blocks until all tasks have completed execution after a shutdown request
- Important that you wait for all tasks to terminate after a shutdown request

# ForkJoinPool for Recursive Divide and Conquer Pattern

```java
import java.util.concurrent.*;

public class Fibonacci extends RecursiveTask<Integer> {
    int n;
    public Fibonacci(int _n) { n=_n; }

    public Integer compute() {
        if(n<2) return n;

        Fibonacci left = new Fibonacci(this.n-1);
        Fibonacci right = new Fibonacci(this.n-2);
        left.fork();
        return right.compute() + left.join();
    }
    public static void main(String[] args) {
        ForkJoinPool pool = new ForkJoinPool(2);
        Fibonacci task = new Fibonacci(40);
        int result = pool.invoke(task);
    }
}
```

- `Fibonacci` class could also extend the class `RecursiveAction`
  - `RecursiveAction` represents a task that doesn’t return any result
- RecursiveTask<T> is better suited in scenarios where there is a need to return results from each task (same return type for all tasks)

# Topic-6: Mutual Exclusion

# Mutual Exclusion

- ***Critical section:*** a block of code that access shared modifiable data or resource that should be operated on by only one thread at a time
- ***Mutual exclusion:*** a property that ensures that a critical section is only executed by a thread at a time.
  - o *Otherwise it results in a race condition!*

Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

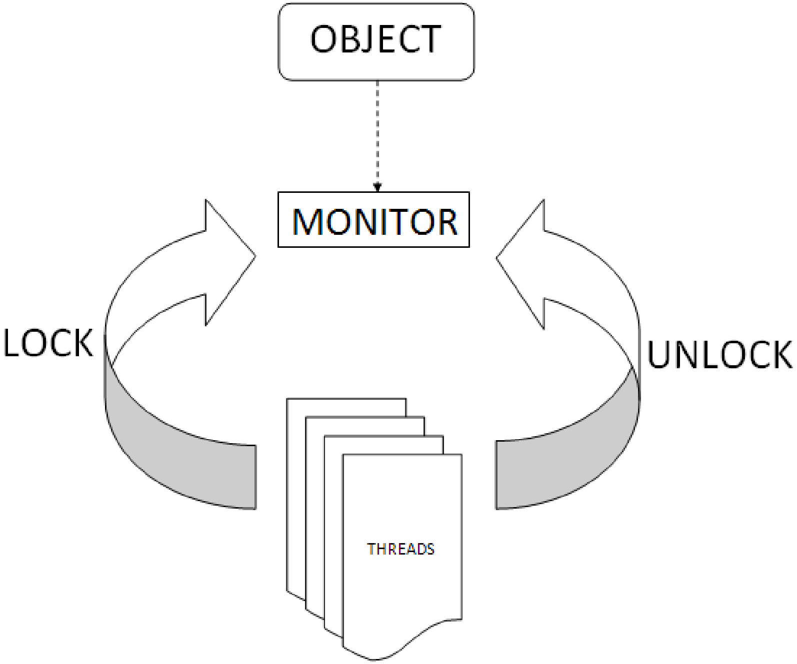
# Implementing Mutual Exclusion

```
class Counter implements Runnable {
    volatile int counter = 0;
    // Both the versions of run method below is correct
    public synchronized void run() { counter++; }
 /* public void run() { synchronized(this) {counter++;} } */
    public static void main(String[] args)
                                 throws InterruptedException
{
ExecutorService exec =
                    Executors.newFixedThreadPool(2);
        Counter task = new Counter();
        for(int i=0; i<1000; i++) {
            exec.execute(task);
        }

        if(!exec.isTerminated()) {
          exec.shutdown();
          exec.awaitTermination(5L,TimeUnit.SECONDS);
        }
        System.out.println(task.counter);
    }
}
```

- **Critical section**
  - The **synchronized** methods (or block) define the critical sections
  - By using **synchronized** keyword we achieved mutual exclusion
- **volatile** keyword for avoiding memory consistency issues
  - For faster data access, memory referenced by a CPU is first copied from main memory (RAM) onto its local cache
  - The updated memory content on cache is not immediately written back to RAM

# Monitors



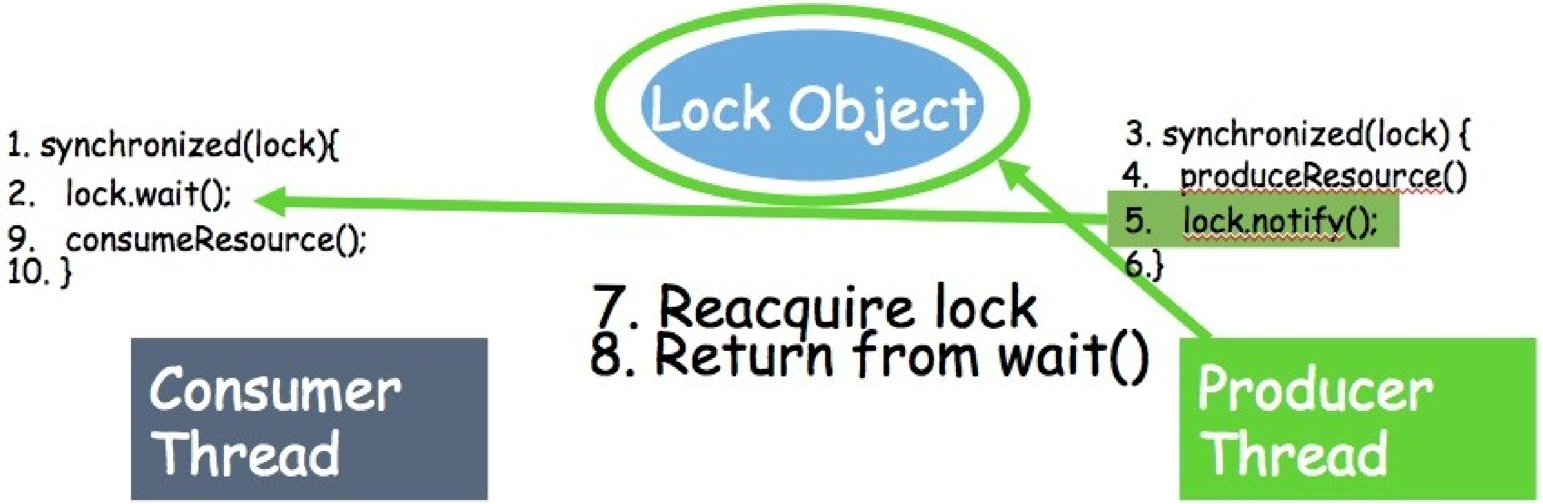

- Each object has a "**monitor**" that is a token used to determine which application **thread** has control of a particular **object** instance
- In execution of a synchronized method (or block), access to the object monitor (lock) must be gained before the execution
- Access to the object monitor is queued
- Demerits
  - Does not guarantee fairness
    - Lock might not be given to the longest waiting thread
  - Might lead to starvation
    - A thread can indefinitely hold the monitor lock for doing some big computation while other threads keep waiting to get this monitor lock
    - Not possible to interrupt the waiting thread
    - Not possible for a thread to decline waiting for the lock if its unavailable

# Demerits of Monitor Lock

- Does not guarantee fairness
  - Lock might not be given to the longest waiting thread
- Might lead to starvation
  - A thread can indefinitely hold the monitor lock for doing some big computation while other threads keep waiting to get this monitor lock
  - Not possible to interrupt the thread who owns the lock
  - Not possible for a thread to decline waiting for the lock if its unavailable

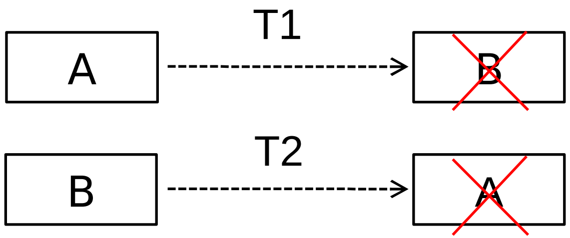
# Producer Consumer Application Using Wait/Notify

- The `wait()` method is part of the class **java.lang.Object**
- It requires a lock on the object's monitor to execute
- It must be called from a synchronized method, or from a synchronized segment of code
- `wait()` causes the current thread to relinquish the CPU and wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object
- Upon call for wait(), the thread releases ownership of this monitor and waits until another thread notifies the waiting threads of the object



# Deadlock Avoidance

```java
class NEFTtransfer {
    Account A, B;
    int amount;
    // prone to deadlock
    void run() {
      synchronized(A) { // A locked
         synchronized(B) { // B locked
            A.debit(amount);
            B.credit(amount);
         } // B unlocked
      } // A unlocked
    }
}
```




- Deadlock occurs when multiple threads need the same set of locks but obtain them in different order
- Deadlock avoidance
  - Lock ordering
    - Ensure that all locks are taken in same order by any thread
      - E.g., in the code on left, first sort both the lock objects (e.g. based on account id of “A” and “B” accounts) and then always lock in a particular order followed by unlock in reverse order

I hope you enjoyed the course..

All the best for your end semester exam and final project deadline!