JAVA Programming

Course Instructor: **Dr. N Nandini Devi**

School of Computer Science

UPES Dehradun

TOPICs to be discussed

- Thread Scheduling
- Problem of Data Races
- Thread Synchronization

Let’s START …!!!

Thread Scheduling

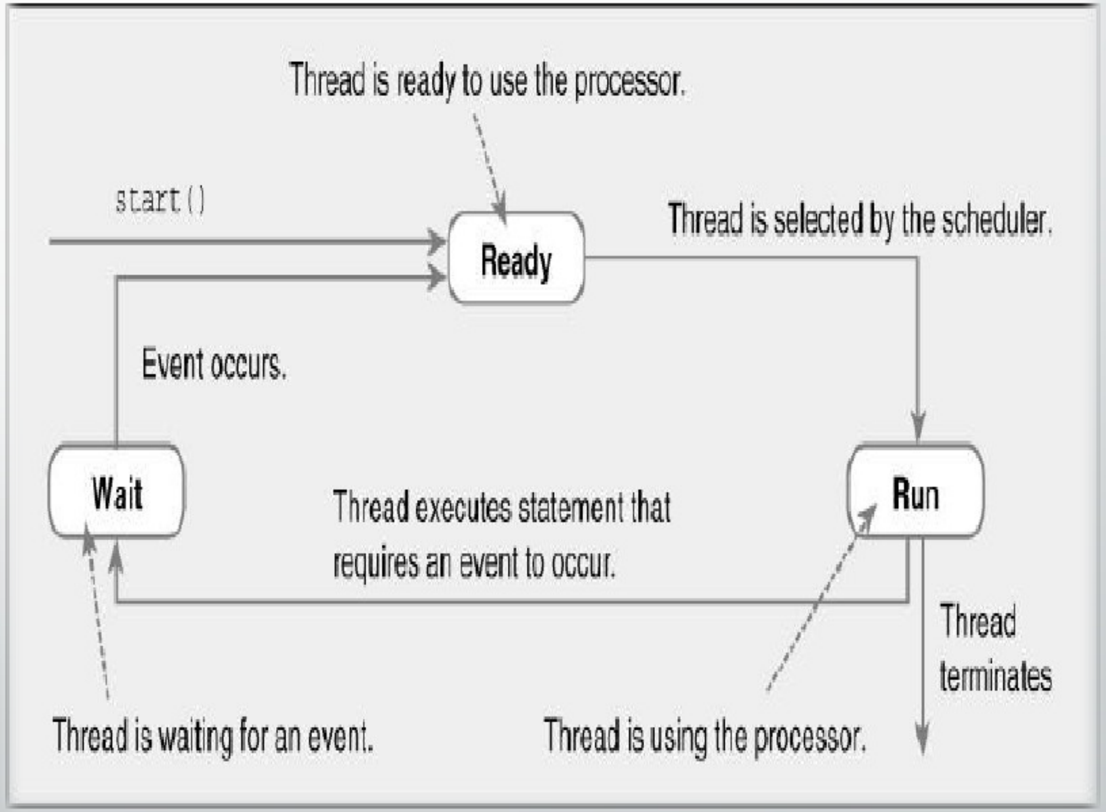
- **Thread Scheduling** is the mechanism used to determine how runnable threads are allocated CPU time.
- There is a **Thread Scheduler** dedicated for this task.

- **Thread Scheduler:**
  - Determines which runnable threads to run.
  - **Thread scheduling** can be based on the **Thread priority**.
  - Part of the OS or the Java Virtual Machine (JVM).

- **Scheduling Policy:**
  - Non-preemptive (Co-operative) Scheduling
  - Preemptive Scheduling

Non-preemptive Scheduling

- Thread continues execution until
  - Thread terminates after finishing its execution
  - Thread executes some instructions causing wait (I/O operation, etc.)
  - Thread invokes methods like **yield()** or **sleep()**, and hands over control of the CPU to any other runnable thread.



Starvation:

A non-preemptive scheduler may cause starvation (runnable threads wait to be executed for a long, sometimes forever). Sometimes, referred to as **Livelock**.

Non-preemptive (Example)

```
class ThreadA extends Thread {
   String name;
   public ThreadA(String name){ this.name = name;  }
   public void run() {
      System.out.println(name +" started.");
      for(int i = 0; i < 2; i++){
         System.out.println(name + " running step " + i);
      }
      System.out.println(name + " completed.");
   }
   public static void main(String[] args)
                                                  throws
InterruptedException {
      ThreadA t1 = new ThreadA("Task 1");
      ThreadA t2 = new ThreadA("Task 2");
      ThreadA t3 = new ThreadA("Task 3");

      t1.start(); t1.join();
      t2.start(); t2.join();
      t3.start(); t3.join();

      System.out.println("All tasks completed sequentially.");
   }
}
```

Output:

Task 1 started.

Task 1 running step 0

Task 1 running step 1

Task 1 completed.

Task 2 started.

Task 2 running step 0

Task 2 running step 1

Task 2 completed.

Task 3 started.

Task 3 running step 0

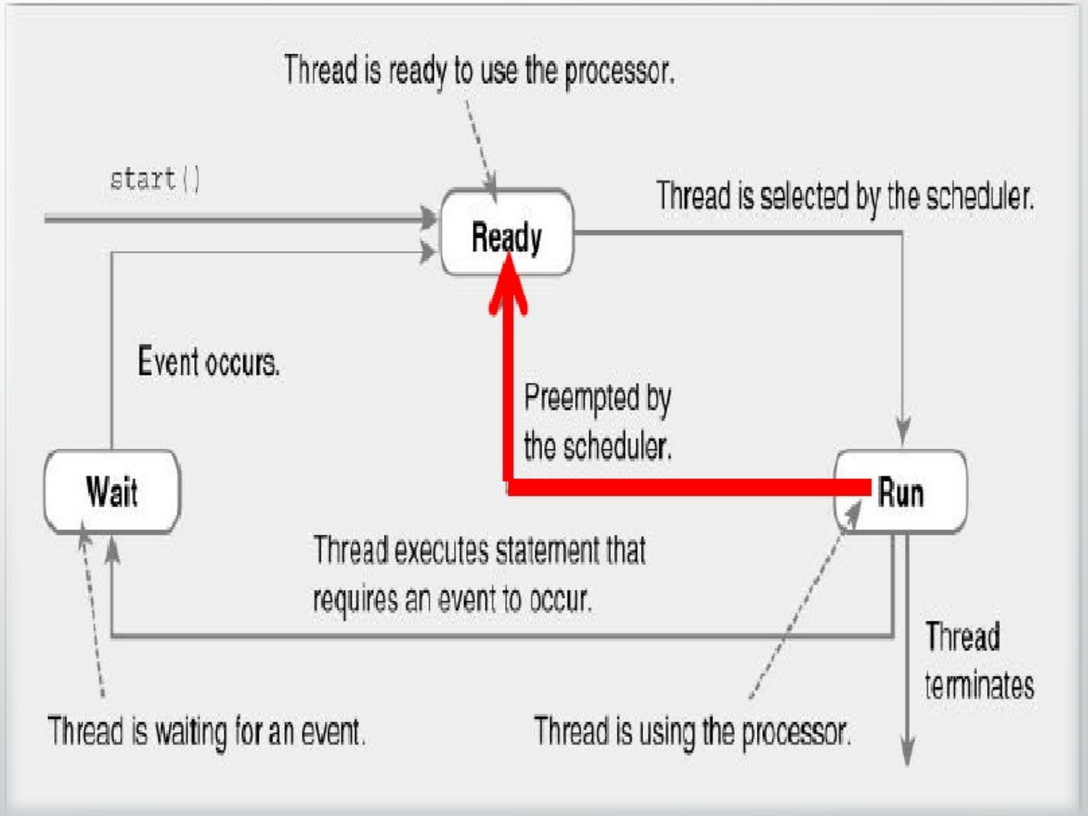
Task 3 running step 1

Task 3 completed.

All tasks completed sequentially.

Preemptive Scheduling

- In this case, the thread can pause for the same reasons as for the non-preemptive case. Also, the **scheduler** can pre-empt (pause) the running thread to allow a different runnable thread to execute.
  - **Time-sliced Scheduling:** The scheduler allocates a time-period that each thread can use the CPU. Once the allotted time is elapsed, the scheduler pre-empts the thread.
  - **Non-time-sliced Scheduling:** The scheduler does not allocate a time-period to the threads but uses other criteria such as priority or I/O status to pre-empt the thread.



Time-sliced (Example)

```
class ThreadB extends Thread {
   String name;
   public ThreadB(String name){ this.name = name;  }
   public void run() {
      System.out.println(name +" started.");
      for(int i = 0; i < 2; i++){
         System.out.println(name + " running step " + i);
      }
      try{
         Thread.sleep(1000);        //A short time-slice
      }catch(InterruptedException e){
         e.printStackTrace();
      }
      System.out.println(name + " completed.");
   }
   public static void main(String[] args) {
      new ThreadB("Task 1").start();
      new ThreadB("Task 2").start();
      new ThreadB("Task 3").start();
   }
}
```

Output:

Task 1 started.

Task 3 started.

Task 2 started.

Task 1 running step 0

Task 2 running step 0

Task 2 running step 1

Task 3 running step 0

Task 3 running step 1

Task 1 running step 1

Task 3 completed.

Task 1 completed.

Task 2 completed.

But can be anything else too (Random)

Non-Time-sliced (Example)

```
class ThreadC extends Thread {
   String name;
   public ThreadC(String name){ this.name = name;  }
   public void run() {
      System.out.println(name +" started.");
      for(int i = 0; i < 2; i++){
         System.out.println(name + " running step " + i);
         Thread.yield();  //Gives other thread chance 
                                                       to 
run
      }
      System.out.println(name + " completed.");
   }
   public static void main(String[] args) {
      new ThreadC("Task 1").start();
      new ThreadC("Task 2").start();
      new ThreadC("Task 3").start();
   }
}
```

Output:

Task 3 started.

Task 1 started.

Task 2 started.

Task 3 running step 0

Task 2 running step 0

Task 2 running step 1

Task 2 completed.

Task 1 running step 0

Task 3 running step 1

Task 1 running step 1

Task 3 completed.

Task 1 completed.

But can be anything else too (Random)

Data Races

```
class DataRace extends Thread {
        static int x;
        public void run(){
                for(int i=0; i<1000; i++){
                        x = x + 1;
                        x = x – 1;
                }
        }
        public static void
main(String[]args) {
                x = 0;
                for(int i=0; i<1000; i++)
                        new
DataRace().start();
                System.out.print(x);
        }
}
```

Output of this code is not always 0 as it should be **(VERIFY …!!!)**

- A **data race** occurs when multiple threads in a program access the same shared data concurrently, and at least one of the threads modifies the data. This leads to unpredictable and incorrect results, as the threads "race" to read and write the shared data without coordination.

- To prevent **data race** in a **multi-threaded environment**, a proper **synchronization** is needed between the threads, so that one thread work on the shared variable at a time.

Thread Synchronization

- **Thread synchronization** is a mechanism that ensures two or more concurrent threads do not execute a particular critical section of code simultaneously, especially when they access shared resources.
- Without **synchronization**, threads may experience **race conditions** leading to incorrect behavior, unpredictable output, or corrupted data.

- **Java** provides several ways to synchronize threads to manage access to shared resources.

  - By using **"synchronized"** keyword
    - synchronized methods
    - synchronized blocks
  - By using **ReentrantLock class**
  - By using **Atomic classes**

Using Synchronized Methods

- A **synchronized method** allows only one thread to execute it at a time on the same instance, ensuring exclusive access.

```
class Table{
  synchronized void printTable(int n){
    for(int i=1;i<=3;i++){
       System.out.print(n*i + "\t");
       try{  Thread.sleep(400);
       }catch(Exception e){ e.printStackTrace(); }

    }
  }
}
class ThreadA extends Thread {
  Table t;
  ThreadA(Table t){  this.t = t;  }
  public void run(){  t.printTable(5);  }
}
class ThreadB extends Thread {
  Table t;
  ThreadB(Table t){  this.t = t;  }
  public void run(){  t.printTable(100);  }
}
```

```
class Main{
  public static void main(String[] args) {
     Table obj = new Table();
     ThreadA t1 = new ThreadA(obj);
     ThreadB t2 = new ThreadB(obj);

     t1.start();
     t2.start();
  }
}
```

Output:

5 10 15 100 200 300

Try without the “synchronized” keyword…!!!

Using Synchronized Blocks

- A **synchronized block** allows you to lock only a specific part of the code instead of the entire method, offering finer-grained control.

```
class Table{
  void printTable(int n){
    for(int i=1;i<=3;i++){
       synchronized(this){
          System.out.print(n*i + "\t");
       }
       try{  Thread.sleep(400);
       }catch(Exception e){ e.printStackTrace(); }

    }
  }
}
class ThreadA extends Thread {
  Table t;
  ThreadA(Table t){  this.t = t;  }
  public void run(){  t.printTable(5);  }
}
class ThreadB extends Thread {
  Table t;
  ThreadB(Table t){  this.t = t;  }
  public void run(){  t.printTable(100);  }
}
```

```
class Main{
  public static void main(String[] args) {
     Table obj = new Table();
     ThreadA t1 = new ThreadA(obj);
     ThreadB t2 = new ThreadB(obj);

     t1.start();
     t2.start();
  }
}
```

Output:

5 100 10 200 15
300

Try without the “synchronized” block…!!!

Using ReentrantLock class

- The ReentrantLock class offers more flexibility than **synchronized**, allowing explicit lock and unlock control.

```
import java.util.concurrent.locks.ReentrantLock;
class Counter {
   private int count = 0;
   private ReentrantLock lock =

         new ReentrantLock();
   public void increment(){
      lock.lock(); //Acquire the lock
      try{
         for(int i=0; i<1000; i++)
            count++;
      }finally{
         lock.unlock(); //Always release the lock
      }
   }
   public int getCount(){
      return count;
   }
}
```

```
class Main{
  public static void main(String[] args)
              throws InterruptedException {
    Counter obj = new Counter();
    Thread t1 = new Thread(obj::increment);
    Thread t2 = new Thread(obj::increment);

    t1.start();  t2.start();
    t2.join();  t2.join();

    System.out.println("Final count: " +

    obj.getCount());
  }
}
```

Output:

Final count: 2000

Try without the loop…!

Using Atomic Classes

- Atomic classes in **Java** provide a way to perform atomic (indivisible) operations on single variables without using explicit synchronization like **synchronized blocks** or **ReentrantLock**.

```
import java.util.concurrent.
                    atomic.AtomicInteger;
class Counter {
   private AtomicInteger count =
                                           new
AtomicInteger(0);
   public void increment(){
       //Atomic operation
       count.incrementAndGet();
   }
   public int getCount(){
       return count.get();
   }
}
```

```
class Main{
  public static void main(String[] args)
             throws InterruptedException {
    Counter obj = new Counter();
    Thread t1 = new Thread(obj::increment);
    Thread t2 = new Thread(obj::increment);

    t1.start();  t2.start();
    t2.join();  t2.join();

    System.out.println("Final count: " +

    obj.getCount());
  }
}
```

Output:

Final count: 2

Summary

Today, we learned about

- Thread Scheduling (Non-Preemptive, Preemptive: time-sliced, non-time-sliced)
- Problem of Data Races
- Thread Synchronization (Synchronized methods, blocks, Reentrant Lock class, and atomic classes)

Thank You!