Threads

Pradeep LN

Topics

- What are threads ?
- Need for Multiple Threads
- Time Scheduling
- Creating multiple threads
- Thread class
- The Runnable interface
- Thread priorities
- sleep() and join()
- Daemon threads
- The problems that comes with parallelism
- What are race conditions?
- Thread synchronization
- Synchronizing critical code
- Synchronized method() Vs Synchronized block

Concurrency

- Many a times a software is capable of doing more than one thing at a time, a software which supports this functionality is called *concurrent software*.
- In concurrent programming, *processes* and *threads* are two basic units of execution.
- A *process* is a self-contained running program with its own address space.
- A *process* generally has a complete, private set of basic run-time resources, in particular each process has its own memory space.

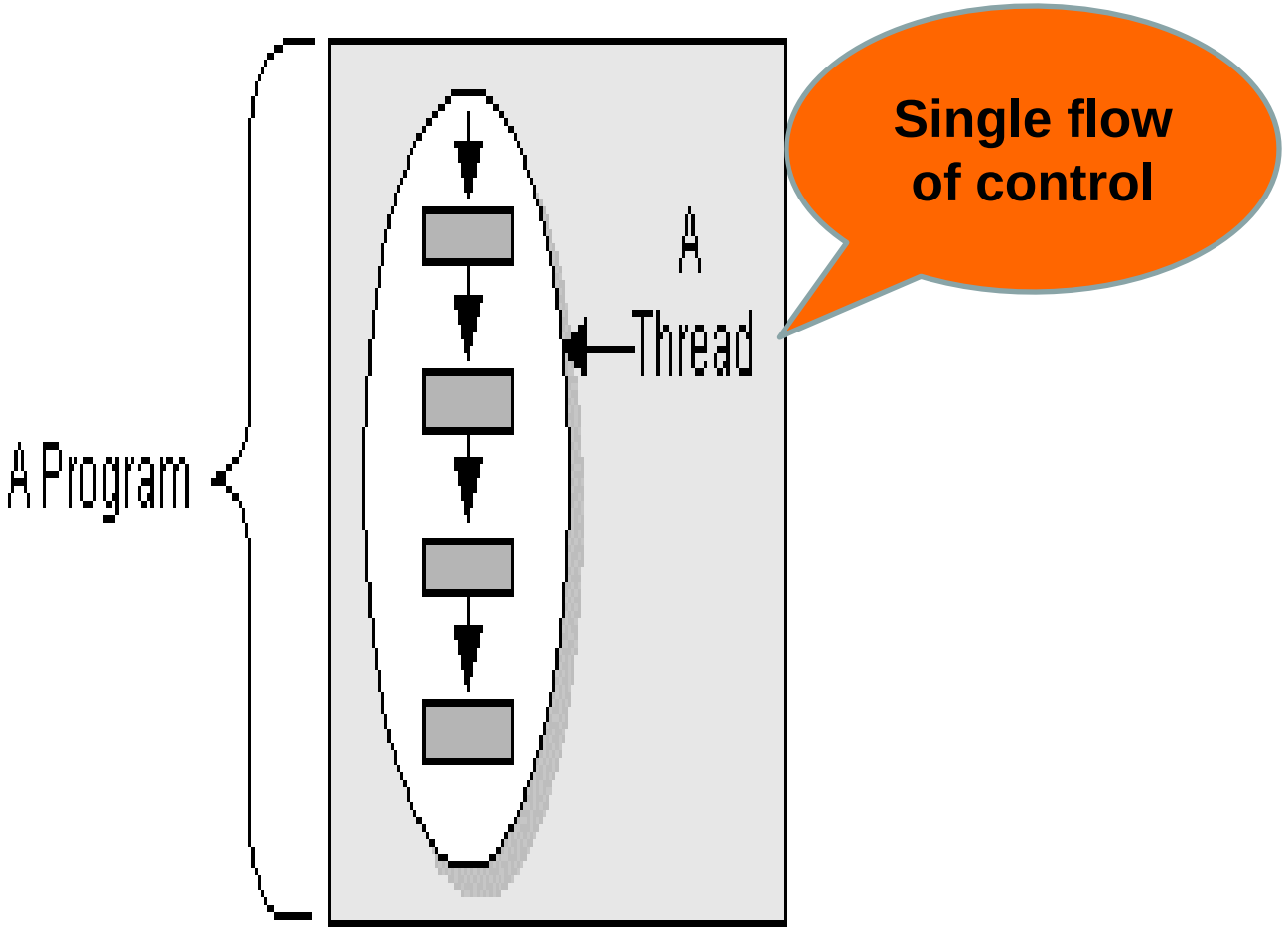
What is a Thread?

- In Java programming language, concurrent programming is mostly supported with *threads*.
- **What is a *thread*?**
  - A thread is a sequential flow of control within a program.
  - A thread itself is not a program, it runs within the context of a program so is considered lightweight.
  - A single thread has a beginning, sequence and an end. At any given time during runtime of the thread, there is single point of execution.
  - The use of multiple threads in a single program means different threads running at the same time and performing different tasks.

Process vs Thread

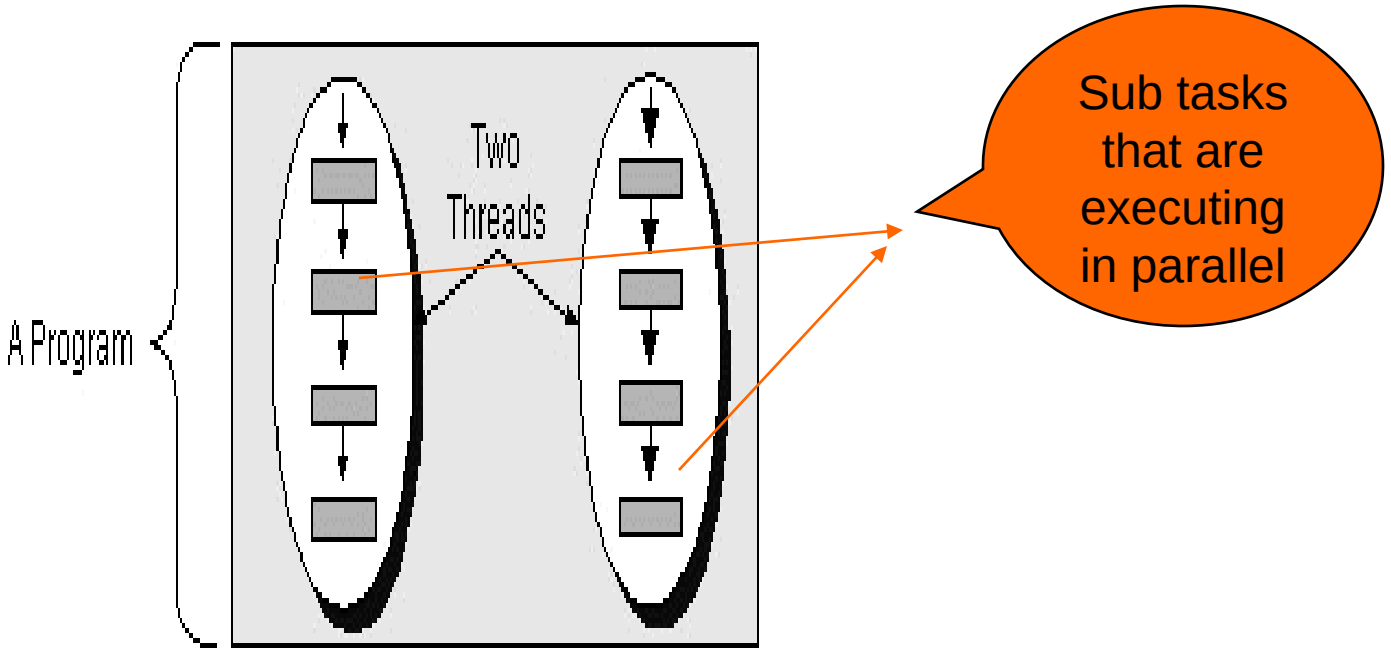
Process	Thread
Synonymous with programs or applications.	Exist within a process.
Heavyweight	Lightweight
Separate address space	Same address space
Interprocess communication is expensive	Interthread communication is less expensive
Context switching is difficult	Context switching is easy

Single Thread



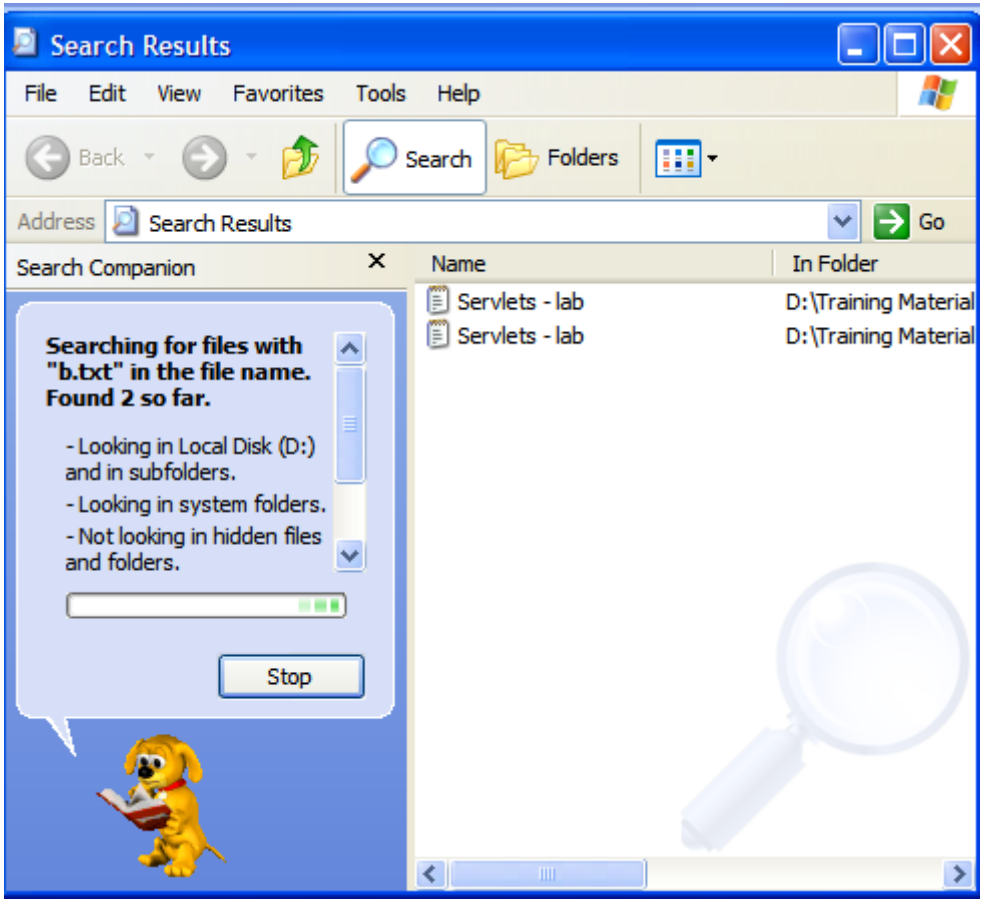
Multiple Threads

- A program having multiple threads implies that there are multiple flows of execution within the program



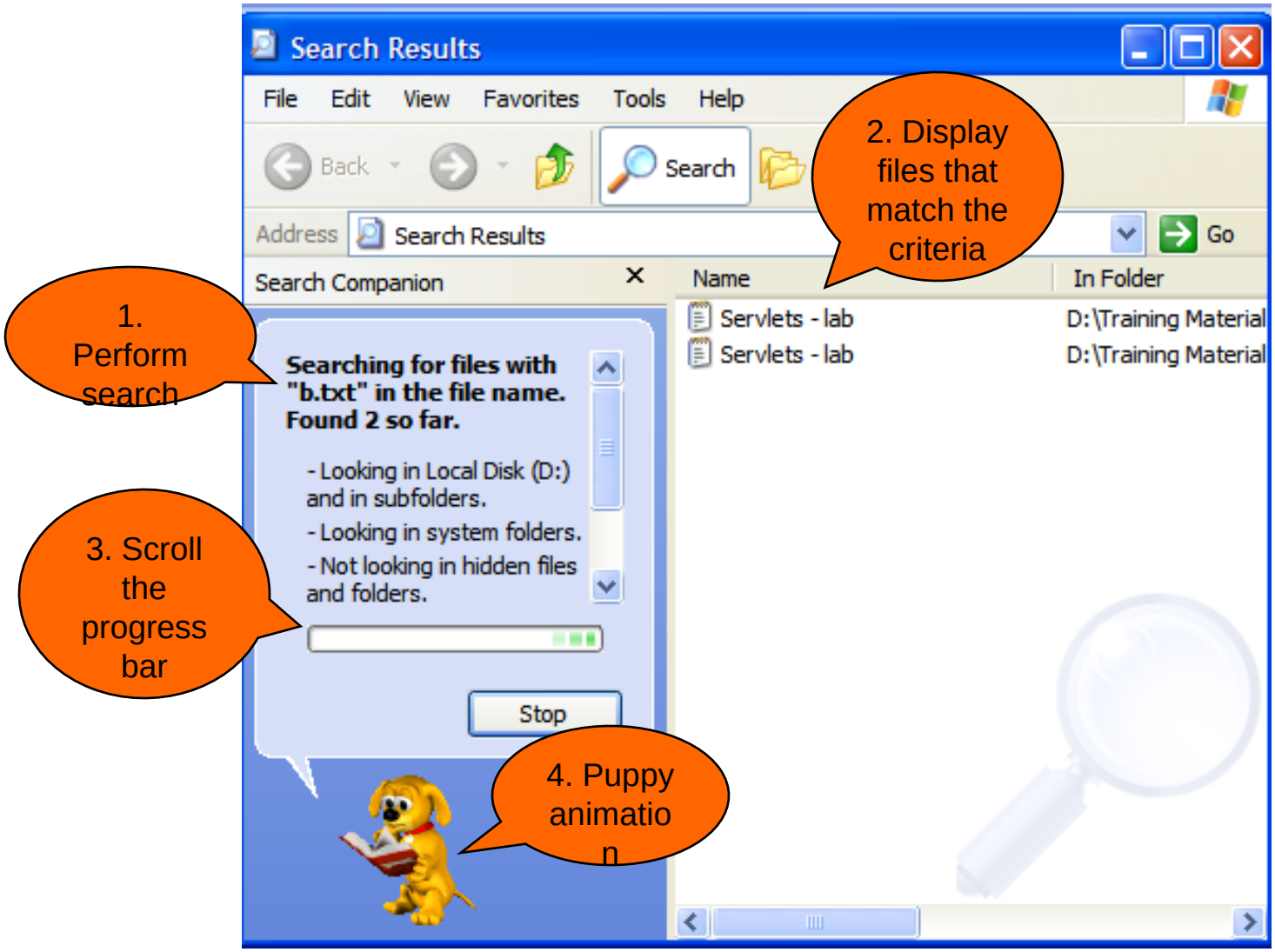
Need for Multiple Threads

- Let us suppose that we have to develop an application that is similar to Windows search



What are the different functionalities that we identify ?

Need for Multiple Threads



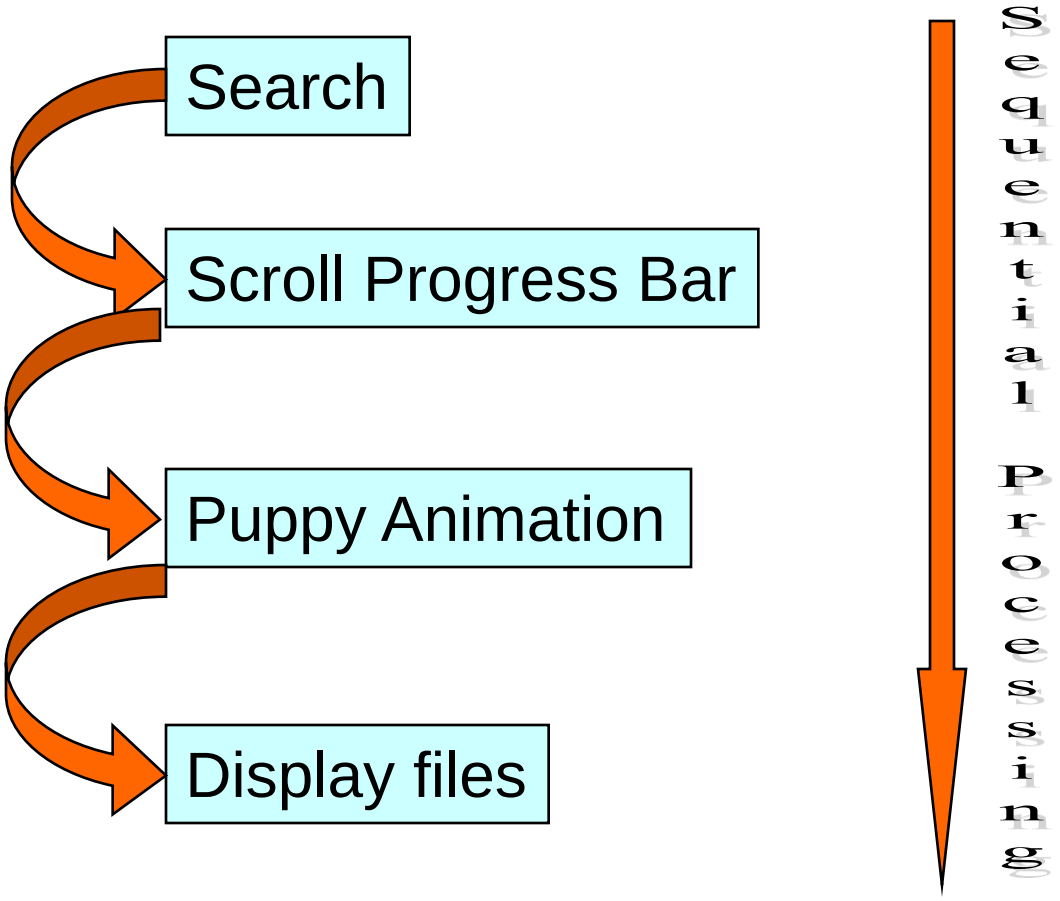
Need for Multiple Threads

- Look at the below code snippet

```
public class WindowsSearch
{
      public void search(String fileName)
      {
              // Implementation
      }
      public void scrollProgressBar()
      {
              // Implementation
      }
      public void animatePuppy()
      {
              // Implementation
      }
      public void displayFiles()
      {
              // Implementation
      }
}
```

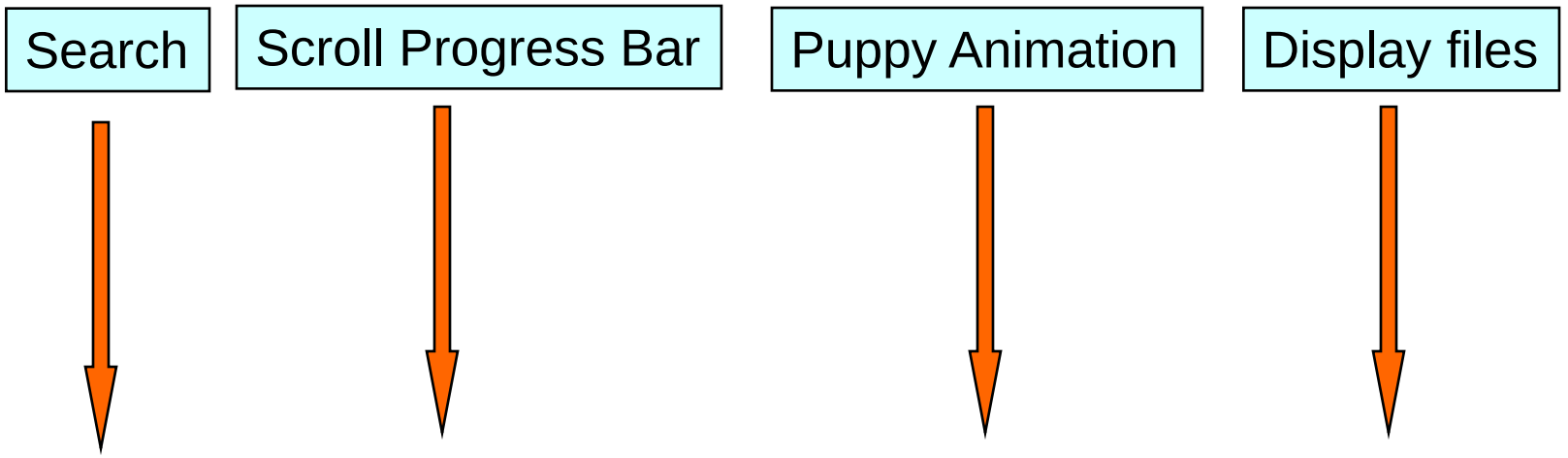
```
class SearchDemo
{
    public static void main()
    {
        WindowsSearch ws =
         new WindowsSearch();

         ws.search(fileName);
         ws.scrollProgressBar();
         ws.animatePuppy();
         ws.displayFiles();
     }
}
```

What is the problem ?



Need for Multiple Threads

- What we desire to achieve is parallel processing



- Every task is processed by one ‘thread’
- All four threads execute ‘almost’ at the same time

Threads and Scheduling

- Each thread is given a small amount of processor time called a quantum or timeslice, with which to perform its task.
- When the thread's quantum expires, the thread's execution is paused.
- The operating system then assigns another thread to the processor.
- In this way, the CPU does a round robin of all threads.

Thread Priorities

- A CPU can execute only one thread at a time
  - The threads that are ready for execution queue up for processor time.
- So the threads are executed depending on their priority, relative to one another.
- A thread's priority is used to decide when to switch from one running thread to the next, which is called *context switching.*
- Higher priority threads get more CPU time

Why use Threads ?

- Threads can be used in numerous scenarios
  - To improve the responsiveness of application.
  - To separate out data processing and input/output operations.
    - For non blocking input/output operations.
  - To handle asynchronous events (event handling such as a mouse click).
  - To perform repetitive or timed tasks (animations).
  - Management of multiple service request with unpredictable arrival.

Threading API

- The Java platform is designed from ground up to support multi threaded programming
- Java support for multi threading is provided by a simple Threading API
- There are two ways in which we can create threads in Java.
  - Creating a subclass of java.lang.***Thread*** class.
  - Implement the java.lang.***Runnable*** interface.

Class Thread

- ***Thread Class***
  - The Thread class itself implements the Runnable interface which defines a single method run().
  - The Thread class run() method does nothing, a class which extends from thread has to override the run() method by providing its own implementation.
  - The Thread class defines a number of methods useful for thread management.
  - The start() method of the Thread class spawns a new thread and automatically calls the run() method in that newly created thread.
- *Constructors*
  - Thread()
  - Thread(Runnable r)
  - Thread(String name)

Creating a Thread

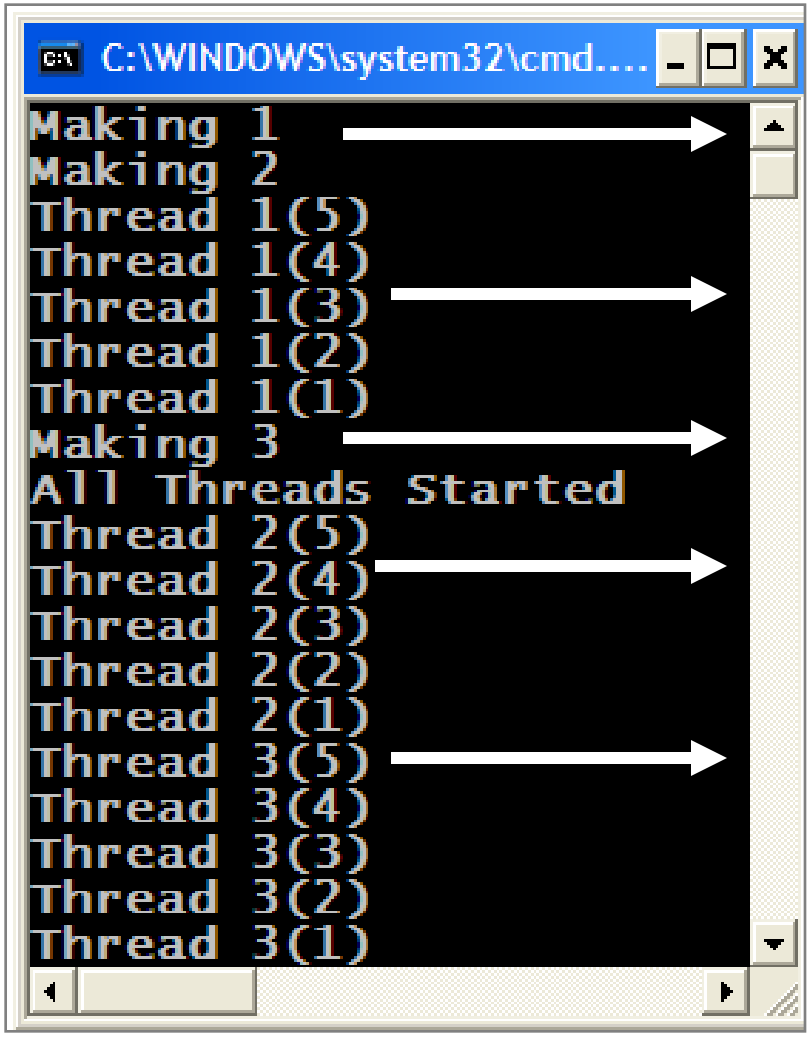
- Steps Involved
  - Define a class that extends Thread.
  - Override the run() method
  - Instantiate the class
  - Spawn the thread by making a call to start() method
  - start() method automatically calls run() and triggers execution of the thread.

Example

```
class SimpleThread extends Thread {
      private int countDown = 5;
      private static int trdCount = 0;
      private int trdNum = ++trdCount;
      SimpleThread() {
            System.out.println("Making
              thread " + trdNum);
      }
      public void run() {
           while(true) {
                System.out.println("Thread” +
                 trdNum + "(“ + countDown+")");
                  if(--countDown == 0) return;
           }
      }
}
```

```
class ThreadDemo
{
     public static void
                main(String[] s)
     {
          for(int i=0;i<=5;i++)
               new SimpleThread().
                          start();
          System.out.println("All
                  Threads Started");
     }
}
```

Executing SimpleThread



Main thread executing…

Thread 1 executing…

Main thread executing…

Thread 2 executing…

Thread 3 executing…

How does it work ?

```
SimpleThread t1 = new SimpleThread();
```

An instance of SimpleThread is created

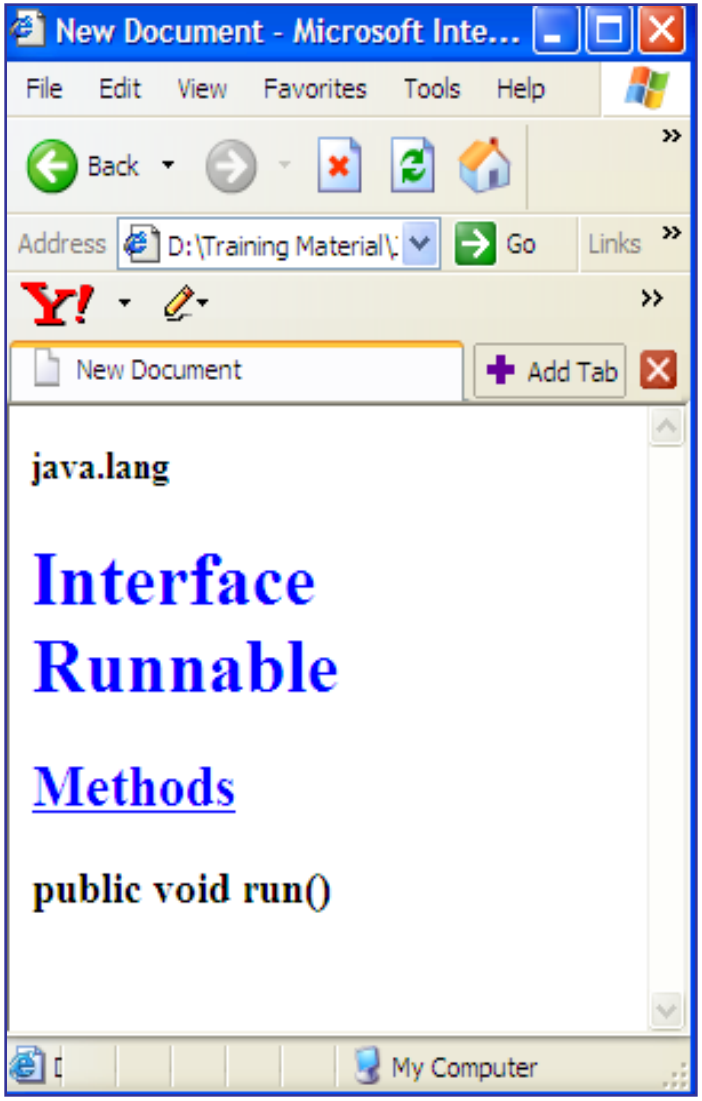
```
t1.start();
```

The JVM interfaces with the OS and an OS level thread is spawned

start() calls run()

```
public void run()
{
    while(true)
    {
        System.out.println("Thread” +
            trdNum + "(“ + countDown+")");
        if(--countDown == 0) return;
    }
}
```

Interface Runnable



- The Runnable interface defines the contract of making an object capable of being run by a thread
- The Runnable interface defines a single method, run()
  - meant to contain the code to be executed in the thread.
- A class that implements this interface has to provide the implementation for run() method
  - An object of this class becomes a ‘runnable’ object

Creating a Thread

- Steps Involved
  - Define a class that implements Runnable interface.
  - Provide implementation for the run() method
  - Instantiate the class
    - The object is now a Runnable object
  - Create a Thread instance and assign the Runnable object to the thread.
  - Spawn the thread by making a call to start() method
  - start() method automatically calls run() and triggers execution.

Example

```
class NewThread implements Runnable
{
    int start,stop;

    NewThread(int start,int stop)
    {
        this.start=start;
        this.stop=stop;
    }
    public void run()
    {
        for(int i=stop;i>start;i--)
            System.out.println(Thread.currentThread() + " : " + i);
        System.out.println("Exiting " + Thread.currentThread());
    }
}
```

Example

```
class RunnableDemo
{
    public static void main(String args[])
    {
        NewThread nt1=new NewThread(5,10);
        NewThread nt2=new NewThread(15,18);
        Thread t1 = new Thread(nt1);
        Thread t2 = new Thread(nt2);
        System.out.println(“Starting Thread 1 “);
        t1.start();
        System.out.println(“Starting Thread 2 “);
        t2.start();
        System.out.println("Main thread exiting");
    }
}
```

How does it work ?

```
NewThread nt1=new NewThread(5,10);
```

Runnable instance is created

```
Thread t1 = new Thread(nt1);
```

Thread object is created and given the runnable reference

```
t1.start();
```

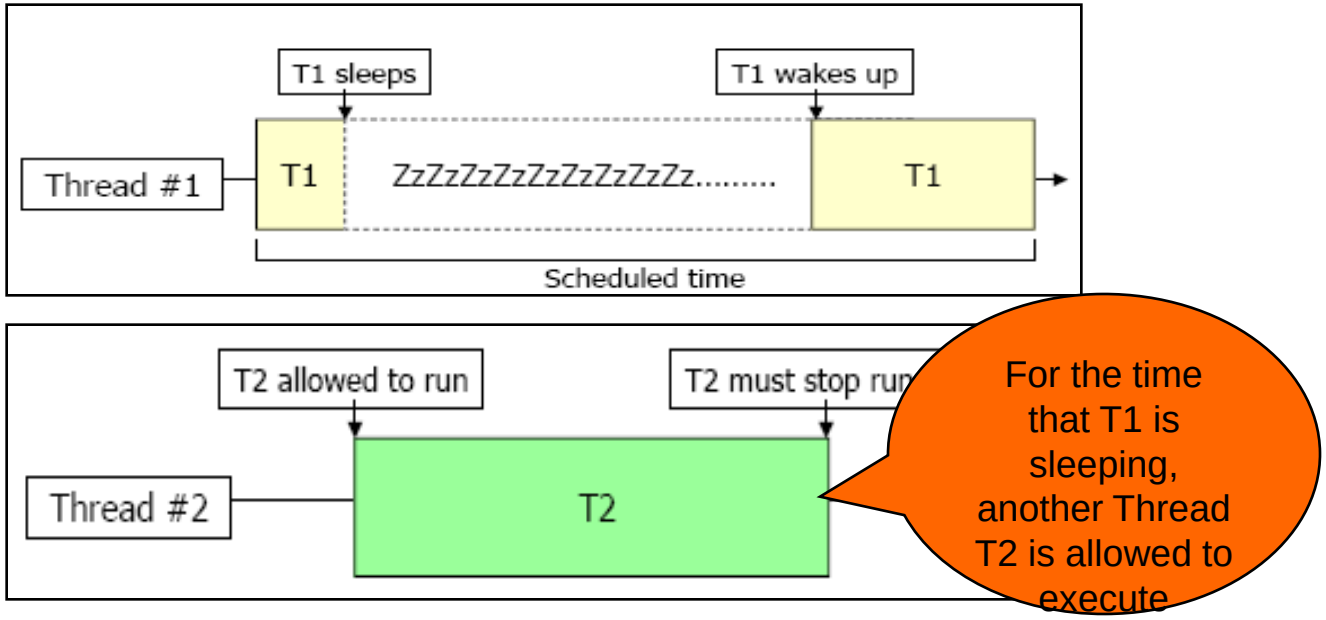
The JVM interfaces with the OS and an OS level thread is spawned

start() calls run()

```
public void run()
{
      for(int i=stop;i>start;i--)
      System.out.println(Thread.currentThread() + " : " + i);
      System.out.println("Exiting " + Thread.currentThread());
}
```

sleep() method

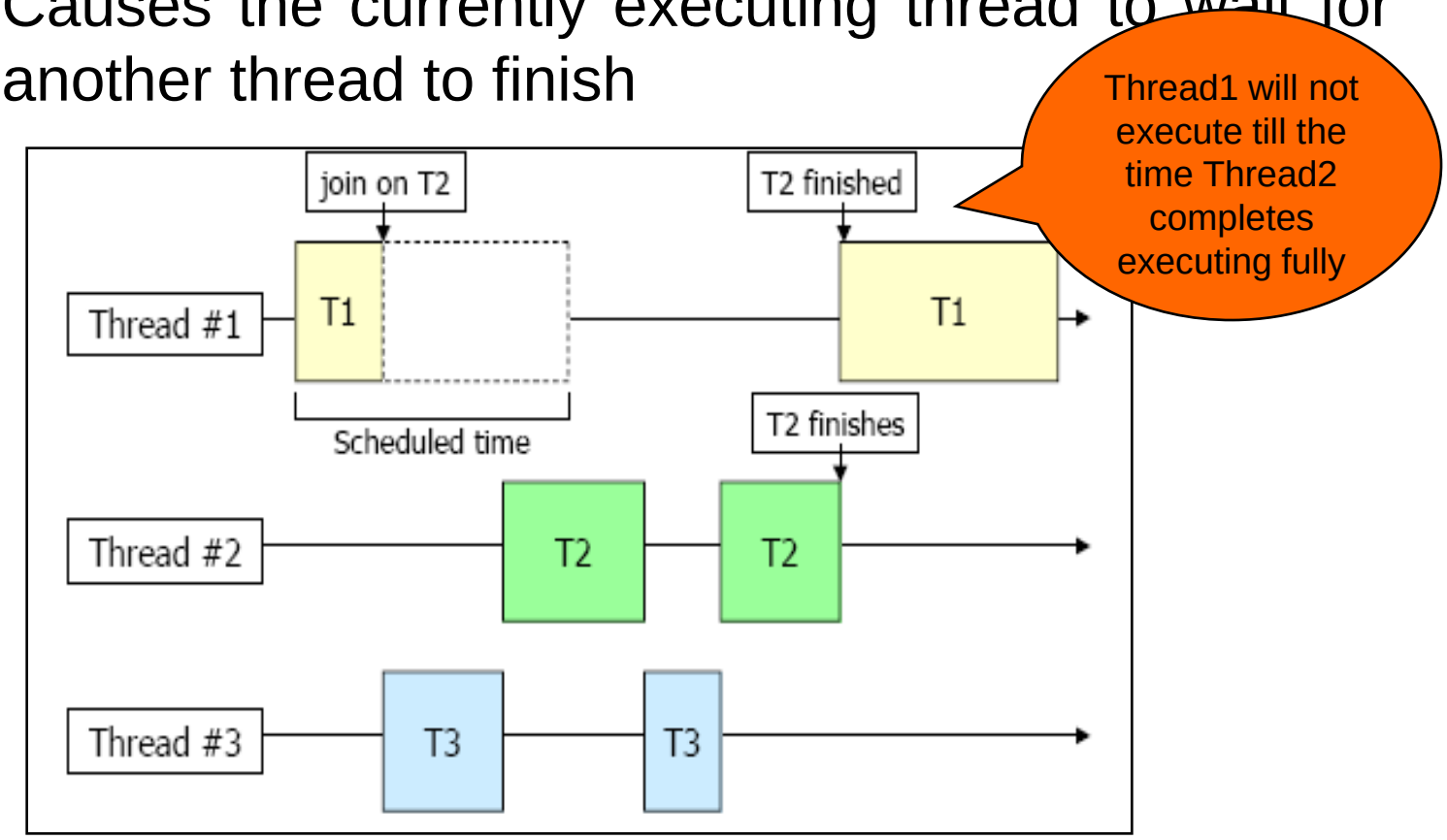
- public static void sleep(long millis)
  - Causes the thread to cease execution for the specified number of milliseconds
  - Throws InterruptedException if interrupted by another thread



See Listing : **SleepDemo.java**

join() method

- public void join()
  - Causes the currently executing thread to wait for another thread to finish



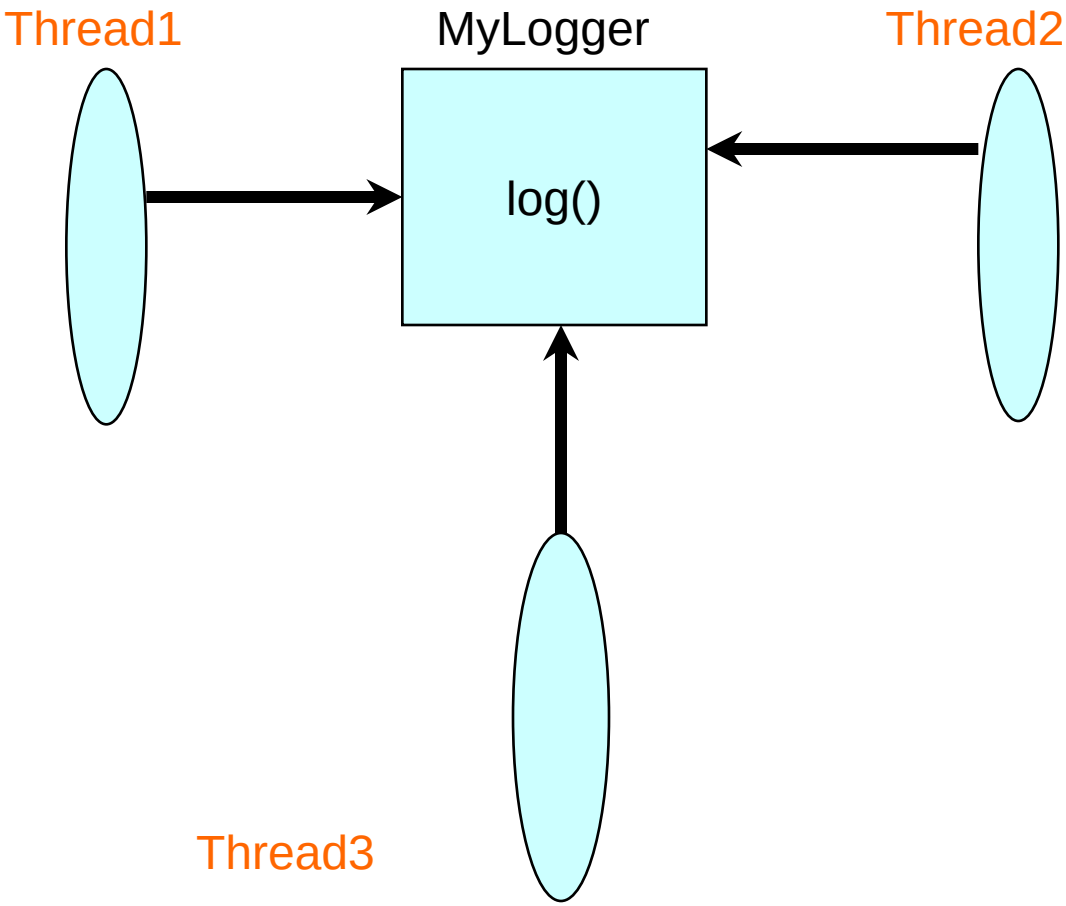
See Listing : **JoinDemo.java**

Other methods

- **boolean isAlive()**
  - Determines if the thread is still running**.**
- **int getPriority()**
  - Returns the thread's priority.
- **String getName()**
  - Returns the thread's name.
- **setName(String name)**
  - Changes the name of this thread to the specified string.
- **yield()**
  - Causes the currently executing object to temporarily pause and allow other threads to execute (needed only for non pre-emptive OS such as DOS).
- **static Thread currentThread()**
  - Returns a reference to the currently executing thread object
- **toString()**
  - Returns a string representation of this thread, including the thread's name, priority, and thread group.

Concurrent Access

- Sharing data among threads could cause inconsistencies, when multiple threads access the same object at the same time.
- Consider the below example

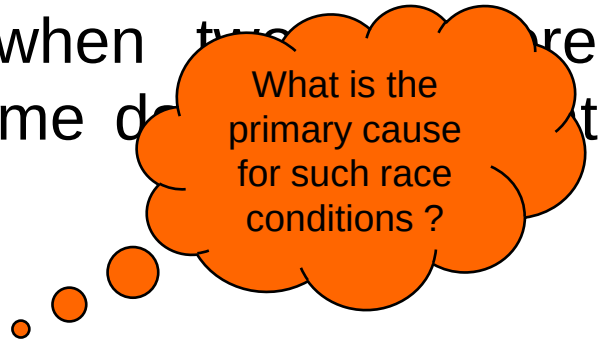


Concurrent Access

```
class MyLogger
{
    PrintWriter out = new PrintWriter(new FileWriter());
    public void log(Message msg)
    {
            out.println(msg.getName());
            out.println(msg.getTime());
            out.println(msg.getException());
            out.println(msg.getStackTrace());
    }
}
```

Race Conditions

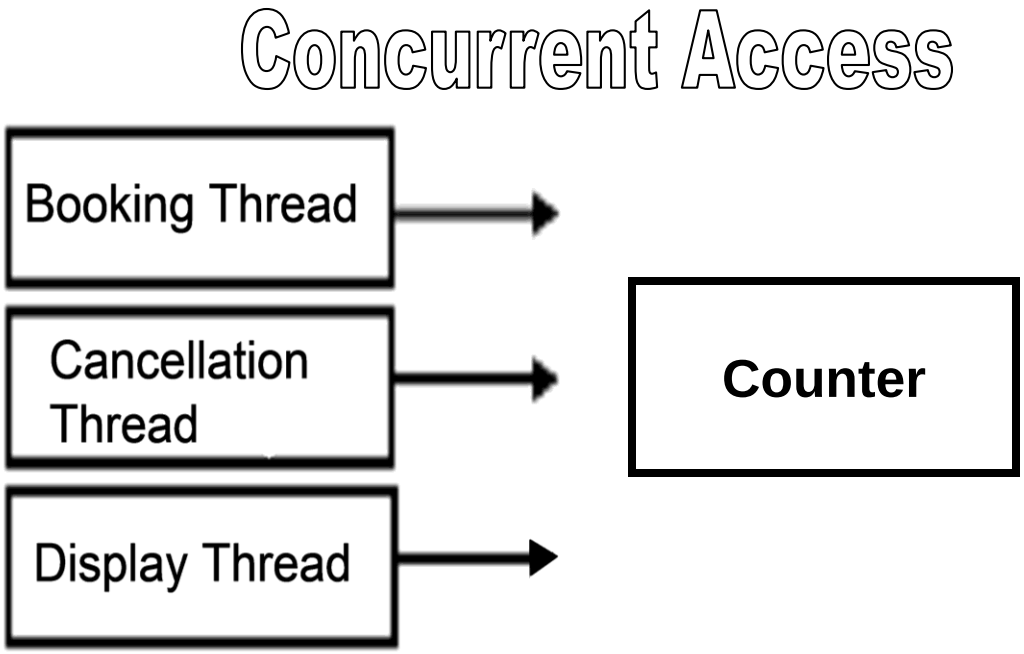
- A race condition is a programming fault which produces unpredictable program state and behavior due to un-synchronized concurrent executions.
- Race Conditions can occur when two or more threads 'race' to update the same data at the same time.



- The result can be partly what one thread wrote and partly what the other thread wrote.
- This garbles the data structure, or can cause the next thread that tries to use it to crash

Race Conditions

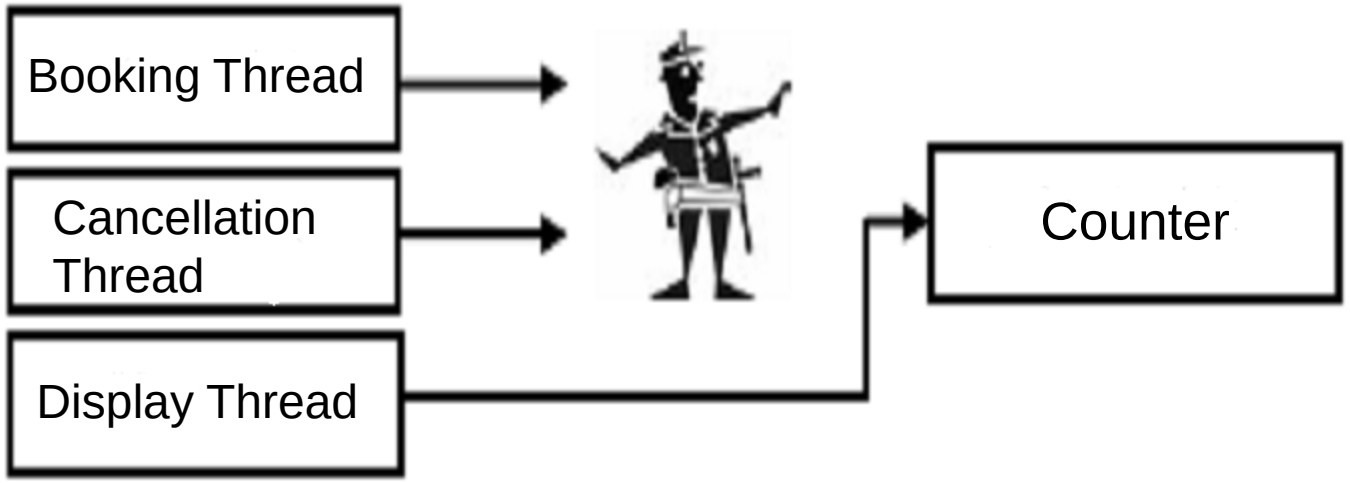
- What is the primary cause for such a race condition ?



- What is the possible work around ?

Synchronizing Access

- Synchronization makes access to the object restricted to only one thread at a time.
- The Java platform associates a lock with every object that has synchronized code.



Synchronization

- The code segments within a program that access the same object from separate, concurrent threads are called critical sections.
- Synchronization avoids race conditions by ensuring mutual exclusion to critical sections.
- Every class that has synchronized code is considered to be a monitor.
- A monitor operates by ensuring that at most one thread can execute the synchronized code within the object at any one time.
- A critical section can be a method or a few statements & is identified with the '**synchronized'** modifier.

Synchronized Methods

- To make a method synchronized, the ***synchronized*** keyword is added to its declaration.

```
class MyLogger
{
    PrintWriter out = new PrintWriter(new FileWriter());
    synchronized public void log(Message msg)
    {
            out.println(msg.getName());
            out.println(msg.getTime());
            out.println(msg.getException());
            out.println(msg.getStackTrace());
    }
}
```

Synchronized Methods

- Once a thread enters any synchronized method on an instance, no other thread can enter any other synchronized method on the same instance.
  - Only one synchronized method of an object can be active at a time
  - All other threads attempting to invoke a synchronized method are forced to wait.
- Whenever a synchronized method is called, then the object to which the synchronized method belongs is '*locked'.*
- When a synchronized method finishes executing, the lock on the object is released
  - The lock release occurs even if the method returns due to an uncaught exception.
  - The highest-priority ready thread attempting to provoke a synchronized method is then allowed to proceed.

Synchronized Blocks

- We could lock on a portion of a method using **synchronized blocks.**
- Rather than declaring the entire method to be synchronized, a few lines of critical code can be synchronized.
- This can increase concurrency and improve performance.
- Synchronized blocks place locks for shorter periods than synchronized methods.

Synchronized Blocks

- Unlike synchronized methods, synchronized block must specify the object on which we wish to hold the lock.

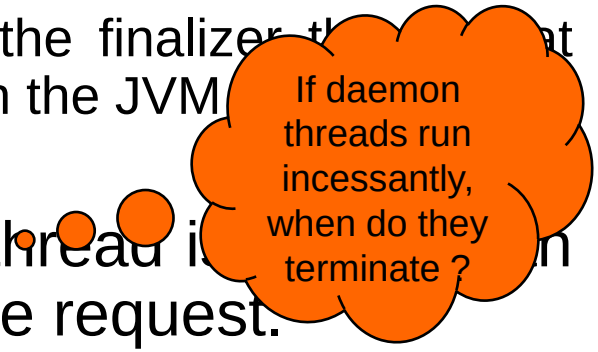
```
class MyLogger
{
    PrintWriter out = new PrintWriter(new FileWriter());
    public void log(Message msg)
    {
            ……….
            synchronized(this) {
                out.println(msg.getName());
                out.println(msg.getTime());
                out.println(msg.getException());
                out.println(msg.getStackTrace());
            }
            ……..
    }
}
```

Thread Priorities

- All threads inherit their priority from the thread that created it.
- Thread priorities are between 1 and 10.
  - Ten is the highest priority (MAX_PRIORITY)
  - One is the lowest (MIN_PRIORITY)
  - Five is the default priority (NORM_PRIORITY)
- Threads can be assigned a priority using the setPriority() method of the Thread class.
  - **void setPriority(int newPriority)**

Daemon Threads

- Daemon threads are service providers for other threads running in the same process as the daemon thread.
  - Like the garbage collector thread, the finalizer th[illegible]t are daemon threads executing within the JVM.
- The run() method for a daemon thread i[illegible]n infinite loop that waits for a service request.
- Daemon threads keep executing until there is atleast one active non daemon thread.
  - When the only remaining threads in a process are daemon threads, the process terminates.



Daemon Threads

- To specify that a thread is a daemon thread, call the setDaemon() method of Thread class with the argument true.
  - This method must be called before invoking the start method on the thread.
- Every thread acquires its ‘daemon’ property from its parent thread.
  - Threads created by daemon threads are all daemon by default.
  - Threads created by non daemon threads are non daemon by default.

Question time