Comp 204: Computer Systems and Their Implementation

Lecture 6: Concurrent Programming and Threads (2)

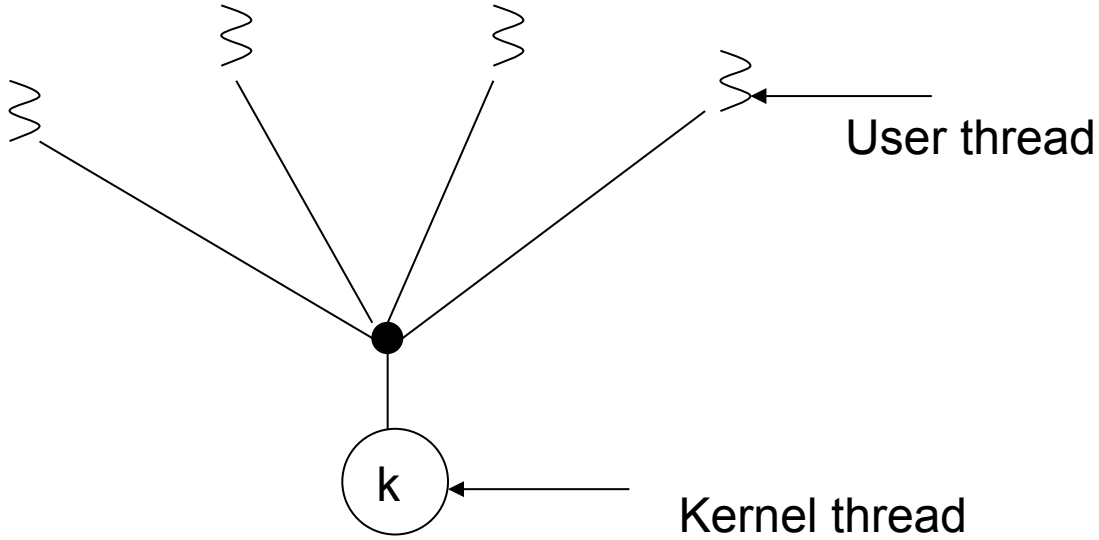
Today

- Threads
  - Multi-threading models
  - Java implementation

Multithreading Models

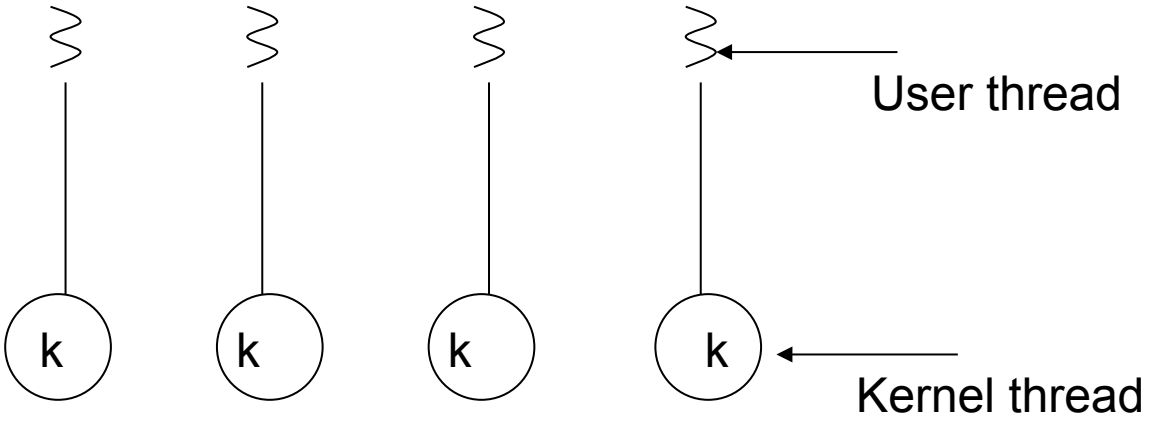
- Many systems support both user and kernel threads, resulting in different multithreading models, such as the following three common models:
  - Many-to-One Model: maps many user level threads to one kernel thread
  - One-to-One Model: maps each user thread to a kernel thread
  - Many-to-Many Model: maps many user-level threads to a smaller or equal number of kernel threads

Many-to-One Model



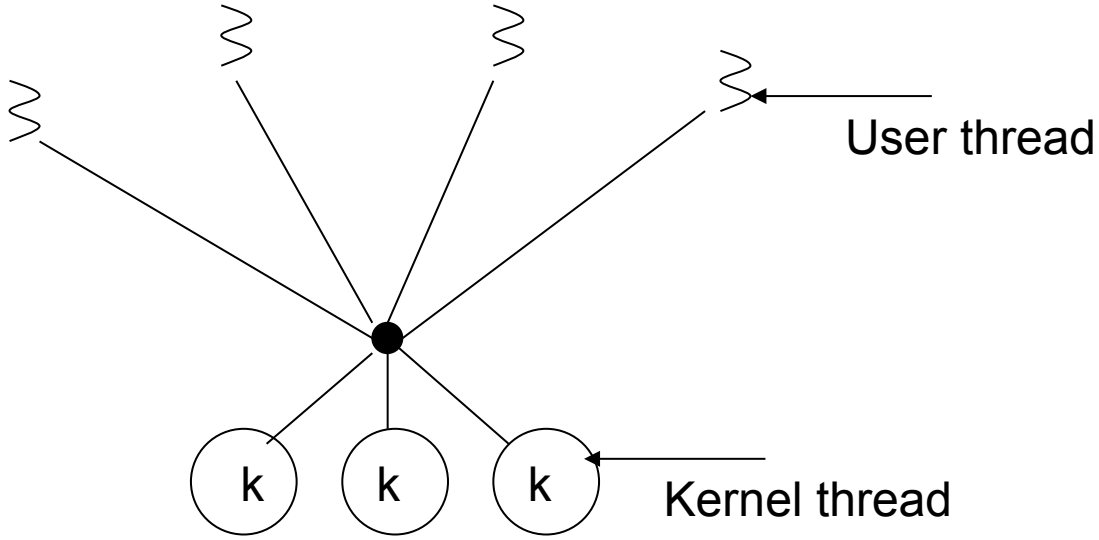
- Thread management done in user space
- Advantages: fast and efficient to create and manage threads
- Disadvantages: entire process will block if a thread makes a blocking system call

One-to-One Model



- Each user thread is mapped to a kernel thread
- Advantages: provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call, and, enables multiple threads to run in parallel on multiprocessors
- Disadvantages: creating a user thread requires corresponding kernel thread to be created, which is costly

Many-to-Many Model



- The number of kernel threads may be specific for a particular application or machine
- Advantages: number of user threads less restricted than in the previous two models, kernel threads can run in parallel on a multiprocessor, and, when a thread performs a blocking system call the kernel can schedule another one to run

Thread Control Block

- Just as information about processes is stored in Process Control Blocks (PCBs), so threads are represented by Thread Control Blocks (TCBs)

- TCBs typically contain the following information:
  - Thread ID: unique thread identifier assigned upon creation
  - Thread state: changes as thread's execution progresses
  - CPU info: info about thread's execution, such as current instruction being executed and current data being used
  - Thread priority: thread's priority status in relation to the other threads
  - Process pointer: points to the process that created the thread
  - Thread pointers: pointers to other threads created by this thread

Java Thread Creation

- When a Java program starts, a single thread is created
  - JVM also has own threads for garbage collection, screen updates, event handling etc.
- New threads may be created by extending the **Thread** class
- Again, threads may be managed directly by kernel, or implemented at user level by a library

Java Threads

```
class Worker1 extends Thread {
  public void run() {
    System.out.println("I am a Worker Thread");
  }
}

public class First {
  public static void main(String args[]) {
    Worker1 runner = new Worker1();
    runner.start();
    System.out.println("I am the Main Thread");
  }
}
```

Explanation

- Class Worker1 is derived from Thread class
- The work of the new thread is specified in the run() method
- In main() we create a new Worker1 object
- Calling the start() method...
  - allocates memory and initialises the new thread
  - causes run() method to be called
- Original thread and new thread now run in parallel

Exercise

```
public static void main(String args[]) {
      Worker1[] runners = new Worker1[100];
      for (int i=0; i < 100; i++)
         runners[i] = new Worker1();

      for (int i=0; i < 100; i++)
         runners[i].start();

      System.out.println("I am the Main Thread");
    }
```

What are we doing here?

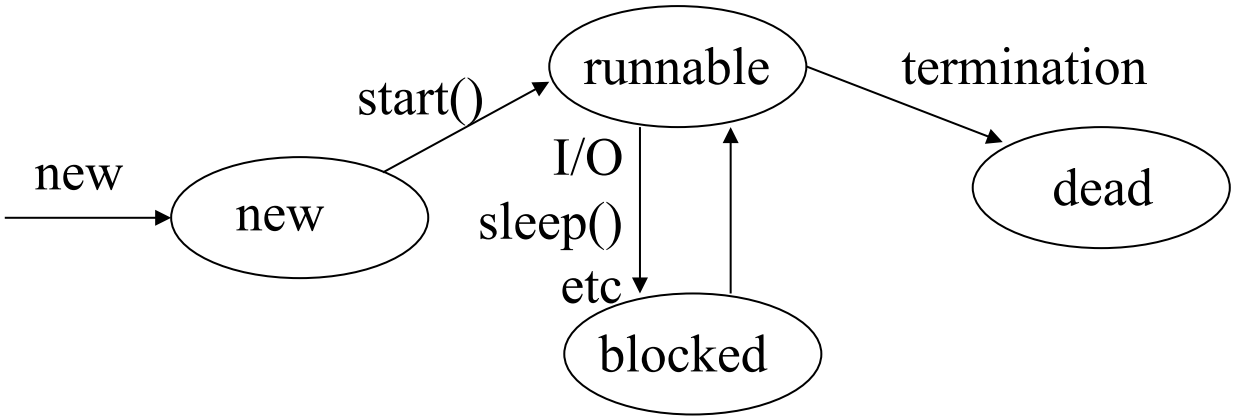
Question

- Which of the following statements about threads is FALSE?

a) A Java program need not necessarily lead to the creation of any threads
b) A thread is sometimes referred to as a lightweight process
c) Threads share code and data access
d) Threads share access to open files
e) Threads are usually more efficient than conventional processes

Answer: a
Every Java program starts as a thread! The rest of the statements are true…

Java Thread States



- All threads capable of execution are in the runnable state
  - Includes currently executing thread

Java Thread States

- A Java thread can be in one of four possible states:
- New: when an object for the thread is created (i.e. through use of the 'new' statement)
- Runnable: when the thread's run() method is invoked it moves from the *new* state to the *runnable* state, where it is eligible to be run by the JVM
- Blocked: when performing I/O the thread becomes blocked, and also when it invokes specific Thread methods, such as sleep() or suspend()
- Dead: when the thread's run() method terminates or when its stop() method is called, the thread moves to the dead state

Problem

- Suppose we have an object (called ‘thing’) which has the following method:

```
public void inc() {
   count = count + 1;
}
```

- Count is private to ‘thing’, and is initially zero
- Two threads, T1 and T2, both execute the following:

```
thing.inc();
```

Question!!!

- What value will ‘count’ have afterwards?

Answer

- We don’t know!
- This is called indeterminacy
- If T1 executes assignment before T2, or vice-versa, then count will have value 2

Question

- A Java object called ‘helper’ contains the two methods opposite, where num is an integer variable that is private to helper. Its value is initially 100.
- One thread makes the call
  - `helper.addone();`
- At the same time, another thread makes the call
  - `helper.subone();`
- What value will num have afterwards?

```
public void addone() {
    num = num + 1;
}

public void subone() {
    num = num - 1;
}
```

a) 100
b) 99
c) 101
d) either 99 or 101, but not 100
e) the value of num is undefined

Answer: d
either 99 or 101, but not 100 – if the two threads are run simultaneously, then it depends on the order in which the threads are executed by the ready queue. However, as “num” is not protected by a semaphore, its final value could be either value