CS 556 – Distributed Systems

Tutorial on
Java Threads

Threads

- A thread is a lightweight process – a single sequential flow of execution within a program
- Threads make possible the implementation of programs that seem to perform multiple tasks at the same time (e.g. multi-threaded Web servers)
- A new way to think about programming

Java Threads

We will cover:

- How to create threads in Java
- The Life Cycle of a thread
- Thread Priority
- Synchronization of threads
- Grouping of threads

How to create Java Threads

There are two ways to create a Java thread:

1. Extend the `java.lang.Thread` class
2. Implement the `java.lang.Runnable` interface

Extending the Thread class

- In order to create a new thread we may subclass `java.lang.Thread` and customize what the thread does by overriding its empty `run` method.
- The `run` method is where the action of the thread takes place.
- The execution of a thread starts by calling the `start` method.

Example I

```
class MyThread extends Thread {
    private String name, msg;

    public MyThread(String name, String msg) {
        this.name = name;
        this.msg = msg;
    }
    public void run() {
        System.out.println(name + " starts its execution");
        for (int i = 0; i < 5; i++) {
            System.out.println(name + " says: " + msg);
            try {
                Thread.sleep(5000);
            } catch (InterruptedException ie) {}
        }
        System.out.println(name + " finished execution");
    }
}
```

Example I

```
class MyThread extends Thread {
    private String name, msg;

    public MyThread(String name, String msg) {
        this.name = name;
        this.msg = msg;
    }
    public void run() {
        System.out.println(name + " starts its execution");
        for (int i = 0; i < 5; i++) {
            System.out.println(name + " says: " + msg);
            try {
                Thread.sleep(5000);
            } catch (InterruptedException ie) {}
        }
        System.out.println(name + " finished execution");
    }
}
```

Example I

```
class MyThread extends Thread {
    private String name, msg;

    public MyThread(String name, String msg) {
        this.name = name;
        this.msg = msg;
    }
    public void run() {
        System.out.println(name + " starts its execution");
        for (int i = 0; i < 5; i++) {
            System.out.println(name + " says: " + msg);
            try {
                Thread.sleep(5000);
            } catch (InterruptedException ie) {}
        }
        System.out.println(name + " finished execution");
    }
}
```

Example I (cont.)

```
public class test {
  public static void main(String[] args) {
    MyThread mt1 = new MyThread("thread1", "ping");
    MyThread mt2 = new MyThread("thread2", "pong");
    mt1.start();
    mt2.start();
  }
}
```

the threads will run in parallel

Example I (cont.)

- Typical output of the previous example:

```
thread1 starts its execution
thread1 says: ping
thread2 starts its execution
thread2 says: pong
thread1 says: ping
thread2 says: pong
thread1 says: ping
thread2 says: pong
thread1 says: ping
thread2 says: pong
thread1 says: ping
thread2 says: pong
thread1 finished execution
thread2 finished execution
```

Implementing the Runnable interface

- In order to create a new thread we may also provide a class that implements the `java.lang.Runnable` interface
- Preffered way in case our class has to subclass some other class
- A Runnable object can be wrapped up into a Thread object
  - **Thread**(Runnable target)
  - **Thread**(Runnable target, String name)
- The thread's logic is included inside the run method of the runnable object

Example II

```
class MyClass implements Runnable {
  private String name;
  private A sharedObj;
  public MyClass(String name, A sharedObj) {
    this.name = name; this.sharedObj = sharedObj;
  }
  public void run() {
    System.out.println(name + " starts execution");
    for (int i = 0; i < 5; i++) {
        System.out.println(name + " says: " + sharedObj.getValue());
        try {
            Thread.sleep(5000);
        } catch (InterruptedException ie) {}
    }
    System.out.println(name + " finished execution");
  }
}
```

Example II (cont.)

```
class A {
  private String value;
  public A(String value) { this.value = value; }
  public String getValue() {
    return value;
  }
}

public class test2 {
  public static void main(String[] args) {
    A sharedObj = new A("some value");
    Thread mt1 = new Thread(new MyClass("thread1", sharedObj));
    Thread mt2 = new Thread(new MyClass("thread2", sharedObj));
    mt1.start(); mt2.start();
  }
}
```

shared variable

Example II (cont.)

- Typical output of the previous example:

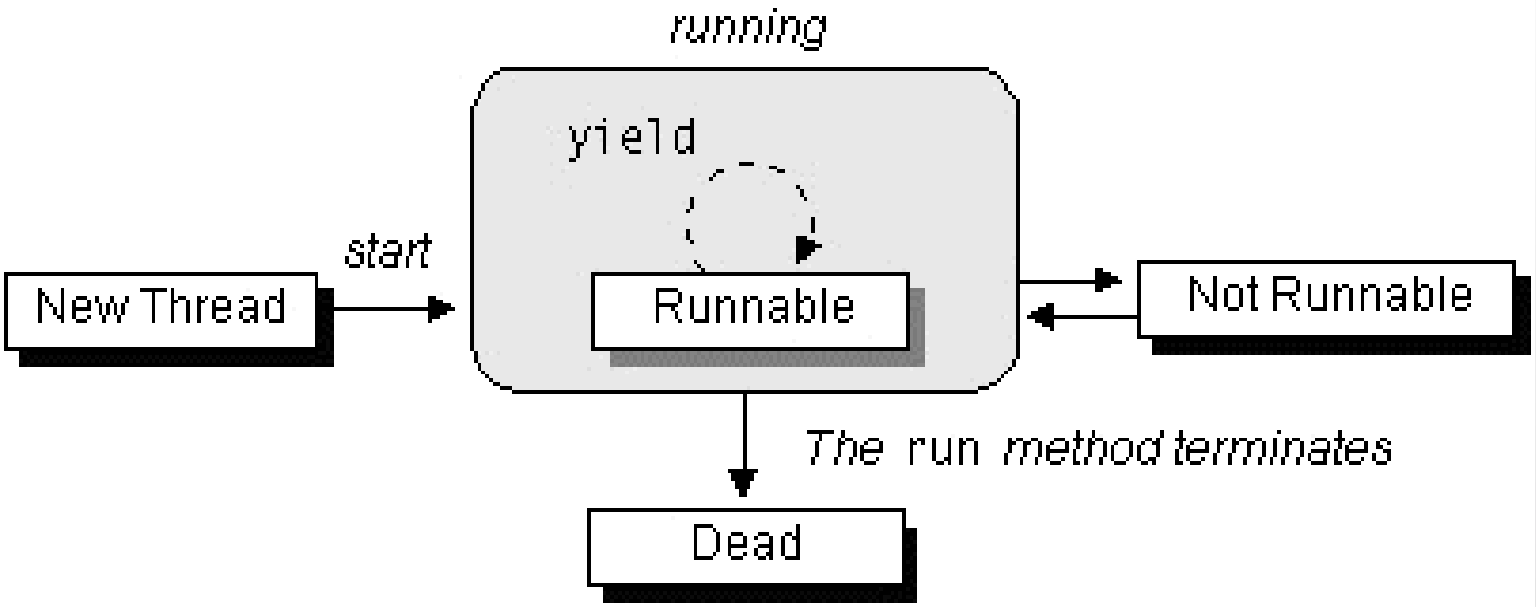
```
thread1 starts execution
thread1 says: some value
thread2 starts execution
thread2 says: some value
thread1 says: some value
thread2 says: some value
thread1 says: some value
thread2 says: some value
thread1 says: some value
thread2 says: some value
thread1 says: some value
thread2 says: some value
thread1 finished execution
thread2 finished execution
```

Java Threads

We will cover:

- How to create threads in Java
- **The Life Cycle of a thread**
- Thread Priority
- Synchronization of threads
- Grouping of threads

The Life Cycle of a Thread



The Life Cycle of a Thread (cont.)

- The start method creates the system resources necessary to run the thread, schedules the thread to run, and calls the thread's run method
- A thread becomes Not Runnable when one of these events occurs:
  - Its `sleep` method is invoked.
  - The thread calls the `wait` method.
  - The thread is blocking on I/O
- A thread dies naturally when the `run` method exits

Java Threads

We will cover:

- How to create threads in Java
- The Life Cycle of a thread
- **Thread Priority**
- Synchronization of threads
- Grouping of threads

Thread Priority

- On a single CPU threads actually run one at a time in such a way as to provide an illusion of concurrency.
- Execution of multiple threads on a single CPU, in some order, is called *scheduling*.
- The Java runtime supports a very simple scheduling algorithm (fixed priority scheduling). This algorithm schedules threads based on their priority relative to other runnable threads.

Thread Priority (cont.)

- The runtime system chooses the runnable thread with the highest priority for execution
- If two threads of the same priority are waiting for the CPU, the scheduler chooses one of them to run in a round-robin fashion.
- The chosen thread will run until:
  - A higher priority thread becomes runnable.
  - It yields, or its `run` method exits.
  - On systems that support time-slicing, its time allotment has expired

Thread Priority (cont.)

- When a Java thread is created, it inherits its priority from the thread that created it.
- You can modify a thread's priority at any time after its creation using the setPriority method.
- Thread priorities are integers ranging between `MIN_PRIORITY` and `MAX_PRIORITY` (constants defined in the `Thread` class). The higher the integer, the higher the priority.

Thread Priority (cont.)

- Use priority only to affect scheduling policy for efficiency purposes.
- Do **not** rely on thread priority for algorithm correctness.

Java Threads

We will cover:

- How to create threads in Java
- The Life Cycle of a thread
- Thread Priority
- **Synchronization of threads**
- Grouping of threads

Synchronization of Threads

- In many cases concurrently running threads share data and must consider the state and activities of other threads.
- If two threads can both execute a method that modifies the state of an object then the method should be declared to be `synchronized`, allowing only one thread to execute the method at a time.
- If a class has at least one synchronized methods, each instance of it has a monitor. A monitor is an object that can block threads and notify them when it is available.

Synchronization of Threads (cont.)

Example:

```
public synchronized void updateRecord() {
    // critical code goes here ...
}
```

- Only one thread may be inside the body of this function. A second call will be blocked until the first call returns or `wait()` is called inside the synchronized method.

Synchronization of Threads (cont.)

- If you don't need to protect an entire method, you can synchronize on an object:

```
public void foo() {
    //...
    synchronized (this) {
        //critical code goes here ...
    }
    //...
}
```

- Declaring a method as synchronized is equivalent to synchronizing on *this* for all the method block.

Synchronization of Threads (cont.)

- The Object class has three synchronization methods:
  - wait()
  - notify()
  - notifyAll()
- These methods allow objects to wait until another object notifies them:

```
synchronized( waitForThis ) {
    try {
        waitForThis.wait();
    } catch (InterruptedException ie) {}
}
```

- To wait on an object, you must first synchronize on it.

Synchronization of Threads (cont.)

- A thread may call `wait()` inside a synchronized method. A timeout may be provided. If missing or zero then the thread waits until either `notify()` or `notifyAll()` is called, otherwise until the timeout period expires.
- `wait()` is called by the thread owning the lock associated with a particular object (it causes the lock to be released).
- `notify()` or `notifyAll()` are only called from a synchronized method. One or all waiting threads are notified, respectively. It's probably better/safer to use `notifyAll()`. These methods don't release the lock. The threads awakened will not return from their wait() call immediately, but only when the thread that called notify() or notifyAll() finally relinquishes ownership of the lock.

Synchronization of Threads (cont.)

A Producer – Consumer Problem

```
public synchronized void put(int value)
    {
    while (available == true) {
        try {
     //wait for Consumer to get
     //value
            wait();
        } catch (InterruptedException e)
         {}
    }
    contents = value;
    available = true;
    //notify Consumer that value has
    //been set
    notifyAll();
}
```

```
public synchronized int get() {
    while (available == false) {
        try {
         //wait for Producer to put
        //value
            wait();
        } catch (InterruptedException e)
         {}
    }
    available = false;
      //notify Producer that value has
      //been retrieved
    notifyAll();
    return contents;
}
```

Synchronization of Threads (cont.)

- The Thread class has a method, join(), which allows an object to wait until the thread terminates

```
public void myMethod() {
    // Do some work here...
    // Can't proceed until another thread is done:
    otherThread.join();
    // Continue work...
}
```

- Equivalent to `waitFor()` of `java.lang.Process`

Java Threads

We will cover:

- How to create threads in Java
- The Life Cycle of a thread
- Thread Priority
- Synchronization of threads
- **Grouping of threads**

Grouping of Threads

- Thread groups provide a mechanism for collecting multiple threads into a single object and manipulating those threads all at once, rather than individually.
- Java thread groups are implemented by the `java.lang.ThreadGroup` class
- When a Java application first starts up, the Java runtime system creates a ThreadGroup named main.
- Unless specified otherwise, all new threads that you create become members of the main thread group.

Grouping of Threads (cont.)

- To put a new thread in a thread group the group must be explicitly specified when the thread is created
  - public **Thread**(ThreadGroup *group*, Runnable *runnable*)
  - public **Thread**(ThreadGroup *group*, String *name*)
  - public **Thread**(ThreadGroup *group*, Runnable *runnable*, String *name*)
- A thread can not be moved to a new group after the thread has been created.

Grouping of Threads (cont.)

- A ThreadGroup may also contain other ThreadGroups allowing the creation of an hierarchy of threads and thread groups:
  - public **ThreadGroup**(ThreadGroup parent, String name)
- To get the thread group of a thread the `getThreadGroup` of the `Thread` class should be called:

ThreadGroup group = myThread.getThreadGroup();

Reference

- Read the Java Tutorial on Threads:

http://java.sun.com/docs/books/tutorial/essential/threads/