CS 580 Client-Server Programming
Fall Semester, 2000
Doc 8 Threads
Contents

References....1
Concurrent Programming....2
Threads....3
- Creating Threads....4
- Solaris, Green & Native Threads....9
- Thread Scheduling....10
- Types of Threads: *user* and *daemon*....14

Thread Control....16
- Yield....18
- Sleep....19
- Join - Waiting for Thread to end....23

References

Cancellable Activities, Doug Lea, October 1998, http://gee.cs.oswego.edu/dl/cpj/cancel.html

Concurrent Programming in Java: Design Principles and Patterns, Doug Lea, Addison-Wesley, 1997

The Java Programming Language, 2nd Ed. Arnold & Gosling, Addison-Wesley, 1998

The Java Language Specification, Gosling, Joy, Steele, Addison-Wesley, 1996, Chapter 17 Threads and Locks.

Java's Atomic Assignment, Art Jolin, ***Java Report***, August 1998, pp 27-36.

Java 1.2 on-line documentation http://java.sun.com/products/jdk/1.2/docs/index.html

Java 1.2 Thread Docs http://java.sun.com/products/jdk/1.2/docs/tooldocs/solaris/threads.html

Concurrent Programming

The ability to perform concurrent programming is part of the Java programming language. That is different parts of the same program can be executing at the same time, or behave if they are executing at the same time. Java uses threads to achieve concurrency. Writing concurrent programs presents a number of issues that do not occur in writing sequential code.

Safety

Two different threads could write to the same memory location at the same time, leaving the memory location in an improper state.

Liveness

Threads can become deadlocked, each thread waiting forever for the other to perform a task. Threads can become livelocked, waiting forever to get their turn to execute.

Nondeterminism

Thread activities can become intertwined. Different executions of a program with the same input can produce different results. This can make program hard to debug.

Communication

Different threads in the same program execute autonomously from each other. Communication between threads is an issue.

Threads

A **thread** is an active entity that shares the same name space as the program that created the thread. This means that two threads in a program can access the same data.

Difference from fork()

fork()

- Child process gets a copy of parents variables
- Relatively expensive to start
- Don't have to worry about concurrent access to variables

thread

- Child process shares parents variables
- Relatively cheap to start
- Concurrent access to variables is an issue

Thread Class Methods

activeCount()
checkAccess()
countStackFrames()
currentThread()
destroy()
dumpStack()
enumerate(Thread[])
getName()
getPriority()
getThreadGroup()
interrupt()
interrupted()
isAlive()
isDaemon()
isInterrupted()
join()
join(long)
join(long, int)
resume()
run()
setDaemon(boolean)
setName(String)
setPriority(int)
sleep(long)
sleep(long, int)
start()
stop()
stop(Throwable)
suspend()
toString()
yeild()

Creating Threads

There are two different methods for creating a thread: extending the Thread class or implementing the Runnable interface. The first method is shown on this slide, the second on the next slide.

In the Thread subclass, implement the run() method. The signature of run() must be as it is in this example. run() is the entry point or starting point (or main) of your thread. To start a thread, create an object from your Thread class. Send the "start()" method to the thread object. This will create the new thread, start it as an active entity in your program, and call the run() method in the thread object. Do not call the run() method directly. Calling the run() directly executes the method in the normal sequential manner.

```
class  SimpleThread  extends Thread {
  public void run() {
    for  ( int  count = 0;  count < 4; count++)
      System.out.println( "Message " + count +
                  " From: Mom" );
  }
}

class  TestingSimpleThread {
  public static void main( String[]  args ) {
    SimpleThread  parallel  = new SimpleThread();
    System.out.println( "Create the thread");
    parallel.start();
    System.out.println( "Started the thread" );
    System.out.println( "End" );
  }
}
```

Output

```
Create the thread
Started the thread
End
Message 0 From:  Mom
Message 1 From:  Mom
Message 2 From:  Mom
Message 3 From:  Mom
```

Second Method for Creating a Thread

First, have your class implement the Runnable interface, which has one method, run(). This run() plays the same role as the run() in the Thread subclass in the first method. Second, create an instance of the Thread class, passing an instance of your class to the constructor. Finally, send the thread object the start() method.

```
class  SecondMethod  implements  Runnable {
  public  void  run() {
    for  ( int  count = 0;  count < 4; count++)
      System.out.println( "Message " + count +  " From: Dad");
  }
}

class  TestThread {
  public static void main( String[]  args ) {
    SecondMethod  notAThread  = new SecondMethod();
    Thread  parallel  =  new Thread( notAThread );

    System.out.println( "Create the thread");
    parallel.start();
    System.out.println( "Started the thread" );
    System.out.println( "End" );
  }
}
```

Output

```
Create the thread
Started the thread
End
Message 0 From: Dad
Message 1 From: Dad
Message 2 From: Dad
Message 3 From: Dad
```

Giving a Thread a Name

We can give each thread a string id, which can be useful.

```
class  SecondMethod  implements  Runnable {
  public  void  run() {
    for  ( int  count = 0;  count < 2; count++)
      System.out.println( "Message " + count +  " From: " +
                  Thread.currentThread().getName() );
  }
}

class  TestThread {
  public static void main( String[]  args ) {
    Thread  a  =  new Thread(new SecondMethod(), "Mom" );
    Thread  b  =  new Thread(new SecondMethod(), "Dad" );

    System.out.println( "Create the thread");
    a.start();
    b.start();
    System.out.println( "End" );
  }
}
```

Output

```
Create the thread
End
Message 0 From: Mom
Message 1 From: Mom
Message 0 From: Dad
Message 1 From: Dad
```

SimpleThread for Use in Future Examples

This class will be used in future examples.

```
public class  SimpleThread  extends Thread
  {
  private  int  maxCount =  32;

  public  SimpleThread(  String  name )
    {
    super( name );
    }

  public  SimpleThread(  String  name, int repetitions )
    {
    super( name );
    maxCount  =  repetitions;
    }

  public  SimpleThread(  int repetitions )
    {
    maxCount  =  repetitions;
    }

  public  void run()
    {
    for  ( int  count = 0;  count < maxCount; count++)
      {
      System.out.println( count + " From: " + getName() );
      }
    }
  }
```

Show Some Parallelism

In this example we show some actual parallelism. Note that the output from the different threads is mixed.

```
class  TestThread
   {
   public static void main( String[]  args )
      {
      SimpleThread  first      = new SimpleThread( 5 );
      SimpleThread  second  = new SimpleThread( 5 );
      first.start();
      second.start();
      System.out.println( "End" );
      }
   }
```

Command Line to Execute the Program

```
java -native TestThread
```

Output- On Rohan Using Native Threads

```
End
0 From: Thread-0
1 From: Thread-0
2 From: Thread-0
0 From: Thread-1
1 From: Thread-1
2 From: Thread-1
3 From: Thread-0
3 From: Thread-1
4 From: Thread-0
4 From: Thread-1
```

Solaris, Green & Native Threads

In the beginning, Java on Solaris machines used Green threads (Green being the name of the originator, not the actual color of the software). A Java program using Green threads runs in one Solaris thread. This means the program runs on one processor. JDK 1.2 supports the use of native Solaris threads. On multiprocessor machines like Rohan, native Solaris threads are scheduled on different processors. Thus, a Java program using native threads can use multiple processors on Rohan.

The default is JDK 1.1 & JDK 1.2 is to use green-threads. To use native threads with JDK 1.1 or JDK 1.2b4 on a Solaris machine use the flag -native:

```
java -native className
```

In unix you can also set an environment variable to specify which type of thread to use.

```
% setenv THREADS_FLAG  native
```

The options are green and native.

Save rohan and used green threads.

Thread Scheduling
Preemptive
Priorities

Each Java thread has a priority. The values of a thread priority are integer values ranging from java.lang.Thread.MIN_PRIORITY and java.lang.Thread.MAX_PRIORITY. That is currently from 1 to 10. When code running in some thread creates a new Thread object, the new thread has its priority initially set equal to the priority of the creating thread. A thread is always run before other threads with lower priorities. A program starts execution in a thread of Thread.NORM_PRIORITY (that is 5). The following example shows how to set the priority of a thread and some interesting output.

```
class  TestThread
  {
  public static void main( String[]  args )
    {
    SimpleThread  first      = new SimpleThread( 5 );
    SimpleThread  second  = new SimpleThread( 5 );
    second.setPriority( 8 );
    first.start();
    second.start();
    System.out.println( "End" );
    }
  }
```

Output

On Single Processor

```
0 From: Thread-5
1 From: Thread-5
2 From: Thread-5
3 From: Thread-5
4 From: Thread-5
0 From: Thread-4
1 From: Thread-4
2 From: Thread-4
3 From: Thread-4
4 From: Thread-4
End
```

On Multiple Processor Rohan

```
End
0 From: Thread-3
1 From: Thread-3
2 From: Thread-3
0 From: Thread-2
3 From: Thread-3
1 From: Thread-2
2 From: Thread-2
4 From: Thread-3
3 From: Thread-2
4 From: Thread-2
```

Thread Scheduling
Single Processor -Time-slicing or not time-sliced

Time-slicing

A thread is run for a short time slice and suspended,
It resumes only when it gets its next "turn"

Threads of the same priority share turns

Java does not specify if the scheduler must use time-slicing, so some implementations, do some do not.

Java running Green threads on Solaris does not time-slice

Non time-sliced threads run until:

- They end
- They are terminated
- They are interrupted
  - Higher priority threads interrupts lower priority threads
- They go to sleep
- They block on some call
  - Reading a socket
  - Waiting for another thread
- They yield

Setting Priorities

Continuously running parts of the program should have lower-priority than rarer events

User input should have very high priority

A thread that continually updates some data is often set to run at MIN_PRIORITY

Testing for Time-slicing & Parallelism

```
class  InfinityThread  extends Thread
   {
   public  void run()
      {
      while ( true )
         System.out.println(  "From: " + getName() );
      }
   }

class  TestThread
   {
   public static void main( String[]  args )
      {
      InfinityThread  first      = new InfinityThread( );
      InfinityThread  second  = new InfinityThread( );
      first.start();
      second.start();
      }
   }
```

Output if Time-sliced

A group of "From: Thread-a" will be followed by a group of "From: Thread-b" etc.

Output if not Time-sliced, Single Processor

"From: Thread-a" will repeat "forever"

Multiple Processor

"From: Thread-a" and "From: Thread-b" will intermix "forever"

Types of Threads: *user* and *daemon*

We have seen several examples now of a program that continues to execute after its main has finished. So, when does a Java program end? To answer this question we need to know about the different types of threads. There are two types of threads: user and daemon.

Daemon thread

Daemon threads are expendable. When all user threads are done, the program ends all daemon threads are stopped

User thread

User threads are not expendable. They continue to execute until their run method ends or an exception propagates beyond the run method.

When a thread is created, it is the same type of thread as its creator thread. The type a thread can be changed before its start() method is called, but not after its start() method has been called. See example on next slide. The main of your program is started in a user thread.

The Java Virtual Machine continues to execute the program until either of the following occurs:

- The exit method of class Runtime has been called and the security manager has permitted the exit operation to take place.
- All threads that are not daemon threads have died, either by returning from the call to the run method or by throwing an exception that propagates beyond the run method.

Daemon example

The thread "shortLived" has the same priority as the thread running main. Hence on a single processor machine, "shortLived" will not start until main ends or main uses up its time-slice. Main is short enough to finish in one time-slice. However, since "shortLived" is a daemon thread, it does not run after all the user threads are done. Hence, "shortLived" never starts and does not print anything.

```
class DaemonExample
  {
  public  static  void  main( String  args[] )
    {
    InfinityThread  shortLived      = new InfinityThread( );
    shortLived.setDaemon( true );
    shortLived.start();
    System.out.println( "Bye");
    }
  }

class  InfinityThread  extends Thread
  {
  public  void run()
    {
    while (true)
      {
      System.out.println(  "From: " + getName()  );
      System.out.flush();
      }
    }
  }
```

Output

```
Bye
```

Thread Control

So far we have just started threads and let them run independently. Threads often have to work together. One thread may need the result of some computation of another thread. The first thread will have to wait until the second one has completed the computation. On a single processor, you need to make sure that one thread does not consume all the CPU cycles, leaving other threads waiting for their turn.

A thread can be in one of three states: executing (running), runnable (but not executing), not runnable. On a multi-processor machine more than one thread may be executing at the same time. A thread that is waiting for another thread, suspended, or sleeping is not runnable.

Methods of Interest

destroy()	join()	stop()
interrupt()	join(long)	stop(Throwable)
interrupted()	join(long, int)	suspend()
isInterrupted()	sleep(long)	yield()
	sleep(long, int)	

Threads Run Once

Once a thread's run method ends, the thread can not be restarted. You can call the thread's run method directly, but that just sequentially executes that method.

```
class  SimpleThread  extends Thread {
   public void run() {
      System.out.println(  "I ran" );
   }
}

public class Test  {
   public static void main( String args[] ) throws Exception {
      SimpleThread onceOnly = new SimpleThread();
      onceOnly.setPriority( 6 );
      onceOnly.start();

      System.out.println( "Try restart");
      onceOnly.start();

      System.out.println( "The End");
   }
}
```

Output

```
I ran
Try restart
The End
```

Yield

Sending the "yield()" method to a thread moves it from the executing state to the runnable state. Often the executing thread sends the "yield()" method to itself. This allows other runnable threads of the same priority a chance to execute.

```
class  YieldThread  extends Thread {
  public  void run() {
    for  ( int  count = 0;  count < 4; count++) {
      System.out.println( count + "From: " + getName() );
      yield();
    }
  }
}

class  TestThread {
  public static void main( String[]  args )  {

    YieldThread  first      = new YieldThread();
    YieldThread  second  = new YieldThread();

    first.start();
    second.start();
    System.out.println( "End" );
  }
}
```

Output - Single and Multiple Processors

```
End
0From: Thread-2
0From: Thread-3
1From: Thread-2
1From: Thread-3
2From: Thread-2
2From: Thread-3
3From: Thread-2
3From: Thread-3
```

Sleep

The sleep() method moves a thread to the not runnable state for specified length of time. At the end of that time, the thread is moved to the runnable state. If it has a higher priority than the current executing thread, the executing thread is moved to the runnable state and the just awakened thread becomes the executing thread.

```
class  SleepyThread  extends Thread
  {
  int maxCount = 4;
  public  SleepyThread(  String  name, int count )
    {
    super( name );
    maxCount = count;
    }

  public  void run()
    {
    for  ( int  count = 0;  count < 4; count++)
      {
      System.out.println( count + " From: " + getName() );

      try
        {
        sleep( 5 );  // in  milliseconds
        }
      catch  ( InterruptedException ignored )  {}
      }
    }
  }
```

// Sleep Example Continued

```
class  TestSleep
   {
   public static void main( String[]  args )
      {
      SimpleThread  alert    = new SimpleThread( "Alert", 32 );
      SleepyThread  sleepy  = new SleepyThread( "Sleepy", 32 );
      sleepy.setPriority( 8 );
      alert.start();
      sleepy.start();
      System.out.println( "End" );
      }
   }
```

Output (read down first)

```
0  From: Sleepy
0  From: Alert
1  From: Alert
2  From: Alert
3  From: Alert
4  From: Alert
5  From: Alert
6  From: Alert
7  From: Alert
8  From: Alert
9  From: Alert
10  From: Alert
11  From: Alert
12  From: Alert
13  From: Alert
14  From: Alert
1  From: Sleepy
End
15  From: Alert
16  From: Alert
17  From: Alert
18  From: Alert
19  From: Alert
20  From: Alert
21  From: Alert
22  From: Alert
23  From: Alert
24  From: Alert
25  From: Alert
26  From: Alert
27  From: Alert
28  From: Alert
2  From: Sleepy
29  From: Alert
30  From: Alert
31  From: Alert
3  From: Sleepy
```

A Clock Example

Using the sleep method and a thread, we can make a clock. The following clock can be used in a simulation. It allows us to simulate a given number of days in an hour.

```
import java.util.Date;
class Clock extends Thread
  {
  static int millisecondsHour = 1000 * 60 * 60;
  static long millisecondsPerDay = millisecondsHour * 24;

  int millSecondsPerSimDay;
  Date simDate = new Date();

  public Clock( int simDaysPerHour )
    {
    millSecondsPerSimDay  = millisecondsHour / simDaysPerHour;
    setPriority( Thread.MAX_PRIORITY );
    }

  public String toString() { return simDate.toString(); }

  public void run()
    {
    try
      {
      while (true)
        {
        // Advance one sim day
        long simTime = simDate.getTime();
        simDate.setTime( simTime + millisecondsPerDay);
        // wait until next day
        sleep( millSecondsPerSimDay );
        }
      }
    catch ( InterruptedException simulationOver )
      { return;}
    }
  }
```

Using the Clock

```
import sdsu.io.Console;

class Test
   {
   public static void main(String  args[] ) throws Exception
      {
      Clock fast = new Clock( 7 * 52 * 60 );
      fast.start();
      for ( int k = 0; k < 5; k++ )
         {
         System.out.println( fast.toString());
         Console.readInt( "Type an int" );
         }
      fast.stop();
      }
   }
```

Output

```
rohan 24-> java Test
Tue Nov 03 09:37:55 PST 1998
Type an int 12
Sun Nov 29 09:37:55 PST 1998
Type an int 32
Fri Jan 29 09:37:55 PST 1999
Type an int 12
Sat Apr 17 10:37:55 PDT 1999
Type an int 10
Thu Jan 20 09:37:55 PST 2000
Type an int 10
```

Join - Waiting for Thread to end

If thread A sends the "join()" method to thread B, then thread A will be "not runnable" until thread B's run methods ends. At that time thread A becomes runnable.

```
class  Godot
  {
  public static void main( String[]  args ) throws
     InterruptedException
     {
     SimpleThread  lowly      = new SimpleThread( "Lowly" );
     lowly.setPriority( 1 );
     lowly.start();

     System.out.println( "now I can go" );

     lowly.join();

     System.out.println( "Done" );
     }
  }
```

Output (Read down first)

now I can go	11 From: Lowly	23 From: Lowly
0 From: Lowly	12 From: Lowly	24 From: Lowly
1 From: Lowly	13 From: Lowly	25 From: Lowly
2 From: Lowly	14 From: Lowly	26 From: Lowly
3 From: Lowly	15 From: Lowly	27 From: Lowly
4 From: Lowly	16 From: Lowly	28 From: Lowly
5 From: Lowly	17 From: Lowly	29 From: Lowly
6 From: Lowly	18 From: Lowly	30 From: Lowly
7 From: Lowly	19 From: Lowly	31 From: Lowly
8 From: Lowly	20 From: Lowly	Done
9 From: Lowly	21 From: Lowly	
10 From: Lowly	22 From: Lowly	