

CS 580 Client-Server Programming

Fall Semester, 2002

Doc 6 Threads part 1

Contents

| | |
|--|----|
| Concurrent Programming | 3 |
| Threads - Light Weight Processes | 4 |
| Creating Threads | 5 |
| Multiple Processors | 12 |
| Thread Scheduling | 14 |
| Priorities..... | 15 |
| Time-slicing..... | 20 |
| Java Types of Threads: <i>user</i> and <i>daemon</i> | 22 |
| Thread Control..... | 24 |
| Thread States..... | 24 |
| Yield..... | 25 |
| Suspend & Resume | 27 |
| Killing a Thread | 29 |

References

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 1.4.1 on-line documentation <http://java.sun.com/j2se/1.4/docs/api/overview-summary.html>

VisualWorks Application Developers Guide, Cincom, Chapter 11 (Smalltalk), see docs/AppDevGuide.pdf in the VW installation

Copyright ©, All rights reserved. 2000 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent (<http://www.opencontent.org/opl.shtml>) license defines the copyright on this document.

Reading

Java Network Programming, 2nd Ed., Harold, Chapter 5. (Java)

VisualWorks Application Developers Guide, Cincom, Chapter 11 (Smalltalk), see docs/AppDevGuide.pdf in the VW installation

Source Code

Smalltalk

See rugby.sdsu.edu_cs580 Store repository
Package ThreadLecture1

Java

CVS root `~whitney/cs580/cvsroot`
Module name `threadExamples`

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 - Light Weight Processes

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 Processes (fork())

Processes (Heavy Weight)

- Child process gets a copy of parents variables

- Relatively expensive to start

- Don't have to worry about concurrent access to variables

Thread (Light Weight Process)

- Child process shares parents variables

- Relatively cheap to start

- Concurrent access to variables is an issue

Creating Threads Smalltalk

fork

- Creates a process (thread) from a block
- Starts the process

Transcript

```
clear;  
show: 'Create the thread';  
cr.  
[1 to: 4 do:  
  [:each |  
    Transcript  
    show: 'Message ' , each printString;  
    cr]] fork.
```

Transcript

```
show: 'Thread started. The End';  
cr
```

Output

```
Create the thread  
Thread started. The End  
Message 1  
Message 2  
Message 3  
Message 4
```

Note later we will discuss some a problem related to threads and the VW GUI. For now I will ignore the problem.

newProcess

```
| hello |
```

```
hello := [Transcript show: 'Hi';cr] newProcess.
```

“hello will not run until it is activated via resume”

```
hello resume.
```

Both fork and newProcess return an instance of Process

With Arguments

```
| printer printProcess |
```

```
printer :=
```

```
[:a :b :c |
```

```
Transcript
```

```
print: a + b + c;
```

```
cr;
```

```
flush].
```

```
printProcess := printer newProcessWithArguments: #(1 2 3).
```

```
printProcess resume
```

Creating Threads Java

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

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

Output

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

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");
    }

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


Giving a Thread a Name

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

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

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

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

Output

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

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.

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

Output- On Rohan

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

Multiple Processors Java

Java on a Solaris machine with multiple processors can run threads on different processors

If you run the last example on a single processor machine the results may be completely different.

Smalltalk

VisualWorks claims to have some native thread support

I have not used this feature

Showing Parallelism – Smalltalk

```
Transcript clear.  
[10 timesRepeat:  
  [Transcript  
    show: 'A '  
    cr]] fork.  
[10 timesRepeat:  
  [Transcript  
    show: 'B '  
    cr]] fork
```

Output

You get the 10 'A's first then the 10 'B's

What is going on?

Thread Scheduling

- Priorities
- Timeslicing

Priorities

Each thread has a priority

If there are two or more active threads

- If one has higher priority than others
- The higher priority thread is run until it is done or not active

Java Priorities

| java.lang.Thread field | Value |
|------------------------|-------|
| Thread.MAX_PRIORITY | 10 |
| Thread.NORM_PRIORITY | 5 |
| Thread.MIN_PRIORITY | 1 |

Smalltalk Priorities

| Priority | ProcessScheduler Methods | Purpose |
|----------|--------------------------|---|
| 100 | timingPriority | Processes that are dependent on real time |
| 98 | highIOPriority | Critical I/O processes, such as network input handling |
| 90 | lowIOPriority | Normal input/output activity, such as keyboard input |
| 70 | userInterruptPriority | High-priority user interaction; such a process pre-empts window management, so it should be of limited duration |
| 50 | userSchedulingPriority | Normal user interaction |
| 30 | userBackgroundPriority | Background user processes |
| 10 | systemBackgroundPriority | Background system processes |
| 1 | systemRockBottomPriority | The lowest possible priority |

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`

Java Examples

```
public class PriorityExample
{
    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
```

Smalltalk Priority Example

Transcript clear.

```
[10 timesRepeat:
```

```
  [Transcript
```

```
    show: 'A ';
```

```
    cr]] forkAt: Processor userSchedulingPriority.
```

```
[10 timesRepeat:
```

```
  [Transcript
```

```
    show: 'B ';
```

```
    cr]] forkAt: Processor userSchedulingPriority + 1
```

Output

First you get 10 'B's then 10 'A's

Threads Run Once

When a Java or Smalltalk thread ends it can not be restarted

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

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

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

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

Output

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

Thread Scheduling Time-slicing

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

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

Java

Does not specify if threads are time-sliced or not

Implementations are free to decide

VisualWorks

Threads are not timesliced

Testing for Time-slicing & Parallelism

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

    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"

Java 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.

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

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

Output

From: Thread-0 (Repeated many times)

Bye

From: Thread-0 (Repeated some more, then the program ends)

Thread Control Thread States

- Executing

Only one thread per processor can be running at a time

- Runnable

A thread is ready to run but is not currently running

- Not Runnable

A thread that is suspended or waiting for a resource

Yield

Allow another thread of the same priority to run

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

    public static void main( String[] args ) {

        YieldThread first    = new YieldThread();
        YieldThread second = new YieldThread();
        first.setPriority( 1);
        second.setPriority( 1);
        first.start();
        second.start();
        System.out.println( "End" );
    }
}
```

Output (Explain this)

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

Smalltalk Yield Example

```
[10 timesRepeat:  
  [Transcript  
    show: 'A '  
    cr.  
  Processor activeProcess yield]]  
fork.
```

```
[10 timesRepeat:  
  [Transcript  
    show: 'B '  
    cr.  
  Processor activeProcess yield]]  
fork
```

Output

```
A  
B  
A  
B  
A  
B (the pattern repeats)
```

Suspend & Resume Smalltalk

suspend

Suspends a processs,
Raises an exception if processes is already suspended

suspendUnconditionally

Like suspend but can be sent to suspended process

resume

Resumes a suspended processs

| a |

Transcript clear.

a := [3 timesRepeat:

[Transcript
show: 'A ';
cr.

Processor activeProcess yield]] fork.

[a suspend.

4 timesRepeat:

[Transcript
show: 'B ';
cr.

Processor activeProcess yield].

a resume] fork

Output

A
B
B
B
B
A
A

Suspend & Resume – Java

The following Thread methods are not thread safe

- suspend
- resume
- stop

These methods can leave your Java program in unstable states

You should not use them

Killing a Thread Smalltalk

terminate

kills a Smalltalk process

Example

```
| a |  
Transcript clear.  
a := [3 timesRepeat:  
    [Transcript  
     show: 'A '  
     cr]] forkAt: Processor userSchedulingPriority - 5.  
  
a terminate.  
Transcript show: 'Killed a'.
```

Output

Killed a

Killing a Thread - Java

stop

This Thread method is unsafe do not use it

destroy

This method does nothing. It was never implemented

There is no good way to really kill a Java thread

Later lectures will cover some suggestions for doing this