OS Week 3 - Lab

Threads and Race Conditions

Introduction

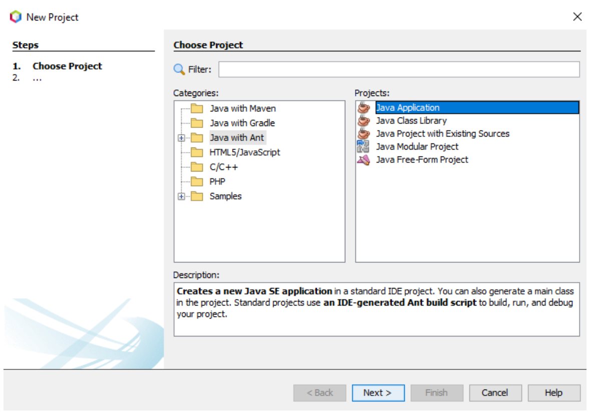
This week's lab starts to illustrate ideas about concurrency. It builds on Java basics reviewed in the last couple of weeks, and you may want to refer back to earlier lab scripts as you proceed.

In the main part of the lab, you will be asked to run some simple Java programs with more than one thread. Then we will go on to illustrate problems that can arise when two or more threads in a concurrent program try to update the same data structure (Race conditions).

Record the results of these experiments, and answer both reflective questions in your lab book. Finally, there are two exercises, which you are strongly encouraged to attempt.

Notes if you are using NetBeans: When creating a Java application project, select "*Java with Ant*". Unlike with earlier versions of NetBeans, it may be necessary to explicitly save edited Java files before running them! If classes still don't update, try running "Clean and Build Project" on the "Run" menu, and try again.

Note: When you create a new Java application for an experiment, don't forget to rename the project name and the main class as the name given in the experiment to run it with no errors.


New Project

Steps
1. Choose Project
2. ...

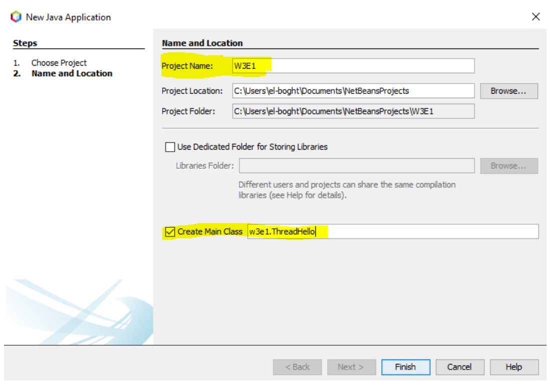
Choose Project

Filter:

Categories: Java with Maven, Java with Gradle, Java with Ant, HTML5/JavaScript, C/C++, PHP, Samples

Projects: [✓ Java Application], Java Class Library, Java Project with Existing Sources, Java Modular Project, Java Free-Form Project

Description:
Creates a new Java SE application in a standard IDE project. You can also generate a main class in the project. Standard projects use **an IDE-generated Ant build script** to build, run, and debug your project.


New Java Application

Steps
1. Choose Project
2. Name and Location

Name and Location

Project Name: W3E1

Project Location: C:\Users\el-boght\Documents\NetBeansProjects

Project Folder: C:\Users\el-boght\Documents\NetBeansProjects\W3E1

☐ Use Dedicated Folder for Storing Libraries

Libraries Folder:

Different users and projects can share the same compilation libraries (see Help for details).

☑ Create Main Class w3e1.ThreadHello

Thread Creation

In Java, a *thread* is represented by a class that extends a special class `Thread`. If you are less familiar with object-oriented programming, don't worry about exactly how the class extension works; just follow the recipe in the examples below. In any case, the class must have a method called `run` which contains the part of the program executed by the new thread.

Experiment 1

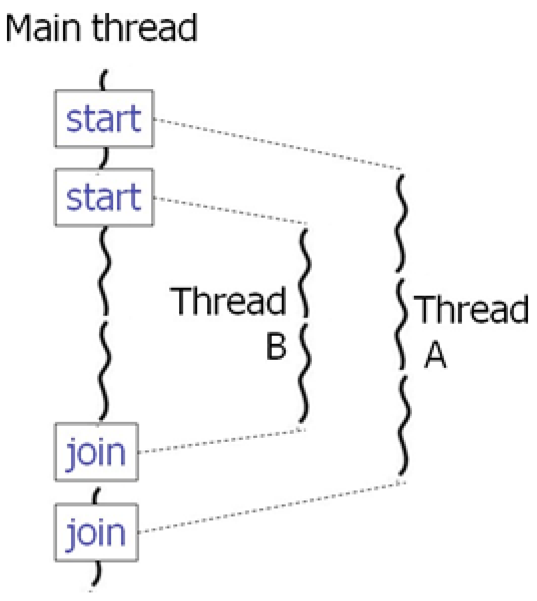
Here is a simple example:

```
public class ThreadHello {
      public static void main(String args []) throws Exception {
      MyThread thread1 = new MyThread();
      thread1.name = "A";
      MyThread thread2 = new MyThread() ;
      thread2.name = "B";
      thread1.start() ;
      thread2.start();
      thread2.join() ;
      thread1.join() ;
}
}

class MyThread extends Thread {
      String name;
      public void run() {
      System.out.println("Hello from thread " + name) ;
}
}
```

The main program creates two objects of class `MyThread` and gives each a name. `MyThread` is a thread class, defining a `run` method, which in our case just prints a hello message.

The two threads are started by calling their `start` method. Then the main program just waits for both threads to finish by calling the `join` method on each. Refer to the week 2 lecture for more explanation of this. The picture here is slightly different to the example in the lecture. It looks like this:



We could have moved the code of one of our threads into the main method, but later it will be convenient to do things in this more symmetric way.

Run the example above. Quite likely you will see output from `thread1` followed by the output from `thread2` because thread1 starts first and gets a head start. But try to run the program a few times and see if the other possible ordering sometimes appears. It may or may not, but that is the nature of non- determinism!

Experiment 2

To make it more interesting, let's add a loop and some random delays. Change the definition of MyThread to this:

```
class MyThread extends Thread {
	String name;
	public void run()
	{
		for(int i = 0 ; i < 10 ; i++)
		{
		delay();
		System.out.println("Hello from thread " + name);
		}
	}

void delay() {

	int delay = (int) (1000 * Math.random());
	try {
		Thread.sleep(delay);
```

```
        }
        catch(Exception e) {
        }
    }
}
```

The trick for creating random delays was introduced in an earlier exercise (week 1). Here it just brings out the concurrent nature of the threads more clearly. Run the new version of the program. You should see interleaved messages from both threads, proving they are both active at the same time.

What happens if you remove the calls to `delay()`?

A Race Condition

Let's try to capture a race condition in a simple counter, as described in the final section of the week 2 lecture. Because the counter will be a static variable it is effectively common to, or shared between, all objects in the `MyThread` class (see examples on static variables in the week 2 lab). But now by sharing this field between two objects, we are also sharing it between two threads.

Experiment 3

Here is a program that is very likely to illustrate a race condition:

```
public class Race {
        public static void main(String args []) throws Exception {
                MyThread.count = 0;
                MyThread thread1 = new MyThread();
                MyThread thread2 = new MyThread();

                thread1.start();
                thread2.start();
                thread2.join();
                thread1.join();
                System.out.println("MyThread.count = " + MyThread.count) ;
        }
}

class MyThread extends Thread {
        volatile static int count;
        public void run()
        {
                for(int i = 0 ; i < 1000000000 ; i++) {
                int x = count;
                count = x + 1;
                }
        }
}
```

The general structure is very similar to the thread program above. But now we have added a static variable called count that is initialized to zero, then incremented 1,000,000,000 times by each of two threads.

See the note at the end of this lab for some discussion of the `volatile` modifier.

If we didn't know about race conditions, we might expect that the final value printed out would be 2,000,000,000. Run the program and record and interpret the outcome (be patient - the program may take a minute or two to run).

(After you have run this experiment, you may want to look at the appendix to the week 3 lecture at the end of the slides).

Experiment 4

You might think that we have "faked" this behaviour by unrealistically splitting the counter update into two statements. Try replacing these two lines:

```
int x = count;
count = x + 1;
```

by this single line:

```
count = count + 1;
```

Then try replacing them with this:

```
count++;
```

In Java and similar languages, the ++ operator is a quick way of increasing the value of a variable by one.

Run the code from experiment 3 after these changes, and record the outcome.

Reflective Question 1

Do the changes made in Experiment 4 help resolve the race condition? What do the results tell you about ***atomicity*** *of various Java statements?* (See the week 3 lecture, slide 7, for a definition of atomicity).

A "Slow Race"

Modern computers run so fast that billions of updates can be performed in a second. In the previous example, we needed to loop millions of times to really see the race condition (otherwise the first thread might do all its work before the second thread has even kicked in, and we don't truly have parallel execution).

Experiment 5

Let's slow things down artificially by putting delays at strategic positions in the loops. We'll also print out some messages as we go. Look at this program:

```
public class SlowRace {

        public static void main(String args []) throws Exception {
        MyThread.count = 0;
        MyThread thread1 = new MyThread();
        thread1.name = "A";
        MyThread thread2 = new MyThread();
        thread2.name = "B";

        thread1.start();
        thread2.start();
        thread2.join();
        thread1.join();
        System.out.println("MyThread.count = " + MyThread.count);
        }
}

class MyThread extends Thread {
        volatile static int count;
        String name;

        public void run() {
                for(int i = 0 ; i < 10 ; i++) {
                delay();
                int x = count;
                System.out.println("Thread " + name + " read " + x);
                delay();
                count = x + 1;
                System.out.println("Thread " + name + " wrote " + (x + 1));
                }
        }

        void delay() {
                int delay = (int) (1000 * Math.random());
                try {
                        Thread.sleep(delay);
                   }
                catch(Exception e) {
                }
        }
}
```

First, convince yourself that, apart from adding delays and printing messages - neither of which we generally expect to change the final outcome of a program - this is essentially the same as the previous program.

If there was no race condition, this version would print out a final value of 20 for the counter. Run this program and carefully study the output. You should be able to see the race condition "in action", and explain more exactly the final result for the count (do so in your lab book, which should record the whole of the output of at least one run).

The exact outcome will change from run to run - precisely what is meant by the term non-determinism.

Although we have used random delays to accentuate this non-deterministic behaviour, you should convince yourself that the non-determinism here isn't fundamentally caused by having random number generators in the program, but by the program's concurrent nature.

Reflective Question 2

Performance issues aside, would correctness in behaviour of a sequential program ever be affected by introducing random delays?

Exercises

1. Without using any further Java features for concurrency (i.e. don't yet use the `synchronized` keyword, or any of the classes in the `java.util.concurrent` package), try to eliminate the race condition in our "Slow Race" example.

   This is possible, just by setting and testing additional shared variables to "guard" the critical section. For our purposes, the critical section is these lines:

```
int x = count;
System.out.println("Thread " + name + " read " + x);
delay();
count = x + 1;
System.out.println("Thread " + name + " wrote " + (x + 1));
```

Possible algorithms will be given in this week's lecture. Try coming up with your own solution.

If your lab session is timetabled *after* this week's lecture, you may try some of the solutions and attempted solutions from the lecture.

Otherwise, after making your own attempt, take a look at this link: http://en.wikipedia.org/wiki/Peterson's_algorithm. Can you program this algorithm in Java?

2. Inspect and run the code below. You will probably want to refer to the separate PowerPoint "appendix" to the week 2 lecture, where a simplified version of this code was considered:

```
public class QueueRace {
      public static void main(String args []) throws Exception {
      MyThread.q = new Queue();
      MyThread thread1 = new MyThread();
      thread1.name = "A";

      MyThread thread2 = new MyThread();
      thread2.name = "B";
      thread1.start();
      thread2.start();
      thread2.join();
      thread1.join();
      }
}

class MyThread extends Thread {
      volatile static Queue q;
      String name;

      public void run() {
            for(int i = 0 ; i < 1000000 ; i++) {
            q.add("message " + i);
            System.out.println("Thread " + name + " added message " + i);
            }
      }
}

class Queue {
      volatile Node front, back;

      public void add(String data) {
            if (front != null) {
            back.next = new Node(data) ; back = back.next;
            }
            else {
            front = new Node(data);
            back = front;
            }
      }
```

```
	public String rem() {

	// returns null if queue empty
	String result = null;

		if (front != null) { result = front.data ; front =
	front.next;
		}
		return result;
	}
}

class Node {

	Node(String data) {
	this.data = data;
	}

	String data;
	Node next;
}
```

You are likely to see intermittent failures. What is the code trying to do, and what is the failure mode?

Note on the `volatile` modifier

In principle, where fields are shared between threads *without* using Java's built-in features for mutual exclusion, those variables should be declared volatile. Intuitively, this is a hint to the system that such variables may be changed unpredictably by other threads. There is a more precise definition of volatile in terms of the *Java Memory Model*, which you may read up on if you are curious (google it, or take a look at the relevant chapter of the Java Language Specification, though it is a tough read!)

SPECIAL TOPIC: Creating threads using POSIX Threads in C

This is optional!

1. Start the Code::Blocks IDE as in earlier weeks.
2. Click on "File" -> "New" -> "Project". Select "Console Application" and select the language as "C" when prompted.
3. Choose a Project title (I used "pthreads-egs"). You can set "Folder to create your project in" as "N:\".
4. Accept defaults on the next screen, and click "Finish".
5. On the file chooser, go to "Sources" and double click on "main.c".
6. Replace the ready-made "Hello World" program with this code:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void* run(void* name) {
      printf("Hello from thread %s\n", (char*) name);
      return NULL;
}

int main() {
      pthread_t thread1;
      char* thread1Name = "A";
      pthread_create(&thread1, NULL, run, thread1Name);

      pthread_t thread2;
      char* thread2Name = "B";
      pthread_create(&thread2, NULL, run, thread2Name);
      pthread_exit(NULL);
      return 0;
}
```

It would be interesting to try some of the other experiments above using POSIX Threads. But you may have some problems with the random number generation in our `delay()` methods because C doesn't have a standard random number generator that is safe to call from multiple threads. Also, the form of the sleep method will depend on which OS you are using. E.g. in Windows, something like this should work:

```
#include <Windows.h>

void delay() {
Sleep(1000) ; // sleep for 1 second.
}
```

It will be different on Linux.