Threads and Concurrency in Java: Part 1

Concurrency

- What every computer engineer needs to know about concurrency:
  
  *Concurrency is to untrained programmers as matches are to small children. It is all too easy to get burned.*

Concurrency: What

- Concurrency means that there are multiple agents running at the same time and interacting.

Concurrency: Where

- Sources of concurrency:
  
  - We have concurrency when we have interacting processes running on different computers (e.g. Apache—a web server—on mona.engr.mun.ca and Firefox—a web browser—on paradox.engr.mun.ca)
Concurrency: Where

- We also have concurrency when we have interacting processes running on the same computer. E.g. Firefox and Windows Explorer.
  - Every interactive program is part of a concurrent system: the user is a concurrent agent.
- Furthermore we can have multiple “threads of control” within one OS process.
  - A.K.A. multithreading
- Concurrency can be intermachine, interprocess, or multithreading.

Concurrency: Why

- Reasons for using concurrency
  - Speed. Multiple threads can be run on multiple processors (or multiple cores). This may give a speed advantage.
  - Distribution. We may wish different parts of a system to be located on different machines for reasons of convenience, security, reliability, ....

- These slides concentrate on intraprocess concurrency in Java.
  - Asynchrony. It is easiest to deal with multiple sources of events by having one thread dedicated to each stream of incoming or outgoing events.
    - For example, a web browser may use one thread to deal with input from the user and one thread for each server it is currently interacting with.
    - Likewise a web server will typically dedicate at least one thread to each current session.
Threads

- Each thread has its own
  - program counter
  - registers
  - local variables and stack
- All threads share the same heap (objects)

Concurrency: How
Multiple processors

- Multiprocessor (and multicores)

Time slicing implementation

- Single processor
  - The CPU is switched between threads at unpredictable times
- Multiple processor
  - Thread may occasionally migrate

Context switch
Thread Objects in Java

- In Java, threads are represented by objects of class java.lang.Thread.
- The run method contains the actual code for the thread to execute.

```java
public class ThreadExample extends Thread {
    private String message;
    ThreadExample(String message) {
        this.message = getName() + " : " + message;
    }
    @Override
    public void run() {
        for (int i = 0; i < 20; ++i)
            System.out.println(message);
    }
}
```

Starting a new thread

- Calling t.start() starts a new thread which executes the t.run()

```java
public class ThreadExampleMain {
    public static void main(String[] args) {
        ThreadExample thread0 = new ThreadExample("Hi");
        ThreadExample thread1 = new ThreadExample("Ho");
        thread0.start();
        thread1.start();
        System.out.println(Thread.currentThread().getName() + ": Main is done");
    }
}
```

Output for example

- Possible output for example:
  - Thread-1: Ho
  - Thread-1: Ho
  - Thread-0: Hi
  - main: Main is done
  - Thread-1: Ho
  - Thread-0: Hi
  - ... (and so on for another 35 lines)
- When t.run() completes the thread stops.
- When all threads have stopped the program exits

Race Conditions

- A system has a race condition when its correctness depends on the order of certain events, but the order of those events is not sufficiently controlled by the design of the system.
Race Conditions

- Often race conditions occur because 2 agents have uncontrolled access to a shared resource
- Consider two train routes that share the same bridge
- Unless access to the shared resource is controlled, disaster may ensue.

Race Conditions

- A solution:
  - Before crossing the bridge, trains acquire a token
  - After crossing the bridge, trains relinquish the token

Race Conditions in Software

- Remember: objects are shared by threads
- In Java, access to an object’s methods is uncontrolled, by default!!!!
- Suppose we have
  ```java
  public class Counter {
      private int count = 0;
      public void increment() {
          ++count;
          System.out.println(count);
      }
  }
  ```

Race Conditions in Software

- Different threads can share the same Counter
  ```java
  public class CounterThread extends Thread {
      private Counter counter;
      CounterThread(Counter c) {
          this.counter = c;
      }
      @Override public void run() {
          for( int i=0 ; i<10 ; ++i ) {
              counter.increment();
          }
      }
  }
  ```
Race Conditions in Software

Execute the following:

```java
public class CounterMain {
    public static void main(String[] args) {
        Counter c = new Counter();

        // Threads p and q share the same counter
        CounterThread p = new CounterThread(c);
        CounterThread q = new CounterThread(c);

        p.start();
        q.start();
    }
}
```

Possible Result:
1
2
3
4
5
6
7
8
9
10
11
WTF?

The reason is that the increment operation results in multiple bytecode (low-level JVM) instructions that can get interleaved

Focus only on `++count`

```java
public class Counter {
    private int count = 0;
    public void increment() {
        ++count;
        System.out.println(count);
    }
}
```

The statement `++count` results in the following bytecode

```
load count to r0
r0 ← r0 + 1
store r0 to count
```

r0 represents register 0
Race Conditions in Software

- Two threads invoke increment at about the same time
  - (Recall: Registers are local to the thread.)
- A “race condition” results.

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>count</th>
<th>r0 (in p)</th>
<th>r0 (in q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>load count to r0</td>
<td>load count to r0</td>
<td>41</td>
<td>41</td>
<td>41</td>
</tr>
<tr>
<td>r0 ← r0 + 1</td>
<td>store r0 to count</td>
<td>42</td>
<td>42</td>
<td>42</td>
</tr>
</tbody>
</table>

41 + 1 + 1 = 42? Of the two increments, one was lost.

Race Conditions: Another Example

Consider transfers on an account

```java
class AccountManager {
    private Account savings;
    private Account chequing;

    public void transferToSavings(int amount) {
        int s = savings.getBalance();
        int c = checking.getBalance();
        savings.setBal(s + amount);
        chequing.setBal(c - amount);
    }
}
```

Two threads execute transfers.

Race Conditions : Another Example

<table>
<thead>
<tr>
<th>One Thread (amount = 500)</th>
<th>Another Thread (amount = 1000)</th>
<th>sav</th>
<th>chq</th>
</tr>
</thead>
<tbody>
<tr>
<td>s = savings.getBalance()</td>
<td>s = savings.getBalance()</td>
<td>4000</td>
<td>6000</td>
</tr>
<tr>
<td>c = chequing.getBalance()</td>
<td>c = chequing.getBalance()</td>
<td>4000</td>
<td>2000</td>
</tr>
<tr>
<td>savings.setBal(s+500)</td>
<td>savings.setBal(s+1000)</td>
<td>5000</td>
<td></td>
</tr>
<tr>
<td>chequing.setBal(c-500)</td>
<td>chequing.setBal(c-1000)</td>
<td>5000</td>
<td></td>
</tr>
</tbody>
</table>

I started with $6000 and ended with $5000. This is not good.

synchronized to the rescue

Methods may be declared synchronized

```java
public class Counter {
    private int count = 0;

    public synchronized void increment() {
        ++count;
        System.out.println(count);
    }
}
```

© 2003–09 T. S. Norvell
Memorial University
Threads. Slide 25

© 2003–09 T. S. Norvell
Memorial University
Threads. Slide 26

© 2003–09 T. S. Norvell
Memorial University
Threads. Slide 27

© 2003–09 T. S. Norvell
Memorial University
Threads. Slide 28
synchronized to the rescue

- Each object has an associated token called its lock.
- At each point in time, each lock either is owned by no thread or is owned by one thread.
- A thread that attempts to acquire a lock must wait until no thread owns the lock.
- After acquiring a lock, a thread owns it until it relinquishes it.

synchronized to the rescue

- When a thread invokes a synchronized method \( x.m() \):
  - If it does not already own the recipient’s \( x \)'s lock,
    - It waits until it can acquire the lock
  - Once the lock has been acquired the thread begins to execute the method
- When a thread leaves an invocation of a synchronized method:
  - If it is leaving the last synchronized invocation for that object
    - It relinquishes the lock as it leaves

synchronized to the rescue

- Hence, for any object \( x \), at most one thread may be executing any of \( x \)'s synchronized methods.

synchronized to the rescue

Example: Two threads invoke \( c.increment() \) at about the same time.
**synchronized** to the rescue

<table>
<thead>
<tr>
<th>Thread p</th>
<th>Thread q</th>
</tr>
</thead>
<tbody>
<tr>
<td>request lock on object c</td>
<td>request lock on object o</td>
</tr>
<tr>
<td>acquire lock on object c</td>
<td>acquire lock on object o</td>
</tr>
<tr>
<td>load count to r0</td>
<td>load count to r0</td>
</tr>
<tr>
<td>r0 ← r0 + 1</td>
<td>r0 ← r0 + 1</td>
</tr>
<tr>
<td>store r0 to count</td>
<td>store r0 to count</td>
</tr>
<tr>
<td>relinquish lock on object o</td>
<td>relinquish lock on object o</td>
</tr>
</tbody>
</table>

The “Room metaphor”

- When a thread not in the room calls a **synchronized** method, it enters an antechamber.
- There it waits until the room is empty.
- When the room is empty, at most one thread is allowed to enter the room.

Design rule: Shared Objects

- For any object that might be used by more than one thread at the same time
  - Declare all methods that access or mutate the data **synchronized**
    - (Note: private methods are exempted from this rule, as they can only be called once a nonprivate method has been called.)

  Constructors need not (and can not) be **synchronized**

Accounts

- In AccountManager add synchronized to all methods
  ```java
  class AccountManager {
      private Account savings ;
      private Account chequing ;

      public synchronized void transferToSavings( int amount ) {
          int s = savings.getBalance() ;
          int c = checking.getBalance() ;
          savings.setBal( s+amount ) ;
          checking.setBal( c-amount ) ; } ... }
  ```