Concurrent and Distributed Systems 2017-2018
Dr Robert N.M. Watson

PRACTICAL EXERCISES - Concurrency in Java

Java provides a broad array of concurrency primitives in order to ease the writing of concurrent programs. This document provides a quick high-level summary of those facilities, and then a set of practice exercises intended to help you gain familiarity with Java concurrency. These are supplemented by exercises you will find in the Further Java course, which includes a section on concurrent programming.

You can learn more about Java and concurrency by reading the Concurrency lesson in the Java Essentials tutorial:

http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

Section 1. Synopsis of Java concurrency primitives

Processes

UNIX processes execute programs loaded from the filesystem; each executing Java Virtual Machine (JVM) occupies a process. Inter-Process Communiation (IPC) between processes can be used to gain concurrency.

Threads

Units of execution within a JVM process -- associated with an execution context including a stack. JVM threads execute compiled Java code.

Java's Thread class may be used to create and manage threads. One way to start a new thread executing code in an object of your choice is to have an object implement the Runnable interface and a method run() that contains the code to execute in the new thread. You can then pass the object to a Thread constructor and invoke the Thread's start() method to cause the thread to be scheduled. For example:

```
public class MyThreadedCode implements Runnable {
  public void run() {
    System.out.println("Runs in a thread.");
  }
}

...
  Thread t = new Thread(new MyThreadedCode());
  t.start();
...
```

It is sometimes desirable to await the termination of one or more threads; you can use the thread object's join() method to do this:

```
t.join();
```

Synchronised methods

Each Java object is associated with an "intrinsic lock"; synchronised methods implement monitor semantics, acquiring the intrinsic lock when invoked, and dropping it on return. The intrinsic lock provides mutual exclusion, but also creates a "happens-before" relationship between sequential acquisitions. You can declare a method as synchronised using the "synchronized" keyword:

```
public class atomicFooCounter {
  private int foo;

  public synchronized void incrementFoo() {
    foo++;
  }
}
```

Synchronised methods can be invoked recursively: invoking one synchronised method from another will "recurse" on the lock rather than blocking waiting for it to become free.

Synchronised statements

Synchronised statements explicitly acquire an object's intrinsic lock around a code fragment; this allows methods on one object to synchronise data access with respect to its own, or another object's, lock. This might be used to embed multiple locks in a single object in order to provide finer-grained locking of its members, addressing contention problems. Synchronised statements likewise use the "synchronized" keyword, only with an argument specifying the lock object to use:

```
synchronized(foo) {
  // Statements within this block are synchronised using foo's lock.
}
```

You can synchronise on the current object by specifying "this", but also on other objects, including private members within the current class. For example, you might declare three locks (La, Lb, Lc) to protect three different pieces of data (Da, Db, Dc), synchronising explicitly on the locks around any access to their respective data.

Atomic access

Simple integer and reference reads and rights are atomic -- you will not get back half of a new value and an old value.

Further, you can declare variables (including integers and references) as "volatile", which provides atomicity for longer types (e.g., long and double) but also ensures a "happens-before" relationship with subsequent reads and writes of the same variable -- and other memory operations leading up to that operation will likewise be visible with respect to the same event. As non-volatile accesses do not imply "happens-before", accesses without use of another primitive implying "happens-before" -- e.g., synchronized() -- may not appear in program order (or at all). If you plan to use atomic reads and writes without explicit synchronisation, "volatile" is always recommended.

Note that even simple arithmetic operations will not be atomic; e.g., "v++" performs an atomic read, increment, and atomic write, rather than an atomic read-modify-write. Protecting compound operations requires (careful) use of primitives such as synchronized().

Guarded blocks

Java's monitor primitive not only associates an intrinsic locks with each object, but also a condition variable. Conditions may be waited for using "wait()", and signals issued using "notify()" or "notifyAll()". When invoking wait(), the intrinsic lock on the object in question must be held or an exception will be thrown -- the simplest way to ensure this is to call wait() only when within a synchronised method or statement.

```
synchronized (obj) {
  while (!condition) {
    obj.wait();
  }
}

synchronized (obj) {
  obj.notify();
}
```

Higher-level concurrency primitives

Java contains an extensive library of higher-level concurrency primitives and design patterns relating to thread pools, barriers, queues, and more -- built on the lower-level primitives described above. Where higher-level primitives exist, using them is almost always preferable! The API reference for java.util.concurrent can be found here:

http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html

Section 2. Concurrency exercises

1. Creating and joining threads

1.a. Write a short program that prints "Hello world" from an additional thread using the Java Thread API.

1.b. Now modify the program to print "Hello world" five times, once from each of five different threads. Ensure that the strings are not interleaved in the output.

1.c. Now modify the printed string to include the thread number; ensure that all threads have a unique thread number.

2. Simple synchronisation

2.a. Write a short program in which two threads both increment a shared integer repeatedly, without proper synchronisation, 1,000,000 times, printing the resulting value at the end of the program. Run the program on a multicore system and attempt to exercise the potential race in the program.

2.b. Now modify the program to use "synchronized" to ensure that increments on the shared variable are atomic.

3. Guarded blocks

3.a. Write a short program in which one thread increments an integer 1,000,000 times, and a second thread prints the integer -- without waiting for it to finish.

3.b. Now modify the program to use a condition variable to signal completion of the addition task by the first thread before the second thread prints the value.

4. More complex constructions

```
 4.a. We have seen several examples of producer-consumer implemented using a
      number of different synchronisation primitives in pseudo-code.
      Implement a ProducerConsumer class using synchronized, wait(), and
      notify() in Java, and use it to pass a sequence of integer values from
      one thread (the producer) to a second that prints them (the consumer).

 4.b. Semaphores are a widely used synchronisation primitive -- but not one of
      the fundamental primitives provided by Java.  Implement a counting
      semaphore using synchronized(), wait(), and notify() in Java.

 4.c. Deadlocks are an inherent problem in concurrent systems using locks or
      other blocking primitives.  Implement a deadlock involving two threads
      and two locks in Java.  What debugging tools does the Java environment
      offer that might help us debug this deadlock?
```