School of Computing and Information Systems
The University of Melbourne

SWEN90004 Modelling Complex Software Systems

Concurrency Workshop 2

Java monitors and semaphores

Plan

The focus of this week's workshop is to explore some more of Java's concurrent features and experiment with concurrent processes.

To begin, download the workshop code from the LMS. Unzip the source, change into a suitable directory and compile the files (`javac *.java`).

1 Java Monitors

Background: Every Java object has a monitor associated with it. Declaring a method as `synchronized` means that a thread must obtain the lock on that object's monitor before the synchronised method can be run. This prevents two threads from simultaneously accessing and executing code in a synchronised method.

Aim: The aim of this exercise is to use Java's monitor functionality to synchronise the behaviour of two threads.

Steps:

1. The file `UseSynchronise.java` contains implementations of two threads, `P` and `Q`; excerpts from both are shown below. The variable `s` is a shared instance of the `Synchronise` class; beyond defining an empty method `synch()`, this class currently does nothing.

```
class P extends Thread {

  public void run() {

    while (true) {
      task1p();
      s.synch();
      task2p();
    }
  }
  ...
```

```
class Q extends Thread {

  public void run() {

    while (true) {
      task1q();
      s.synch();
      task2q();
    }
  }
  ...
```

   The intent is that process `P` repeatedly executes tasks `1p` and `2p`, and `Q` tasks `1q` and `2q`. However, for each iteration of the loop, `P` cannot start task `2p` until `Q` has finished task `1q`, and similarly, `Q` cannot start task `2q` until `P` has finished task `1p`. That is, for each iteration of the loop, `P` and `Q` must synchronise after tasks `1p` and `1q`.

2. Compile the classes in the file and run the `main` method in the `UseSychronise` class (included in `UseSynchronise.java`). Note the ordering of the events.

3. Also included in `UseSynchronise.java` is the skeleton of the monitor below.

```
class Synchronise
{
  // any useful variables go here

  public synchronized void synch()
  {
    // the code to synchronise goes here
  }
}
```

4. Complete the definition of the monitor `Synchronise` so that the above algorithm behaves correctly.

2 Java Semaphores

Background: Last week's lecture discussed a semaphore implementation of a *bounded buffer*. A bounded buffer maintains a fixed number of "slots". Items can be inserted into and removed from the buffer. The buffer has a maximum size.

A bounded buffer is useful as a way of making producer-consumer interaction more robust (and also more smooth when transmission rates vary but the producer and consumer run more or less at the same speed). For example, a video player does not read each frame over a network when it needs it. Instead, it reads many more than it needs, so that if the network connection drops, even for a small time interval, it can consume the buffered frames.

Aim: The aim of this exercise is to modify a semaphore implementation of a bounded buffer so that it behaves correctly.

Steps:

1. The file `UseBuffer.java` contains an implementation of a bounded integer buffer that is broken. There are three methods: `put(int input)`, which inserts an integer to the end of the buffer if the buffer is not full; `get()`, which gets the integer at the front of the buffer; and `size()`, which returns the current number of items in the buffer.

   The classes `Producer` and `Consumer` define respective threads that produce integers at random intervals, and consume integers at random intervals. The class `UseBuffer` simply starts `Consumer` and `Producer` threads with references to the same shared instance of a bounded buffer.

2. Compile and run the `main` method of the `UseBuffer` class.

   The bounded buffer class is broken for two reasons:

   (a) when the thread is full, it ignores new inputs;

   (b) when the thread is empty, it still tries to take the front element of the buffer.

3. Modify the `BoundedBuffer` class so that it works correctly. First, do this using `wait()` and `notifyAll()`, as used in Exercise 1.

Extra: Create a new copy of `UseBuffer.java` and modify it to create a correct implementation using Java's `Semaphore` class (from `java.util.concurrency`). Hint: you should use three semaphores... two to track whether the list is empty/full, and one to ensure mutual exclusion.