School of Computing and Information Systems
The University of Melbourne

SWEN90004 Modelling Complex Software Systems

Concurrency Workshop 1

Java threads and the mutual exclusion problem

Plan

The focus of this week's workshop is to explore 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 Concurrent Quicksort

Background: Quicksort is the archetypal divide-and-conquer algorithm—we partition an array's elements according to size, relative to a chosen *pivot* element. The segment that contains elements smaller than the pivot are then sorted recursively, and so is the segment with the larger elements. With some luck, the two segments are of similar size, in which case we obtain an $O(n \log n)$ algorithm, very fast in practice. Even better, quicksort lends itself naturally to parallelism.

Aim: The aim of this exercise is to modify the provided implemention of quicksort to make it concurrent.

Steps:

1. The file `SeqQuick.java` contains a sequential implementation of quicksort. Copy `SeqQuick.java` to a new file called `ConQuick.java`.

2. Create a new sorting thread after each partitioning step. This new thread should then be given the task task of sorting the lower segment (say), while the parent thread concentrates on sorting the higher segment.

3. Specifically, you will need to make the sorter class extend `Thread`, and you will need to change the `quicksort()` method. However, you should not have to change the partitioning code at all, or touch the insertion sort.

Notes: If the standard setup allocates insufficient space for Java, you can ask for more heap space like so: `java -Xmx4g`. Independently of space issues, you may sometimes find that concurrent sorting needs to be curbed, so as to not generate an excessive number of threads.

Extra: If you run the program under Unix, use the `time` command to profile the time taken for both the sequential and concurrent quicksort implementation. For that part you will want to increase the array size and disable the printing.

2 The Mutual Exclusion Problem

Background: In last week's lecture we discussed the mutual exclusion problem: how to ensure that only a single process in a concurrent system executes statements in its critical section at any time. We described a series of solution attempts, culminating in Dekker's algorithm.

Aim: The aim of this exercise is to apply each of these solution attempts to the (originally unsafe) shared counter example from the lecture, and to explore the conditions under which these solution attempts fail or succeed.

Steps:

1. The file `Count.java` contains an unsafe implementation of the shared counter example. In this example, the critical section of code in each thread updates a shared variable `counter`. Each of the two threads attempts to increment the value of the counter N times. At the conclusion of the execution, the final value of the counter should therefore be $2N$. Run this code enough times to observe it producing "incorrect" output. Try modifying N to be much larger (eg, 1000). What do you observe?

2. Copy `Count.java` to a new file named `Attempt1.java`. The source code contains comments indicating where protocol variables and code should be inserted. Create a shared variable `turn` (indicating which thread's turn it is to enter the critical section) and associated pre- and post-protocol code as per lecture Con.02. Try running this solution attempt and observe its behaviour. Does it produce the expected output?

3. Create additional copies of `Count.java` and modify them to implement the second, third, fourth and fifth (Dekker's algorithm) attempts at a solution to the mutual exclusion problem. Do they produce the expected output? Try inserting `System.out.println` statements to identify where (and why) each attempt fails. It may be helpful to give each thread a unique identifier string when doing this. In some cases, you may have to add code to the non-critical section in order to bring about the failure conditions described in the lecture.

Notes: It is possible that even the "correct" solution (ie, Dekker's algorithm) will not always work correctly if your protocol variables are not declared as `volatile`. In brief, this is due to optimisations that the Java compiler makes which may be correct for sequential code, but which are not necessarily correct for concurrent code.

(Optionally, more detail on this problem can be obtained from this web page: `https://www.enseignement.polytechnique.fr/informatique/INF431/X13-2014-2015/TD/INF431-td_3a-1.php`)

Implementing these solutions to the mutual exclusion problem should highlight some of the difficulties encountered when writing and debugging low-level concurrent code. Fortunately, Java provides some high-level constructs for writing concurrent code, which we will discuss in this week's lectures.

Extra: Implement Peterson's solution to the mutual exclusion problem (also from lecture).