CAB302 Software Development

Semester 1 2017

Practical 10 — Understanding Threads

Exercise 1 – Some Basic Threadwork

Change some of the data and times in ScheduleTasks to see the effect of the Timer class

The larger the number the longer the time it takes to print out. The conversation 'stops' when the time in the Thread.sleep method is reached.

Alter the delay times on the messages in JumbledMessage to see the effect, in both the unsynchronized and synchronized versions.

In order to see a change you should set a big differences in the delay times (i.e set one to 10 and one to 1000) . In synchronised version the delay acts as you would expect (i.e. just delays the time that a message is printed). In the unsynchronised version the longer delay means that the message is more likely to be printed unjumbled – but certainly doesn't guarantee it.

Exercise 2 – A Question of Priorities

The code is available in the Java file RunThreeThreads

Run the programme several times and note the behaviour. Check the results when you vary MAX from say 20 to 10 and up to 50. Are consistent patterns observable?

Generally the thread started first gets the lion's share of the time slices early on, and so tends to complete earlier than the others. But this is not guaranteed, and some swaps may be observable. Nothing can be taken for granted.

What is happening? Relate your answer not merely to the actual priorities being assigned, but also to their relative magnitudes.

The priorities are equal, so the chances of a swap depend on the time that each thread actually takes to execute.

Now change the priority of one and three to Thread.MIN_PRIORITY. Note that this time there is one clear winner in the priority battles. Is this respected by the JVM?

No, not really. More consistency is observed for larger values of MAX. In practice we see the higher priority thread being 'largely' scheduled earlier, but this is not at all guaranteed. It is more likely for larger values of MAX that a majority of the relevant cases will be executed early.

Try additional variations, using a random number generator and/or the sleep() method to schedule changes to thread priorities. Try to explain your results.

According to the specification, a major difference in thread priority should result in pre-emptive scheduling: i.e. we dump the low priority thread and replace it with another of higher priority. Do we always see this behaviour? If not, what is happening and can we fix it?

Using explicit thread sleeps and joins is the only way of ensuring reproducible behaviour. The thread scheduler will almost never be free of OS-dependent influences.

Exercise 3 – Dining Philosophers

The Philosophers class is the driver class for the problem and all three classes were originally given for the lecture. Fork has had an accessor added to determine if the Fork is available for pick up. The run method in Philospher has been altered to call one of runOriginal() [the code given in the lecture], runRandom() [implements random selection of the first fork to pick up] or randomCheckFirst() [checks to see if the second fork is available].