COMP25111 Lab Exercise 2: Scheduling

Duration: 2 sessions

Aims

To give you some insights into process scheduling and into semaphores, and some practice in the use of threads in Java.

Learning outcomes

On successful completion of this exercise, a student will:
- Have modified an illustrative non-preemptive scheduler
- Have implemented semaphores to allow threads to rendezvous at a barrier

Summary

Using Java, and in particular using Java threads to simulate processes,

1. implement different scheduling algorithms by writing `RoundRobinScheduler.java` and `PriorityScheduler.java` (starting from `RandomScheduler.java`) for use by the `TestScheduler.java` program, and
2. complete the implementation of semaphores in `Semaphore.java` for use by the `TestBarrier.java` program.

There are very brief descriptions of scheduling and semaphores below. For more information, consult the course textbook(s).

The unextended deadline is the end of your (second) scheduled lab session.
If you attend the lab you will, if you need it, automatically get an extension until one week after your (second) scheduled lab session commences - you must use submit to prove you finished in time.
You must get your work marked in this lab or in your next scheduled lab.
You can also get an extension for good reason e.g. medical problems.

Description

For this lab exercise you should do all your work in your `COMP25111/ex2` directory.

Copy the starting (Java) files into your `COMP25111/ex2` directory from `/opt/info/courses/COMP25111/ex2`
These files are (for part 1): `TestScheduler.java`, `Scheduler.java`, `RandomScheduler.java`, `SimProc.java`
(for part 2): `TestBarrier.java`, `Semaphore.java`, `SemScheduler.java`, `SemProc.java`

Introduction

The scheduler used in an operating system is not an easy thing on which to experiment in a modern networked PC. Instead this exercise will modify a Java class which acts like a scheduler to "processes" which are simulated by Java threads.

Unlike real processes, these threads do not get timesliced - but instead relinquish the processor at points of their own choosing by calling a method on the scheduler. Thus we are doing non-preemptive scheduling.

The threads are also responsible for introducing themselves to the scheduler when they start, and removing themselves when they finish. In a real system, the Operating System ensures that these things are done without the process calling the scheduler - but this artificiality does not really change how the scheduler itself operates.

Part 1: Alternative scheduling stategies

You are provided with an example scheduler in `RandomScheduler.java` (a sub-class of `Scheduler.java`) which chooses randomly which "process" to run next from those that are available.

The program `TestScheduler.java` currently uses `RandomScheduler` to run a number of simulated processes (described below). Start by running it as it stands, to get some understanding of the set-up.

Read the code you are given for hints to write two different schedulers (variants of `RandomScheduler`):

`RoundRobinScheduler.java`
: which uses round-robin scheduling instead of random. (Round-robin scheduling should be preemptive, but the simulation doesn't use that. Instead, we get the same effect because the processes that are being scheduled explicitly call the scheduler's `endslice` method - see "Details of the Simulated Processes" below.) scheduling should be preemptive, but the simulation doesn't use that. > Instead, we get the same effect because the processes that are being > scheduled repeatedly do some work and then explicitly call the scheduler's > `endslice` method - see)
Note: `RoundRobinScheduler` needs to get a different process from the list each time around the loop. If you simply increment an index, you might skip a process when the previous process finishes. (Why? If you don't know, try it and see what happens.) Don't just check the length of the list. (Why? You might get away with it in this simple simulation, but what happens in a more general situation?) Instead, you need to find a well-engineered way to do whatever is necessary each time a process enters or leaves the list.

`PriorityScheduler.java`
: which does scheduling among processes using "priorities", where priority is a property of the process which can change as it runs (see `getCount` in the description of the Simulated Processes below). If there are several processes with the same, highest priority, it doesn't matter which is run.

Modify `TestScheduler.java` to run with your new schedulers to demonstrate their working.

Details of the Simulated Processes

The simulated processes are instances of `SimProc.java`, and should not be changed. Each process has a name, and computes a number of values of the `collatz` function.

(*It doesn't matter what the `collatz` function computes - but you can see from the code that it is quite simple. The interesting fact about it is that the loop terminates for all known arguments if large enough intermediate values can be handled on the way - and there is no known proof of this termination in general. This is known in the literature as the "3x+1" problem, but also as the Collatz problem, the Syracuse problem, Kakutuni's problem, Hasse's algorithm, and Ulam's problem!*)

The process continues to call the `collatz` function, relinquishing the processor after each call, until it has computed 4 non-negative results for random inputs. Thus the number of time-slices required by each process is not easily determined.

When using the priority scheduler on these processes, the priority should be the result of calling the `getCount()` method. This returns an integer showing how many non-negative results the process still needs to compute.

Part 2: Semaphores and Barriers

You are given a program which runs several "processes" and uses semaphores to provide "barrier synchronisation" - i.e. all the processes should wait for the last to arrive each time around a loop before proceeding. However, in the code you are given, the method bodies for the `P()` and `V()` semaphore operations are empty, so each process, once started, will run to completion before the others run.

To get some understanding of the set-up, start by running `TestBarrier.java` (which uses "processes" provided in `SemProc.java` and a scheduler in `SemScheduler.java`) without modifying the semaphore code in `Semaphore.java`.

Barriers

A barrier is a mechanism whereby all the threads that reach it stop and wait for the last to arrive. They then all proceed and the barrier is re-initialised to hold up the next batch of threads to reach it (typically the same threads, each executing a loop containing the barrier). This way, one thread does not get too far ahead of any other - regardless of the underlying scheduling mechanism.

You can see the barrier code, which uses 2 semaphores, in the method `reachBarrier()`. **You do not need to understand how the barrier is implemented from semaphores to do this exercise.**

Semaphores

Although Java uses a rather different mechanism for synchronization between threads, it is still possible to implement semaphores in this context. Below is some pseudocode for the operations on a general semaphore called "sem", which has a value and a queue of threads which are waiting. The trick it uses is that negative values of the semaphore are used to count how many threads are queuing - this simplifies the logic and is quite standard.

```
P():
    decrement sem.value ;
    if (sem.value < 0) {
       this thread needs to stop running and join the queue, so move
       current thread from the scheduler's ready queue to sem.queue
    }

V():
    increment sem.value ;
    if (sem.value <= 0) {
       there is a thread to awaken from the queue, so move the front
       thread from sem.queue to the scheduler's ready queue
    }
```

Modify `Semaphore.java` to implement the methods `P()` and `V()` as above. **You must not modify any other code in this class, or any code in the other associated classes.** In translating this into Java, be careful to be thread-safe, i.e. to ensure that threads cannot accidently interfere with the operation of P() and V() by other threads. Re-run `TestBarrier` to see the results.

Assessment

You must use `labprint` and `submit` as normal. They will look for: `RoundRobinScheduler.java`, `PriorityScheduler.java` and `Semaphore.java`

The marks are awarded as follows:
Part 1:
4 - Correct Round Robin scheduler
4 - Correct Priority scheduler
Part 2:
8 - Correct operation of semaphores
Both parts:
4 - Sensible style of code
Total 20

Remember to get your solution marked as soon as possible after it has been submitted.