CONCSYS - ASSIGNMENT 1

Upload your solution on ILIAS as a ZIP or TAR.GZ including all your files (sources, text file with explanations, ...). In case of questions, please address them during the lab or by email to yaroslav.hayduk@unine.ch

Deadline: 9:00 AM, March 10, 2015

Exercise 1

1. Write a java program that spawns n and m threads of two different types as described below, where n and m are program arguments. These threads access a shared counter (initialized as 0) in a loop (100 000 iterations). In each iteration, the threads of the first type read the counter to a local (stack) variable, increment it, and store it back to the counter. The threads of the second type are performing the same set of operations but instead of incrementing the counter are decrementing it. When all threads finish, the program prints the value of the shared counter and the execution duration. (Try to check the value for equal number of incrementing and decrementing threads) Save the source to Ex1NoSync.java.
2. Modify this program using the keyword synchronized to protect the counter. The counter must be 0 at the end of the execution. Save your code to Ex1Sync.java.
3. Modify the first program to use java.util.concurrent.locks.ReentrantLock class. The counter must be 0 at the end of the execution. Save your code to Ex1ReentrantLock.java.
4. Run the 3 programs with equal sets of 1, 2, 4, and 8 threads and report the results in a textual table in a file Ex1.txt. Please report run time in ms, calculate the speedup (Ex1NoSync as reference) and your machine specification (processor specs and operating system - e.g: the result of cat /proc/cpuinfo for linux OS)

Exercise 2

The dining savages problem considers the next situation: A tribe of savages eats meals from a single large pot that has a capacity of N portions. When a savage eats, he takes a portion from the pot if the pot still has at least one available. If the pot is empty the savage orders the cook to refill the pot and waits until this is full again. The cook does only full refills of the pot (N portions). To summarize the constraints: the savages can't take a portion from the pot if the pot is empty and the cook can't refill the pot unless the pot is empty.

Idea → two primitives : eat, refill 1 => binary semaphores (mutex)

but not fully correct

eat :=
lock.wait()
if pot > 0
pot --
lock.signal()

not sure that is really eaten

refill :=
lock.wait()
if pot = 0
pot ...
lock.signal()

→ cook has to inform using semaphore

→ Consumer / Producer (on Wiki → Semaphore) is a good starting point

1. Write a program (Ex2Savages1.java) to simulate the behavior of the savages and the cook, where each one of them is a thread and the pot is a shared resource, respecting the constraints above. Consider that each savage wants to eat only one meal, but the number of savages should be higher than the capacity of the pot, so the pot will need refillings. Try to write a short description (Ex2.txt) of the reasoning you used as an informal proof for the synchronization constraints required.

2. Considering that the savages are always hungry (the threads are continously looping trying to take another portion from the pot after they eat) amend your program (Ex2Savages2.java) so that every savage will eventually eat from the pot (hint: think on a way to make the execution fair, so one savage will not eat more often than another). The number of savages is fixed, and it is known by every one of them. Add to the previous short description (Ex2.txt) some ideas about the reasoning you used, as an informal proof for the no starvation constraint required.

Some tips and notes

- You are not allowed to use in your solutions synchronization mechanisms available in Java that were not discussed during the lab (specifically Atomic classes, as in *AtomicInteger*, etc)
- You should not rely for the solution on Exercise 2.2 on the fairness implementation provided by the Java classes (e.g. using a Semaphore with fairness support initialized from the constructor)
- For exercise 2.2 you can think on a mechanism that forbids one savage thread to "eat" again (start another iteration over the "pot") until all the other savages have "eaten" for the current round.
- Use System.nanoTime() before and after the execution to measure the run time.
- Java API documentation : http://java.sun.com/javase/6/docs/api.
- The names of the files are just suggestions for better organizing the solutions. You can obviously split your code in multiple files as you think it is fit, but in this case is recommended to add a note to the submitted assignment stating what and were was solved.