Concurrent Programming

COMP 409, Winter 2021

Assignment 2

Due date: Tuesday, February 23, 2021
6pm

General Requirements

These instructions require you use Java. All code should be well-commented, in a professional style, with appropriate variables names, indenting, etc. Your code must be clear and readable. **Marks will be very generously deducted for bad style or lack of clarity.**

There must be no data races: all shared variable access must be properly protected by synchronization. Any variable that is written by one thread and read or written by another should only be accessed within a synchronized block (protected by the same lock), or marked as volatile. At the same time, avoid unnecessary use of synchronization or use of volatile. Unless otherwise specified, your programs should aim to be efficient, and exhibit high parallelism, maximizing the ability of threads to execute concurrently. Please stick closely to the described input and output formats.

Your assignment submission **must** include a separate text file, `declaration.txt` stating "This assignment solution represents my own efforts, and was designed and written entirely by me". Assignments without this declaration, or for which this assertion is found to be untrue are not accepted. In the latter case it will also be referred to the academic integrity office.

Questions

1. Suppose you use multiple threads to move checker pieces around an 8×8 checkerboard. To avoid conflicts, each individual square is protected by an associated lock, using blocking synchronization to ensure that the contents of that square can only be modified by one thread at a time. **15**

   A checker in this simulation can move either by sliding to an empty square diagonally adjacent, or by capturing other checkers, which it does by jumping diagonally over an adjacent checker, provided the square it jumps over is occupied and the square it jumps to is empty. Note that this is a single-player simulation that only models the movement/capturing mechanics of the actual checkers game—there is only one kind of checker (no colours, no kings), and movement can be in any direction.

   Build a multi-threaded simulation of this. First, populate the board with $2 \leq t < 32$ checkers at random locations, all on the same colour square, and not overlapping. Then launch t threads, each controlling one of the checkers.

   Each thread sleeps for m milliseconds and then attempts to move its checker in one of the 4 diagonal directions, either a simple (if the square is unoccupied), or capturing (if possible). The direction should be randomly chosen, but you may need to check more than one direction if a move is not possible (if no move is possible after checking the 4 diagonals then just wait sleep and try again).

   Both the capturing and moving actions need to guarantee the state of any affected squares in a given diagonal direction is not changed by other threads/checkers while it tries to move by ensuring exclusive accesss to each relevant square. Threads that do not conflict must, however, be able to move concurrently. If a checker is captured, the thread managing that checker waits $2m$–$4m$ milliseconds, and then repeatedly tests random squares (of the same original colour) to find and place its checker back on the board, while avoiding any potential overlap. After successfully moving or being captured or replacing its checker on the board, a thread prints out to the console a message, formatted like:

```
Tx:  moves to (x,y)
Tx:  captures (x,y)
Tx:  captured
Tx:  respawning at (x,y)
```

Where x is a thread id, and coordinates are based on the origin $(0, 0)$ in the bottom-left corner. Note that you must only using blocking synchronization (monitors, locks, semaphores)—all attempts to acquire a lock must proceed until the operation succeeds—no spinning!

Provide a Java program `checkers.java` that takes three integer command-line parameters t, k, and n, where n is a count of the number of moves to be made by thread; once a thread has made n moves or captures, it exits. Your solution should allow for maximal concurrency—operations should not be serialized unless necessary. Your program should work correctly and terminate for all reasonable values/combinations of parameters, but to test it choose a small value for k, let $t = 16$, and let the simulation run for $n \geq 200$.

Include a sample of the resulting output with your submission, along with a text file `deadlock.txt` which briefly explains how you guarantee your program does not deadlock.

2. A break-out room is created as a space shared by students within the faculties of Arts, Science, and Engineering. Students in each faculty may use the room, individually deciding whether to enter or leave once inside, but at any given time all the students in the room must be from the same faculty, so if the room is occupied by one faculty's students then students from other faculties have to wait outside until it is free. It is also important that the room be fairly shared; no faculty's students should be starved from entering the room.

   Model this using 12 threads, with 4 threads per faculty. Each thread repeatedly sleeps for k–$10k$ milliseconds (time randomly chosen) and then attempts to enter the room. Once in the room, the thread sleeps for a random time, w–$10w$ milliseconds, and then leaves and repeats its cycle.

   Choose reasonable k and w to ensure high room competition and let the simulation run for n seconds, keeping track of which faculty (or empty) is currently occupying the room at each point in time: each time the room "ownership" changes, print out the faculty name or `empty` to the console (each time on a separate line). After n seconds the simulation should gracefully terminate. Include a sample of the resulting output with your submission.

   Use only blocking synchronization. Your solution should accept 3 command-line parameters, n, k, and w. Provide two solutions:

   - A version called "breakoutMon.java" using monitors for synchronization. **10**
   - A version called "breakoutSem.java" using semaphores for synchronization. **10**

What to hand in

Submit your declaration and assignment files to *MyCourses*. Note that clock accuracy varies, and late assignments will not be accepted without a medical note: **do not wait until the last minute**. Assignments must be submitted on the due date **before 6pm**.

Where possible hand in only **source code** files containing code you write. Do not submit compiled binaries or .class files. For any written answer questions, submit either an ASCII text document or a .pdf file. Avoid .doc or .docx files. Images (plots or scans) are acceptable in all common graphic file formats.

This assignment is worth 15% of your final grade. **35**