PS 3 -- OpenMP and Pthreads (3 pages)

TDT4200, Fall 2017
Deadline: 12.10.2016 at 20.00 Contact course staff if you cannot meet the deadline.
Evaluation: Pass/Fail

PS 3 Part 1b -- **Do not hand in, but good to practice before final..**

Problem 1
a) What is cache memory?
b) What is the difference between spatial and temporal locality?
c) What is cache coherence?
d) What is false sharing?

Problem 2
Write code to show how semaphores can protect a critical section, where at most two threads should be allowed to execute the code in the critical section at the same time.

a) Using binary semaphores.
b) Using regular/non-binary semaphores.

You do not need to write a complete, compilable program, only the relevant lines.

Problem 3
Write code using semaphores that will always deadlock. Explain why you code causes a deadlock. You do not need to write a complete, compilable program, only the relevant lines.

PS 3 Part 2 -- Programming

- **All problem sets are to be handed in INDIVIDUALLY.**
  You may discuss ideas with max. 1-2 collaborators and note them in the comments on the top of your code file, but code sharing is strongly discouraged. **Preferably seek help from TAs rather than get unclear/confusing advice from co-students!**
- **Code must compile and run on course servers.**
- **Do not add third-party code or libraries.**

In this part of the exercise you will build on the RPS cellular automata from last exercise. This time around you're tasked with implementing two versions of RPS:

- one using OMP and
- one using Pthreads

In contrast to the MPI version, the threaded version is much simpler to implement due to the shared memory model employed by threading. Additionally, we have supplied a serial version that is more similar to the version you will develop in order to lessen the burden of porting. While porting code is a very helpful exercise, the Head TA might have gone a little over board for PS2, adding a burden that detracted from the core learning goal. This means that the functions contained in CA.c should be a lot simpler to utilize this time around, and we actively encourage you to use them rather than implementing your own as a lot of people did for PS2.

Your task is to create two programs: RPS_omp and RPS_pthread. For both programs you should parallelize the calculation for each iteration, which means you need to break down the work performed in `iterate_image`. How you do this is up to you, however, we encourage you to think about what constraints we face in multithreaded programs opposed to multi-processes programs!

Note that this time around we are including a very basic random function. rand() calls are much more expensive than we realized when making PS2, and in the PS3 solution the time spent went from ~1 minute to 4 seconds upon using the more primitive rand function. Oops.
Optional: We encourage you to come up with a more sophisticated **rand** function than the one supplied in CA.c

Hand-in

You must hand in your code as a zip file named $(your_username)_ps3 containing a folder again named $(your_username)_ps3. This makes grading easier for us.

Running **make omp** should create the **omp** executable. Running **make pth** should make the Pthread executable. Both the **Pthread** and the **OpenMP** program should take a single command line parameter upon running, stating the desired amount of threads. Both programs should produce an image upon running named RPS_pthread.bmp and RPS_omp.bmp, respectively.

Recommendation:

- Start with the OMP version, you should be able to reuse most of the code between the OMP and Pthread version, and OMP does a lot of things for you which eases the programming burden.

- Read the documentation for Pthreads and OMP. Threads are treacherous, it's easy to get lost, especially if you don't have a clear idea where you're going!

If you run out of time: To pass this exercise, we will accept a Pthread version of PS0/1, but you still need to do an OpenMP version of PS2 to get a pass.