CS 744 - Autumn 2020
Programming Assignment 2
Threads and Synchronization

1. hello pthreads!

In this section, you will use the pthread (POSIX threads) library to understand the working of threads, share data across threads and synchronization.

(a). Write a program which creates **n threads**, where n is passed as a command line argument to your program. Each of these threads would increment a shared counter variable (initialized to zero) once and then exit.
Write your code in the threads.c file present in the assignment resources folder provided to you.

(b). Write a program in the **nlocks.c** file provided in the assignment resources folder that does the following

- Creates and initializes 10 shared counters and 10 locks.
- The parent process creates 10 threads.
- Each thread adds 1 to its corresponding data value, a 1000 times. E.g., thread 0 updates data[0].
- The parent also updates the data items in the following manner—adds 1 to data[0], then to data[1] etc. for all 10 data items and then repeats for a total of 1000 iterations.
- With correct synchronization each of the data values should be 2000
- Don't forget to reap all your the threads created by the parent program

2. just a barrier away!

- A barrier for a group of threads stops execution at the point where the barrier is invoked and the threads cannot proceed until all them reach/invoke the barrier.
  (lookup testcase code snippet for example usage)

- Using the pthreads library Implement the ***barrier*** functionality using the following functions.

  barrier_init(struct barrier*, int nthreads)
  initializes the barrier structure variable, nthreads is the number of threads on which the barrier should hold.

barrier_wait(struct barrier*) wait on the barrier if all **n** threads have not reached this point of execution.

- Implementation files: **barrier.c** and **barrier.h**
- Test cases in: barrier-testcase1.c, barrier-testcase2.c . . .
- Compile using **make** and run using **make run**
- Don't change any of the testcase files.

3. Hare and Turtle

You will have to write a few cooperating threads:

I. Turtle
II. Hare
III. Randomizer
IV. Reporter

As we know, since our childhood, a hare and a turtle went for a race, and the turtle won!! **Why?** Maybe luck worked out for the turtle. Now, let's recreate the scene and assert that it is random luck (Randomizer) that decides the winner.

Input specification

1. Time unit intervals in which Reporter must report race standings (Current hare and turtle distance)
2. Turtle speed: Distance turtle moves every unit time
3. Hare speed: Distance hare moves every unit time
4. Hare sleep time: Time units for which hare rests after moving once
   (As in the story! The hare rests after every time it moves)
5. Minimum distance between hare and turtle when hare panics and moves
   (Remember! It can only move after it has finished resting)
6. Finish distance
7. Number of times Randomizer intervenes/makes an impact (N)
8. **N** lines of "player, time and distance" specifying N interventions of the randomizer
    a. Player: Decide whether to move hare or turtle
    b. Time: time unit at which this particular intervention must happen
    c. Distance: The distance by which the player has to be displaced (can be negative)

Pseudocode for the five threads involved are as follows:

```
Turtle () {
        while (no_winner) {
                Move at the specified speed every unit time
```

```
        }
}
```

```
Hare () {
        while (no_winner)
                if (turtle is far off)
                        Sleep (for the specified time units);
                else
                        Move at the specified speed for a single time unit
                        Sleep (for the specified time units)
}

Randomizer () {
        while (no_winner)
                if (time_to_reposition)
                        Select hare or turtle and reposition;
}

Reporter () {
        while (race is on) {
                Report the positions of hare and turtle in distance from the start in specific time
unit intervals given in input;
        } }

Init () {
        Create 4 parallel threads and run each of the four threads (Hare / Turtle / Randomizer
and Reporter.
        while (no_winner)
                Synchronise all four threads
        return winner and reap all threads;
}
```

Note: If hare and turtle cross the finish line at the same time unit then we'll let the turtle win, because slow and steady wins the race, in stories and in assignments!
Randomizer must intervene and reposition a player at **the beginning of a particular time unit** (ie when it's the randomizer's turn to reposition)**.**

You can communicate using shared variables. However, you will need to ensure that different threads synchronize their access to shared variables to avoid inconsistencies. In particular, you should use mutexes and conditional variables to protect all accesses to shared variables.

Tests:

- The current outcome of the race should be displayed after each specified interval of N time units
- Few test files are provided in the Testcases folder.
- Passing these test cases doesn't mean the solution is correct. Your solution will be tested against hidden test cases. Check for corner cases.
- Feel free to make use of the cs_thread.h file provided in the resources folder.
- Compile using **make**
- Run using **./hare_tortoise test-file=<path to file> or make run (**this will run your code against the test files present in the resources folder. You can add some of your own test file here to test your code against them**)**
- Make sure there are no spaces in the file path. The specification of the format for the test file is present in the resources folder in a file named test_format.txt