Homework 3 Wet

Due Date: 10/1/2020

Teaching assistant in charge:

- **Reda Igbaria**

Important: the Q&A for the exercise will take place at a public forum Piazza only. Critical updates about the HW will be published in pinned notes in the piazza forum. These notes are mandatory and it is your responsibility to be updated. A number of guidelines to use the forum:

- Read previous Q&A carefully before asking the question; repeated questions will probably go without answers
- Be polite, remember that course staff does this as a service for the students
- You're not allowed to post any kind of solution and/or source code in the forum as a hint for other students; In case you feel that you have to discuss such a matter, please come to the reception hour
- When posting questions regarding **hw3** , put them in the **hw3** folder

Only the TA in charge can authorize postponements. In case you need a postponement, please fill out the following form: https://forms.gle/tgzxrfqkfufzSK1o8

Assignment Objectives

In this assignment you will develop a simple version of a **thread pool** and utilize it to complete a workload in parallel, utilizing the synchronization techniques you have learned in the lectures and recitations. This assignment will allow you to gain experience in the following areas:

• **Threads**: This includes creating threads, joining threads, and cancelling threads. You will be using the **pthreads** library.

• **Synchronization data structures**: You will use semaphores, condition variables and mutexs to synchronize communication between an arbitrary number of threads.

• **Deadlock and Starvation:** As you develop, you will likely encounter deadlock (perhaps for the first time ever). You will need to determine why you got the deadlock, and determine a solution. The behavior on screen is similar to an infinite loop – no progress, the system is stuck. Remember - if you are ever getting deadlock, or often/always getting starvation, then your solution is wrong.

• **Parallel Debugging**: You will have to debug your code, which is parallel. This is challenging, and using a standard debugger is often not helpful. Using well-placed printf() statements in your code can help, but it is far from efficient. In the last few years many IDEs have been fitted with state of the art tools to help you with your plight. A few good examples are Eclipse, Visual Studio and ThreadSanitizer. It is your decision whether to explore them or to simply utilize prints to screen. The course staff will not provide any technical help regarding these tools.

• **Thread Safe and Not-Thread Safe**: A new concept – not all code was written to be parallel. For example, **std::cout** is not "thread-safe". This means that there is no guarantee that writes from concurrent different threads will produce the prints you expect (even in "random" order). Other mechanisms have problems as well, such as **exceptions**. It is best to be responsible, and check up on the functions you use from the standard libraries, to make sure that they act responsibly with concurrent execution.

Part 1

We are first going to prepare our basic synchronization tool set. The synchronization primitives you are allowed to use throughout the **entirety** of this homework assignment are **only:**

pthread_mutex_t , **pthread_cond_t**

You are not allowed any others: pthread semaphores, barriers, atomic variables and specifically, anything from the C++ 11 synchronization libraries. You may add any other primitives such as integers, booleans, structures and the like to your implementations as you see fit, and of course use the Semaphore and PCQueue synchronization classes.

Your tasks:

1. Create a basic semaphore by implementing the **Semaphore.hpp** API, in **Semaphore.cpp**.
2. Create a basic Single- Producer, Multiple-Consumer queue by implementing the **PCQueue.hpp** API, in **PCQueue.cpp**. You may or may not use your implementation of the Semaphore class, as you see fit.

Advice and notes on this section:

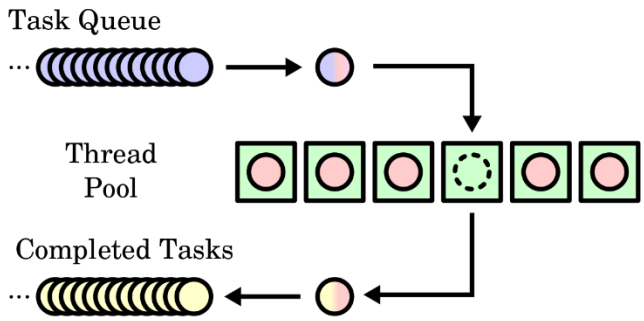
1. Seeing this is the core of your synchronization tools in the next part, we recommend you make sure your implementation **fully works** before advancing on to Part 2.

Before you start coding: Read Part 2 along with the assignment constraints and advice.

Part 2

1. Thread Pool

A **thread pool** is a design pattern where a number of threads are created to perform a number of tasks, usually organized in a Producer-Consumer queue. In real applications, the number of tasks would be much higher than the number of workers working on them, achieving constant reuse of the threads. As soon as a thread completes its task, it will request the next task from the queue until all tasks have been completed. The thread will then sleep until there are new tasks available.



2. Game of Life

The **Game of Life** (or simply Life) is not a game in the conventional sense. There are no players, and no winning or losing. Once the "pieces" are placed in the starting position, the rules determine everything that happens later.

Life is played on a grid of square cells-like a chess board but extending infinitely in every direction. A cell can be alive or dead. A live cell is shown by putting a marker on its square. A dead cell is shown by leaving the square empty. Each cell in the grid has a neighborhood consisting of the eight cells in every direction including diagonals, each cell belongs to a certain species, and each species is color-coded (see source code).

The cycle of life in this game is divided into 2 phases, the first phase is where new cells are created and some cells die :(, and in the second phase the cells get to know their lovely neighborhood and change its properties according to the neighborhood it is in.

To move from one generation to the other each cell goes through the following 2 phase process:

Phase 1:

1. A dead cell with exactly three live neighbors becomes a live cell (birth), and its species is the dominant species in the neighborhood.
2. A live cell with two or three live neighbors stays alive (survival).
3. In all other cases, a cell dies or remains dead (overcrowding or loneliness).

Phase 2:

Each live cell changes its species to be the average of the species of the alive cells in its neighborhood (including itself), i.e. : $new\ species\ = round(\frac{\sum_{c \in \{alive\ neighbours\}} c \to species}{|alive\ neighbours|})$.

Note: The number of live neighbors is always based on the cells before the rule was applied. In other words, we must first find all of the cells that change before changing any of them.

Note: The dominant species in a neighborhood of a certain location is the species with the most presence: $dominant\ species = \frac{argmax}{c \in species\ in\ neighbrhood}(number\ of\ cells\ from\ c * c)$, if two species a, b have equal presence ($(\#cells\ from\ a) * a == (\#cells\ from\ b * b)$), the dominant species is the species with the lower value representing it, i.e. : $dominant\ species = min(a, b)$

Explanation for the case of 1 species:

You may see a more detailed explanation here.

You may also see some wacky examples of what people have created with this simple set of rules here.

You may also play the game yourself with the .jar file attached to this exercise: GameOfLife.jar. This can provide a simple baseline for you to test different scenarios.

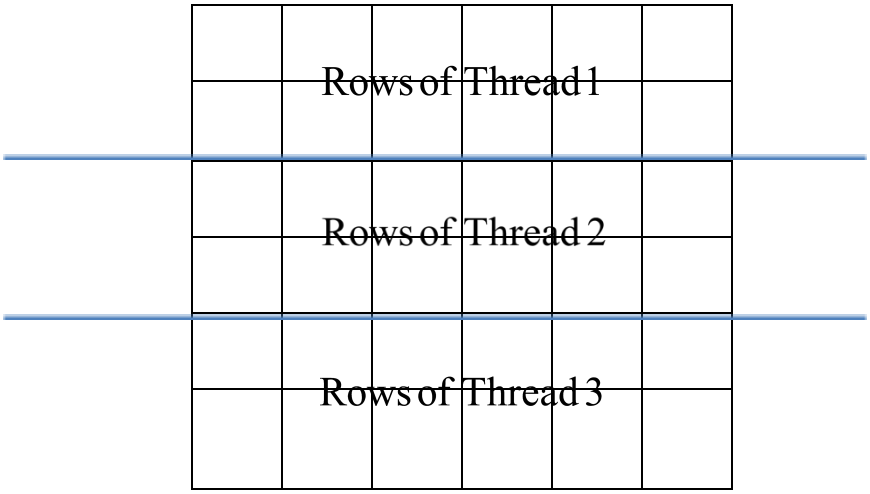
3. Parallel Game of Life

You will implement a parallel Game of Life utilizing the thread-pool pattern.

In this exercise the field size is not infinite but known and provided as an **input file**. All cells outside the field are assumed to be always dead.

The field is split between different threads so that in every **generation** (iteration), each thread works on its own rows of the field (which are split as equally as possible between all the working threads) calculating the next generation's field. We will call the rows the thread works on: the thread's "**tile**".

For example, for a field of 6x6 cells with N=3 threads:



Corner Cases:

1. If the field height dimension cannot be split into equal parts, you should leave the remainder to the last partition.
2. The minimum granularity for each thread is a row, so upon a low-height field (height < num_of_threads), the most threads that can work on it concurrently is the number of rows in the field. This means that the **effective** number of threads is always min(num_of_threads,height).

Input:

IO in this exercise has been mostly implemented for you. The input file is composed of an unsigned integer matrix of arbitrary height and width. You may see examples attached to this exercise. Activation of the code is to be done by command line via the following:

./GameOfLife <matrixfile.txt> <number_of_generations> <number_of_threads> Y Y

Where the arguments are as follows:

1. The matrix file (examples are supplied with the source files)
2. The number of generations g to run the game where $g \in \mathbb{N}$
3. The number of threads n to run the game where $n \in \mathbb{N}$. You can assume that this number is well below the maximal number of threads allowed by the OS.
4. Whether to print in interactive mode or not (When on, the field is displayed as an animation, not simply dumped to the STDOUT). Either Y/y to turn on, or N/n to turn off. If you would like to output the prints to a file, turn this off and use:

./GameOfLife <matrixfile.txt> <number_of_generations> <number_of_threads> N Y > myfile.txt

5. Whether to print to the screen or not. You will need to turn off the prints only on the conclusion exercise at the end of Dry 3.

The file is to be parsed by the **utils::parse_lines**(see source files) and inputted into an unsigned integer matrix (the "field") on which you will calculate consecutive generations.
The program should perform the prescribed number of iterations (generations), while printing each generation's field to STDOUT.
The calculation time of each generation and each tile is appended to a history vector, which you will analyze in the Dry part of HW3.

Parallel Algorithm Sketch

Producer:

1. Create two fields: curr, next
2. Create N threads
3. for *t=0* → *t=n_generations*
   start timer
   Insert N tile-jobs to queue // phase1
   // ?
   Swap curr & next pointers
   Insert N tile-jobs to queue // phase2
   swap curr & next pointers
   stop timer
   append duration to generation history vector
   Print newly calculated field
4. Destroy field and threads.

Consumer (One of N):

while(1)
   // ?
   pop job from queue
   start timer
   execute job
   stop timer
   append duration to shared tile history vector

This algorithm is incomplete, and missing some small details you will have to figure out for yourself. A partial implementation has been supplied for your comfort, handling all output to the screen. The main logic and synchronization is left to you. More information be found in the attached source files.

Assignment Constraints and Grading Policy

1. The APIs are explained in detail in each file. You are expected to implement the functions as detailed by the API, and we presume such an implementation in our testing. You may add additional classes, members and methods as you see fit. You may also add in as many **other** hpp or cpp files as you need.
2. You must provide a **well-structured, modular, clean** and **readable** code. The correctness of multi-threaded code depends on its behavior under all possible interleaving of all the different threads. It is difficult (usually impossible) to verify the correctness of such code under normal testing, and might require us to read your code. Unreadable code results in our inability to assert its correctness.
3. This exercise is about synchronization, and not perfecting corner cases. We will not test against MATAM style errors, such as memory leaks and stack overflows. You are, however, programmers, and should not supply lousy leaky code.
4. You are furthermore prohibited from using **global variables** anywhere in the code. The usage of **pthread_cancel** or any other usage of signals is also prohibited.
5. The code provided was meant to be run with a C++ 11 compiler, which is available on the VM image you installed in HW0. **Make sure your code is working on it before submitting.**

Grading Policy:
You may lose points due to the following:

1. The lack of correctness
2. Unreasonably bad performance (usually due to a lack of sensibility in design)
3. Direct contradictions to the constraints above.

Informative Notes and Technical Details

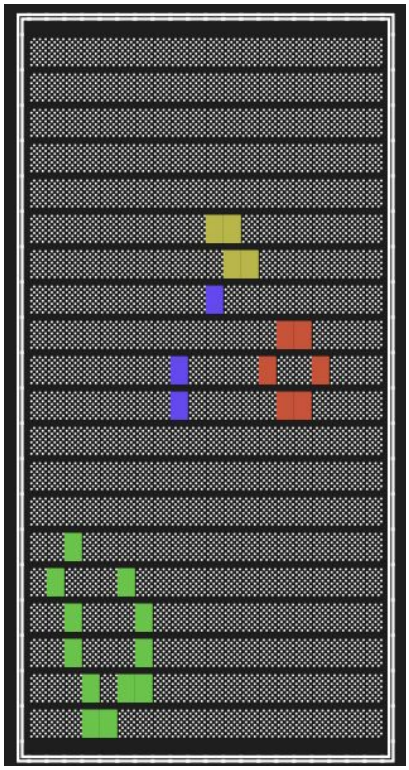
1. The file "**Headers.hpp**" holds all recommended libraries and macros for this assignment. You may change it if you feel this is imperative for your needs.
2. The "gen_hist" and "tile_hist" timing durations have been detailed in the "Algorithm Sketch" above. You will not be tested upon any aspect of the results file created, but you will utilize it in the dry part of this HW assignment. Exact placement of the time ticks is not critical, but do try to stick to the algorithm above.
3. In your implementation, you can ignore any errors that the pthreads functions may return; we will not check such cases in our tests.
4. You may presume that the input file we provide exists, and holds at least 1 row of numbers.

5. The "**Thread.hpp**" is a C++ wrapper for the C pthread_t. You can inherit it, and add members to it which would then be available for you in the thread_workload() function. **You must allocate this**

object with the new operator. A vector of your **inherited** thread class would serve as the actual "Thread Pool" which was mentioned before, along with the producer-consumer queue you have implemented. Both may be placed in the Game class as private variables.

6. Input files may be very large, so it is also recommended to allocate the field matrix on the heap, and not the stack.
7. The internet is riddled with patterns for you to train you program on, for example, here. These files usually arrive in Run-Length-Encoding which may be decoded by a Perl script we provide. More instructions may be found inside the script. We do not give any guarantees about the script's ins and outs, and it is provided solely for your experiments.

8. The printing of the field is done with non-standard characters, which may be problematic when copied via CTRL-C to simple IDEs such as Notepad. The board, when printed on your VM, should look similar (in terms of characters printed to the screen) to this: (this is samll.txt, from the supplied files)



9. Very large fields probably won't fit on the screen, so test your code on small fields, where the structure field and the printing of it is clear.

Suggested Strategy of Advancement in this Exercise

1. First read this pdf **thoroughly**.
2. Play around with the GameOfLife simulation provided, to understand what is this game about, and understand how the logic you are about to implement should look like
3. Read all header and cpp files, to understand what is given to you, and what should you implement.
4. Create a **working** serial version of this code, which means the producer also calculates the entirety of the next field. **Make sure it works**. This would allow you deal only with the synchronization and its related bugs.
5. Plan out the general synchronization scheme, by using the hints in this pdf file. If the instructions are still not clear, additional help may be found in the last part of the dry exercise. More advice below.
6. Implement the scheme, and be deadlocked for a while. Don't panic.
7. Test your implementation to make sure you receive a 100 on this exercise.
8. Submit – Don't forget your IDs.

Advice for Step (5) above

1. Most important**:** Plan out your locking scheme in advance. Make sure to protect variables that may be accessed concurrently from different threads. You will save yourself a lot of hassle if you are systematic about your approach rather than haphazardly putting in locks.
2. Review the lecture & recitation notes on the dead(live)lock subject. Assert your knowledge about the conditions where upon it can take place.
3. Use helper functions to factor out your locking code. Try to keep the number of places that you have to worry about to a minimum. Maintaining clear and modular codes helps spot odd synchronization bugs.

Submission

- You are to electronically submit a single zip file named **XXX_YYY.zip** where XXX and YYY are the IDs of the participating students.
- The zip should contain all source and header files you wrote **with no subdirectories of any kind**. The main.cpp would be supplied by us (you should not submit it!).
- The zip file should also contain a working makefile which we will use to compile your code.
- Make sure to also add to the zip a file named **submitters.txt** which includes the ID, name and email of the participating students. The following format should be used:

```
Linus Torvalds linus@gmail.com 234567890
Ken Thompson ken@belllabs.com 345678901
```

If you missed a file and because of this, the exercise is not working, **you will get 0 and resubmission will cost 10 points.** In case you missed an important file (such as the file with all your logic) we may not accept it at all. In order to prevent it you should open the zip file in a new directory and try to build and test your code in the new directory, to see that it behaves as expected.

Important Note: when you submit, **retain your confirmation code and a copy of the file(s)**, in case of technical failure. Your confirmation code is **the only valid proof** that you submitted your assignment when you did.

Have a Successful Journey,
The course staff