Operating Systems – 234123

Homework Exercise 3 – Wet

Teaching Assistant in charge:

Ido Imanuel

Assignment Subjects & Relevant Course material

Threads, Synchronization, pthread Library

Recitations 5-6, Lectures 4-5

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 (such as <mutex>, <atomic>). 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:

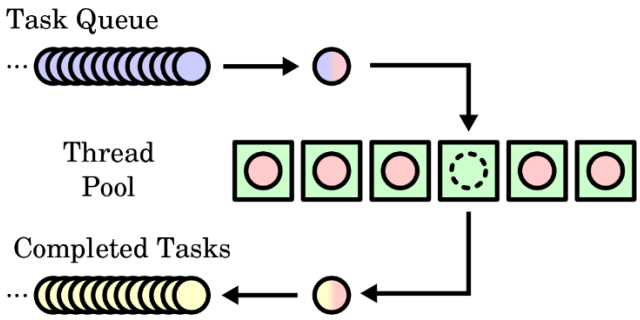
1. Seeing this is the core of your synchronization tools in the next part, we recommend you make sure your implementation **fully works** before advancing on to Part 2.
2. Please implement a **simple** Semaphore. This means that it isn't required to promise FIFO ordering.

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.

To apply one step of the rules, we count the number of live neighbors for each cell. What happens next depends on this number:
1. A dead cell with exactly three live neighbors becomes a live cell (birth).
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).

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.

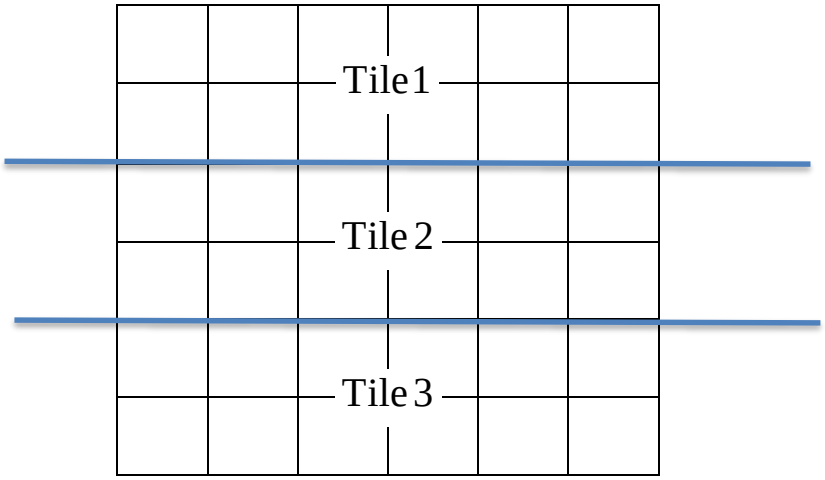
You may see a more detailed explanation here.
You may also see some wacky examples of what people have created with this simple set of rules here.

You may also play the game yourself here. This can provide a simple base line for you to test different scenarios. The GameOfLife.jar attached is also recommended.

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 as being always dead.
Every **generation** (iteration) the field is split into N equal parts, (where N is the number of threads in the threadpool) as to allow for the maximal number of concurrent threads working together. In theory, each thread would work on some set of rows of the field, calculating the next generation's field. We will call an element of this set a "**tile**".

For example, for a field of 6x6 binary 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), and this is the "true" number of threads you should work with.

Questions to think about:

1. **Does your hardware support the concurrency?** What happens if you have C cores, and N threads, where C<N? How many threads can run concurrently?
2. What happens if you only have a single core to run on? Is there any benefit in this purposed concurrency model? Is there any loss?
3. Let's say a great many generations were computed, and for each tile computed, the thread "signed" the tile completion with its ID (some integer). If you were given a timeline of these signatures over many generations, would you be able to tell the number of cores in the system from the distribution of the thread IDs? Return to this question after you have progressed some in the exercise.

Input:
IO in this exercise has been mostly implemented for you. The input file is composed of a binary 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 requested 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. After parsing the board, you will compute the effective number of threads $N \in \mathbb{N}, N \leq n$ and use it as the number of threads in the thread pool.
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::read_lines**(see source files) and inputted into a boolean 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
    - / / ?
    - 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 along with the thread's id

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. Notice that "signals" relates to the material in R7, and has nothing to do with cond_signal, which you've seen in R6.
5. The code provided was meant to be run with a C++ 11 compiler, which is available on **CSL3** or the **LUX** servers. **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. Direct contradictions to the constraints above.
3. Unreasonably bad performance (usually due to a lack of sensibility in design). This will be tested in two ways: (1) A stress test will be supplied, computing a very large matrix with N>600. (2) All tests will timeout after 2-5 minutes (larger matrices = more time). You will fail these tests if your computation takes too long. Presume that the input will not be larger than the supplied tests attached with this exercise.
   In short, please make your code **efficient**.
   A possible example: A PCQueue that delays the producer unnecessarily behind N=600 threads would take extreme amounts of time to run, and would fail the test. Please devise a mechanism to allow the producer in **with minimal delay**. This relates to the PCQueue alone, and must not relay on Game.hpp.

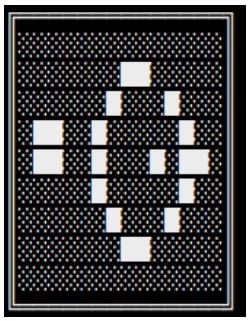
You may gain **bonus** points in this exercise for the following:

1. Fast computation time on large boards – 15 points will be supplied to the fastest implementation, 10 points for the 2nd and 3rd fastest and 7 points for places 4-10.
2. Well-structured and clean code. Usage of encapsulation and well-designed classes is recommended. Bonus here is up to 5 points.

The maximum grade in this wet part of the exercise is **120**, where bonus points over 100 **will** be added to your global HW grading computation, allowing you to enhance your final grade in the course.

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 boolean 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. Anything that is by default copied by value and added to an **std::** container such as **std::vector** or **std::queue** is allocated on the heap (the **pushback** method in std::vector, for example, takes care of this) .
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 the CSL3/LUX servers, should look similar (in terms of characters printed to the screen) to this: (this is tiny.txt, from the supplied files). Please download **MobXterm** which supports UTF-8, and can display these characters properly, and use it to log on CSL3/LUX.



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

- First read this pdf **thoroughly**.
- 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
- Read all header and cpp files, to understand what is given to you, and what should you implement.
- Setup your coding environment if needed (On the next page).
- 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.
- 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.
- Implement the scheme, and be deadlocked for a while. Don't panic.
- Test your implementation to make sure you receive a 100 on this exercise.
- Make your implementation fast, efficient and clean, to make yourself eligible for the bonus points in this exercise.
- Submit – Don't forget your IDs.

Advice for Step (6) 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.

Environment Setup (Step (4) above):

Seeing this exercise is in User Space, you are not attached to the luxurious RedHat 8.0 kernel from HW1 and HW2. If you are MacOS or Linux users, pthreads is supported and you will have no problem. Remember to assert that the code compiles on CSL3/LUX.
If you are Windows users, pthreads is **not** supported and you will need to utilize any of the following 4 options:

- Use a "modern" Linux version for a more user-friendly experience with Eclipse (via a virtual machine).
- Another possibility is to use Windows, and insert support to the Linux-only pthreads library and any other UNIX only functions, which enables compilation and debugging on Windows. CLion is recommended, as it supports pthreads on windows.
- A third – write the code in Windows (Visual Studio/Eclipse/CLion are recommended), and compile & test it on CSL3/LUX.
- The fourth and last option – Use the RedHat 8.0 kernel and write and test the code on it. Please notice that the kernel uses the incomplete LinuxThreads implementation of pthreads, as detailed in Recitation 5. For the purposes of this exercise, this is enough.
- Please note that the staff would not provide technical support with these options.
- Please take care to download **MobaXterm**, and use it as your sole tool of logging on CSL3/LUX. The printed UTF-8 characters are not supported on your good old MATAM terminal.
- If testing on a VM, please make sure that enough cores are allocated to this VM. You wouldn't want to run concurrent code with only one CPU for this exercise.

Server and Server load

You will need to test your code on the CS servers. You have two options to choose from:

1. CSL3 at **csl3.cs.technion.ac.il**
2. LUX at **lux.technion.ac.il**

They are both equipped with C++ 11 compilers, as needed to compile the exercise code. As you noticed, this exercise requires writing complex multi-threaded code. When this code is buggy, it may allocate hundreds and thousands of threads, severely taxing the servers. Due to the embedded resource limits, the servers may crash under **extreme** loads.

Note – a crashed server = You are unable to test the code until IT resets the server. Therefore, please act responsibly with the servers. You can check server load with the **uptime** command:

```
[cs234123@csl3 ~]$ uptime
 18:23:54 up 54 days,  4:01, 46 users,  load average: 3.30, 3.39, 3.46
```

You should choose the server with the lowest number marked in yellow. The crashes are rare, but usually occur on the last day or two of submission, right before the submission deadline (guess why). Therefore, please **do not** wait for the last minute to submit your exercise. **Additional submission time will not be granted due to server crash.**

FAQ

This last part of the instructions is dedicated for those who have progressed some through the exercise, and should be only be read if you have done so. It could simply be confusing otherwise.

(1) **Can I add classes, variables, methods, files, etc etc?**

Of course. Please don't destroy the encapsulation and logical separation of the data structures – it's just bad practice.

(2) **Do I have to use utils.cpp/bool_mat?**

Nope. They are there for your comfort and smiles.

(3) **Can I change supplied function signatures/ variables/ the threadpool vector etc etc?**

No. The code structure must be kept intact so we will be able to automatically test your code. You should have no troubles inserting a pointer of your inherited class into a std::vector of the parent class.

(4) **The files you supplied don't compile**.

The code skeleton, is, as the name implies is an **unfinished** piece of code, which you shouldn't expect to compile. After implementing and fixing all needed links/includes/ etc, all files should be in one directory along with the makefile and a simple "make" would do the trick. If it doesn't work immediately then - debug a bit with the output of the make command. You can also throw away the makefile and write one of your own. Please alert me on any bugs you find in the skeleton – they sadly might have slipped through.

(5) **My files don't compile/ I'm getting a weird GLIBCXX error which gives me a headache**

You forgot to use the supplied makefile – The pthread library requires non-standard flags. An external library link is also needed to support some of the code skeleton.

(6) **How can we join the consumers if they never exit? The "while(1)" means – that they are always trying to pop jobs from the queue, without stopping**

You've hit upon one of the main problems of the algorithm. The most standard and correct way to end a Producer Consumer queue of a threadpool is to supply "**poison**". This means - insert an agreed upon mark to the queue that the threads recognize as the mark to end their execution.

(7) **How do I divide work between threads? Let's say we have 20 rows and N=13. Each number in the following arrays is the number of rows that a thread will work on:**

{2,2,2,2,2,2,2,2,2,2,0,0,0}
{1,1,1,1,1,1,1,1,1,1,1,1,8}
{2,2,2,2,2,2,2,1,1,1,1,1,1}

The second option is correct. You are correct to point out that this is not the avant-garde pinnacle of modern engineering, but this decision is there to simplify the exercise. If you'd like – you may implement a different, smarter logic, which might help in making the code more efficient.

(8) **Seeing the field is split into N parts, and we have N threads – do I have to force each thread to take only <u>one</u> job?**

Nope – that's a bad idea. Image the following case: Your computer has 2 cores, and the inputted N is 200. With the forcing idea – The computation would be done at a rate of 2 threads (due to the core constraint) but we add an **additional** ≈ 198 redundant contex switch operations (at the end of every tile computation, which would then block the computing thread – seeing it is now waiting for the next batch of tiles). These are very computationally expensive. If only 2 were running – they would calculate as many tiles as they can, and only get switched if they ran out of timeslice/exited for IO. The computation – is still being done at a rate of 2 threads. Small question – are the other threads redundant? (hint: Latency Hiding)

(9) **Should I implement a reader-writers lock here?**

You are not supposed to create a readers-writers lock at all (acts like a multi-reader mutex). You are supposed to create a synchronized multiple consumer single producer queue, which, in fact - does not act like a mutex - for example, is must have some sort of condition variable or semaphore handling the "**Queue is empty"** scenario.
The consumers would be the worker threads, from class **YourImplementedThreadClass**. There should be **m_thread_num** of them, trying to extract jobs from the queue, and basically the only ones using "pop".
The producer is the main thread, that both parses the game arguments, and initializes the game and runs it and also destroys it. It plays the part of the "producer" seeing it is he that places jobs into the queue, thus "producing" some needed "goods" for the consumers. It is also the only thread that runs "push".
Note: Both Producer and Consumers are **writers**. They both write to the memory of the PCQueue – One with "I've placed something" and the other – "I've taken something". Both change the PCQueue memory.

(10) **I'm getting odd prints when outputting to a file:**

These are the clear screen "symbols" used in interactive mode. Please remember to turn it off (Pass N as a the 2nd to last argument).

(11) **Is it OK to join the threads as the generation ends, and immediately re-start the threads, for them to serve the upcoming generation?**

This means that you are not reusing any of your threads – they are there only for a single generation, after which, you again pay the price of creating them. Such a solution goes against the assignment spirit and will result in a heavy loss of points.

Questions & Answers

- The Q&A for the exercise will take place at a public forum Piazza **only**. Please **DO NOT** send questions to the private email addresses of the TAs.
- 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

Late Days

- Please **DO NOT** send postponement requests to the TA responsible for this assignment. Only the **TA in charge** can authorize postponements. In case you need a postponement, please fill out the attached form : https://goo.gl/forms/eW76r9cRNPTw9vAW2

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 does not work, **you will receive a 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