Assignment 3

Student: *Alicia Rodriguez - 5162522*

Due Date: *07/09/2017 by 11:55pm*

Part 1: Simple Multi-threaded Programming using Pthreads

Task 1.1: Simple Multi-threaded Programming without Synchronization

1 THREAD

See file `part_1/task1_1_results/1_thread_results.txt` for results.

Since there was only 1 thread created, `thread #1` will see each consecutive number and therefore a final value of 20. The value does not get affected by the `usleep` if the `if condition` becomes true because there is only 1 thread. The final value is 20 since the for loop goes up to 20 and there's only 1 thread.

2 THREADS

See file `part_1/task1_1_results/2_threads_results.txt` for results.

In this case, threads will get affected by the `usleep` and therefore some of them will finish prematurely and not reach a final value of 40.

The final value for at least the last thread that gets executed will be 40 because since the for loop goes up to 20 and there's 2 threads, then it will be 20 * 2.

5 THREADS, 10 THREADS, 50 THREADS

See the following files for results:

- `part_1/task1_1_results/5_threads_results.txt`
- `part_1/task1_1_results/10_threads_results.txt`
- `part_1/task1_1_results/50_threads_results.txt`

The final value increases exponentially as the number of threads increases. However, from the previous executions, I was expecting for at least the last thread's final value to be 100 (for 5 threads) but in this case it was not. The other executions for threads greater than 5 show the same behavior. Since there is no synchronization involved, we cannot predict what the final value will be for any thread.

Task 1.2: Simple Threads Programming with Proper Synchronization

1 THREAD

See file `part_1/task1_2_results/1_thread_results.txt` for results.

Since there is only 1 thread, the output is the same as it is without synchronization. Nothing is affecting the thread from having a final value of 20.

2 THREADS

See file `part_1/task1_2_results/2_threads_results.txt` for results.

Since we now have synchronization, all threads will finish with a final value of 40. Each `thread #1` and `thread #2` share the `shared_variable` and synchronically handle it's increment in the critical section.

5 THREADS, 10 THREADS, 100 THREADS

See the following files for results:

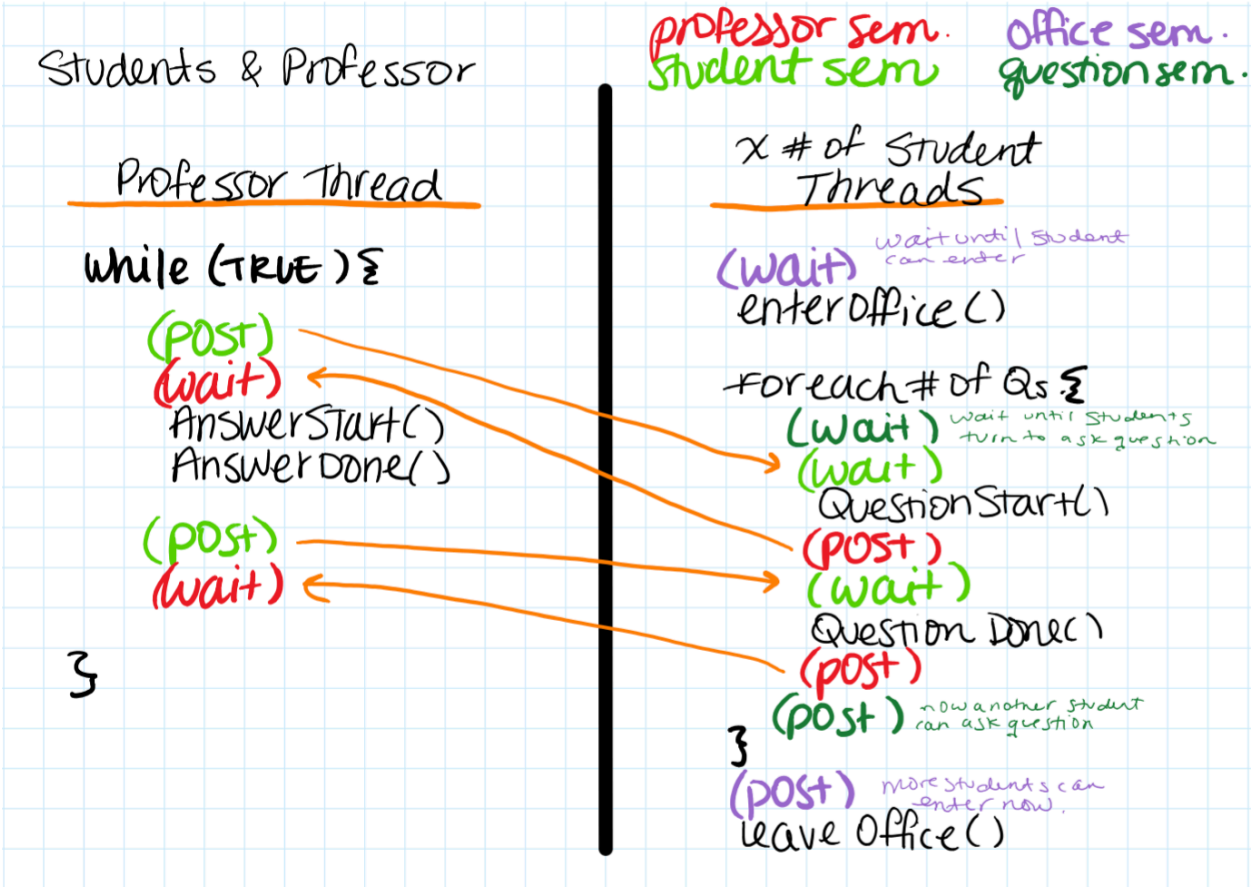
- `part_1/task1_2_results/5_threads_results.txt`
- `part_1/task1_2_results/10_threads_results.txt`
- `part_1/task1_2_results/100_threads_results.txt`

Now, the final value for each number of threads can be predicted with synchronization. Each thread will have a final value of `20 * <number_of_threads>`.

Task 1.3: Queue Scheduling Multi-Threaded Programming

For this problem, I tackled it by using only semaphores.

The following is an image of how the professor and student threads communicate. This illustrates my strategy to solve this problem.



The following is what my output looks like for 3 students and an office capacity of 2.

```
[[root@localhost part_1]# ./a.out 3 2

*** PROFESSOR'S OFFICE HOURS HAVE BEGUN ***

Student 0 enters the office.
Student 0 asks a question.
Student 1 enters the office.
Professor starts to answer question for student 0.
Professor is done with answer for student 0.
Student 0 is satisfied.
Student 0 leaves the office.
Student 2 enters the office.
Student 1 asks a question.
Professor starts to answer question for student 1.
Professor is done with answer for student 1.
Student 1 is satisfied.
Student 2 asks a question.
Professor starts to answer question for student 2.
Professor is done with answer for student 2.
Student 2 is satisfied.
Student 2 asks a question.
Professor starts to answer question for student 2.
Professor is done with answer for student 2.
Student 2 is satisfied.
Student 2 asks a question.
Professor starts to answer question for student 2.
Professor is done with answer for student 2.
Student 2 is satisfied.
Student 2 leaves the office.
Student 1 asks a question.
Professor starts to answer question for student 1.
Professor is done with answer for student 1.
Student 1 is satisfied.
Student 1 leaves the office.

*** PROFESSOR'S OFFICE HOURS HAVE CONCLUDED ***
```

Part 2: Multi-threaded/Parallel Programming using OpenMP

Task 2.1: Compile a “hello world” program

> *`helloworld.c` can be found in `part_2/OpenMP/helloworld.c`.*

Compiling the `helloworld.c` program given with `gcc -fopenmp -o helloworld helloworld.c` and then running it with `./helloworld`, these were the results:

```
Hello World from thread = 0
Number of threads = 2
Hello World from thread = 1
```

It looks like there were only 2 threads. The first thread executed and since it was the master thread, it printed out the message with the number of threads. Then it repeated again by printing out the last thread number.

Next, after setting the `OMP_NUM_THREADS` to 4, these were the results by re-running the program:

```
Hello World from thread = 3
Hello World from thread = 0
Number of threads = 4
Hello World from thread = 1
Hello World from thread = 2
```

Then, I re-ran the program a couple of more times and got a different output:

```
Hello World from thread = 0
Number of threads = 4
Hello World from thread = 2
Hello World from thread = 1
Hello World from thread = 3
```

And another time:

```
Hello World from thread = 0
Number of threads = 4
Hello World from thread = 3
Hello World from thread = 1
Hello World from thread = 2
```

It looks like there is no particular order in which the threads are being executed.

Task 2.2: Work Sharing

> *The code for work sharing can be found in `part_2/task2_2.c`.*

Output from the program has been placed in `part_2/task2_2_dynamic_results`. I ran the program 5 times, the files have been numbered in the order that they were executed.

- `task2_2_results-0.txt` shows that the only thread working was thread #3.
- `task2_2_results-1.txt` is a good example on how the threads interleave when multiple threads (2 of them) are being used. It shows that `thread #3` starts, then midway, `thread #2` starts and starts adding elements to the vector at some other index. Then when `thread #2` is done, which means it reached the end of the vectors, `thread #3` picks up where it left off from. In between, `threads #0 and #1` had begun but did not add anything to the vector, since `thread #2` had already completed adding.
- Similar behavior is shown to on `task2_2_results-2.txt` and `task2_2_results-3.txt`.
- `task2_2_results-4.txt` is also a another good example on how threads interleave, but this time it is with 3 threads.

After altering the code from `dynamic` to `static`, the results have been placed in `part_2/task2_2_static_results`.

From looking at the results, it looks like most of the time, all of the threads are interleaving and being used. Additionally I also noticed that they fill in the vector files in 10s.

So, for example:

```
Thread 1:  c[10]= 20.000000
Thread 1:  c[11]= 22.000000
Thread 1:  c[12]= 24.000000
Thread 1:  c[13]= 26.000000
```

```
Thread 1: c[14]= 28.000000
Thread 1: c[15]= 30.000000
Thread 1: c[16]= 32.000000
Thread 1: c[17]= 34.000000
Thread 1: c[18]= 36.000000
Thread 1: c[19]= 38.000000
```

and then...

```
Thread 1: c[50]= 100.000000
Thread 1: c[51]= 102.000000
Thread 1: c[52]= 104.000000
Thread 1: c[53]= 106.000000
Thread 1: c[54]= 108.000000
Thread 1: c[55]= 110.000000
Thread 1: c[56]= 112.000000
Thread 1: c[57]= 114.000000
Thread 1: c[58]= 116.000000
Thread 1: c[59]= 118.000000
```

This pattern happens if the thread does not get interrupted.

In regards to **time of execution**, the program has been altered to get the `start_time` and `end_time` of the program as well as display the execution time by subtracting the `start_time` from the `end_time`.

Task 2.3: Work Sharing with Sections Construct

> *The code for work sharing with sections construct can be found in `part_2/task2_3.c`.*

The output of the code has been placed in `part_2/task2_3_results`.

All of the outputs are different but they are similar in the way that each thread, either #0 or #1, will either do its calculation for `c[]` or `d[]`. They might interleave each other in the process but it does not alter the output.

Task 2.4: Parallelizing Matrix Multiplication

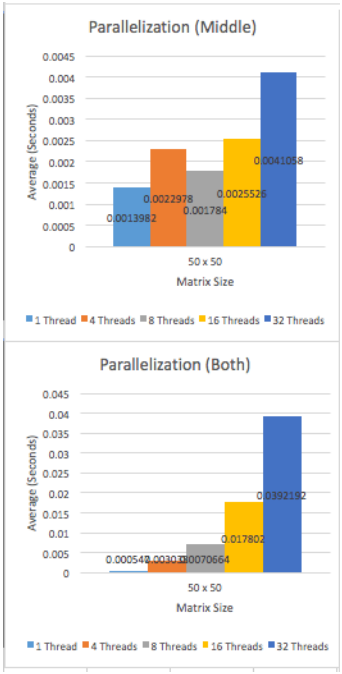
> *Data has been collected and can be found in* `part_2/task2_4_results.xlsx`.

For this part of the assignment, even though the specifications say to collect 10 data values, I only collected 5. The reason for this being is that, it is unnecessary for us to run it 10 times because we are not on a shared environment; therefore, the outputs will not vary that much. Since we are working on our own virtual machines, it does not make sense to run it many times, since there aren't other processes that can fluctuate the results. Additionally, as you can see from the excel data document, for the 5000x5000 matrices, in respect to time, I only collected 1 data value for those cases.

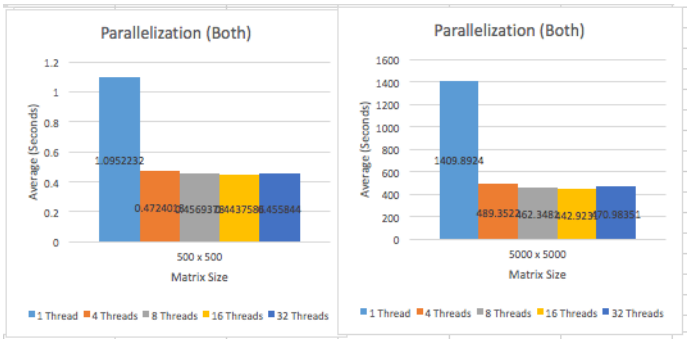
The maximum number of cores on my computer is 2. Therefore, the 2 datasets collected were with 1 CPU and 2 CPUs. Bar graphs have can be seen on the excel document, which depicts the average amount of time for each scenario.

CONCLUSIONS MADE BY OBSERVATIONS AND DATA:

- The number of cores is irrelevant for single threads. The outputs will still be the relatively the same.
- In some cases, increasing or decreasing the number of threads, increased or decreased the execution time. However, if there are more threads, usually the execution time increases. A good depiction of this can be captured by the following graphs for 1 CPU:



- In the case for Max CPU and parallelization for both loops, when executing 500 x 500 and 5000 x 5000 matrices, executing with 1 thread was almost the double amount of execution time for multiple threads. This makes sense because we are parallelizing both of the outer and middle loops. This can be depicted by the graphs below:



- Executing any of the 5000 x 5000 took almost 2 hours to complete.
- Running 50 x 50 matrix size, irrelevant of the number of threads and cores, the average execution time range was between 0.05 and 0.0005. For 500 x 500 matrix size, it was almost and sometimes exact a second and change. So the ranges stayed consistent in those cases.

Part 3: OpenMP Solution for Queue Scheduling Problem in Task 1.3

The strategy used for this part was essentially the same as task 1.3 except removing the pthreads and semaphores. Since OpenMP already handles the creation of threads with `omp_set_num_threads`, this simplified the code a lot.

Additionally, I replaced the semaphores with simple OpenMP locks and that did the trick.

The following is what my output looks like for 2 students and an office capacity of 3.

```
[[root@localhost part_3]# ./a.out 2 3

*** PROFESSOR'S OFFICE HOURS HAVE BEGUN ***

Student 1 enters the office.
Student 0 enters the office.
Student 1 asks a question.
Professor starts to answer question for student 1.
Professor is done with answer for student 1.
Student 1 is satisfied.
Student 0 asks a question.
Professor starts to answer question for student 0.
Professor is done with answer for student 0.
Student 0 is satisfied.
Student 0 leaves the office.
Student 1 asks a question.
Professor starts to answer question for student 1.
Professor is done with answer for student 1.
Student 1 is satisfied.
Student 1 leaves the office.

*** PROFESSOR'S OFFICE HOURS HAVE CONCLUDED ***
```