Hunter Black
113229605

Homework 4

Description of how I wrote and compiled my code:

To complete this homework, I had to modify the code found in thread_incr_psem.c in a number of ways. First, I created an int variable that would hold the total number of threads that are to be created by the program, naming it numThreads. I filled this variable by checking to ensure that there are indeed more than 2 command line arguments entered, and if so taking the 3rd command line argument and placing it in the numThreads variable. If there are not more than 2 command line arguments, than no number of threads are specified, and numThreads is defaulted to 2. I left the code that determined the total number of loops alone, and copied the structure of said code to achieve what I described above (i.e. obtaining the total number of threads from the cmd).

Next, I created a for loop and iterated through it numThreads times, creating a thread each time and running the threadFunc that was already in the code. I then wait for the thread to join back up, and then print out glob at the end before going back up to the top of the for loop.

I compiled this code using the makefile found within the /projects/tlpi-dist/psem directory

I created and ran the provided bashfile by creating runtest.sh in vim, copying over the script provided, and editing some minor errors that existed due to copy-pasting. I then changed the permissions of runtest.sh by running the command *sudo chmod u+x runtest.sh*, which made the bash file executable. To run the code, I used the suggested command from the assignment description, *sudo ./runtest.sh ./thread_incr_psem experiment`date +%Y%m%d%M`.csv*.

Set of all the assumptions I made to run the code:

For this homework, I am assuming “perfect’ user input. By this I mean that I am assuming that the bashfile script provided runs in the way that it is intended, and that the user is running the command correctly (i.e. by running the command found in the above section).

Bugs / corner cases handled:

There was one big corner case that popped up, which was when the threads ran through the final loop, an overflow occurs due to the number being interpreted is larger than the upper limit that computers can handle (a little over 2 billion). In this case, the expected number would be 2.2 billion. To get around this, I “ignore” the last data set pulled from running thread_incr_psem, and instead looking purely at the time it took to reach that data value.

There were a few bugs located in the provided script, namely the header file has two instances of *#!/bin/bash* which was fixed by simply removing one instance, and missing permission commands (sudo) in front of directory movement which was fixed by adding the keyword *sudo* in front of these

commands. Finally, when downloading the sqlite extension function, there was a new line added that I needed to remove, as well as a \ slash that needed to be removed.

Definition of semaphores:

A semaphore, according to lecture 12, is a variable that has a integer variable upon which only 3 operations are defined (all of which are atomic):

- May be initialized to a nonnegative integer value
- The semWait operation decrements the value
- The semSignal operation increments the value

Semaphores cannot be inspected or manipulated in any other way outside of the above three operations.

Additionally, semaphores can be broken down into 2 groups: strong and weak. A strong semaphore takes place when a process that has been blocked the longest is released from the queue first (FIFO). A weak semaphore takes place when the order in which processes are removed from the queue is not specified.

Definitions of “real time”, “kernel time”, and “user time”:

“Real Time” - According to *The Linux Programming Interface*, “Read time is measured either from some standard point (*calendar* time) or from some fixed point, typically the start, in the life of a process (*elapsed* or *wall clock* time). On UNIX systems, calendar time is measured in seconds since midnight on the morning of January 1, 1970, UCT, and coordinated on the base point for time zones defined by the longitudinal line passing through Greenwich, England”. Essentially, this is all time that a process takes up from start to finish (including time spent blocked as well as time that the CPU spends on other processes before finishing the original process)

“Kernel Time” - Kernel time, also known as Sys time, is the total amount of time the CPU spent in the kernel within the process. This includes time the CPU spends executing system calls within the kernel (rather than user-mode code / library code). Only CPU usage is clocked. Thus, time that a process spends blocked or time that the CPU spends on other processes do not count towards this time.

“User Time” - User time refers to the total amount of CPU time spent in user-mode code within a particular process. This time does not include any amount of time that the process is blocked, or any time in which the CPU is focused on other processes

Description of trends of the plots:

The trends of the plot are as expected. That is, the more loops that a certain number of threads needed to run through, the longer overall that it took for said threads to run through said loops. Additionally, the more threads that are being created, the longer it takes to run through the specified number of loops. This trend is clearly visible when examining the plots produced by running the bash script, as seen in **Figure 1.1** below.

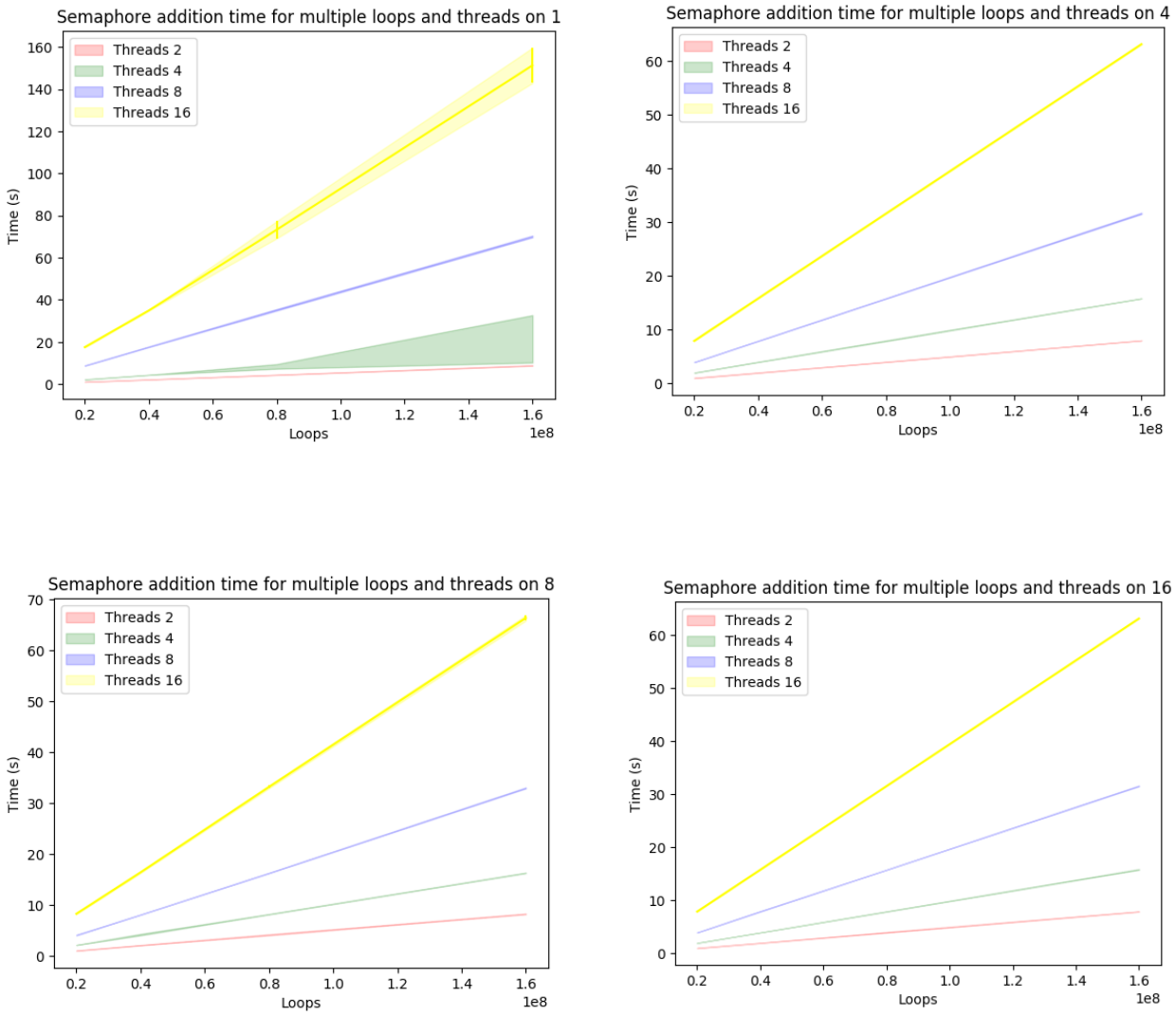

Figure 1: 1/4/8/16-core instance trends plot, moving clockwise starting from the top left plot

The trends themselves shown in the plots are nearly linear, as is made evident when viewing the csv file associated with each plot. The only main deviation from this is seen in the 1-core instance when running 4 threads, which displays two different slopes in the graph, indicating a jump in speed in the middle of running through the specified number of loops.

Explanation of the trend results:

The results of the trends seen above indicate that the total time it takes to run loop through a function with a given number of threads linearly increases as the threads increase and the number of loops increase. As is seen in the plots, the 1-core instance took the longest to run through the program, with each subsequently larger core instance taking less time. There is a noticeable improvement in speed when jumping from the 1-core instance to the 4-core instance, however the improvements in speed after the 4-core instance (i.e. the 8 and 16-core instances) is barely noticeable. This leads me to the conclusion that the overall improvement that can be expected from increasing the number of cores available is not linear but rather is inversely exponential, bottoming out after 4 cores.

Bonus

See the above paragraph for the bonus information, as well as the trend plots provided in **Figure 1**.

Code:

The .c file for thread_incr_psem is submitted separately along with this document.