Homework 4

Christopher Tse

The code

The code used for this assignment is a fork of the original `thread_incr_psem.c` file found in the tlpi sample code folder. The `threadFunc` has been left unchanged while some modifications were made to the main function.

First, instead of two `pthread_t` variables on line 51, a pointer `threadarr` is declared instead so that we can use an array to hold the variable number of threads used. On line 52, a new `int` variable `threads` is declared to store the number of threads passed in as a command line argument. It is then assigned to on line 55 with a ternary operator, storing the second command line argument into it if it exists. Then, `malloc` is used on line 57 to allocate memory for `thread` number of threads. Following that, instead of creating and joining two threads, we use a loop and iterate through the `threadarr` array to create and join the desired number of threads.

The code was edited and compiled within the `psem` folder in the tlpi code folder. It is assumed that the code is compiled using the `Makefile` found in that folder with the file name of the code added to it. It is also assumed that a compatible version of `gcc` is installed for compilation.

Some edge cases are handled through the use of functions from the tlpi header file. For example, when parsing the command line arguments for number of loops and number of threads, the built-in `getInt` function will return an error if an invalid value is parsed. The ternary operator also handles the cases where a value is not given, in which case it will assign a pre-defined default value. In the main loops for creating and joining the threads, if an error occurs where there are too many threads specified in the arguments, then it will print an error message before exiting the program.

Semaphores

A semaphore is an integer value used for signalling among processes. This is done through three atomic operations on the semaphore: initialize, decrement, and increment. The value can be initialized to a non-negative integer. There is a `semWait` operation which decrements the semaphore. If the value becomes negative, then the process which called `semWait` will become blocked; otherwise the process continues as normal. There is also the `semSignal` operation. This increments the semaphore, and if the value becomes less than or equal to

zero, a process which was previously blocked by `semWait` becomes unblocked.

There is also the distinction between strong and weak semaphores. For a strong semaphore, the unblocking process follows a FIFO pattern where the process that has been blocked the longest will be unblocked first. For a weak semaphore, it is not specified what order blocked processes become unblocked.

Time

When using the `time` function in Unix, there are three "types" of time: real time, kernel time, and user time.

- **Real time**: Real time is the time that can be measured with a wall clock. It is the complete elapsed time of all processes including time spent being blocked.
- **Kernel time**: Kernel time (or sys time) is the amount of CPU time spent in kernel mode executing system calls.
- **User time**: User time is the amount of CPU time spent in user-mode executing the process with library code.

The results

From the generated data, it seems that the time taken follows a linear trend across all 4 thread counts. Not taking into account the hiccup near the beginning of the 2-thread trial, in general it seems that the more threads are run, the longer it takes to run.

Creating more threads should theoretically decrease the time needed since there should be more threads available to do work concurrently. However, in the program the semaphore is always initialized to 1. The initial value determines how many threads can be run concurrently, therefore the same initial value causes the excess threads to stay blocked. If the semaphore value were to be changed, we should be able to see a change in the elapsed time.