FIT2070 Assignment 3 Report

Callum White
24571520

This report is to be read as a supporting document to the included C file named assignment3.c. In this report I will detail the steps I took to complete the assignment task and how I tested the code.

When the program is run by the command line it also expects two integers as arguments. e.g. the executable may be run as so: "./assignment3 5 100". If the program is not supplied with these two arguments when it is run by via. command line, the program will Segmentation Fault. The first integer is the number of threads that should be used. The second integer is the number of values each thread should compute the sum of, minus 1. For our example above, the program will create 5 threads. The first one will compute the sum of integers from 0 to 99, the second from 100 to 199, the third from 200 to 299, and so on. Therefore, the total range of values that the program returns the sum of is:

*0 to (number of threads * initial range) -1*

To implement this I created an array of threads which I would then iterate over to create each thread, along with passing it a pointer to the addValues() function, and giving it an array of type long and a size of 3. In this array, the first value was the start value for the sum calculation, the second value was the end value+1, and the third value was the thread's unique number which is used when printing the runtime of the sum calculation in each thread. At the end of each iteration, the values in the two-value array was then changed so it was ready for the next thread creation. By using this method I ended up with a weird issue: all of the threads were mysteriously receiving the same values, even though the array being passed in as a parameter to the threads was being changed in each iteration. I decided to put a sleep of 1 second within the for-loop creating the threads. Sure enough, the threads were now receiving unique values. This suggested to me that pthread_create() was taking so long to create each individual thread that by the

time it went to access the parameter's location, the for-loop had finished and the parameter's location then contained unexpected values.
To fix this issue I decided to create a two-dimensional array. This array was of the size [number of threads] and each element contained an array of type long and a size of 3, just like the array being used as a thread parameter earlier. Then, before creating all of the threads, I would iterate over this two-dimensional array, assigning each inner-array the values that would be needed for each respective thread, and each thread would be given a pointer to its own unique array when it was created.

I also tested that threads were running concurrently by putting prints within the addValues() function so that I could see threads occasionally being switched between during the program's execution.

After using a separate test file to figure out how to accurately calculate time using gettimeofday(), implementing timing was trivial. I simply had to create two arrays of timeval structs; one for the start time of each thread, and another for the end time of each thread. The start time would be calculated directly before a thread starts calculating the sum of the range given, and the end time calculated after the sum has been calculated by the thread. The start time is the subtracted from the end time, formatted, and displayed on the screen.

The total running time of the entire program is also calculated, with the start time being calculated as soon as the program starts to run, and the end time calculated just before the program is ready to print the result and then exit.

Extensive comments have been provided within the C code to detail the functionality.

Please be aware that when the program is compiled using GCC a number of warnings may be returned. The program still functions as expected, so these warnings are to be ignored.

Thank you for reviewing my assignment.