SAN DIEGO STATE UNIVERSITY

Computer Engineering 571
Professor Aksanli
Programming Assignment #1

Ethan Nagelvoort, RedID: 821234668
Ivan Cruz, RedID: 823794173

Time.h

Before discussing the three main cases to create a summation of variables, we would like to first speak upon how we calculated the run time at which the summation takes place. Due to similarity in how we calculated the run time, to implement the time function within each case, we inserted the '*sys/time'* header that has its built-in function of *gettimeofday()*. This function was provided in the link on PA1.pdf, which takes in two arguments with a data type of struct. The two arguments that are passed are the address of start/end followed by a NULL for its second argument. The two function calls are placed before and after the program body that you want to perform. For example, in each case (Baseline, Multithread, Multitasking), you provide the *gettimeofday(&start, NULL)* just before running the summation of variables and call *gettimeofday(&end, NULL)* after to end the duration calculation. After this, you are to take the difference using the formula *total = (end.tv_sec - start.tv_sec) * 1E6* where *'tv_sec'* is time calculated in seconds.

Baseline

For this case, we created a nested for loop that allows an iteration to read into the bytes of N[] and calculate its sum. Meaning, depending on the index, it will add from 0 up to the value of the current indexed position. For example, the zero index of N = 100000000 therefore our *'total'* variable will sum up the values of our iterator *i* up to N.

For the calculation of run time Baseline, please refer back to the *Time.h* heading.

Multithreading

For this case, our interpretation of assignment was that since we were given two defined array variables N and NUM_THREADS we needed to run a summation from 0:N depending on how many threads there are running. To implement this, we created two variables of N and NTH selects. The purpose of these two variables is to select the index we would like to run, which will then ultimately be passed as an argument into the parameter of our thread function, *'addup'*. For our code, we created another nested *for loop* similar to the baseline code. Such that it will run through each index of N and NUM_THREADS array. The main difference between the two codes was creating the function using *pthread_create() and pthread_join()*.

We began by first setting the N and NTH_SELECT variables with respect to the index of the iterator. The purpose of this was to set the function's condition and incrementing values.

Next, since we created a nested *for loop*, the first loop was created to run the *pthread_create()* function that served the purpose of passing in the index into our thread function (*addup)*.

Then, another loop to add the summation that each thread was totaling. In this loop, we received the values of total by inserting the *pthread_join()* function that essentially waits for each thread to complete and spews out the summation total in each thread. Our sum variable within this loop holds the total from each thread.

For the calculation of run time for Multithreading, please refer back to the *Time.h* heading.

Multitasking:(using option 1)

In the case of Multi-tasking, we first created three separate functions that are based on the number of tasks we are supposed to create. These three functions are: *'twoTasks(),' 'fourTasks(),' 'eightTasks().'* Each function accepts the *'N_Value'* in each parameter. Important to note that multitasking code size varies depending on the number of tasks. A low number of tasks can have less code than multithreading while a large number can have more.

When *twoTasks()* is called in the main function, *fork() is* used to create 2 different processes. In the child process it will obtain the partial summation using a *for loop* that will eventually write its value into a pipe. Next, the parent process will then accumulate the other portion that has not been included in the *for loop* which will be added to the value that was written in *fd[1]* to obtain the grand total of summation. We get fd[1]'s value using the read function during the parent process.

. In fourTasks(), since two fork()s are called we create 4 different processes. With two fork()s, we created three pipes that will read and write summation values that are accumulated from each different for loops. These loops start at different indexes and iterate by +4, this divides the workload evenly per process. These three pipe values are obtained from the child-child, child-parent, parent-child tasks (it is good to note that each task increments the value of initial iteration). When we reach the final task of parent-parent, we read the three pipes created, take the remaining summation starting at i=3, read all values stored in each pipe to total the grand summation value into sum5.

Last, in the *eightTasks(), since three fork() pid's* are created we have a total of 8 different processes. We first initialize an array of *fd to 16*. The purpose of this is to create a read and write index. For every even index, we use it as our *read* function during piping. On the other hand, for every odd index, we use it as our *write* function when piping. There is overall a total of 8 pipes. There is then an else if statement that determines the value of '*t'*. This value is determined on the value of the pid's and so makes t represent a different process that is happening.

t will then be used in a switch statement that has each case represent what happens in each different process. In child cases, summations occur with a for loop and that sum is stored into a pipe. In parent cases, summations occur the same way, but parents read pipes their child tasks used to get their values and add it to their sums. If this parent is a child of another task, then their sum is stored in a pipe as well. It is also important to note that summations in each process use a for loop with a different starting index for each process and is iterated with +8. This is what evenly divides the workload for all processes.

In our main function, this is where we manually select the value of N, NUM_TASKS, and the function we would like to use to accumulate the sum. Please note that for four tasks, this is in a separate C file.

Results

Baseline:

Average: N[0] = 0.239sec, N[1] = 2.500sec, N[2] = 23.932sec (of Baseline)

Standard Deviation: N[0] = 0.002, N[1] = 0.204, N[2] = 0.311

After running the baseline code and obtaining the standard deviation, we notice that as you increase the value of N your standard deviation also increases. This is due to the duration at which our code runs. We

noticed that when N = 1000000000, the duration at which the code runs takes much longer due to a much bigger space in calculation resulting in a longer time. Although our standard deviation slightly increases, we believe that our data shows a valid report as its mean nearly results in equal run times.

Multi-threading:

Average: With our results, we found that the resulting duration of run time increases in time due to the values of N being factored by 10. And also the addition of *2 threads.

	N[0]	N[1]	N[2]
2 Threads	0.301	2.784	28.671
4 Threads	0.297	2.822	27.944
8 Threads	0.298	2.903	28.975

Standard deviation: The standard deviation

STD DEV			
	N[0]	N[1]	N[2]
2 Threads	0.0152	0.1062	2.3035
4 Threads	0.0044	0.2385	1.9031
8 Threads	0.0217	0.6012	2.5699

Multi-tasking:

Average: For our result, the average have nearly had the same value even if we increase the N value 10x. Also, when each thread is being incremented by 2x, it has also kept its average. We believe that this is due to the hardware equipped with the computer being used to run the codes.

Standard deviation: The standard deviation has not increased, but in fact has decreased when we increase in threads.

AVERAGE			
	N[0]	N[1]	N[2]
2 Tasks	0.238	2.370	23.502
4 Tasks	0.238	2.385	
8 Tasks	0.242	2.380	15.013
STD DEV			
	N[0]	N[1]	N[2]
2 Threads	0.003	0.001	0.013
4 Threads	0.002	0.002	

8 Threads	0.001	0.000	0.019

1. **Explain what you observe when you increase N and why it happens that way.**
    a. As you increase N, the duration of run time increases. This happens because you're increasing memory space which will take longer to accumulate.
2. **Explain what you observe when you increase NUM_THREADS. Explain whether what you observe has any connection with an underlying hardware property of the system you are using.**
    a. For our results, we found that as we increase the number of threads, our run time decreases. I think that this happens because of the amount of CPU our computer has. Having an effect on the duration of run time.
3. **Explain what you observe when you increase NUM_TASKS. Explain whether what you observe has any connection with an underlying hardware property of the system you are using.**
    a. As you increase NUM_TASKS, the run time also becomes longer. This is also due to the fact that each computer has different CPU cores. In this case, when running the code on my Macbook, the resulting data shows the correlation of increase in NUM_TASKS results in a longer run time duration.
4. **Compare multithreading vs. multitasking. Is there a clear winner between these two? And why? Would the answer to this question change in a different system with different hardware characteristics?**
    a. Based on observations from our data, we found that Multi-threading is a clear winner between the two. We believe that this is due to the amount of core/cpu power my macbook is equipped with. Although we thought that multi-threading should run faster since due to parallelism, this wasn't the case. We believe that yes, it would change with different hardware characteristics.
5. **The values of NUM_THREADS and NUM_TASKS (with respect to the value of N) are selected to make your implementation easier. Discuss how your implementation would change if NUM_THREADS and NUM_TASKS would not evenly divide N.**
    a. If NUM_THREADS and NUM_TASKS were not evenly divided N, we believe that it would take up much more run time because the program itself is trying to loop through an enormous number byte. The implementation would change due to N not being an integer, it would have to be a different data type such as floating point or double.

Reference Page:

Understanding the use of run time code
https://www.geeksforgeeks.org/measure-execution-time-with-high-precision-in-c-c/

Understanding Multi-threading:
https://www.geeksforgeeks.org/multithreading-c-2/
Professor Aksanli's "*09_Threads_signals.pdf*"