Assignment 2 Part 1

Zachary Kaarvik

301134028

zkaarvik@sfu.ca

Question 1:

CLOCK_REALTIME : System wide clock representing nanoseconds since the Epoch (January 1, 1970). Will be affected by changes in system time.

CLOCK_MONOTONIC : Time in nanoseconds since some unspecified point. Will not be affected by changes to the system clock. Best for calculating relative time between two events on the same computer in the same session.

CLOCK_PROCESS_CPUTIME_ID : Represents the total time the current process has consumed active on the CPU. Appropriate to use to find the time that code running in the current process takes to execute, ignoring the time spent computing other processes.

CLOCK_THREAD_CPUTIME_ID: Represents the total time the current thread has consumed active on the CPU. Appropriate to use to find the time taken for the code running in the current thread to execute, ignoring time spent computing other threads or processes.

Question 2:

I found the cost of a minimal function call by timing a minimal function called in a loop many (~10000000) times. Since this does not account for the overhead implied by the loop, I also calculated the time it took for a loop of the same number of cycles to complete. The returned answer was the total time it took for the loop with the minimal function minus the bare loop time.

For both this question and question 3 I found that the results were affected by the number of loops performed. I believe this difference is due to the OS optimizing this repetition when it observes the same action reoccurring many times. Additionally, the results may be affected by the system load changing in between calculating the loop with function call and bare loop times.

Question 3:

I found the cost of a minimal system call in the exact same manner as question 2, except instead of a minimal function call I used the system call *getpid()*, I found that this operation performed consistently slower (by nearly one order of magnitude on my home machine).

I believe the difference in time taken by these minimal function/system calls is due to the system call having to switch out of user mode and into kernel mode, a much more expensive process.

Question 4:

For questions 4 and 5 I have made sure that all processes and threads spawned by the parent process will be run on the same core by setting the CPU affinity such that only core 0 of the CPU is used. In my code I have done this twice, with *sched_setaffinity()* and *pthread_setaffinity_np()*, I used both functions so that all spawned processes and threads will inherit this affinity and I don't have to manage it on a per-process or per-thread basis.

Since I have guaranteed the measurements will take place within a single core I have followed the basic measurement methodology outlined in the assignment document. I am first creating two pipes in the main thread, then forking to create a child process. The child process repeatedly waits for the main process to write a single char to its pipe (until EOF), then writes back for the main process to receive. I am measuring the time for many (~1000000) repetitions of this and taking the average to obtain my result. I did not consider loop overhead as it accounts for an insignificant fraction of the total time. Results may also be affected by the OS switching to threads besides the ones we are measuring, but we cannot prevent the OS from doing this.

Question 5:

I have followed similarly to the assignment's suggested measurement methodology, I have two threads, each wants to set the same variable and to do so they try and claim a mutex lock. Using conditional varaibles to signal current lock ownership, the threads will attempt to lock the mutex once the other thread is holding it. This causes the thread attempting the lock to have to wait until the other thread releases the lock, causing a context switch for the other process to run and unlock the mutex. This process repeats many (~10000000) times and the average switching time is taken from the total overall time.

As in question 4, the CPU affinity has been set such that the threads will only run on core 0. I have not taken into account any loop overhead into my measurement, as it will be minuscule compared to the context switch time. My results may be affected by the OS switching to other processes' threads during execution, potentially skewing my data.

Conclusion:

Included in my submission are four files corresponding to questions 2 through 5. These graphs show a histogram of the average times computed by my timing analysis.

Each calculation as described above was repeated 100 times and the results were put into histograms.

For a function call the results are distributed from 0.1 to 1.2 ns.

For a system call the results are distributed from 1 to 2.2 ns, showing the extra time required for switching to kernel mode instead of remaining in user mode.

For process switching my results are spread from 1500 to 1550 ns, or about 1.5us.

For thread switching my results range from 110 to 114ns, much faster than a process switch due to less data needing to be moved on and off the cache during this kind of process.