Operating Systems - 1: CS3510

Programming Assignment 2:
Multi-Process & Multi-Threaded
Computation of Statistics

Assignment Report

Sagar Jain - CS17BTECH11034

November 26, 2018

Program Design

Using Processes

The following points describe the code in detail.

1. The file Assgn2-ProcStat-CS17BTECH11034.cpp begins by including the necessay libraries for input-output, few system calls(creating processes), algorithms and file systems.
2. First we create a struct to store in the shared memory, ***stats***.
3. In main, we start off by initilizing the following,
   (a) ***SIZE*** i.e. the size of the shared memory we would like to create.
   (b) ***name*** i.e. the name of the shared memory object to be created or opened.
   (c) ***fd*** i.e. a nonnegative value, the file descriptor of the shared memory object created.
   (d) ***ptr*** i.e. a pointer to the mapped shared area.
   (e) ***pid*** to store the pid when a process is forked.
4. We then call shm open to create a new shared memory region. We use the arguments O CREAT / O RDWR to create a new shared memory object if one doesnt already exist and open the object for read-write access.On success, shm open returns a nonnegative file descriptor and we store it in fd.
5. We then check for errors in creating of the shared memory.
6. We then truncate the file to the required size in bytes, using ftruncate.
7. Then using mmap we store the value of the pointer to the mapped shared area in ptr. We use the argument PROT READ / PROT WRITE to allow both read and write access. We use MAP SHARED since we want to share the mapping.
8. We type cast ***ptr*** into ****stats*** and assign it to ***statsistics***.
9. We then use ***cin*** to take input from user/file and store the numbers in a ***vector***.

10. Now we use fork() to create a new child process and using the value of pid we differentiate between the child and the parent process.In the child process we then compute the mean using a loop. We store the value of the computed mean into the shared memory using the pointer statisctics.

11. In the parent process we once again use fork() to create a new child process. In this chiild process we compute the standard deviation using two for loops. Once again we store the computed standard deviation into the shared memory using the pointer statistics.

12. In the parent process we use ***fork*** once again to create another child process. In this child process we compute he median of the given numbers using the ***sort*** function. Once we have the median of the data we store it into the shared memory using the pointer statistics.

13. At every ***fork*** we use the value of the variable pid to differentiate between the child process and the parent process.

14. Now in the parent process we wait for the three child processes to terminate by invoking three ***wait*** calls.

15. Once the three child processes terminate we can start writing the output safely from the shared memory into the file.

16. We use ***ofstream*** to write the output into the file.

Using Threads

1. The file Assgn2-ThStat-CS17BTECH11034.cpp begins by including the necessay libraries for input-output, few system calls(creating threads), algorithms and file systems.

2. We first declare the following global varibales,

   (a) ***mean***

   (b) ***median***

   (c) ***standard_deviation***

3. Now we define the functions we need to compute the required statistics.

4. All the three functions return ***void**** and take only one parameter ***void**** as specified by the pthred signature.

5. In the functions we cast the void pointer into vector pointer and dereference this to perform any operations.
6. In all the three functions we end by calling ***pthread_exit*** this signifies the end of the thread.
7. In ***main***, we start off by initilizing the following,
   (a) ***n***, the number of numbers in the input.
   (b) ***nums***, a vector to store all the numbers.
   (c) ***tid1***, ***tid2***, ***tid3*** the ids of the threads we would be creating.
   (d) We also initialize the default attributes using ***pthread_attr_init***.
8. Now using pthread_create we create three threads to perform the tasks parallely.
9. pthread_create takes in four arguments,
   (a) Pointer to a pthread_t variable, i.e. pointer to the variable storing tid.
   (b) Pointer to a pthread_attr_t variable storing the attributes for the creation of the pthread.
   (c) Name of the function to be called by the thread, this function must me a voidy* returing function, and it must accept only one argument of type void*.
   (d) The void* type argument to the fucntion.
10. We then use pthread_join to wait for all the the pthreads to terminate.
11. Once all the pthreads have terminated, we can safely assume that all the global variables have been updated with the required values.
12. Now, we proceed to write the values into the file.
13. We write the output into the file using ofstream and cout.

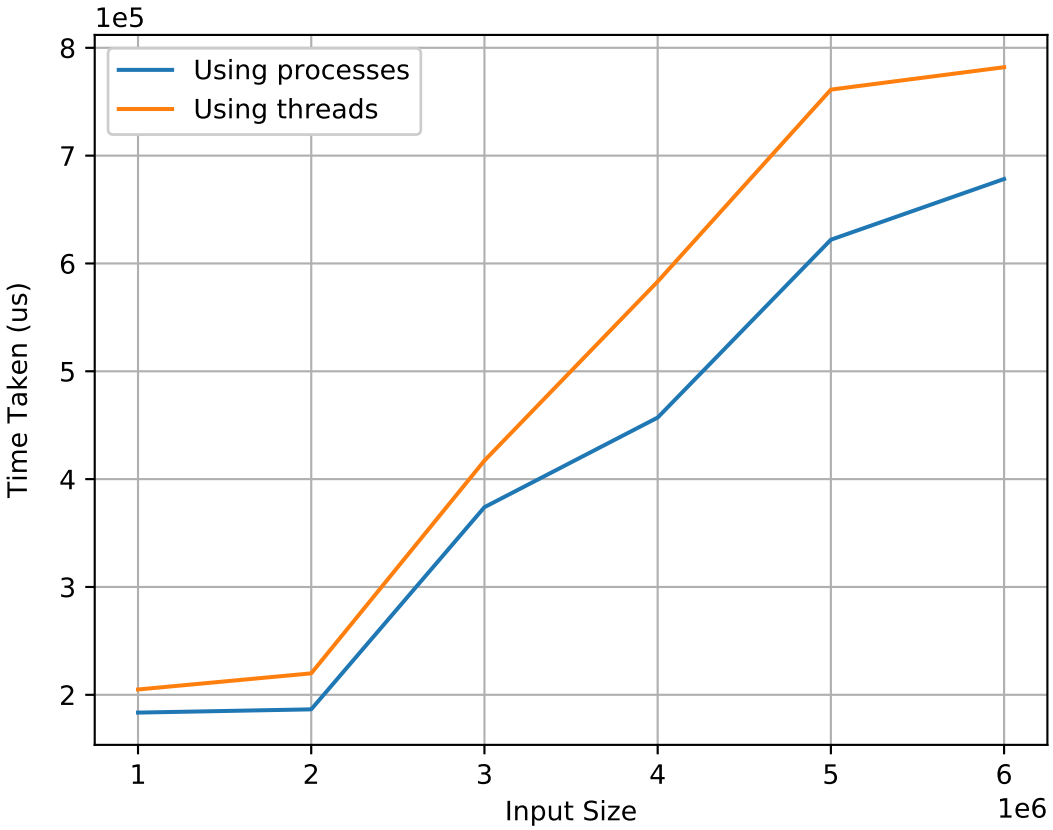
Note

For both the programs, the commented out lines are used to measure the total time taken by the code, I have used ***chrono*** to accomplish this task.

Output Analysis

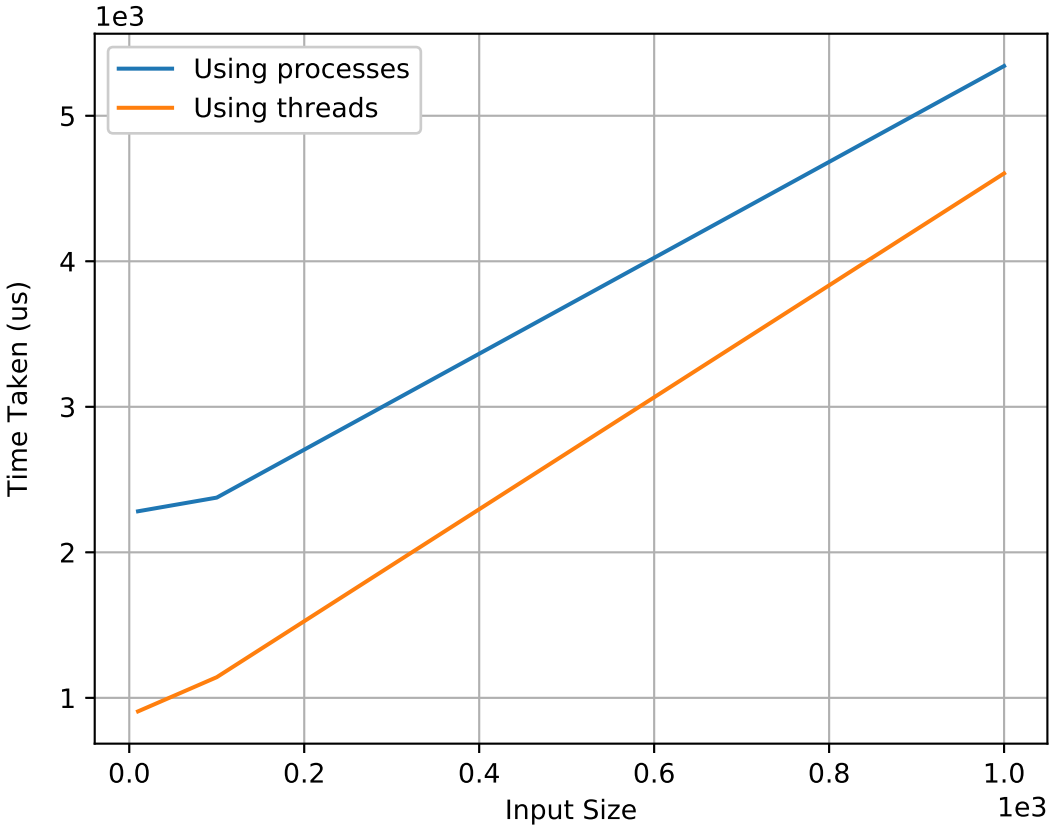
The following is the output for both the programs,

Input Size	Test Cases	Mean Time - Threads (us)	Mean Time - Processes (us)
10	10	907	2282
100	10	1142	2376
1000	10	4604	5342
10000	10	47103	39059
100000	10	204933	183488
200000	10	219885	186505
300000	10	417203	373982
400000	10	583214	457077
500000	10	761222	622016
600000	10	781995	678334



The above is the plot of the execution times for both programs.

1. The given plot shows the execution time of the two programs for different inout sizes. The output times are in microseconds.
2. It is evident that the total time taken increases with increase in input size for both the programs.
3. It is clear to see that the program using processes to compute the statistics performs better at large inputs.
4. Not only does the program using processes perform better but the difference in the performance widens with increase in input size.
5. For inputs of the order of 10^6, the program using processes executes around 25% faster than the one using threads.
6. The following is the plot of time takens by both the programs to execute for smaller inputs.



Explaination of Results

1. Before commenting on the different times taken, it must be noted that in linux, a process is not too different from a thread. The system call ***clone*** clones a task, with a configurable level of sharing, it is only in this level of sharing that a process and thread are different.

2. Performance of threads and processes may be different in other operating systems, depending on the way the two are implemented.

3. At a small input size, we see that threads perform better, this could be due to the following reasons,

   (a) Faster Context Switch between threads.

   (b) Since threads have the same address space, writing to the same address space is faster than writing to a shared memory space as is the case with IPC.

4. As we move to larger input sizes, the execution time of the program using processes gets better than the one using threads, this can be due to the fact that a process essentially has more resources than a thread($\because$ less sharing of resouces), therefore for heavy computation it performs better than a thread.

5. From the gathered results, a thread could be considered as *lite* process.