CS342-PROJECT 2

Ensar Kaya- 21502089

Implementation notes:

In this project, I implemented 2 different programs to do the same job. For all 2 programs, I used the insertion sort algorithm to sort the integer values.

For the thread program, I first created all the n threads in a for loop with parameters inputfile path and k value. Then I used pthread_join method for thread joing. I send all thread into operation function. I read the input files in a while loop and at the first line in the loop I added pthread_mutex_lock. I unlock the mutex at the end of all write operations done. So every thread can read from the files but cannot read at the same time.

For the second program, I used an array in shared memory. I open a shared memory object with shm_open(). A file descriptor has been returned if shm_open() creates a shared memory object successfully. I setted the shared memory object size which is 10000(max value of k) with ftruncate(). I mapped the shared memory object into the current address space with mmap(). After that I used fork() method in a for loop for n times. I sent each child processes to operation function. Inside the functin, I created a pointer to shared memory object and another pointer to memory map of the shared object. When I started to read from the files. At the first line of the reader while loop add added a sem_wait which means that every process can read from the files but they cannot write what they have into shared memory in case of mutual exculution. After all reads and writes done I sorted the integers one more time and print them into a output file. At the end I unmapped the shared memory with munmap() and I deleted the shared memory object with shm_unlink().

Results and Observations:

The following tables demonstrates the time spent (in miliseconds) in program executions. Count is the count of the numbers in the input files.

K=500,Count=50.000	Topk_thread	Topk_process
N=2	43618	647152
N=4	70947	172054
N=6	115558	2718931
N=8	133875	3810654
N=10	153271	4929346

Figure1- N is observed

N=5,Count=50.000	Topk_ thread	Topk_process
K=100	75857	741793
K=300	80735	2276051
K=500	117418	2553628
K=750	112532	3201319
K=1000	129505	3936884

Figure2- K is observed

N=5, K = 500	Topk_ thread	Topk_process
Count=1.000	7687	95667
Count=5.000	18301	225605
Count=20.000	50520	1024611
Count=50.000	89417	2235168
Count=100.000	154590	4898981

Figure3- Count of integers in input files are observed

The first table shows the execution time differences between the number of input files for thread and process programs for the constant k and count values. The second table shows the execution time differences between k values for the constant n and count values. The third table shows the execution time differences between count values for the constant n and k values.

First I choose insertion sort for finding minimum integer in the shared memory. Since I sorted the array repeteadly, I cared more about insertion sort's best time complexity which is O(N). For first table, I took k as 500 and count as 50.000, n is variable. I observed that multiple threads are way more faster than multiple process. Also they are much more easy to implement and handle. Since creating and destroying a process is costly and creating and destroying theads are cheaper, the results show that threads are almost 10 times faster than processes. Increasing number of input files also increases the time spent which is logical since tha amount of data must be handle is increasing.

For second table, I took n as 5 and count as 50.000, k is variable. I faced with a similiar table. Again threads were much more faster than processes. Increasing number of k is also increases the time spent which is logical since the calculation must be done increases. K value has a lot of effect on time spent when k=10000 for a count = 500000, n=5 the topk_process takes almost 8.5 minutes and when k = 5000 for a count = 500000, n=5 the topk_process takes almost 4 minutes. That's why I did not increment k value much. I think the reason is using an array in shared memory. Minheap or BST would be a better solution for this problem. Sorting an array each time costs a lot in terms of time.

For third table, I took n as 5, k as 500 and coun is variable. Threads were at least 9 times faster than processes for each cases. Increasing the count increases the time spent.

The most surprising result was the differences between topk_thread, and topk_process. The possible reasons are the followings:

-The findtopk_thread requires thread creation and management which I learnt from first project that a quite costly operation in terms of time. I wroted that "Creating and destroying n short-lived threads cost lots of time. It would be more useful when the task is more complex." . Now, I observed that when the task is complex, threads are more useful and creating, destroying a process is more costly than threads. It is more convinient to use multiaple threads in a process than use multiple processes.