Umang Srivastava
2019101090

OSN Assignment 4

Q1 : Concurrent Merge Sort

Overview

This code sorts a given array of integers using the algorithm of merge sort, first by using concurrent processes and then by using threads. Later, the run time of both methods is compared by normal merge sort and we derive a conclusion.

Approach

1. Normal Merge Sort

Merge sort is a divide and conquer algorithm. In it, we divide the input array into 2 halves, sort the 2 halves and then merge them. This goes on recursively. Here, as asked in the question, if the length of the operating array is <5, I perform a selection sort where the minimum element is put into its correct position each time the array is iterated.

2. Concurrent Merge Sort using Processes

To implement concurrent merge sort using processes, I used shmget (for shared memory allocation) and shmat (for shared memory operations). A shared memory space between parent process and child process (that is forked) is created because using a normal fork, child and parent process have their own memory space, which cannot work here. Each segment of the array is split into left and right subarray, where the left subarray is handled by the child process and the

right subarray is handled by the parent process. In the end, I merge both the subarrays to get the resultant array.

3. Concurrent Merge Sort using Threads (BONUS)

To implement concurrent merge sort using threads, I create 2 threads for each segment of the array, one for left subarray and one for right subarray and then join the threads. Since threads already share the memory space, we don't need to use shmget and shmat, as was in the case of using processes.

NOTE: In all the 3 methods, if the size of a segment is <5, we use selection sort.

Time analysis

Some examples: (All times in seconds)

N	Array	Normal Merge Sort	Merge Sort using **processes** (Normal Merge Sort was faster by ____ times on an average)	Merge Sort using **threads** (Normal Merge Sort was faster by ____ times on an average)
5	[7,5,8,6,4]	0.000004	0.000005 **(1.154772 times)**	0.000096 **(23.970730 times)**
6	[10,12,14,16,18,20]	0.000026	0.000273 **(10.545778 times)**	0.002920 **(112.902145 times)**
10	[10,9,8,7,6,5,4,3,2,1]	0.000004	0.000201 **(51.879378 times)**	0.000207 **(53.467342 times)**
20	[1,5,7,8,4,23,6,94,14,25,7,4,8,4,75,54,25,15,96,73]	0.000004	0.000776 **(190.005814 times)**	0.000538 **(131.749832 times)**
50	[50,49,.....,2,1]	0.000008	0.001961 **(237.218907 times)**	0.000971 **(117.491454 times)**

50	[27,26.....2,1,50,49....,29,28]	0.000008	0.002337 **(278.377429 times)**	0.000826 **(98.435876 times)**
100	[100,99.....2,1]	0.000014	0.004942 **(344.404047 times)**	0.008563 **(596.791570 times)**

Conclusion

After observing the time required to sort arrays by all 3 methods, the general conclusion is that time taken by using threads and the time taken using processes is more than time taken by normal merge sort.

This is because by using processes, we are creating a new child process and waiting for it, this increasing the time taken. Also , accessing shared memory takes time.

In the case of threaded merge sort, creation of threads takes time. Also in case of large sizes of array, the overhead from creating all those threads far outweighs the gains of parallel processing.