YEDİTEPE ÜNİVERSİTESİ

FALL - 2024

CSE331 - Operating Systems Design

ASSIGNMENT - 1

SECTION - 2

NAME SURNAME : ULAŞ CAN DEMİRBAĞ

ID : 20200702119

INSTRUCTOR : Prof. Dr. ŞEBNEM BAYDERE

Table of Contents

1. Introduction

In this assignment, three programming paradigm multi-process, multi-threaded, and single process programming were compared. Four programs were developed to facilitate this comparison. In an effort to minimize differences in execution times, each process was run five times. The execution times of the five runs were averaged. In each program, a 50 million-element array containing integers between 1 and 100 was created. The code was enhanced to allow the array to be divided into various sub-array sizes (10000, 25000, 50000, 100000, 250000, 500000, 1000000), with these sizes being taken as external arguments by the programs. Rather than printing the entire 50 million-element array to the screen, each program saved the initial version of the array in the file 'arrays/initial/ArrA.txt', organized by program name, and then saved the updated array in the file 'arrays/updated/ArrA.txt'. To automate this process, a sh-bash script was developed, which provided each program with different sub-array sizes as arguments, ran each program five times for each argument, and averaged the output (execution time).

2. Design and Implementation

2.1 Multiprocess (hw1a.c):

In multi-process programming, a 50-million-element array was created and divided into sub-arrays according to the given argument. The arguments are: (10000, 25000, 50000, 100000, 250000, 500000, 1000000). On every sub-array, the increment operation was performed by different sub-processes that ran simultaneously. Since each process possesses its process control block, memory assigned to every process is unique. This infers that changes in part of the array by the child process do not affect the original array. To handle this challenge, the solution used inter-process communication through shared memory. In this shared memory technique, a portion of memory is assigned to all the processes and shared among them to access. This is because, for this assignment, the array was defined as shared memory; hence, when the array gets updated in the child processes, the original array also got updated. That means each child process updates a subsection of the same array. Countdown of array size was achieved by the use of fork system call in order to create sub-arrays for the child processes. The program was run five times and then averaged.

A problem was encountered when working with sub-array sizes between 1000 and 10000 due to a constant fork() system error. As a result, the argument values had to be limited to sizes greater than 10000 to avoid this error.

Additionally, a function was written to save the entire array to a text file before and after updates, but only the first 10 elements are saved for easier testing. If viewing the entire array is desired, line 31 of the program can be modified (the code has been commented with instructions on how to make this change). This process is followed in the same way for all programs.

```
Running ./hw1a.out with argument 10000
Iteration 1: Execution time = 251.700000 ms
Iteration 2: Execution time = 224.853000 ms
Iteration 3: Execution time = 245.421000 ms
Iteration 4: Execution time = 293.627000 ms
Iteration 5: Execution time = 245.735000 ms
./hw1a.out with argument 10000: Average execution time = 252.267200 ms
Running ./hw1a.out with argument 25000
Iteration 1: Execution time = 113.271000 ms
Iteration 2: Execution time = 119.254000 ms
Iteration 3: Execution time = 110.375000 ms
Iteration 4: Execution time = 110.206000 ms
Iteration 5: Execution time = 118.290000 ms
./hw1a.out with argument 25000: Average execution time = 114.279200 ms
Running ./hw1a.out with argument 50000
Iteration 1: Execution time = 76.208000 ms
Iteration 2: Execution time = 77.741000 ms
Iteration 3: Execution time = 82.078000 ms
Iteration 4: Execution time = 82.173000 ms
Iteration 5: Execution time = 82.176000 ms
./hw1a.out with argument 50000: Average execution time = 80.075200 ms
Running ./hw1a.out with argument 100000
Iteration 1: Execution time = 78.203000 ms
Iteration 2: Execution time = 63.361000 ms
Iteration 3: Execution time = 63.647000 ms
Iteration 4: Execution time = 62.911000 ms
Iteration 5: Execution time = 63.841000 ms
./hw1a.out with argument 100000: Average execution time = 66.392600 ms
Running ./hw1a.out with argument 250000
Iteration 1: Execution time = 59.216000 ms
Iteration 2: Execution time = 59.888000 ms
Iteration 3: Execution time = 60.962000 ms
Iteration 4: Execution time = 54.537000 ms
Iteration 5: Execution time = 55.249000 ms
./hw1a.out with argument 250000: Average execution time = 57.970400 ms
Running ./hw1a.out with argument 500000
Iteration 1: Execution time = 51.616000 ms
Iteration 2: Execution time = 51.437000 ms
Iteration 3: Execution time = 51.521000 ms
Iteration 4: Execution time = 50.859000 ms
Iteration 5: Execution time = 51.443000 ms
./hw1a.out with argument 500000: Average execution time = 51.375200 ms
Running ./hw1a.out with argument 1000000
Iteration 1: Execution time = 49.248000 ms
Iteration 2: Execution time = 50.410000 ms
Iteration 3: Execution time = 49.965000 ms
Iteration 4: Execution time = 49.499000 ms
Iteration 5: Execution time = 52.093000 ms
./hw1a.out with argument 1000000: Average execution time = 50.243000 ms
```

Figure 1 the output for hw1a.c from the index.sh output

Subarray Size	Average Execution Time (ms)
10000	252.2672
25000	114.2792
50000	80.0752
100000	66.3926
250000	57.9704
500000	51.3752
1000000	50.243

Table 1 the average execution time table of hw1a.c

The average execution time of the program for all subarray sizes is approximately **0.096** seconds.

2.2 Multithreaded Program (hw1b.c & hw1c.c):

In multithreaded programming, threads share the same memory space, so the use of shared memory techniques, as required in multiprocessing, is not necessary. Each thread is able to update the same globally defined array. However, because each thread can access the array simultaneously, a race condition problem may arise. This issue has been resolved by placing a lock on the critical section where sub-arrays are updated by the threads. In our case, such a problem does not occur, as each sub-array accesses a specific index range, making the likelihood of a race condition quite low. To examine the execution time, the program was tested in two different ways. Here as well, the program test was automatically executed using index.sh, which provided different sub-array sizes as external arguments.

If you want the entire array to be saved to a text file before and after the execution in hw1b.c, it will be sufficient to update line 40 (details are provided in the comment). Similarly, for hw1c.c, updating line 37 (details are provided in the comment) will achieve the same result.

2.2.1 Mutual Exclusion (hw1b.c):

```
Running ./hw1b.out with argument 10000
Iteration 1: Execution time = 3350.024000 ms
Iteration 2: Execution time = 3347.073000 ms
Iteration 3: Execution time = 3321.483000 ms
Iteration 4: Execution time = 3314.740000 ms
Iteration 5: Execution time = 3424.023000 ms
./hw1b.out with argument 10000: Average execution time = 3351.468600 ms
Running ./hw1b.out with argument 25000
Iteration 1: Execution time = 3297.681000 ms
Iteration 2: Execution time = 3286.514000 ms
Iteration 3: Execution time = 3521.333000 ms
Iteration 4: Execution time = 3243.240000 ms
Iteration 5: Execution time = 3282.741000 ms
./hw1b.out with argument 25000: Average execution time = 3326.301800 ms
Running ./hw1b.out with argument 50000
Iteration 1: Execution time = 3407.967000 ms
Iteration 2: Execution time = 3280.220000 ms
Iteration 3: Execution time = 3397.964000 ms
Iteration 4: Execution time = 3355.029000 ms
Iteration 5: Execution time = 3349.284000 ms
./hw1b.out with argument 50000: Average execution time = 3358.092800 ms
Running ./hw1b.out with argument 100000
Iteration 1: Execution time = 3393.131000 ms
Iteration 2: Execution time = 3400.517000 ms
Iteration 3: Execution time = 3353.137000 ms
Iteration 4: Execution time = 3420.580000 ms
Iteration 5: Execution time = 3413.928000 ms
./hw1b.out with argument 100000: Average execution time = 3396.258600 ms
Running ./hw1b.out with argument 250000
Iteration 1: Execution time = 3352.175000 ms
Iteration 2: Execution time = 3265.521000 ms
Iteration 3: Execution time = 3368.366000 ms
Iteration 4: Execution time = 3372.703000 ms
Iteration 5: Execution time = 3441.265000 ms
./hw1b.out with argument 250000: Average execution time = 3360.006000 ms
Running ./hw1b.out with argument 500000
Iteration 1: Execution time = 3375.383000 ms
Iteration 2: Execution time = 3254.865000 ms
Iteration 3: Execution time = 3272.773000 ms
Iteration 4: Execution time = 3326.059000 ms
Iteration 5: Execution time = 3401.750000 ms
./hw1b.out with argument 500000: Average execution time = 3326.166000 ms
Running ./hw1b.out with argument 1000000
Iteration 1: Execution time = 3259.257000 ms
Iteration 2: Execution time = 3201.582000 ms
Iteration 3: Execution time = 3252.262000 ms
Iteration 4: Execution time = 3259.479000 ms
Iteration 5: Execution time = 3146.163000 ms
./hw1b.out with argument 1000000: Average execution time = 3223.748600 ms
```

Figure 2 the output for hw1b.c from the index.sh output

Subarray Size	Average Execution Time (ms)
10000	3351.47
25000	3326.3
50000	3358.09
100000	3396.25
250000	3360.0
500000	3326.17
1000000	3223.75

Table 2 the average execution time table of hw1b.c

The average execution time of the program for all subarray sizes in is approximately **3.33** seconds.

2.2.2 Without Mutual Exclusion (hw1c.c):

```
Running ./hw1c.out with argument 10000
Iteration 1: Execution time = 348.882000 ms
Iteration 2: Execution time = 473.298000 ms
Iteration 3: Execution time = 319.212000 ms
Iteration 4: Execution time = 319.726000 ms
Iteration 5: Execution time = 310.505000 ms
./hw1c.out with argument 10000: Average execution time = 354.324600 ms
Running ./hw1c.out with argument 25000
Iteration 1: Execution time = 135.817000 ms
Iteration 2: Execution time = 134.062000 ms
Iteration 3: Execution time = 140.186000 ms
Iteration 4: Execution time = 133.885000 ms
Iteration 5: Execution time = 134.319000 ms
./hw1c.out with argument 25000: Average execution time = 135.653800 ms
Running ./hw1c.out with argument 50000
Iteration 1: Execution time = 71.113000 ms
Iteration 2: Execution time = 77.522000 ms
Iteration 3: Execution time = 78.460000 ms
Iteration 4: Execution time = 72.863000 ms
Iteration 5: Execution time = 79.665000 ms
./hw1c.out with argument 50000: Average execution time = 75.924600 ms
Running ./hw1c.out with argument 100000
Iteration 1: Execution time = 48.617000 ms
Iteration 2: Execution time = 44.964000 ms
Iteration 3: Execution time = 47.917000 ms
Iteration 4: Execution time = 49.779000 ms
Iteration 5: Execution time = 50.184000 ms
./hw1c.out with argument 100000: Average execution time = 48.292200 ms
Running ./hw1c.out with argument 250000
Iteration 1: Execution time = 36.742000 ms
Iteration 2: Execution time = 39.118000 ms
Iteration 3: Execution time = 35.801000 ms
Iteration 4: Execution time = 35.195000 ms
Iteration 5: Execution time = 34.238000 ms
./hw1c.out with argument 250000: Average execution time = 36.218800 ms
Running ./hw1c.out with argument 500000
Iteration 1: Execution time = 33.019000 ms
Iteration 2: Execution time = 32.976000 ms
Iteration 3: Execution time = 33.253000 ms
Iteration 4: Execution time = 33.479000 ms
Iteration 5: Execution time = 33.030000 ms
./hw1c.out with argument 500000: Average execution time = 33.151400 ms
Running ./hw1c.out with argument 1000000
Iteration 1: Execution time = 33.426000 ms
Iteration 2: Execution time = 32.679000 ms
Iteration 3: Execution time = 33.163000 ms
Iteration 4: Execution time = 32.631000 ms
Iteration 5: Execution time = 32.107000 ms
./hw1c.out with argument 1000000: Average execution time = 32.801200 ms
```

Figure 3 the output for hw1c.c from the index.sh output

Subarray Size	Average Execution Time (ms)
10000	354.32
25000	135.65
50000	75.92
100000	48.29
250000	36.21
500000	33.15
1000000	32.8

Table 3 the average execution time table of hw1c.c

The average execution time of the program for all subarray sizes in is approximately **0.102** seconds.

2.3 Single Process Program (hw1d.c)

An array was created, and the elements were updated within a single for loop. The results, shown below, were obtained by running the program 5 times and averaging the execution times. Here as well, the argument 'sudizi' is used, but it is not actually utilized within the program. This was designed intentionally to prevent index.sh from generating errors during execution.

If you would like the entire array to be saved to a text file before and after the program runs, simply update line 21 (details are provided in the comments).

```
Running ./hw1d.out with argument 10000
Iteration 1: Execution time = 116.263000 ms
Iteration 2: Execution time = 114.479000 ms
Iteration 3: Execution time = 113.660000 ms
Iteration 4: Execution time = 113.602000 ms
Iteration 5: Execution time = 184.145000 ms
./hw1d.out with argument 10000: Average execution time = 128.429800 ms
Running ./hw1d.out with argument 25000
Iteration 1: Execution time = 113.049000 ms
Iteration 2: Execution time = 111.338000 ms
Iteration 3: Execution time = 111.095000 ms
Iteration 4: Execution time = 112.278000 ms
Iteration 5: Execution time = 110.472000 ms
./hw1d.out with argument 25000: Average execution time = 111.646400 ms
Running ./hw1d.out with argument 50000
Iteration 1: Execution time = 110.957000 ms
Iteration 2: Execution time = 110.251000 ms
Iteration 3: Execution time = 110.633000 ms
Iteration 4: Execution time = 110.620000 ms
Iteration 5: Execution time = 113.553000 ms
./hw1d.out with argument 50000: Average execution time = 111.202800 ms
Running ./hw1d.out with argument 100000
Iteration 1: Execution time = 114.671000 ms
Iteration 2: Execution time = 113.701000 ms
Iteration 3: Execution time = 114.445000 ms
Iteration 4: Execution time = 114.043000 ms
Iteration 5: Execution time = 113.533000 ms
./hw1d.out with argument 100000: Average execution time = 114.078600 ms
Running ./hw1d.out with argument 250000
Iteration 1: Execution time = 114.978000 ms
Iteration 2: Execution time = 113.760000 ms
Iteration 3: Execution time = 113.817000 ms
Iteration 4: Execution time = 114.911000 ms
Iteration 5: Execution time = 114.993000 ms
./hw1d.out with argument 250000: Average execution time = 114.491800 ms
Running ./hw1d.out with argument 500000
Iteration 1: Execution time = 115.078000 ms
Iteration 2: Execution time = 115.205000 ms
Iteration 3: Execution time = 113.876000 ms
Iteration 4: Execution time = 114.418000 ms
Iteration 5: Execution time = 115.043000 ms
./hw1d.out with argument 500000: Average execution time = 114.724000 ms
Running ./hw1d.out with argument 1000000
Iteration 1: Execution time = 115.294000 ms
Iteration 2: Execution time = 115.250000 ms
Iteration 3: Execution time = 112.932000 ms
Iteration 4: Execution time = 116.752000 ms
Iteration 5: Execution time = 115.955000 ms
./hw1d.out with argument 1000000: Average execution time = 115.236600 ms
```

Figure 4 the output for hw1d.c from the index.sh output

Subarray Size	Average Execution Time (ms)
10000	128.43
25000	111.65
50000	111.2
100000	114.08
250000	114.49
500000	114.72
1000000	115.24

Table 4 the average execution time table of hw1d.c

The average execution time of the program for all subarray sizes in is approximately 0.116 seconds.

2.4 Index.sh & Makefile:

In this project, a bash script and a Makefile were used to automate the compilation and testing of four C programs. The bash script executes each program with various subarray sizes as arguments, runs each program five times, and calculates the average execution time. The results are stored in a text file. The Makefile simplifies the compilation process by providing targets to compile each program and clean up the workspace, as well as a command to run the bash script for testing. This setup ensures that the programs are tested efficiently, with consistent results.

To compile and run the programs, the command **make** is used to compile all the executables. After compilation, **make run** is executed to test the programs with various subarray sizes, calculating the average execution times. Finally, **make clean** is used to remove the compiled files and clean up the workspace.

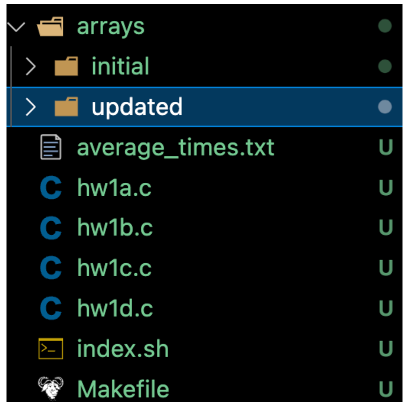

Figure 5 folder structure

The directory structure should be organized as shown in the image to maintain proper file management and execution flow. The arrays folder contains two subfolders: **initial** (to store the array's initial state) and **updated** (to store the updated array). The **average_times.txt** file records the average execution times of the programs, while the C source files (hw1a.c, hw1b.c, hw1c.c, hw1d.c) contain the code for the respective programs. The **index.sh** script automates running the programs with different subarray sizes, and the Makefile handles compilation and cleanup tasks. This structure ensures clear organization of source code, output files, and automation scripts.

3. Discussion

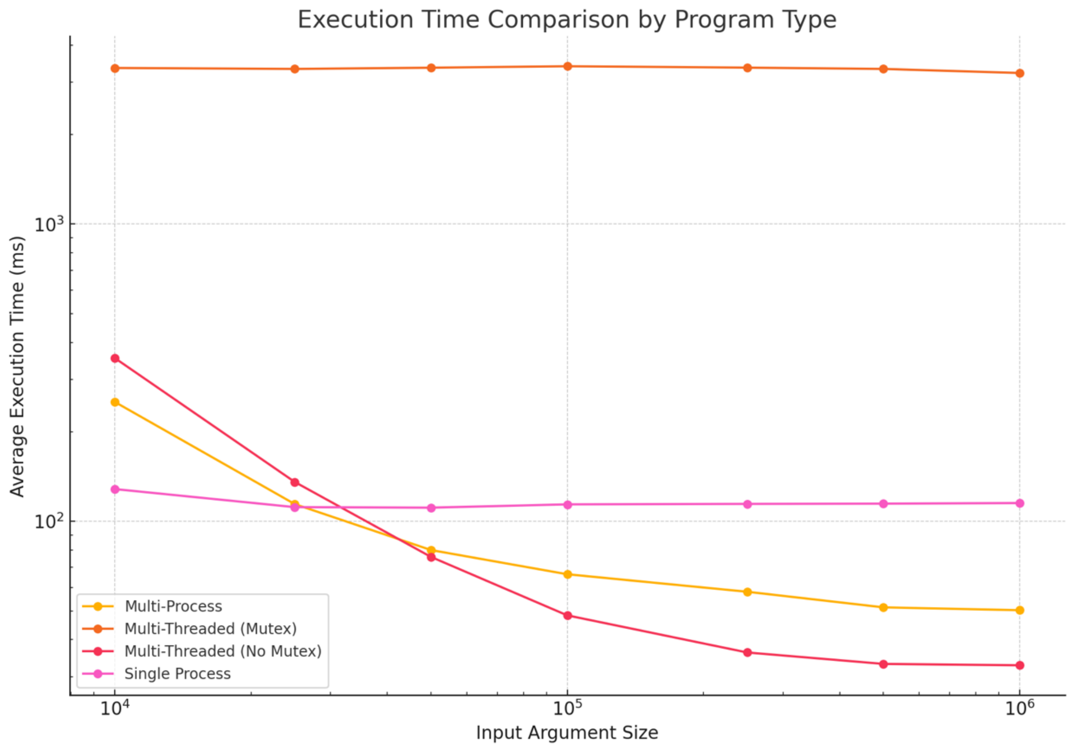

Figure 6 Execution Time Comparison by Program Type Graph

Execution time comparison shows that the multi-threaded approach without a mutex executes the best, especially for larger input sizes, as can be clearly seen from the graph in most cases by showing overall lower execution times compared to other approaches. This is due to the absence of any locking overhead, though it has the risk of race conditions:. The multi-threaded version with mutex shows slightly worse performance: the execution times at an input size of around 10^5 vary and then stabilize because operations with mutexes usually take some extra time, which is quite insignificant but provides the guarantee of data integrity because possible race conditions will be avoided. While the multiprocess approach demonstrates process isolation, its higher overhead is added due to the expense of creating and maintaining multiple processes and inter-process communication; it yields relatively higher execution times, even if the times stabilize after a value of 10^5. The single-process approach will always perform the worst since, due to its inherent lack of concurrency, its execution time plateaus around 10^5; hence, for large inputs, it is much slower. Indeed, the multi-threaded approach with proper synchronization achieves the best balanced and efficient performance in both cases, while for larger input sizes than 10^5, definite improvements are expected.

4. Conclusion

This provides the highest performance in the case of larger input sizes for the multi threaded approach with no mutex applied; there is a possibility of race conditions, though. The significant advantage of the multi-threaded version with the mutex applied is that it guarantees correctness in data while maintaining reasonable execution times. While very useful for process isolation, the multi-process approach results in a lot of redundant work, making it less than ideal for applications that require massive inter-process communication. Lastly, the latter solution using one thread, though simple, gives an idea of how slow a non-parallel execution is with large data sets. Most typically, the multi-thread solution utilizing a mutex is recommended to avoid the problems of race conditions. But here, the addition of a mutex resulted in a big loss of time. Since, in this case, race conditions are not an issue, using the start and end indexes for the sub-arrays can give the right result without using a mutex.

The conclusion is that usually, the use of a mutex ensures that the data is correct but adds time. In all cases where race conditions can be safely ignored, such as in this example, not using a mutex produces a significant performance gain. Thus, for this experiment, the avoidance of the mutex could improve the speed.