Operating Systems :

Process and Thread Management

Name : Osaid Hasan Nur

ID : 1210733

Instructor :

Dr. Bashar Tahaynah

Section : 4

Date : 25/11/2023

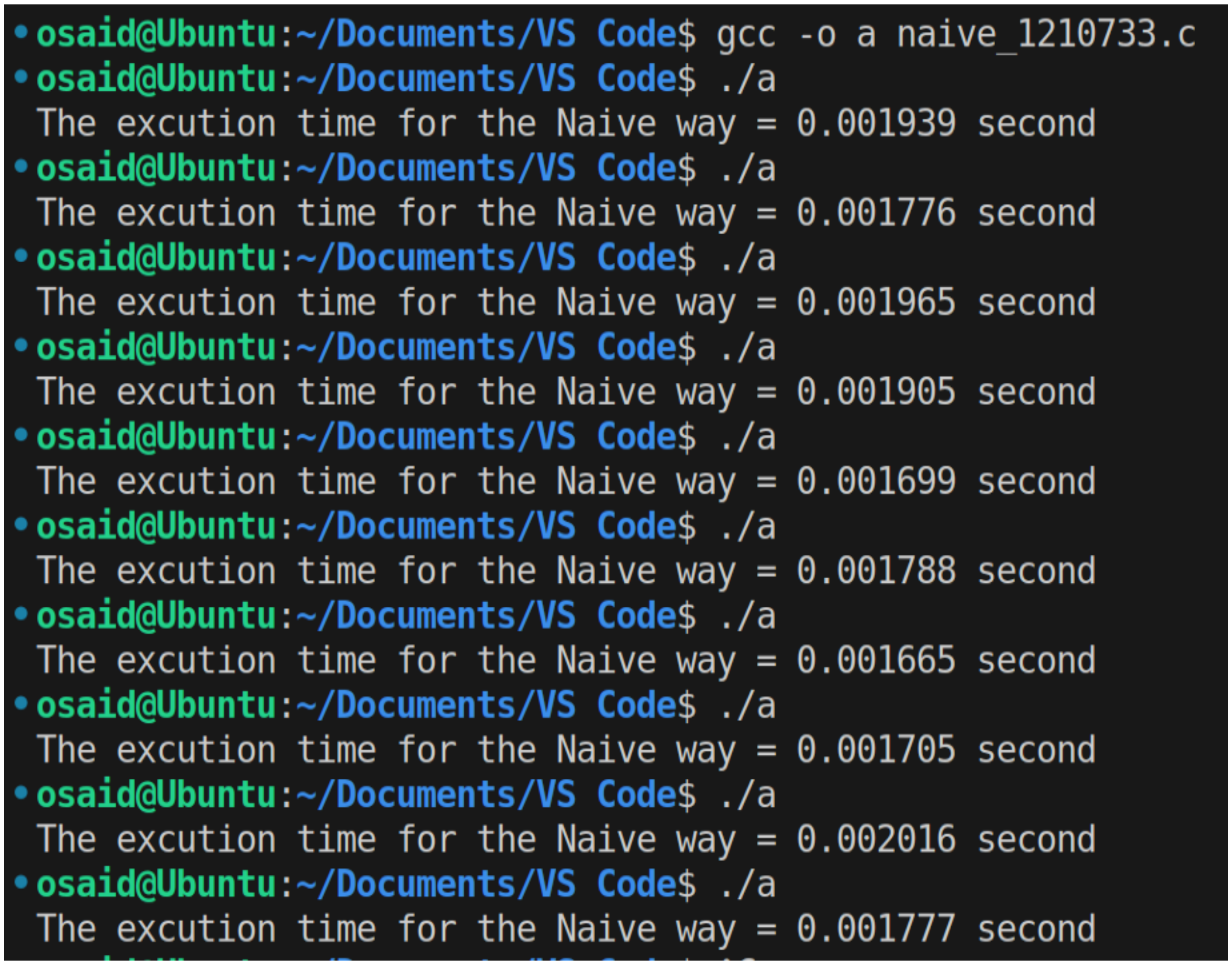
Table of Content

Introduction

In this task , we will build a program that implements matrix multiplication operation , and we will use some different ways to make this operation . First ,we will use the naive way which is the normal way that we get used to , then we will divide the operation on multiple processes and see the effect on time , after that we will divide the same operation on multiple threads , and we will use two types , joinable and detached threads , and also see the effect on the time and throughput .

Part 1 : Naive way

In this part , I implement a simple matrix multiplication function that takes two array and return an array representing the result .The implementation of a basic matrix multiplication function using a single-threaded, single-process approach may lead to poor performance, particularly for large matrices, given the O(n^3) time complexity .



```
osaid@Ubuntu:~/Documents/VS Code$ gcc -o a naive_1210733.c
osaid@Ubuntu:~/Documents/VS Code$ ./a
The excution time for the Naive way = 0.001939 second
osaid@Ubuntu:~/Documents/VS Code$ ./a
The excution time for the Naive way = 0.001776 second
osaid@Ubuntu:~/Documents/VS Code$ ./a
The excution time for the Naive way = 0.001965 second
osaid@Ubuntu:~/Documents/VS Code$ ./a
The excution time for the Naive way = 0.001905 second
osaid@Ubuntu:~/Documents/VS Code$ ./a
The excution time for the Naive way = 0.001699 second
osaid@Ubuntu:~/Documents/VS Code$ ./a
The excution time for the Naive way = 0.001788 second
osaid@Ubuntu:~/Documents/VS Code$ ./a
The excution time for the Naive way = 0.001665 second
osaid@Ubuntu:~/Documents/VS Code$ ./a
The excution time for the Naive way = 0.001705 second
osaid@Ubuntu:~/Documents/VS Code$ ./a
The excution time for the Naive way = 0.002016 second
osaid@Ubuntu:~/Documents/VS Code$ ./a
The excution time for the Naive way = 0.001777 second
```

As we noticed in the previous screenshot that the execution time is between 1700-2000 micro second . the average time is about 1850 micro second so the throughput is 1/execution time = 1/1850 = 540.54 sec^{-1} .

Notes about time calculations :

in my code I used the "struct timeval" to calculate time , including the keyword gettimeofday() , and the size of array is 100 as mentioned in the specification of the task .

I used a virtual machine to run Ubuntu on my laptop , and I used the VS code to execute my task , actually , the execution times that I had is very unstable , I don't know the reason , so I will provide each way with multiply runs so that we can take the average execution time to be more accurate .

Part 2 : Multiple processes way

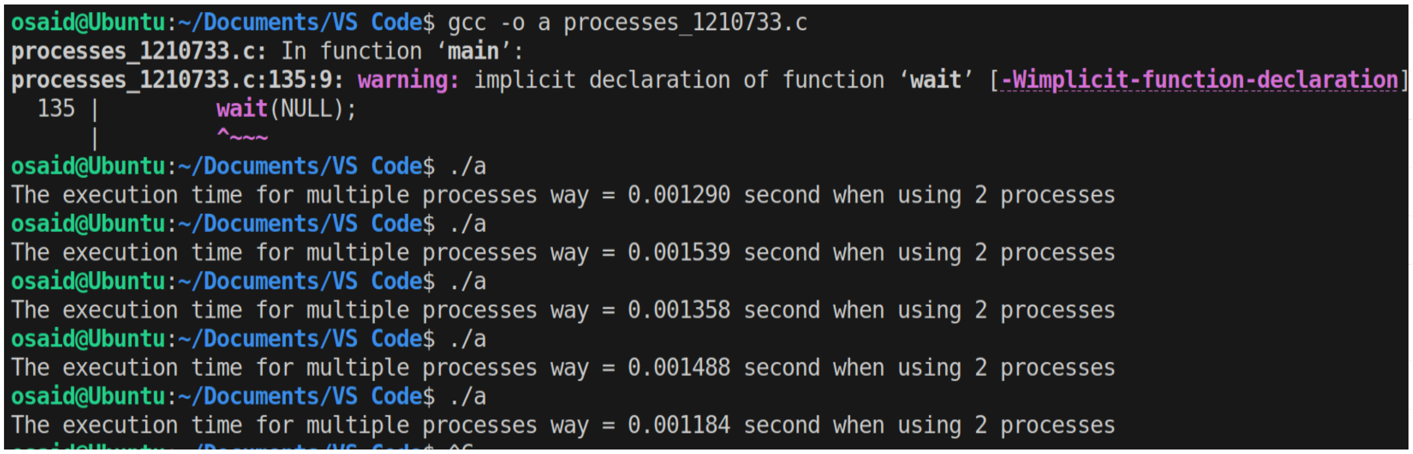
In this part , I divided the multiply operation into multiple processes , my method is to divide the rows of matrix equally between the child processes and the parent process, so I want to mention that every process in my program is working on the array , I used the pipes for IPC(InterProcess Communication) between parent and childs , I used a different pipe for each process .

My program is working as following , the main process which is the parent , make child processes as the number of processes using fork() , every child process takes its portion of the array , it perform the multiplication on this range only , and return the result array to the pipe for this children , actually , it writes the array "result" on the write side for the pipe to let the parent get this array , after it did that , it terminates , so we return the parent process which also makes another child depending on the number of processes, the new child perform the multiplication for the next range of array , and store it to the result array which will be written on the pipe , and it continues in that way .Every child don't have to communicate to any other children during childs working , as it finishes child creating , and return to the parent process , it had to perform the multiplication for the last time on the last range on array , after this step , the final array "result" have only the last range written on it , so I iterate over all pipes for all children and reads the temporary arrays that they made , every array produces by any child has only a specific range with nonzero numbers , and the rest of elements is zeros , so I combine them to form the final array , and now the final array is ready.

Notes on time calculations :

In this way I start recording the time before the creation of child processes , and end recording when the parent process finishes it's work on the last part of the array.

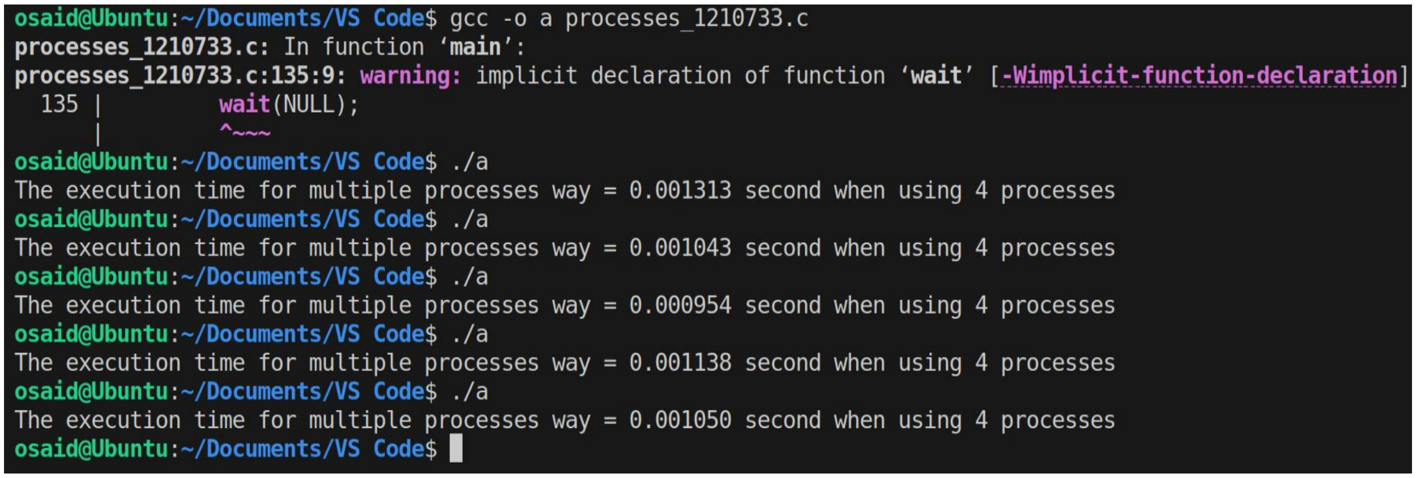
The execution time for this way is less than the naïve way , so we notice an improvement on the time , that's because the child processes are working concurrently .



```
osaid@Ubuntu:~/Documents/VS Code$ gcc -o a processes_1210733.c
processes_1210733.c: In function 'main':
processes_1210733.c:135:9: warning: implicit declaration of function 'wait' [-Wimplicit-function-declaration]
  135 |         wait(NULL);
      |         ^~~~
osaid@Ubuntu:~/Documents/VS Code$ ./a
The execution time for multiple processes way = 0.001290 second when using 2 processes
osaid@Ubuntu:~/Documents/VS Code$ ./a
The execution time for multiple processes way = 0.001539 second when using 2 processes
osaid@Ubuntu:~/Documents/VS Code$ ./a
The execution time for multiple processes way = 0.001358 second when using 2 processes
osaid@Ubuntu:~/Documents/VS Code$ ./a
The execution time for multiple processes way = 0.001488 second when using 2 processes
osaid@Ubuntu:~/Documents/VS Code$ ./a
The execution time for multiple processes way = 0.001184 second when using 2 processes
```

The average execution time when using 2 processes = 1372 micro seconds

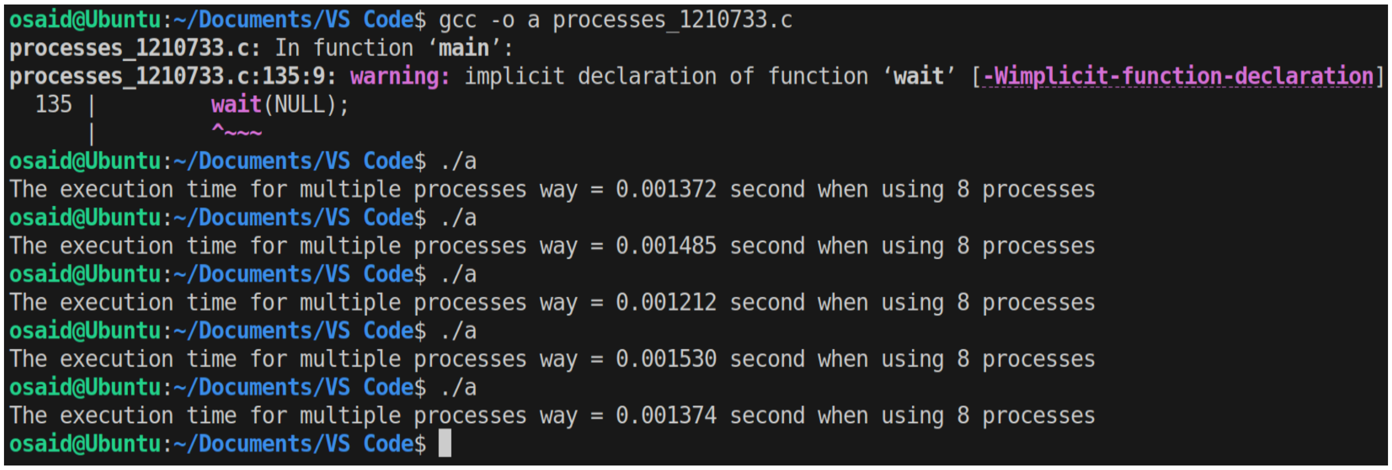
and throughput = 728.86 sec^{-1}



```
osaid@Ubuntu:~/Documents/VS Code$ gcc -o a processes_1210733.c
processes_1210733.c: In function 'main':
processes_1210733.c:135:9: warning: implicit declaration of function 'wait' [-Wimplicit-function-declaration]
  135 |         wait(NULL);
      |         ^~~~
osaid@Ubuntu:~/Documents/VS Code$ ./a
The execution time for multiple processes way = 0.001313 second when using 4 processes
osaid@Ubuntu:~/Documents/VS Code$ ./a
The execution time for multiple processes way = 0.001043 second when using 4 processes
osaid@Ubuntu:~/Documents/VS Code$ ./a
The execution time for multiple processes way = 0.000954 second when using 4 processes
osaid@Ubuntu:~/Documents/VS Code$ ./a
The execution time for multiple processes way = 0.001138 second when using 4 processes
osaid@Ubuntu:~/Documents/VS Code$ ./a
The execution time for multiple processes way = 0.001050 second when using 4 processes
osaid@Ubuntu:~/Documents/VS Code$ 
```

The average execution time when using 4 processes = 1100 micro seconds

and throughput = 909 sec^{-1}



```
osaid@Ubuntu:~/Documents/VS Code$ gcc -o a processes_1210733.c
processes_1210733.c: In function ‘main’:
processes_1210733.c:135:9: warning: implicit declaration of function ‘wait’ [-Wimplicit-function-declaration]
  135 |         wait(NULL);
      |         ^~~~
osaid@Ubuntu:~/Documents/VS Code$ ./a
The execution time for multiple processes way = 0.001372 second when using 8 processes
osaid@Ubuntu:~/Documents/VS Code$ ./a
The execution time for multiple processes way = 0.001485 second when using 8 processes
osaid@Ubuntu:~/Documents/VS Code$ ./a
The execution time for multiple processes way = 0.001212 second when using 8 processes
osaid@Ubuntu:~/Documents/VS Code$ ./a
The execution time for multiple processes way = 0.001530 second when using 8 processes
osaid@Ubuntu:~/Documents/VS Code$ ./a
The execution time for multiple processes way = 0.001374 second when using 8 processes
osaid@Ubuntu:~/Documents/VS Code$ 
```

The average execution time when using 8 processes = 1394 micro seconds

and throughput = 717.36 sec^{-1}

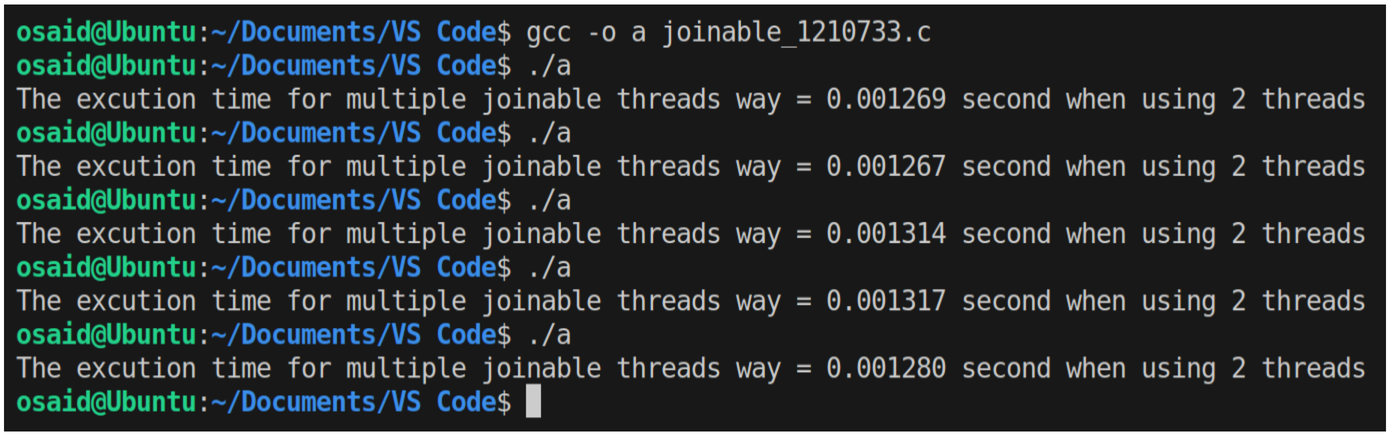
As we have seen , the throughput is improving till we reached 4 processes, and this is the optimal number of processes for the best execution time and throughput , the improvement of time is because that all childs runs concurrently , so the final array is produced almost in the same time .

Part 3 : Multiple Joinable threads

In this way , I divided the multiplication operation on multiple joinable threads , and same as processes , I depend on the matrix rows to divide the matrix equally between all threads . in the joinable threads , the main thread which is main() is creating threads from the pthreads library which are able to execute concurrently within the same program. Each thread represents an independent flow of control, allowing different tasks to be performed simultaneously. A joinable thread allows another thread (often the main thread) to wait for its completion using a function like pthread_join in POSIX library , When a thread completes its execution, it may terminate and release the resources it holds but with a joinable thread , it allows the system to keep track of its termination status and resource usage.

In my programe , I used the function threadFunc() as a thread function that takes a parameter that represents the actual thread number , and I use it to specify the range for each thread , every thread will have a unique parameter so I avoided the Race Condition in my code .

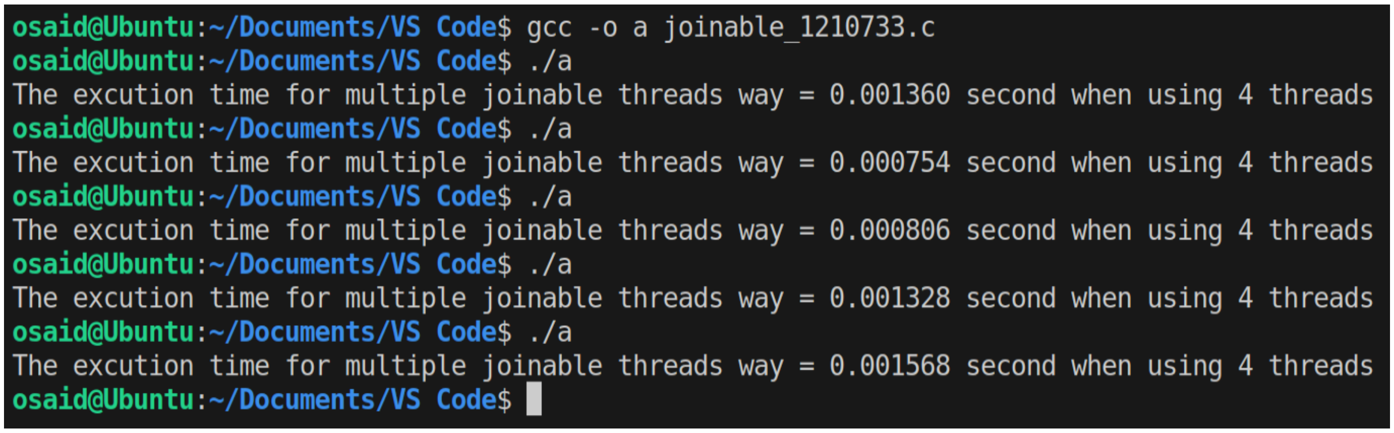
I defined the arrays in this part as global arrays to get the advantage of shared resources between threads, so that all threads can see these arrays and could modify the result array at any time without limitations. I want to mention that in this part , it’s not necessary that the first thread will take the first range of array ,and second will take the second part …etc , that’s because all threads runs concurrently , and the operating system is responsible about threads execution order . The main thread(the main function) will wait all joinable threads to finish executing , and after they all finished , it will join them to the main thread , and finish executing there .



```
osaid@Ubuntu:~/Documents/VS Code$ gcc -o a joinable_1210733.c
osaid@Ubuntu:~/Documents/VS Code$ ./a
The excution time for multiple joinable threads way = 0.001269 second when using 2 threads
osaid@Ubuntu:~/Documents/VS Code$ ./a
The excution time for multiple joinable threads way = 0.001267 second when using 2 threads
osaid@Ubuntu:~/Documents/VS Code$ ./a
The excution time for multiple joinable threads way = 0.001314 second when using 2 threads
osaid@Ubuntu:~/Documents/VS Code$ ./a
The excution time for multiple joinable threads way = 0.001317 second when using 2 threads
osaid@Ubuntu:~/Documents/VS Code$ ./a
The excution time for multiple joinable threads way = 0.001280 second when using 2 threads
osaid@Ubuntu:~/Documents/VS Code$
```

The average execution time when using 2 threads = 1289 micro seconds

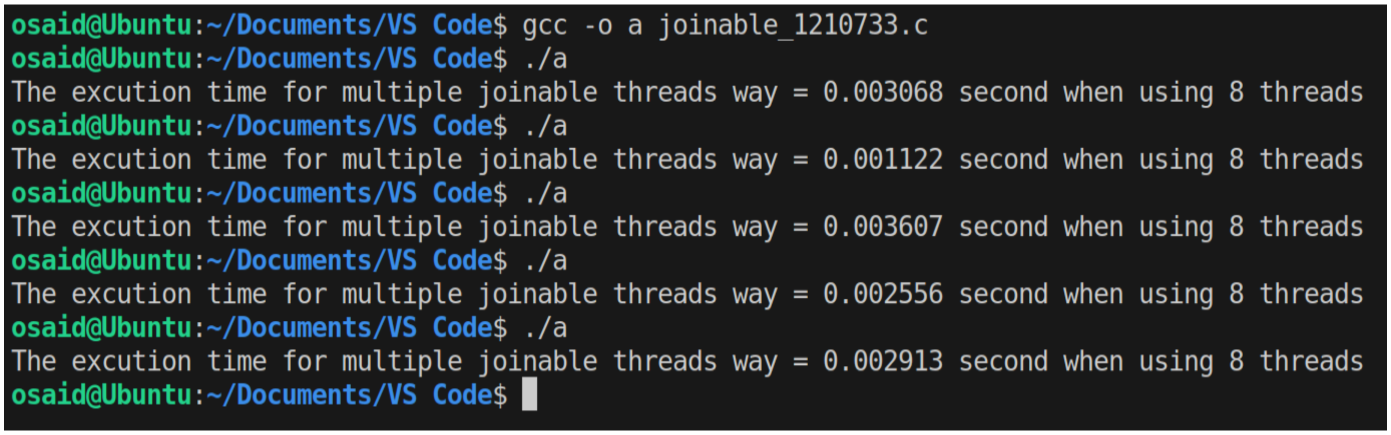
and throughput = 775.8 sec^{-1}



```
osaid@Ubuntu:~/Documents/VS Code$ gcc -o a joinable_1210733.c
osaid@Ubuntu:~/Documents/VS Code$ ./a
The excution time for multiple joinable threads way = 0.001360 second when using 4 threads
osaid@Ubuntu:~/Documents/VS Code$ ./a
The excution time for multiple joinable threads way = 0.000754 second when using 4 threads
osaid@Ubuntu:~/Documents/VS Code$ ./a
The excution time for multiple joinable threads way = 0.000806 second when using 4 threads
osaid@Ubuntu:~/Documents/VS Code$ ./a
The excution time for multiple joinable threads way = 0.001328 second when using 4 threads
osaid@Ubuntu:~/Documents/VS Code$ ./a
The excution time for multiple joinable threads way = 0.001568 second when using 4 threads
osaid@Ubuntu:~/Documents/VS Code$
```

The average execution time when using 4 threads = 1163 micro seconds

and throughput = 859.84 sec^{-1}



```
osaid@Ubuntu:~/Documents/VS Code$ gcc -o a joinable_1210733.c
osaid@Ubuntu:~/Documents/VS Code$ ./a
The excution time for multiple joinable threads way = 0.003068 second when using 8 threads
osaid@Ubuntu:~/Documents/VS Code$ ./a
The excution time for multiple joinable threads way = 0.001122 second when using 8 threads
osaid@Ubuntu:~/Documents/VS Code$ ./a
The excution time for multiple joinable threads way = 0.003607 second when using 8 threads
osaid@Ubuntu:~/Documents/VS Code$ ./a
The excution time for multiple joinable threads way = 0.002556 second when using 8 threads
osaid@Ubuntu:~/Documents/VS Code$ ./a
The excution time for multiple joinable threads way = 0.002913 second when using 8 threads
osaid@Ubuntu:~/Documents/VS Code$ 
```

The average execution time when using 8 threads = 2653 micro seconds

and throughput = 376.93 sec^{-1}

Notes on time calculations:

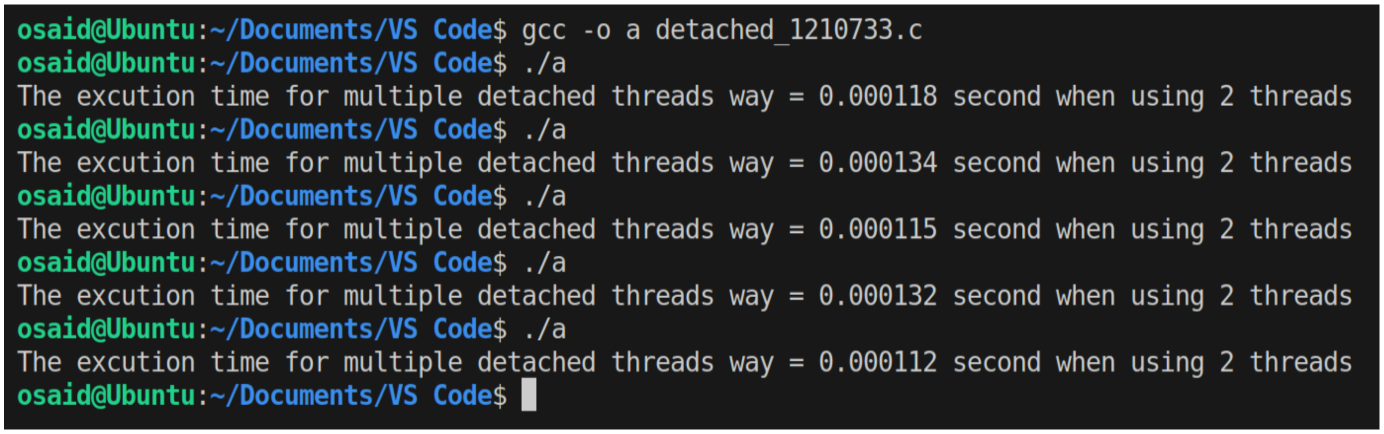
In this way I start recording the time before the creating the threads , and end recording after the main thread join the threads .

We notice that the execution time for 2 or 4 threads is the best between processes and naïve way , and the throughput has the highest value when we used 4 threads .

If we increased the number of threads higher than 4 threads , the execution time will start increase , and the throughput will get worst .

Part 4 : Multiple detached threads

In this part , I just made few changes to the joinable thread code , like adding thread attributes for the detached threads . Detached threads are threads that operate independently and don't need to be explicitly joined by another thread to clean up resources when they finish execution because when a detached thread completes its execution, the system automatically reclaims its resources without the need for another thread to call a join function. Depending on that , the main thread may terminate before another threads , because of that , we can't calculate the time in the main() block , and if we calculate the time just for testing , we'll get a very small time , and this time is representing the life time of main thread that containing thread creation , but it doesn't containing threads execution time . Because of all that , we cannot calculate the execution time .



```
osaid@Ubuntu:~/Documents/VS Code$ gcc -o a detached_1210733.c
osaid@Ubuntu:~/Documents/VS Code$ ./a
The excution time for multiple detached threads way = 0.000118 second when using 2 threads
osaid@Ubuntu:~/Documents/VS Code$ ./a
The excution time for multiple detached threads way = 0.000134 second when using 2 threads
osaid@Ubuntu:~/Documents/VS Code$ ./a
The excution time for multiple detached threads way = 0.000115 second when using 2 threads
osaid@Ubuntu:~/Documents/VS Code$ ./a
The excution time for multiple detached threads way = 0.000132 second when using 2 threads
osaid@Ubuntu:~/Documents/VS Code$ ./a
The excution time for multiple detached threads way = 0.000112 second when using 2 threads
osaid@Ubuntu:~/Documents/VS Code$ 
```

Notice that the execution time is wrong in detached threads .

Important Note

In this task , I am using the Virtual box (from Oracle) on my laptop to run the Ubuntu . When I install it , I give it 6 processors , and actually, they are 6 logical processors(threads) . To be accurate , my laptop has the processor i5 12500h with 12 physical cores and 16 threads , and this processor has a hybrid architecture with two types of processors , the first type is P-cores (Performance cores) which have high frequencies , and gives a high performance in processing , from this type the processor has 4 cores and 8 threads . The other type is E-cores(Efficient cores) which have low frequencies and it's used for background system tasks that don't need powerful processing , and from this type , the processor has 8 cores and 8 threads.

The result of all threads in the processor is 16 threads, 8 E-threads , and 8 P-threads . When I specified the resources for the VM , I give it 6 threads, and here is the problem , I don't know actually what is the type of threads assigned to the VM , because there wasn't an option to choose between P-threads and E-threads(the hybrid technology is first realeased in the 12th generation series of intel) .

As a result of that , the VM may run in different type of threads depending on the original Operating System (windows 11) which is responsible to decide which task runs on which thread , so when I execute my code , the execution time results was not stable , and to minimize the impact of that , I made multiple tests and took the average , after that , the results is more realistic now .

Summary and Conclusion

Way	Execution Time ($\mu\ sec$)	Throughput
Naïve	1850	540.54
Multiple processes	**2 processes =** 1372	728.86
	4 processes = 1100	909
	8 processes = 1394	717.36
Multiple threads (Joinable)	**2 threads =** 1289	775.8
	4 threads = 1163	859.84
	8 threads = 2653	376.93

We notice that the best execution time and throughput is when using processes , then then the multiple thread way , and the worst time and throughput is the naïve way , the reason for the efficiency of processes, is that all processes runs in the same time , and every process produce its portion from the array and give it to the parent , which combine all arrays to form the final array , in threads , it's related to hardware implementation , the main thread created some threads and give every thread its portion , and each thread will run in a single thread in the processor , as a result of that , we notice that there is a big difference between the time in 4 threads and the time in 8 threads, that's because I have 6 threads in my Virtual Machine .