CSE 331 – Assignment 1

Mahny Barazandehtar – 20210702004 – Section 4

Abstract

This project aims to compare three programming concepts: Multi-Process, Multi-Thread, and Single Process. The evaluation focuses on total execution time of each program for varying matrix sizes. The programs are written in C language.

Introduction

A **Process**, also known as a Heavy Weight Process, is an instance of a running program. Each process has its own memory space and execution context. Processes run in isolated and protected address spaces, meaning that a process cannot directly access the memory or resources of another process without using inter-process communication mechanisms provided by the OS such as pipes and shared memory.

Processes comprise two parts; Sequential Execution Steam (Threads) and Process State.

A **Thread**, also known as a Light Weight Process, represents a single sequence of execution within a process. Threads within the same process share the same memory spaces and resources, which allows for communication and efficiency. Unlike processes, many threads can run concurrently, giving the illusion of simultaneous execution of tasks.

Implementation

a) Multi-Process

The program comprises two functions: randomMatrixGenerator and printResult.

Child processes are created using the fork() system call and the wait() system call manages them.

The switch statement handles two different cases depending on the return value of the fork() system call:

Case -1 happens when the forking is unsuccessful so it prints an error regarding that and then the program terminates because of exit(1).

Case 0 happens when the child processes creation is successful.

When testing for matrices of size 900, segmentation fault (core dumped) occurred.

The execution time of the program was measure using the clock_gettime() function.

b) Multi-Thread with Mutual Exclusion

The program comprises three functions: randomMatrixGenerator, threadFunction, and printResult.

The threadFunction represents the task each thread will execute.

A Mutual Exclusion (mutex) is created for thread synchronization purposes and it is locked at the beginning of the critical section of the threadFunction to prevent multiple threads from accessing the shared resources at the same time. The mutex is unlocked after the critical section.

The pthread_create creates N new threads in a loop and pthread_join waits for the termination of those threads.

The execution time of the program was measure using the clock_gettime() function.

c) Multi-Thread without Mutual Exclusion

The program comprises three functions: randomMatrixGenerator, threadFunction, and printResult.

The threadFunction represents the task each thread will execute. The pthread_create creates N new threads in a loop and pthread_join waits for the termination of those threads.

The execution time of the program was measure using the clock_gettime() function.

d) Single Process

The program comprises three functions: randomMatrixGenerator, MatrixMultiplier, and printResult.

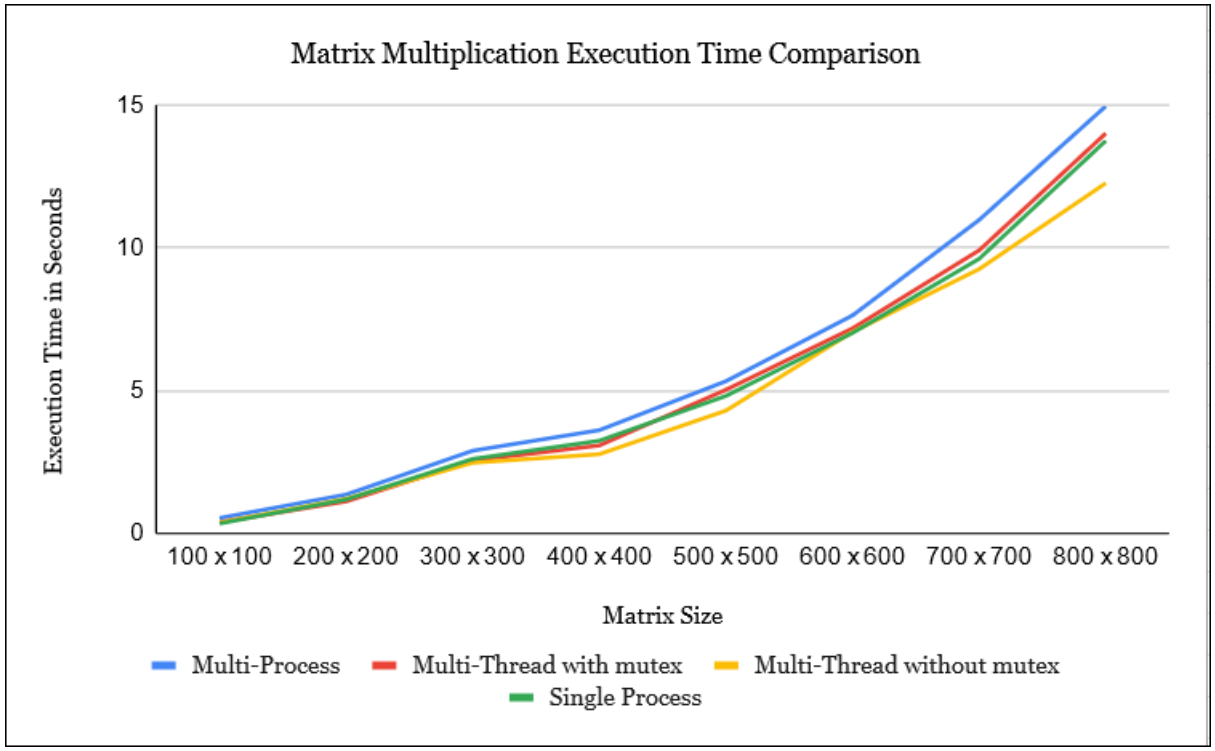
The execution time of the program was measure using the clock_gettime() function.

Results

Table

Matrix Size	Multi-Process (in seconds)	Multi-Thread with mutex (in seconds)	Multi-Thread without mutex (in seconds)	Single Process (in seconds)
100×100	0.510922	0.363938	0.345431	0.317882
200×200	1.33644	1.094904	1.18904	1.161463
300×300	2.879036	2.530787	2.455122	2.589161
400×400	3.600704	3.062744	2.75634	3.229373
500×500	5.32213	5.019678	4.288211	4.804304
600×600	7.633285	7.180137	7.052201	7.019957
700×700	10.991108	9.922853	9.262296	9.624596
800×800	14.989358	14.033666	12.284189	13.772607

Graphs



Conclusion

In the **Multi-Process** approach, N child processes are created. Each child process handles row multiplication concurrently. Since the processes are forked and never merged together, the results are always going to be zero. Due to their isolation, Processes have a high overhead as they each require their own memory space and resources making this the slowest approach. Since my laptop runs on an "Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz 2.81 GHz" processor with 4 cores, I was not able to test my code for matrices larger than 800.

In the **Multi-Thread with Mutual Exclusion** approach, N threads are created and run concurrently within a single process. A mutual exclusion mechanism has been employed to ensure that only one thread can access the critical section at a time. While this prevents race conditions, it introduces a synchronization overhead, resulting in a slightly worse performance compared to Multi-Thread without Mutual Exclusion and a slightly better one compared to Multi-Process.

In the **Multi-Thread without Mutual Exclusion**, N threads are created and run concurrently within a single process with no synchronization mechanism. Meaning that threads can access shared resources concurrently, thereby risking race conditions and inaccurate results. Despite its superior performance due to reduced overhead compared to Multi-Process and Multi-Thread with Mutual Exclusion, this method remains susceptible to errors.

The **Single Process** approach executes the task sequentially and within one process without any concurrency. It has a modest performance speed and doesn't make any compromises when it comes to safety of the memory. Due to its reliability and decent speed, it seems to be the best option for this task.