Faculty of Engineering & Technology

Electrical & Computer Engineering Department

Operating Systems

ENCS3390

Task #1

Multiprocessing/Multiprocessing task

Prepared by :
Yazan AbuAlown 1210145

Instructor : Bashar tayahneh

Date: 5/12/2023

Section : 4

1) Overview

2) Naive Approach:

When creating a program to multiply matrices without employing child processes or threads, I design a straightforward matrix multiplication function that accepts two arrays and produces an array representing the resultant matrix. However, this implementation may suffer from suboptimal performance, especially with large matrices, due to its $O(n^3)$ time complexity.

3) Child Processes:

In this section, I've introduced a modification to the matrix multiplication process by distributing the workload among distinct processes. Rather than executing the entire multiplication in a single step, each process, including the main one, is assigned a specific section of the matrix. To facilitate communication between these processes, I implemented pipes, creating individual pipes for each process. This allows them to exchange information and collaborate effectively in accomplishing the matrix multiplication task.

4) Joinable Threads:

In this section, I've restructured the matrix multiplication task by leveraging multiple joinable threads. Similar to the process-oriented approach, I evenly distributed the matrix workload among these threads, allocating tasks based on rows. Utilizing joinable threads involves the main thread, essentially the core of the program, creating distinct threads through a library named pthreads.

These threads operate concurrently, each responsible for a specific portion of the matrix multiplication. The advantage of joinable threads lies in their synchronization capabilities. The main thread can await the completion of each thread using the pthread_join function from the POSIX library. This enables the main thread to monitor their progress, ensuring orderly execution and facilitating any necessary cleanup procedures.

5) Detached Threads:

In this section, I introduced some refinements to the joinable thread implementation by incorporating thread attributes, specifically setting the threads as detached. Detached threads are designed to operate independently, eliminating the need for another thread, such as the main one, to handle resource cleanup upon their completion.

By designating threads as detached, the system automatically manages the cleanup of their resources once they finish their assigned tasks. This eliminates the necessity for another thread, like the main one, to explicitly join them. However, it's important to note a potential caveat – the main thread might complete its execution before these detached threads, and their cleanup might occur independently.

2) Analysis:

In response to the observed variability in execution times, while running Ubuntu on a virtual machine through VS Code, I took measures to enhance the accuracy of performance measurements. Specifically, I increased the number of runs for each method, doubling the iterations. This adjustment aimed to provide a more reliable and stable estimate of the average execution times.

1. **Naive Approach:**

Test number	Execution Time	Throughput
1	**0.003038**	**329.1639236**
2	**0.003053**	**327.5466754**
3	**0.003238**	**308.8326127**
4	**0.003331**	**300.2101471**
5	**0.002995**	**333.8898164**
6	**0.003121**	**320.410125**

The execution times range between 0.002995 and 0.003331 seconds. and a throughput calculation based on an average time of about 1850 microseconds (0.00185 seconds).

2. **Child Processes:**

Number Of Processes	Average Execution Time	Throughput
2	**0.000830**	1204.819277
4	**0.000786**	1272.264631
5	**0.000744**	1344.086022
10	**0.000789**	1267.427123
20	**0.001070**	934.5794393

As the number of processes increases from 2 to 10, the execution time generally decreases. This is a typical behavior, as parallel processing often leads to faster execution times.

The throughput, on the other hand, increases as the number of processes increases. This is expected since throughput is inversely related to execution time.

However, the trend changes when the number of processes becomes very large (50 and 100). In this case, the execution time increases, and throughput decreases. This could be due to overhead associated with managing a large number of processes, communication costs, or other factors.

In summary, the table suggests that parallelizing the tasks up to a certain point (around 10 processes) leads to improved performance in terms of execution time and throughput. Beyond that point, there might be diminishing returns or even performance degradation due to increased overhead.

3. **Joinable Threads:**

Number Of Threads (Joinable)	Average Execution Time	Throughput
2	**0.001116**	896.0573477
4	**0.000645**	1550.387597
5	**0.000607**	1647.446458
10	**0.000627**	1594.896332
20	**0.001204**	830.5647841

Similar to the analysis with processes, as the number of threads increases from 2 to 10, the execution time generally decreases. This is a typical behavior, as parallel processing often leads to faster execution times.

The throughput increases as the number of threads increases, indicating improved parallelism.

However, like the process analysis, when the number of threads becomes very large (50 and 100), the execution time increases, and throughput decreases. This could be due to overhead associated with managing a large number of threads, contention for resources, or other factors.

In summary, the table suggests that parallelizing the tasks using threads (joinable) up to a certain point (around 10 threads) leads to improved performance in terms of execution time and throughput. Beyond that point, there might be diminishing returns or even performance degradation due to increased overhead.

It's worth noting that the specific performance characteristics can depend on the nature of the tasks being parallelized, the architecture of the system, and other factors.

4. **Detached Threads:**

Number Of Threads (Detached)	Average Execution Time
2	**0.000255**
4	**0.00038**
5	**0.000248**
10	**0.000454**
20	**0.001345**

3) Conclusion:

In summary, the outcomes of the experimentation indicate that multiprocessing surpasses multithreading in the context of this task. Additionally, joinable threads prove to be more appropriate, efficient, and manageable compared to detached threads for this type of workload. The findings suggest a general trend where augmenting the number of processes or threads tends to enhance performance. However, it's noteworthy that pushing beyond a certain threshold may lead to detrimental effects on overall efficiency.

Crucially, all the methods adopted in this study leverage data parallelism, demonstrating their superiority over the more simplistic naïve approach. This collective evidence underscores the importance of thoughtful consideration and optimization strategies when implementing parallel processing techniques for matrix multiplication tasks.